Programming Environment, Run-Time System and Simulator for Many-Core Machines

Olivier Certner

▶ **To cite this version:**

Olivier Certner. Programming Environment, Run-Time System and Simulator for Many-Core Machines. Distributed, Parallel, and Cluster Computing [cs.DC]. Université Paris Sud - Paris XI, 2010. English. NNT : . tel-00826616

HAL Id: tel-00826616

https://theses.hal.science/tel-00826616v1

Submitted on 27 May 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE PARIS-SUD

U.F.R. SCIENTIFIQUE D'ORSAY

THÈSE

présentée pour obtenir le grade de

DOCTEUR EN SCIENCES

DE L'UNIVERSITÉ DE PARIS XI ORSAY

SPÉCIALITÉ : *Informatique*

par

OLIVIER CERTNER

Sujet :

Environnement de programmation, support à l'exécution et simulateur pour machines à grand nombre de cœurs

Programming Environment, Run-Time System and Simulator for Many-Core Machines

Soutenue le 15 décembre 2010 devant le jury composé de :

M.	Lieven	Eeckhout	Rapporteur
M.	François	Bodin	Rapporteur
M.	Pascal	Sainrat	Examinateur
M.	Jean-José	Bérenguer	Examinateur
M.	Bruno	Jégo	Examinateur
M.	Olivier	Temam	Directeur

© 2010–2011, 2013 Olivier Certner

Tous droits réservés pour tous pays.
Imprimé en France.

All rights reserved.
Printed in France.

Revisions of this document:

0.1 This is the first version released to the jury. It missed the content of the "Distributed Work Management" chapter, the "General Conclusion", some appendices and a summary of the thesis in French.

0.2 The contents of the "Distributed Work Management" (Chapter 9) and of the "General Conclusion" are now released. An appendix containing some example Capsule code has been added.

1.0 This is the version that was released to the jury on the defense day. It includes minor modifications. Most of the suggestions by some members of the jury are among them.

1.1 Added dedications and acknowledgements.

1.2 Added an appendix (some example code for the Capsule distributed-memory run-time system). Added an index. Reworked some figures. Minor corrections throughout the manuscript.

À mes grands-parents
À mes professeurs

Résumé

L'accroissement régulier de la fréquence des micro-processeurs et des importants gains de puissance qui en avaient résulté ont pris fin en 2005. Les autres principales techniques matérielles d'amélioration de performance pour les programmes séquentiels (exécution dans le désordre, antémémoires, prédiction de branchement, etc.) se sont largement essouflées. Conduisant à une consommation de puissance toujours plus élevée, elles posent de délicats problèmes économiques et techniques (refroidissement des puces). Pour ces raisons, les fabricants de micro-processeurs ont choisi d'exploiter le nombre croissant de transistors disponibles en plaçant plusieurs cœurs de processeurs sur une même puce.

Par cette thèse, nous avons pour objectif et ambition de préparer l'arrivée probable, dans les prochaines années, de processeurs multi-cœur à grand nombre de cœurs (une centaine ou plus). À cette fin, nos recherches se sont orientées dans trois grandes directions. Premièrement, nous améliorons l'environnement de parallélisation Capsule, dont le principe est la parallélisation conditionnelle, en lui adjoignant des primitives de synchronization de tâches robustes et en améliorant sa portabilité. Nous étudions ses performances et montrons ses gains par rapport aux approches usuelles, aussi bien en terme de rapidité que de stabilité du temps d'exécution.

Deuxièmement, la compétition entre de nombreux cœurs pour accéder à des mémoires aux performances bien plus faibles va vraisemblablement conduire à répartir la mémoire au sein des futures architectures multi-cœur. Nous adaptons donc Capsule à cette évolution en présentant un modèle de données simple et général qui permet au système de déplacer automatiquement les données en fonction des accès effectués par les programmes. De nouveaux algorithmes répartis et locaux sont également proposés pour décider de la création effective des tâches et de leur répartition.

Troisièmement, nous développons un nouveau simulateur d'évènements discrets, SiMany, pouvant prendre en charge des centaines à des milliers de cœurs. Nous montrons qu'il reproduit fidèlement les variations de temps d'exécution de programmes observées sur un simulateur de niveau cycle jusqu'à 64 cœurs. SiMany est au moins 100 fois plus rapide que les meilleurs simulateurs flexibles actuels. Il permet l'exploration d'un champ plus large d'architectures ainsi que l'étude des grandes lignes du comportement des logiciels sur celles-ci, ce qui en fait un outil majeur pour le choix et l'organisation des futures architectures multi-cœur et des solutions logicielles qui les exploiteront.

Abstract

Since 2005, chip manufacturers have stopped raising processor frequencies, which had been the primary mean to increase processing power since the end of the 90s. Other hardware techniques to improve sequential execution time (out-of-order processing, caches, branch prediction, etc.) have also shown diminishing returns, while raising the power envelope. For these reasons, commonly referred to as the frequency and power walls, manufacturers have turned to multiple processor cores to exploit the growing number of available transistors on a die.

In this thesis, we anticipate the probable evolution of multi-core processors into *many-core* ones, comprising hundreds of cores and more, by focusing on three important directions. First, we improve the CAPSULE programming environment, based on *conditional parallelization*, by adding robust coarse-grain synchronization primitives and by enhancing its portability. We study its performance and show its benefits over common parallelization approaches, both in terms of speedups and execution time stability.

Second, because of increased contention and the memory wall, many-core architectures are likely to become more distributed. We thus adapt CAPSULE to distributed-memory architectures by proposing a simple but general data structure model that allows the associated run-time system to automatically handle data location based on program accesses. New distributed and local schemes to implement conditional parallelization and work dispatching are also proposed.

Third, we develop a new discrete-event-based simulator, *SiMany*, able to sustain hundreds to thousands of cores with practical execution time. We show that it successfully captures the main program execution trends by comparing its results to those of a cycle-accurate simulator up to 64 cores. SiMany is more than a hundred times faster than the current best flexible approaches. It pushes further the limits of high-level architectural design-space exploration and software trend prediction, making it a key tool to design future many-core architectures and to assess software scalability on them.

Contents

Remerciements xv

Introduction 1

I Capsule: Parallel Programming Made Easier 7

1 Parallel Programming Is Hard 9

1.1 Work Splitting and Dispatching 10
1.2 Working on the Same Data at the Same Time 10
1.2.1 Atomic Operations and Mutual Exclusion Primitives 11
1.2.2 Transactional Memory 11
1.3 Task Dependencies 12
1.4 The Capsule Environment 13

2 The Capsule Programming Model 15

2.1 Tasks 16
2.2 Conditional Parallelization 17
2.3 Recursive Work Declaration 19
2.4 Coarse-Grain Task Synchronization 22
2.5 Other Primitives and Abstractions 27

3 Run-Time System Implementation 29

3.1 Task Abstraction and Scheduling 30
3.2 Conditional Parallelization 32
3.3 Synchronization Groups 35
3.4 Portability Abstractions 37

4 Performance Study 41
4.1 Performance Scalability and Stability . . . 41
4.1.1 Motivating Example . . . 43
4.1.2 Benchmarks and Experimental Framework . . . 46
4.1.3 Iterative Execution Sampling . . . 48
4.1.4 Experimental Results . . . 50
4.2 Performance Dependence on the Run-Time Platform . . . 54
4.2.1 Run-Time System Implementation . . . 54
4.2.2 Hardware Architecture . . . 56
4.2.3 Task Granularity and Other Program Characteristics . . . 58

5 Related Work 63
5.1 Data Parallel Environments . . . 63
5.1.1 Languages . . . 63
5.1.2 STAPL . . . 64
5.2 Asynchronous Function Calls . . . 65
5.2.1 Cool . . . 65
5.2.2 Cilk . . . 66
5.2.3 Thread Building Blocks . . . 68
5.3 Parallel Semantics Through Shared-Objects . . . 69
5.3.1 Orca . . . 69
5.3.2 Charm++ . . . 70

6 Conclusion and Future Work 73

II Distributed Architecture Support in Capsule 75

7 Towards Distributed Architectures 77

8 Distributed Data Structures 81
8.1 Data Structure Model . . . 84
8.1.1 Concepts . . . 85
8.1.2 Programming Interface . . . 87
8.2 Implementation Traits . . . 90
8.2.1 Object Management Policy . . . 91
8.2.2 Object References . . . 92
8.2.3 Cell Structuration . . . 96

8.2.4 Hardware Support . . . 98
8.3 Status and Future Directions . . . 100

9 Distributed Work Management 103
9.1 Probe Policy and Task Queues . . . 104
9.2 Design Considerations . . . 106
9.2.1 Classical Strategies . . . 106
9.2.2 Push or Pull? . . . 108
9.3 Load-Balancing Policy . . . 108
9.3.1 Migration Decision Algorithm . . . 109
9.3.2 From Local Decisions to Global Balance . . . 111
9.4 Migration Protocol and Interactions With Task Queues . . . 113
9.4.1 Concurrent Task Migrations . . . 113
9.4.2 Preserving Locality . . . 115

10 Related Work 117
10.1 SPMD and Task-Based Languages . . . 117
10.1.1 SPMD Languages . . . 117
10.1.2 Cilk . . . 119
10.2 Memory Consistency . . . 120
10.2.1 Sequential Consistency . . . 120
10.2.2 Strong Ordering . . . 122
10.2.3 Practical Sufficient Conditions for Sequential Consistency . . . 124
10.2.4 Weak Ordering . . . 125
10.2.5 Processor Consistency . . . 127
10.2.6 Slow and Causal Memories . . . 128
10.2.7 Release Consistency . . . 129
10.2.8 Entry Consistency . . . 129
10.2.9 Scope Consistency . . . 131
10.2.10 Location Consistency . . . 132
10.2.11 Total Store, Partial Store and Relaxed Memory Order . . . 134
10.2.12 Local Consistency . . . 134
10.3 Distributed-Shared Memory . . . 135
10.3.1 Ivy . . . 135
10.3.2 Munin . . . 139
10.3.3 TreadMarks . . . 142
10.3.4 Other DSMs with Relaxed Consistency . . . 144

10.3.5 Fine-Grain Coherency 145
10.4 Distributed Objects 148
10.4.1 Emerald 148
10.4.2 Amber 149
10.4.3 Orca 150
10.4.4 Charm++ 153
10.4.5 CRL 155

III SiMany: Simulating Many-Core Architectures 157

11 The Need for Many-Core Simulation 159

12 Virtual Timing 163
12.1 Principles 163
12.1.1 Timing Annotations 163
12.1.2 Distributed Timing 164
12.1.3 Distributed Spatial Synchronization 165
12.1.4 Non-Connected Sets of Active Cores 168
12.1.5 Time Drift of Dynamically Created Tasks 168
12.2 Ensuring Correct Simulation 170
12.2.1 Program Execution Correctness 170
12.2.2 Locks and Critical Sections 171
12.2.3 Deadlock Avoidance Proof 172

13 Simulator Implementation 175
13.1 Implementing an Efficient Simulation 175
13.1.1 Direct Execution of Computations 175
13.1.2 Software Architecture 175
13.1.3 Overhead of Network Communications and OS 177
13.1.4 Userland Threading and Scheduling 179
13.2 Modeling a Network of Cores 180
13.2.1 Simulated Architecture Overview 180
13.2.2 Network Interface Implementation 181
13.2.3 Bandwidth and Concurrency Limits 182
13.2.4 Control Messages 184
13.2.5 Network Interface and Core Interactions 185
13.2.6 Programming Model Support 186

14 Experimental Evaluation **189**

14.1 Framework and Methodology 189

14.1.1 Simulator Parameters 189

14.1.2 Benchmarks 191

14.2 Experimental Results and Hardware Exploration 192

14.2.1 Simulator Validation 192

14.2.2 Simulation Speed 194

14.2.3 Speedups on Regular 2D meshes 195

14.2.4 Simulation Time/Accuracy Trade-Off 196

14.2.5 Clustered Architectures 202

14.2.6 Polymorphic Architectures 202

15 Related Work **205**

15.1 General Discrete-Events Simulation 205

15.2 Single-Core Simulation 208

15.2.1 Monolithic Simulation 208

15.2.2 Modular Simulation 209

15.2.3 Speeding up the Simulation 210

15.3 Multi-Core and Many-Core Simulation 212

15.3.1 Sampling Techniques May Not Scale 212

15.3.2 Conservative Discrete-Events Based Simulators 214

15.3.3 Relaxed Synchronization 215

15.3.4 Other Approaches 216

16 Conclusion And Future Work **219**

General Conclusion **221**

Appendices **231**

A Quicksort Example Code **233**

B Dijkstra Example Code **243**

Selected Personal Bibliography 259

Bibliography 263

Index 291

Remerciements

C'est avec une grande satisfaction et une certaine émotion que j'écris ces lignes, quelques mois après le jour effectif de ma soutenance. Se consacrer quatre ans à l'exploration de sujets passionants et complexes fut une aventure extrêmement enrichissante sur les plans scientifique et technique. Une telle durée autorise en effet la poursuite de projets scientifiques relativement ambitieux, pour peu que l'on jouisse de la liberté suffisante, ce qui fut mon cas. Elle permet aussi d'explorer leurs applications pratiques dans un cadre assez général, même si cela ne fait (malheureusement ?) pas partie des objectifs premiers d'une thèse.

Avant de parler plus avant du déroulement de la thèse et de remercier les personnes qui y ont directement ou indirectement contribué, je souhaiterais expliquer ce qui m'a conduit à la faire et l'enchaînement des évènements préalables. J'ai ainsi l'espoir de servir l'intérêt d'éventuels lecteurs qui se demanderaient s'ils souhaitent suivre la même voie. D'autres, scientifiques ou ingénieurs plus expérimentés, pourront y trouver un témoignage que j'espère utile sur les facteurs qui peuvent pousser un étudiant à entreprendre un doctorat.

J'avais commencé à envisager un projet de thèse quelques années avant de le concrétiser, à peu près au moment de ma scolarité à l'ENST à Paris (2002-2003). Il était alors possible de suivre des cours de certains DEA d'informatique en même temps que ceux de l'école, avec comme débouché possible le démarrage d'une thèse. Poursuivant en cela la grande tradition française de l'enseignement supérieur scientifique, le contenu de ces DEA était fort théorique. Ils n'abordaient malheureusement pas, ou seulement de manière superficielle, des problématiques informatiques plus concrètes et pourtant très importantes comme la programmation parallèle, l'ordonnancement de tâches dans un cadre général où aucune information n'est disponible a priori sur leur durée d'exécution, les différentes techniques d'optimisation de code lors de la phase de compilation ou encore le fonctionnement des différents modules d'un système d'exploitation.

Or ces problématiques n'avaient justement cessé d'être l'objet de mon intérêt et de ma curiosité ; elles étaient celles que je souhaitais étudier. En outre, assister à des cours d'autres DEA, sans aménagement de la scolarité à l'ENST, paraissait difficile. C'était également se condamner à n'avoir jamais le temps d'approfondir quoi que ce soit. C'est ainsi que je renonçais provisoirement à ce projet. À la fin de ma scolarité, je démarrais ma carrière en tant qu'ingénieur de développement chez un grand éditeur de logiciel français.

Après deux ans passés à ce poste, qui m'apporta une solide expérience des bonnes

pratiques de production et de gestion de code à une échelle industrielle, il m'apparut plus clairement que l'intégration seule ne me suffisait pas et que l'approfondissement de problématiques informatiques fondamentales était important pour moi. Le temps passant, il deviendrait plus difficile de se consacrer à une thèse. Le moment semblait propice, d'autant qu'à la même époque, les fabricants de micro-processeurs s'étaient orientés vers un nouveau type de puce, les multi-cœurs, intégrant plusieurs processeurs destinés à travailler en parallèle. Cette évolution m'intéressant énormément, je réactivais mon projet de thèse initial.

Il me fallait à présent trouver une équipe qui pourrait m'accueillir et dont les grands axes de recherche seraient compatibles avec mes centres d'intérêt. Assez rapidement, j'en isolais deux. La première, à Rennes, CAPS, dirigée par André Seznec, se consacrait à l'étude de mécanismes matériels micro-architecturaux permettant d'améliorer la performance des processeurs classiques, c'est-à-dire séquentiels. Plus précisément, leur travaux portaient, entre autres, sur les mécanismes de remplacement des données au sein des mémoires caches, sur la prédiction de branchement et sur le SMT (Simultaneous MultiThreading).

La seconde équipe, ALCHEMY (Architecture, Languages and Compilers to Harness the End of Moore Years), basée à Saclay en région parisienne et dirigée par Olivier Temam, visait à inventer et explorer des techniques pouvant potentiellement prendre le relais des architectures séquentielles classiques, dont les progrès deviennent de plus en plus difficiles et de moins en moins rentables. Parmi les axes de recherche qu'elle suivait, on trouve par exemple les processeurs SMT, les multi-cœurs et les modèles de programmation associés, la simulation d'architectures et de systèmes, les réseaux de neurones, la mutualisation d'accélérateurs ou encore l'amélioration du modèle polyhédrique (optimisation de la compilation de boucles), en théorie et en pratique, notamment par l'intégration dans GCC (voir par exemple Graphite, intégré par défaut depuis la version 4.5).

Mon choix s'est assez rapidement porté sur la seconde équipe, dont les centres d'intérêt correspondaient mieux à mes attentes. En effet, les techniques d'optimisation des processeurs séquentiels m'étaient en grande partie déjà connues. Le parallélisme m'apparaissait comme un terrain d'études a priori plus obscur et donc potentiellement plus excitant et formateur. Le principe n'était pas nouveau (comme on pourra le constater à la lecture de l'introduction de cette thèse) mais l'environnement matériel, logiciel et commercial dans lequel il allait être mis en œuvre différait sensiblement de celui qu'avaient connu les pionniers de la discipline.

L'équipe CAPS mentionnée plus haut a aujourd'hui laissé la place à un nouveau projet centré sur les problématiques des multi-cœurs, ce qui semble a posteriori valider mon choix. Il est cependant trop tôt pour jauger correctement la pertinence à long terme de cette évolution, comme l'indiquent un nombre grandissant d'experts. La conclusion de cette thèse le rappelera : le futur des multi-cœurs est incertain et de nombreux problèmes restent à résoudre avant d'être véritablement assuré de leur pérennité.

Il restait enfin à trouver un financement pour ma thèse. Olivier Temam, directeur d'ALCHEMY, était alors en relation avec Bruno Jégo, de l'équipe AST (« Advanced System Technology ») chez ST Microelectronics, car ils collaboraient à un même projet de recherche européen. L'équipe AST s'occupait des aspects logiciels d'un projet d'ar-

chitecture multi-cœur embarquée nommé « xStream ». À ce titre, Bruno s'intéressait aux modèles de programmation susceptibles d'être utilisés pour ces architectures. Sur ses recommendations, ST s'est engagée dans le financement de ma thèse, au travers du dispositif CIFRE (convention entre deux partenaires privé et public, avec subvention de l'État).

La première année de thèse a commencé par la familiarisation avec l'environnement de programmation et d'exécution parallèle CAPSULE, développé initialement par Pierre Palatin et Yves Lhuillier sous la direction d'Olivier Temam. Les approches précédentes les plus connues, Cilk et Charm++, ont été rapidement abordées. Elles seront décrites de manière détaillée aux Chapitres 5 et 10. Les premiers travaux ont concerné l'étude de l'influence de l'utilisation de la division conditionnelle, le principe central de CAPSULE, sur la stabilité du temps d'exécution de programmes parallèles. J'ai également consacré quelques semaines à la parallélisation de MoGo au sein de l'environnement CAPSULE. MoGo est un des meilleurs programmes de jeu de go au monde et a été développé à l'INRIA par Sylvain Gelly et Olivier Teytaud (entre autres). Cette entreprise n'a malheureusement pas donné de résultat directement exploitable et n'a donc pas été poursuivie.

Enfin, c'est à cette époque qu'a commencé à prendre forme l'un des axes principaux de mes travaux, à savoir l'adaptation de CAPSULE à des architectures distribuées. Pierre Palatin n'avait disposé que de peu de temps à la fin de sa thèse pour commencer à réfléchir à des techniques générales de gestion des données faciles à mettre en œuvre et efficaces dans ce contexte. De mon côté, je souhaitais en sus aborder la gestion des tâches à exécuter par les différents cœurs, une problématique cruciale pour des architectures distribuées, puisque le coût de communication entre les différents cœurs y est largement plus important que sur des architectures à mémoire partagée ou à bus.

Dans un contexte d'accroissement rapide du nombre de cœurs par puce, cet axe semblait devoir prendre une importance croissante. Il n'avait jusque là que peu attiré l'attention des chercheurs se consacrant spécifiquement aux modèles de programmation pour multi-cœurs, peut-être parce qu'ils considéraient difficile voire illusoire de toucher au modèle de mémoire partagée si répandu dans l'industrie. Au contraire, ceux qui s'étaient intéressés à la programmation par flux (streaming), bien avant l'avènement des multi-cœurs, avaient naturellement été parmi les premiers à envisager une telle remise en question. Cependant, leur modèle de programmation, bien adapté pour des applications spécifiques comme le traitement d'image ou de son, s'avère relativement incommode dans un cadre général. Il ne semblait pouvoir constituer une réponse universellement acceptable.

L'équipe ALCHEMY ne disposait initialement que d'une seule machine à mémoire distribuée. De plus, elle ne présentait aux programmeurs qu'une interface de mémoire partagée classique. En outre, nous souhaitions anticiper l'accroissement du nombre de cœurs par puce, bien au delà de ce qui se faisait à l'époque (et de ce qui est d'ailleurs disponible même au moment où j'écris ces lignes). Les travaux sur les architectures distribuées devaient prendre toute leur mesure pour un nombre de cœurs potentiellement élevé. Enfin, conduire des expérimentations réalistes nécessitait de pouvoir faire varier certaines caractéristiques de l'architecture elle-même, ce qui était bien évidemment impossible avec des machines physiques. Pour toutes ces raisons, il m'est apparu essentiel, au commencement de la

seconde année de thèse, de disposer d'un simulateur de machines distribuées, afin de tester et valider les développements précédents.

Peu de simulateurs étaient capables de supporter des dizaines, voire des centaines de cœurs, et ils n'étaient pas tous librement disponibles. Aucun n'avait été conçu pour simuler à la fois des modèles d'exécution à base de tâches créées dynamiquement lors de l'exécution d'un programme et des architectures distribuées. J'ai donc dû concevoir et réaliser un tel simulateur dans son intégralité. Ce projet aura finalement dûré près d'un an et demi, en comptant la réalisation du système d'exécution lié à CAPSULE, c'est-à-dire les mécanismes de répartition de tâches et le support de structures de données distribuées, développements qu'il est difficile de dissocier complètement du simulateur proprement dit. Olivier Temam, mon directeur de thèse, au départ favorable à ce projet, y a ensuite manifesté une vive opposition à cause du temps qu'il me prenait, pensant qu'il n'était pas consacré directement à la recherche sur les structures de données distribuées. Aussi a-t-il fallu, à certains moments, faire preuve de persuasion et de ténacité.

Le simulateur et les résultats obtenus grâce à lui forment la Partie III de ce manuscrit. L'aboutissement de ce projet a également rendu possible un avancement significatif de la recherche sur les structures de données, par l'évaluation quantitative qu'il a permise sur des architectures variées. Ces travaux demandent cependant à être poursuivis, afin d'exploiter le potentiel du simulateur et des techniques proposées dans cette thèse, ainsi que d'élargir les horizons qu'ils ont permis d'entrevoir. Les derniers mois de recherche effective ont servi à parachever les résultats expérimentaux et à rédiger les articles.

La fin de la thèse a été classiquement consacrée à la rédaction de ce manuscrit. Exercice obligé du doctorat, elle était aussi l'occasion d'approfondir les différents points techniques abordés, de mentionner les pistes suivies, y compris celles qui se sont avérées infructueuses, et de faire apparaître le cheminement et la cohérence globale de cette aventure de quatre ans. J'ai essayé, lors de la rédaction, de mettre en exergue ces derniers, tout en satisfaisant autant que possible aux exigences de clarté et de précision habituellement attachées aux travaux scientifiques. Voilà qui explique la longueur de cette phase (un peu plus de neuf mois de travail), mais aussi celle du manuscrit lui-même, qui comprend en outre une abondante bibliographie.

Je souhaiterais à présent remercier chaleureusement et solennellement les différentes personnes qui ont permis ou accompagné cette aventure. J'espère seulement ne pas en oublier ! Si d'aventure cela était le cas, je leur prie de m'excuser et promets de mettre à jour ce texte en conséquence.

Pour commencer, je voudrais naturellement remercier Olivier Temam de m'avoir accueilli au sein du projet ALCHEMY. Je l'avais connu bien avant le démarrage de ma thèse, dès 2001, alors que je suivais son cours d'architecture des ordinateurs à l'École Polytechnique, qu'il donnait d'ailleurs pour la première fois. Je me souviens avoir été très favorablement impressionné par la qualité et la clarté de ses explications. Aussi ai-je eu l'agréable surprise de constater qu'il dirigeait ALCHEMY, au moment où je m'interrogeais sur mon équipe de destination. Olivier m'a fait confiance, en acceptant immédiatement ma candidature et en me mettant en contact avec Bruno Jégo. Il m'a laissé une grande liberté dans mes travaux, pratiquement comparable à celle d'un chercheur permanent.

Il m'a également transmis une partie de son expérience, me conseillant notamment sur la rédaction des articles ou sur les écueils à éviter pendant une thèse. Je dois saluer son énergie et sa grande motivation, qu'il parvient à focaliser avec une redoutable efficacité.

Chez ST Microelectronics, Bruno Jégo a suivi avec constance mes progrès, qu'il a su encourager y compris aux moments les plus difficiles. Il m'a également laissé une grande liberté sur le contenu de la thèse, tout en veillant à l'utilisation de programmes de test représentatifs d'applications utilisées par ST. Je remercie Jean-José Bérenguer, qui a accepté de suivre mes travaux d'un point de vue plus technique, et qui m'a fait le plaisir de consacrer beaucoup de temps à la relecture de ce manuscrit, alors même que son rôle d'examinateur ne l'y obligeait en rien. Le rôle de correspondant technique chez ST était initialement assuré par Thierry Strudel, qui a été rapidement appelé à d'autres fonctions.

Chez Alchemy à nouveau, je souhaite vivement remercier Albert Cohen de m'avoir proposé la charge d'une partie des travaux dirigés correspondant aux cours qu'il donnait à l'École Polytechnique (« Composants d'un système informatique » et « Principes et programmation des systèmes d'exploitation »). Cette collaboration, qu'il a renouvelée jusqu'à la fin de ma thèse, m'a permis d'acquérir une bonne expérience de l'enseignement, tâche que j'affectionne particulièrement. Avec nos compères, Fabrice Le Fessant, Patrick Carribault, Guillaume Quintin, Boubacar Diouf, Frédéric Brault, Erven Rohou, Louis-Noël Pouchet et Philippe Dumont, nous avons fait semblant de mener la vie dure à nos étudiants (en particulier Fabrice !), qui nous l'ont bien rendu par leur enthousiasme et leur appréciation de nos prestations. Je me souviendrai longtemps de ces inénarrables cafés, avant les TD ou entre deux groupes, passés à discuter pédagogie, OCaml, logiciel libres ou d'autres sujets n'ayant pas grand-chose à voir avec l'informatique et donnant parfois lieu aux conversations les plus délirantes. Je n'oublierai pas non plus les longues journées passées à remanier ou écrire le texte d'un TD et des programmes associés, puis à vérifier, de préférence au dernier moment, que les expériences seraient conformes à ce que nous attendions et pensions expliquer aux étudiants ; elles ne le furent pas toujours pour autant !

Pendant ces quatre ans, j'ai eu le plaisir de côtoyer les quelques thésards et post-doctorants que l'équipe a successivement accueillis. Je souhaite affectueusement remercier mes camarades de fortune (et d'infortune !), à commencer par Zheng Li (approximativement « Djong » pour la prononciation du prénom), le deuxième thésard qui a travaillé sur Capsule pendant la même période à quelques mois près, avec comme focalisation le support matériel. Comme il s'en souvient probablement, nous nous sommes connus avant même le démarrage officiel de nos thèses, à l'université d'été ACACES en 2006, où nous nous étions rendus par le même avion. Je dois louer chez Zheng sa grande ténacité dans le travail, son bon sens, son enthousiasme et sa bonne humeur, même aux moments où il jonglait avec la rédaction de son manuscrit, des développements prenants pour un projet européen et les joies de la paternité ! Sans parler, bien sûr, du petit aperçu de la culture chinoise qu'il nous a offert.

Zheng et moi avons côtoyé plusieurs mois l'un de nos prédécesseurs sur Capsule, Pierre Palatin, remarquable pour ses qualités de codeur et d'administrateur système, mais aussi pour ses fameux T-shirts « PhD Comics » approximativement repassés. Son

enthousiasme et sa serviabilité ont été appréciés de toute l'équipe. Avec son compère Sylvain Girbal, certainement une des personnes qui ait passé le plus de temps dans l'équipe à l'exception des chercheurs permanents qui l'ont fondée, ils sont les artisans du premier cluster d'ALCHEMY, composé de PC assemblés à la main, et fonctionnant grâce à un système de double démarrage et d'images qui permettait de mettre à jour de manière simplifiée tout le parc de machines, lui aussi artisanal. Ce système a ensuite été réutilisé sans nécessiter de gros aménagements pour le deuxième cluster, cette fois quasi-exclusivement composés de puissantes machines Dell, dont j'avais assuré la commande. Même si elles n'ont clairement pas contribué à ma ligne, soumise à rude épreuve durant la thèse (particulièrement à la fin), j'ai beaucoup apprécié les virées à trois au centre commercial des Ulis à discuter informatique ou à les écouter parler de jeux de rôle autour « d'un bon burger », comme Pierre avait coutume de dire.

Initialement installé au bout du couloir « architecture » d'ALCHEMY au rez-de-chaussée du bâtiment N à Saclay, j'ai partagé un bureau avec Luidnel Maignan et Mouad Bahi. L'ambiance y était studieuse, sans pour autant interdire d'intéressantes conversations, parfois éloignées de mes thèmes de travail. Elles avaient pour effet bénéfique de m'aérer l'esprit, quelquefois aussi celui de me pousser à m'intéresser à des domaines assez éloignés de mes sujets, parfois même de l'informatique en général, comme lors d'une journée passée à étudier les fondements de la théorie des ensembles et comment ceux-ci pouvait servir à démontrer la convergence d'une suite là où les axiomes de Péano étaient insuffisants. Ces escapades sont heureusement restées rares, sans quoi je n'aurais pu finir cette thèse ou, du moins, y aborder un spectre aussi large de thèmes.

Zheng, quant à lui, occupait un bureau adjacent, en compagnie de Taj Khan, qui a travaillé sur la simulation statistique par échantillons (« sampling »), domaine dont on pourra trouver un bref aperçu en section 15.2.3. J'ai pu apprécier la gentillesse de Taj, son calme et sa patience, très apaisants. Ont également occupé une place dans ce bureau Benjamin Dauvergne, finissant sa thèse alors que je commençais à peine la mienne, Benoît Siri, travaillant sur l'exploitation de réseaux de neurones pour le calcul, but qui nécessitait au préalable une meilleure compréhension de leur dynamique, et Fei Jiang, dont j'ai honteusement oublié le sujet précis des travaux.

Puis, durant la rédaction de la thèse, les derniers mois, j'ai déménagé mon bureau et rejoint Philippe Dumont, dans l'aile « compilation ». Cela facilitait nos interactions lors de la préparation de TDs, mais aussi l'échange d'expérience sur toutes les phases d'une thèse et sur la recherche en général. J'apprécie chez Philippe son ouverture et son intérêt pour de nombreux sujets, qu'ils soient culturels (la photo), scientifiques (son entrain à tester de nouveaux systèmes), ou culinaires (excellents caramels !). Nous avons passé tous deux un temps considérable à administrer le cluster d'ALCHEMY, à la fois en ce qui concerne le matériel (transport de machines, connexions réseaux, alimentations et autres réjouissances) et les logiciels systèmes (distributions Linux, boot par initrd customisés, DNS, DCHP, NTP, NFS, `rsync` et pas mal d'autres), avec l'aide de Philippe Lubrano, Frédérique Morin et Christian Poli, des « moyens informatiques ».

À ces tâches a également largement participé Louis-Noël Pouchet, dont je salue la réactivité ainsi que la capacité à faire marcher des systèmes assez instables, même s'il

faut en passer parfois par des solutions, disons… pas très propres ! Louis-Noël est une des rares personnes que je connaisse qui ait un rythme de travail aussi contrasté, capable de disparaître plusieurs jours pour résoudre un problème, au point de ne plus dormir, comme de prendre le temps d'agréables discussions avec ses collègues. Poursuivant des recherches sur l'optimisation de programmes par transformations automatiques de boucle (utilisant et étendant notamment le fameux modèle polyhédrique, sur laquelle l'expertise de la section « compilation » est mondialement reconnue), il a quitté le laboratoire après sa soutenance pour les États-Unis et l'Ohio State University. Philippe, quant à lui, continue ses travaux sur les méthodes de flux (« streaming ») et sur la synchronisation discrète en Allemagne. La France y a malheureusement perdu deux chercheurs prometteurs coup sur coup !

D'autres thésards et stagiaires travaillaient dans la section « compilation ». Même si j'eus moins d'occasion de les rencontrer, j'ai apprécié discuter avec Mounira Bachir, Boubacar Diouf, Konrad Trifunovic ou encore Frédéric Brault et François Galéa de sciences et de nombreux autres sujets. Le tableau, enfin, serait incomplet sans faire mention des autres membres permanents de l'équipe : Cédric Bastoul et son impressionnant musée de machines Sun, Hugues Berry et ses recherches à l'interface entre l'informatique et la biologie, Christine Eisenbeis et ses collaborations plutôt variées, y compris avec des physiciens (c'est dire !). Nos assistantes, Stéphanie Meunier puis Valérie Berthou, ainsi que Stéphanie Druetta, secrétaire de l'école doctorale d'informatique de l'Université de Paris-Sud, ont assuré avec discrétion et efficacité tous nos travaux administratifs et veillé à ce que nous n'oublions pas de suivre les différentes procédures et de fournir tous les documents nécessaires, parfois jusqu'à l'absurde (même renseignements d'état civil inlassablement demandés, papiers à en-tête de l'université à imprimer obligatoirement en couleur ou encore les 17 (!) exemplaires de résumé de thèse à fournir avant la soutenance).

Ces années de thèse, à l'exception de la première, prirent souvent l'allure d'un sacerdoce. Outre le bonheur et la chance d'en avoir été récompensé par la publication de trois articles dans des conférences de renommée internationale, je dois à tous les gens que je viens de citer, et ceux qui le seront plus bas, de les avoir rendues plus riches, supportables, en un mot plus vivantes.

Abordons maintenant la dernière partie de ces remerciements, qui concerne mes proches et toutes les personnes qui m'ont aidé, consciemment ou non, à devenir ce que je suis. La liste est certainement beaucoup trop longue pour être donnée de manière exhaustive. Néanmoins, je souhaite au moins exprimer à cette occasion ma gratitude pour tous les gens qui ont œuvré à mon éducation et à mon instruction. Par ailleurs, les lecteurs ayant poursuivi jusqu'à ces lignes souffriront bien quelques paragraphes de plus !

Je voudrais donc remercier, tout en les priant de m'excuser si je n'orthographie pas correctement leurs noms, faute d'arriver à m'en souvenir précisément après toutes ces années, tous mes professeurs, à qui cette thèse est dédiée, parmi lesquels, pêle-mêle : de l'école primaire Fabre, à Toulouse, Mme Descamps, Mme Lloubes, Mme Ducès, M. Escoubas, qui m'a initié à l'informatique à l'âge de 8 ans et sans qui je ne serais peut-être pas en train d'écrire ces lignes, M. Catala, dont je n'ai pas oublié l'humour ni les heures consacrées à l'initiation à la musique classique ; au collège Pierre de Fermat, Mme Thibaut, pour sa rigueur et la qualité de son enseignement, Mme Porcheron, M. Grialou, Mme Salis,

Mme Cabos, pour sa rigueur mais aussi pour m'excuser de l'avoir embêtée maintes fois afin d'obtenir des exercices supplémentaires, Mme Calveyrac, M. Lacassagne, qui a fortement contribué à mon niveau d'anglais, Mme Bacri, Mme Diez ; au lycée Pierre de Fermat, M. Pignères, pour son bon sens et son humanité, M. Dumoulin, pour sa joie d'enseigner la physique, ses célèbres sorties et ses expériences aux résultats souvent aléatoires, Mme Husson, Mme Vergé ; en classes préparatoires, toujours à Fermat, Mme Bonnier-Rigny, pour ses cours précis et sa générosité, M. Gonnord, M. Mercier, pour les efforts et le temps qu'il a consacrés à la présentation la plus épurée et la plus rigoureuse de concepts physiques clés, ainsi que pour ses connaissances encyclopédiques ; à l'École Polytechnique, MM. Finkielkraut, pour son bouillonnement et ses idées stimulantes, Rincé, Lebeau, pour son excellent cours sur la théorie des distributions, Salençon, Basdevant, Kopper, pour sa clarté et ses connaissances en mécanique quantique, Maranget, pour savoir introduire juste ce qu'il faut de théorie et pour ses talents de programmeur.

Je souhaite remercier tous mes amis, qui m'ont soutenu directement et indirectement, alors même que j'ai été, plus encore que de coutume, tour à tour occupé, indisponible, injoignable ou absent. François, Didier, Charles, Mathilde, Jean-François, Nadège, Éric, Éric, Pauline, Philippe, Aurélia, Samy, Anne-Claire, ainsi que, de manière plus épisodique, Pierre, Guillaume, Patrice, Jean-Christophe, Jihane, Sophie, Hervé, Thomas, Amélie, Stéphane, Fred, Aline, Mélissa, Kate, vous avez contribué à votre manière à cette thèse, tout simplement en redonnant le sourire à son auteur aux moments où il en avait besoin et en l'aidant à préserver sa santé mentale. Quand je leur ai annoncé mon projet de thèse en programmation parallèle, deux personnes, dont je tairai le nom par charité, se sont imaginées avec malice que j'allais occuper tout mon temps à cheminer le long de droites parallèles dans l'espoir de découvrir enfin un endroit où elles se toucheraient... Je ne saurais dire si la conversation qui a suivi leur a fait entrevoir malgré tout la portée et l'utilité de mon projet !

Je n'aurais pas traversé ces années avec le même enthousiasme sans le soutien des incommensurables génies que sont Rachmaninoff, Chopin, Ravel, Debussy, Beethoven ou Mozart (et quelques autres dont j'épargne, par charité à nouveau, la liste au lecteur, qui pourra néanmoins se la voir fournie sur demande), ni sans l'acuité et la générosité de Vincent, mon professeur de piano, ainsi que la compagnie des autres membres de l'association APE à Suresnes.

Enfin, je souhaite remercier les membres de ma famille et proches, qui m'ont soutenu et encouragé dans mes passions, même s'ils ne les comprirent pas toujours. Je pense, parmi d'innombrables épisodes, à mon père me répétant inlassablement que « science sans conscience n'est que ruine de l'âme », à mes parents qui m'ont offert mon premier ordinateur (j'avais à peine 8 ans), à ceux qui m'ont offert des encyclopédies et des livres, de science ou de littérature, à ceux qui m'ont offert de merveilleux disques (Annick, Grand-mère), à ceux qui m'ont fait voyager et découvrir le monde. Merci également à mon frère Nicolas et ma sœur Victoire pour tous les moments passés ensemble, ainsi qu'à Roxane et Mister Simon. Plus particulièrement, je dédie cette thèse à mes grands-parents, à Robert, qui m'a fait partager sa passion pour la musique classique et qui m'a transmis énergie, curiosité et humour, à Grand-mère pour son affection, ses histoires et sa générosité,

à Malou pour sa redoutable volonté et ses voyages aux quatre coins de la planète, et enfin à Joseph, que j'aurais beaucoup aimé connaître. Je salue également Denis pour son insatiable curiosité et les nombreuses discussions que nous avons eues, ainsi que Jeanine pour les débats politiques et la cuisine !

Merci enfin à Florence, ma compagne pendant ces années, qui a supporté ma peine, mes fréquentes absences, mes sautes d'humeur, mon manque chronique de sommeil, le stress des soumissions d'article à la dernière seconde après des nuits blanches de travail, la colère et parfois l'indignation face aux appréciations reçues, mais aussi les joies, la fierté du travail bien fait, l'accomplissement de quelques vieux rêves, les présentations publiques et les voyages à l'autre bout du monde. Il ne fait aucun doute que, sans son soutien, ce travail n'aurait pû prétendre au degré de qualité et à la largeur qui sont les siens.

À vous tous, avec toute ma reconnaissance et ma profonde gratitude.

— Olivier

Introduction

The computer industry has been driven for more than 30 years by a law originally formulated by Moore in his famous article of 1965 [187]. This law said that the number of components per integrated circuit would double every year. Moore revised the law in 1975 [188], indicating that the doubling would rather happen every other year. Since then, this law has reflected industry's progress, and derivatives of it for a variety of metrics, such as the size of memories, the storage capacity of hard disks, the network capacity or the cost per transistor, also have held true.

During the same period, the computing power of processors has increased exponentially. This feat has been possible thanks to several fundamental technology progresses. The most influential of them has been miniaturization, by which ever smaller transistors could be engraved onto chips. Besides contributing to the raise of transistor density, this progress, along with gating and better high-level circuit design, enabled processor frequency to rise exponentially until a few years ago. Frequency increases, when possible, mechanically augment the computing power of a processor *without changing the programming abstraction*, which is inherently sequential and rooted in the work by Turing [251] on the machines of the same name. This explains why frequency increase has been one of the major reasons for performance progress, particularly between 1999 and 2005. Other microarchitectural innovations that do not change the programming model have supported the performance growth as well: Out-of-order processing, bigger caches, wider pipelines that can execute more instructions per clock cycle, improved branch prediction and other techniques to improve pipeline efficiency.

Frequency increase has however almost completely stalled since 2005 [215]. Frequency is constrained by the maximum size of pipeline stages. Stages can be reduced gradually through design improvements, such as performing the same operation with fewer logic gates, but stage shrinking has been realized mostly by splitting work into more stages. This technique however brings diminishing returns. The pipeline must be flushed on a branch misprediction, which diminishes throughput. All stages must keep state information, which increases chip complexity. Finally, transistor switching time is no longer the factor that prevents the information transmission latency to decrease within a chip. Rather, the wire delay has become preponderant [180] and obliges architects to revise older designs by putting gates frequently in interaction closer to each other when possible, or introducing more buffers between gates by default. The maximum distance between two communicating gates without buffering is roughly inversely proportional to the frequency. For all these reasons, frequency increase has hit a wall, called the *frequency wall* [93].

Power consumption of a silicon processor is the main other factor that has put a stop to the rise of frequency. It increases linearly with it and quadratically with voltage [11]. Lowering the voltage thus could, in principle, largely compensate for greater frequencies. The power-delay product also decreases with new generations of transistors. Unfortunately, a lower voltage V implies a higher transistor delay T, according to the relation: $T \propto \frac{1}{V}$, and this delay limits the attainable frequency. In the end, power increases as the square of V and, equivalently, as the square of f, the frequency. The net result is that increasing the processing power through frequency is exceedingly costly and generates a considerable amount of heat through static leakage and dynamic switching. The difficulty of dissipating

the heat is exacerbated by the higher transistor density. This problem has been known under the denomination of *power wall* [93].

In the meantime, transistors have been getting cheaper, so manufacturers could use more of them to deliver the performance growth once provided by frequency. Most technologies to improve single processor performance have reached relative maturity and are unable to sustain alone the expected performance increase [241]. Consequently, manufacturers have settled to use the additional transistors to form more processing units, which lead to the development of multi-core processors.

It is not the first time in history that computers integrate several processing units. Supercomputers with multiple processors have existed since the 60s. Entry-level servers and powerful workstations of the 90s commonly included 2 or 4 processors. Still, founders would succeed in improving performance of single processors so that they would finally outperform several ones from the previous generation combined. Having multiple processors used to be the mean to be ahead of single processor performance by a few years. By contrast, the recent turn to many-core is a forced move and it is affecting the whole range of computer devices, from embedded systems and low-entry PCs to supercomputers.

In the previous decades, only expert programmers had been exposed to parallelism and parallel programming, which were used essentially in big but mostly niche scientific applications. In the coming years, it is likely that most programs, including irregular ones, will have to be modified or even mostly rewritten to take advantage of multi-cores, and that many more programmers will have to be introduced to parallel programming. There has been recent progress to automate parallelization of legacy sequential code [44], which may provide a speed relief for the short term and for processors with several cores. However, roadmaps of major founders mention tens to hundreds of cores per chip [134, 250] for the coming years. The main graphic chip manufacturers are already shipping dies with general-purpose computing capabilities comprising tens of cores, each made up of tens of SIMD execution units [13, 196]. The current automatic techniques cannot cope with such a high number of processing units, except for very specific application types, such as those that use linear algebra algorithms [260].

Therefore, our first goal for this thesis is to propose a generic programming approach that is simple yet effective in order to facilitate the transition to multi-core and many-core systems and bring parallel programming to a wider audience. Since lots of the design trade-offs of these systems have still unexplored consequences, both at the hardware and software levels, there is also a pressing need for tools to experiment with and explore a large combined design space. For this reason, our second goal is to investigate new simulation techniques to be able to study in advance upcoming architectures and program behavior on them.

For more than 20 years, there has been a great amount of research in programming parallel and distributed systems. We review some of the important approaches and directions in Chapters 5 and 10. A programming model, called CAPSULE, has been developed at the INRIA within the Alchemy project team by Pierre Palatin, Yves Lhuillier and Olivier Temam. It is a task-based environment that enables to express parallelism in a simple, concise and intuitive way. The role of the programmer is to declare *potential*

parallelism and coarse-grain synchronization constraints. The associated run-time system dynamically decides which amount of parallelism is profitable, a feature we call *conditional parallelization*, and schedules work to the available cores. Initially, the team thought that hardware support was critical in providing scalable performance, so they developed a hardware run-time system consisting of a slightly modified SMT processor [202]. Pierre Palatin then started to implement a software prototype to support his idea that an efficient software implementation was possible as well [201].

The contribution of this thesis is threefold. First, we improve the CAPSULE programming model and the implementation of the software run-time system. We show how, in addition to allowing an efficient parallel execution of programs written with the CAPSULE constructs, it reduces the high performance variability exhibited by irregular applications on multi-core processors. This last work was a joint effort with Pierre Palatin and Zheng Li. The influence of the run-time platform on software scalability is finally studied. All this work forms and is described in Part I.

Second, we extend the whole environment to be able to support distributed-memory, which we foresee as the future organization for multi-core and many-core architectures. We propose a set of new primitives that enable a programmer to manipulate data structures in a generic and platform-independent way. The new constructs also automatically enforce proper fine-grain data access synchronization. We improve the run-time system by adding transparent support for dynamic data movement. We also propose a new class of local and distributed task load-balancing mechanisms which are more scalable than the original central scheme. The inception of this last work was done as a collaboration with Zheng Li. We show that the combination of both approaches can yield linear speedup for some of our irregular applications on general-purpose multi-core processors with tens to hundreds of cores, which are to appear soon. All the work related to the adaptation of CAPSULE to distributed architectures is presented in Part II.

Third, we develop a new discrete-event simulator, called SiMany, able to support the simulation of thousands of cores on commodity computers with reasonable simulation time. It is based on the intuition that, past a certain architecture size, properly modeling the interactions between hardware components is more important than modeling accurately the components themselves. The user, or a relatively simple processor model, assigns timings to pieces of code, which are executed mostly natively. SiMany keeps track of the clock advances of each core and models the interconnection network. It ensures that all the cores make progress at a similar pace through a novel distributed synchronization technique, called *spatial synchronization*. Currently, SiMany is the fastest many-core simulator that can model a large range of architecture organizations, such as shared-memory and distributed-memory architecture, and different network topologies. This simulation work is detailed in Part III.

Part I

Capsule: Parallel Programming Made Easier

Chapter 1

Parallel Programming Is Hard

The advent of multi-core and many-core processors is essentially shifting the burden of improving program execution speed from hardware manufacturers to software developers. The latter group now has to uncover parallelism to exploit the capabilities of the new chips, either directly by programming in parallel with appropriate languages and environments, or indirectly by using or developing tools that automatically parallelize current sequential code. As explained in the Introduction, we focus on the first approach in this thesis, and specifically in this Part.

Writing parallel code is a notoriously difficult task, whether done from a project's start or introduced to optimize sequential program performance as an afterthought. Its complexity comes from the multiple processing units operating concurrently and asynchronously to achieve a coherent goal. As an analogy, let us consider a factory that employs a single worker to realize a technical task. The increase of single processor performance corresponds to the worker regularly improving its performance. There are of course physical and intellectual limits to a worker's efficiency, and the company will eventually have no other choices than hiring more workers to achieve a higher job throughput.

This new situation also creates new problems. It must be clear for all workers what their role is precisely and which parts of the work they have to perform. Unless their tasks are essentially independent, the best possible case but also the rarest, the workers will have to spend time talking or partially monitoring what the others are doing to coordinate. If the workers are manufacturing goods, some of them will depend on the work of others and will have to wait for them to complete their stage. In other organizations, workers may not be affected to a particular task but rather may assist different teams over time, depending on the levels of market demand.

Programming multiple processors or multi-cores introduces the same problems that a factory manager may face, but transposed to a technical level. There have been some studies on the overhead of parallel over sequential programming [127], reporting that it takes 10 to 15 times more time to code parallel versions of small and simple programs compared to sequential ones. In the software gaming industry, a designer of the 3D engine of Unreal 3, Tim Sweeney, estimated that multi-threading the renderer's scene traversal loop, content streaming, physics, animation and sound updates doubled to tripled the

software development and testing costs at Epic games (quoted in [74]).

In the following Sections, we briefly expose the various problems posed by parallelism. The last Section (1.4) is a general presentation of the CAPSULE environment. It gives a brief overview of the work detailed in this Part.

1.1 Work Splitting and Dispatching

A program's work must be split and dispatched to the available processing units.

Work splitting is a difficult task in itself. It requires identifying which program parts are inherently sequential and which are independent or may be transformed to be so. It may be done at compile time, for regular codes, or at run time, for irregular ones [159]. The links of the chain that carve up work into units need a mean to transmit the information to the run-time system.

Work distribution includes mapping a task to a processing unit, i.e., deciding where a task will be dispatched to, and then scheduling its execution, i.e., determining when it will be processed by the unit. It must balance the expected gains from taking advantage of multiple units and the overhead of communication and synchronization it may introduce. The architecture's units may offer different functional possibilities. General-purpose cores can accept any work but may not be optimized for the particular task assigned to them, while specialized units (DSPs, FPGAs or other accelerators) accept only specific codes but run them more efficiently. The processing units may also differ in computing throughput or other technical characteristics affecting performance, such as varying access latency to memory banks.

1.2 Working on the Same Data at the Same Time

In shared-memory architectures, execution units share the memory and may inadvertently step on each other's toes. At a low level, there must be mechanisms to arbitrate access to a given memory address by the different processors/cores[1]. At a slightly higher level, there must be mechanisms allowing to maintain coherency of data structures. Several processors writing to the same data structure (the same memory area) may do so only in restricted ways, so that another processor reading concurrently the information it contains is able to interpret them meaningfully. In particular, it usually shouldn't see "intermediary" states, where only part of the structure has been updated. To achieve this concretely, a processor has to monitor the actions of other processors on the same structure. The structure's expected semantics dictate which operations are allowed to overlap and which are not. Programmers and/or compilers must then have some mean to specify the restrictions to enforce. We will now briefly describe the current possibilities.

[1] All modern processors automatically implement such mechanisms.

1.2.1 Atomic Operations and Mutual Exclusion Primitives

The use of hardware-assisted atomic operations makes one or multiple operations at a given address appear to be executed instantaneously at some point in time. It is an efficient yet simple technique to achieve atomicity. However, it is difficult to compose such operations, because of other accesses potentially occuring concurrently between consecutive modifications.

For this reason, more general, and now standard, mutual exclusion primitives, such as locks, are used to protect a set of accesses. They are most often implemented thanks to the above-mentioned hardware-assisted instructions[2]. While a processor executes instructions within a critical section (e.g., when it holds a lock), other processors cannot enter the same critical section (e.g., acquire the same lock). The data to be protected is conceptually associated to a particular lock, by ensuring that the code accessing them can only be executed as long as the corresponding lock is held. The previous rule on locks then obviously guarantees that no more than one processor can be accessing the data.

Using locks may cause several problems, such as priority inversion and deadlocks. Among these, deadlock is the most serious. The simplest deadlock situation occurs when a thread/process holding a lock never releases it and another one wants to acquire it. Such a situation highlights the fundamentally *cooperative* nature of locks. No processors can require any exclusive access directly, but rather have to wait for the lock to become available. If the lock is never released, the waiting processor is blocked undefinitely. This first deadlock example is usually caused by a simple programming error: The programmer has forgotten to release a lock immediately after acquiring it and using the associated data. Nonetheless, for concurrent accesses with complex control flow, localizing the place where to insert the release may be a tough task. Deadlocks can also occur in the more complex case where a thread/processor must acquire simultaneously multiple locks to perform some operation. If another thread/processor needs to acquire the same set of locks, it must do so in the same lock order so as to avoid deadlock. When the order of lock acquisitions is run-time-dependent, avoiding deadlock may be hard and costly performance-wise. This is typically the case when composing several independent software components, such as libraries.

1.2.2 Transactional Memory

Transactional memory [124] is a generalization of classical hardware atomic operations that has become popular in the research community recently. It permits a processor to read and modify arbitrary memory locations atomically with respect to other transactions, i.e., transactions never appear to overlap. One of its most important benefits is to solve the deadlock problem. Programmers indicate which sections should be performed atomically,

[2] There exists implementations relying only on one-way atomic memory operations, such as read or write (see Dijkstra [76]), and even an implementation that doesn't require any atomicity (see Lamport [163]). They are extremely inefficient compared to the dedicated atomic instructions provided by all modern processors, and today only present a theoretical interest.

but do not specify the implementation to achieve this. Instead, they rely on the run-time system to do so.

A transaction is delimited by special primitives. Read and write operations belonging to it are also different from the regular ones. Write special operations do not modify memory as seen by other processors; their values are temporarily stored elsewhere, at least conceptually[3]. When a transaction ends, if the accessed locations were not modified by other processors, it *commits*: Values written are effectively made available to other processors atomically. In the other case, the transaction *aborts*, and all values written while it was in progress are discarded. The program then generally retries the transaction at a later time.

Several hardware and software implementations of transactional memory have been proposed over time [72, 78, 189, 232]. A survey can be found in Harris et al. [119]. A position paper [52] explains that transactional memory has not reached mainstream programmers yet because hardware vendors are reluctant to modify their design to support it, while software implementations are not scalable for large and complex applications and do not always enforce the same semantics[4], causing portability concerns. Moreover, if transactional memory solves the deadlock problem, it doesn't eliminate possible livelocks and introduces problems on its own[5]. Lots of researchers are currently working on these problems and satisfactory solutions are likely to be found in the coming years.

1.3 Task Dependencies

Some pieces of work handled by the different units are dependent on each other, as mandated by the program or algorithm employed. Again, this information must be passed by the programmer or inferred by the compiler through program analysis.

In sequential programming, dependencies are partly implicit at the language level. Instructions that use the same variables or objects are dependent[6] and the order in which they appear in the instruction flow determines the dependency direction. In parallel programming, two pieces of code are not necessarily ordered. They may execute concurrently, or in any order, and access the same data. Dependency specification can be achieved by explicitly imposing an execution order or by detailing which apparent interleavings are acceptable to produce correct results.

[3]Some schemes store the modifications directly into the caches or memory [189]. They nonetheless have to backup the original values in order to be able to restore them if the transaction later fails.

[4]One cause is that they do not try to work around the weak memory model most often presented by the hardware in order to preserve performance. For a survey of memory models, see Section 10.2.

[5]As an example, exception handling may be problematic (see [42]).

[6]Some dependencies are called *false dependencies* if they are not conceptual but rather implementation ones. As an example, a write after a read doesn't conceptually depend on the read value. But the read still must be performed before the write to return the appropriate value according to program order. False dependencies can be worked around through several techniques (static single assignment, register renaming, spilling, etc.).

1.4 The CAPSULE Environment

This rest of this Part presents CAPSULE, an environment to ease parallel programming and to spread it to a greater number of software developers. It specifies an abstract programming model that can supplement usual sequential languages and coexist with or replace current parallel constructs.

The name CAPSULE stems from the concept of *encapsulation*, which is a continuous inspiration for the framework. Encapsulation is a popular software trend that manifests itself in a number of software technological evolutions, such as the introduction of the object paradigm and object-oriented languages, the notion of interface or the appearance of component-based programing frameworks. In full generality, it consists in separating a program's actions and/or data into identifiable groups whose possible forms of interactions are limited in number and completely characterizable.

Encapsulating parts of programs has a considerable advantage in software engineering: Programmers can focus on the parts they are responsible for and code them as they please, independently of the parts they use or that rely on theirs. For large organizations, productivity generally increases and debugging time decreases. Our intuition is that it is profitable for parallel programming as well. Because component interactions are explicit, the occurence of potential conflicts can be forecasted or detected easily. Moreover, complete independence or limited interaction enables program parts, objects or components to be executed in parallel efficiently.

Although some future CAPSULE environment versions may be able to work directly at the level of tightly encapsulated components for appropriate languages, the current versions are mainly concerned with providing an intuitive interface for the programmers to declare work directly and with using this information to perform most of the work splitting job and task scheduling. These versions also handle coarse-grain task dependencies through hierarchical synchronization groups.

The original CAPSULE was a much simpler prototype hardware version [202] based on a SMT processor that would allow work declaration through a special assembly instruction (`nthr`). It was initially believed that hardware support was key to reach high levels of performance. Later, a software version was developed showing that, with appropriate speculative techniques, good performance could be reached also on unmodified multi-core architectures [201]. This software version extended the programming model to include a construct to handle coarse-grain synchronization of tasks in an intuitive way.

The current versions are an extension of these concepts, with an important overhaul of the coarse-grain synchronization primitives and their semantics to ensure their viability and usefulness when code is encapsulated and reused, and other additions to improve program portability. All the concepts and their associated constructs are presented in Chapter 2, along with an implementation based on the C language. Chapter 3 then describes a supporting software-only run-time system that assumes an architecture with global address space, i.e., common shared-memory architectures as well as distributed-shared memory

interfaces[7], which can hide hardware memory distribution. A second implementation, which adapts the current version to distributed-memory architectures with an explicitly distributed interface, is the subject of Part II. It adds to the core concepts presented in Chapter 2 some primitives to manage data structures independently of the precise memory distribution, while retaining the programming simplicity of global addressing.

The version that assumes global addressing does not offer any new mechanisms to regulate concurrent accesses to data. Programs are simply provided with traditional locking primitives. The current choice of included features, however, does not preclude future research on (or integration of) automatic work splitting and automatic parallelization techniques. The support for fine-grain synchronization is also mostly independent of the other interfaces of the environment. As an example, the current locking primitives could easily be replaced by a transactional memory interface and its associated implementation. We indeed propose an automatic locking scheme based on data structures in Section 8.1.1, as part of the distributed-memory version.

Another distinguishing feature of the multi-core era is to regularly ship processors with more cores. Unfortunately, with traditional low-level parallel programming environments, such as the POSIX threads [129] or MPI [94], changing the number of processing units or the network characteristics is not transparent to programs in terms of performance. Therefore, another very important goal is that CAPSULE's interface and run-time system realize *platform-independent high performance*. With this property, the time spent into program parallelization will be a long-term investment since programs will adapt to new architectures without modifications, as they did when single execution units were improved.

Chapter 2 details the base CAPSULE programming model and how we achieve intuitive programming and mostly platform-independent work splitting. Chapter 3 details the implementation of a software-only run-time system for regular shared-memory architectures. Chapter 4 studies the performance scalability and variability of programs on multi-core processors. Scalability of both regular and irregular programs is close to ideal on up to 4 cores. Performance variability, which is greater than on single-cores, can be reduced considerably by the CAPSULE approach. This property enables better execution time predictions and allows multi-core embedded systems to obey soft real-time constraints more easily. Finally, the influence of the run-time platform on performance is studied on an example benchmark. The performance of alternative run-time system implementations are compared. The effect of task granularity on obtainable speedups can be important. Several mechanisms are proposed to mitigate it. Chapter 5 describes some previous approaches and environments. Finally, Chapter 6 concludes this Part.

[7] Please consult Part II for more information on distributed-shared memory environments. Some prominent software implementations are surveyed in Section 10.3.

Chapter 2

The CAPSULE Programming Model

The CAPSULE programming model aims at overcoming all the parallelization difficulties summarized in Chapter 1 through a simple and intuitive programming framework. Besides the foremost goal of obtaining better performance through parallelism, it is designed with an emphasis on simplicity, to foster a wide adoption thanks to simple concepts that even non-expert programmers can grasp, and portability, to preserve the investment in software parallelization from the rapid and unceasing changes in machine architecture. Finally, it is designed to be general enough so that any kind of applications can be coded with it, including irregular ones whose ability to exploit multi-cores will be key for the mass adoption and commercial success of such chips.

Lots of languages especially designed for parallel programming have been proposed in the past 20 years[1], but it is fair to say that not a single one has gained wide acceptance to date. Their major ideas have nonetheless survived into newer languages or have been partly integrated into existing popular ones. Besides possibly inadequate constructs or abstractions, we think they did not spread probably because of the very fact that they are quite different languages than the mainstream sequential ones, and that they often do not include the expressive constructs or paradigms that the latter do.

Learning a new language and becoming highly productive with it may take months to years, a time most engineers are usually not able to dedicate to a task whose benefit is uncertain. Introducing a new language is an even greater effort encompassing a proper language specification but also a complete toolchain, including a compiler and a run-time system in the form of shared libraries or reusable components providing basic services (I/O, mostly). Consequently, we instead chose to use a widely-used language, C, and augmented it with special constructs. These new primitives do not introduce a new syntax, but rather leverage the usual C syntax for function calls. Programs coded with the C variant of the CAPSULE programming model are compiled, linked and run with C's common tools and run-time environment. This approach also has the benefit that new concepts and constructs can be implemented and tested more quickly.

The next Sections describe the programming model's concepts and illustrate them by introducing the corresponding primitives and giving concrete code as usage examples. The

[1] We review a few of them in Chapter 5.

reader should bear in mind, however, that the concepts are not tied to the current syntax, which is presented here for pedagogical purposes. Other implementations of the same concepts, in other existing languages such as Java, or in other new parallel languages, would be equally possible.

2.1 Tasks

The traditional paradigm is to present programmers with hardware execution units or the slightly higher abstraction of threads. This low-level approach has however several drawbacks.

First, programmers must undertake all the splitting, dispatching, and synchronization steps themselves. This task is already hard and time-consuming for small to moderate programs or algorithms. It becomes dauting for big ones to the point that it is often not viable economically [74].

Second, depending on the programming model, the resulting version may not be portable to other architectures because of the use of specific constructs or hardware functionality. Even if the resulting program is portable, the performance improvements it yields on the architecture it was developed and tested on may not be preserved when it is run on another one, because of a varying number of processors/cores, different communication bandwidth and latency and other differing hardware and software characteristics.

A common solution to adapt to a greater number of cores is to create a large number of threads. However, creating lots of threads amounts to split work in tiny pieces, which is inefficient if the actual number of cores is much lower. The reasons are the scheduling overhead, which increases with the number of threads, and the fact that dispatching a little piece of work to a dedicated core may cost alone more than just executing it sequentially. In the latter case, even if the dispatching cost is lower, it introduces more work in the overall execution and uses more execution units that may be better spent on larger units of work.

Third, irregular programs with complex control flow and data structures, for which the total amount of work and the dependencies between parts vary with the input data, cannot be parallelized efficiently at programming or compile time. They benefit tremendously from a dynamic approach that dispatches new work as it appears in unpredictable ways during the execution[2]. Programmers may want to create new threads dynamically to handle the just-uncovered load. Unfortunately, thread creation is very expensive on common platforms and OSes. Its typical cost is on the order of magnitude of 10^5 cycles [33]. Such an approach will not allow to exploit fine-grain parallelism, which is key to obtain scalable speedups for irregular applications [160]. Although seasoned programmers can devise a scheme where existing threads are recycled for new work instead of being constantly destroyed and recreated[3], it is a time-consuming and tedious job that would have to be

[2]Examples of such programs are presented in Section 4.1.

[3]This is indeed part of what is done within our run-time system, whose implementation is described in Chapter 3.

```
for (i = 0; i < n; ++ i)
  C[i] = A[i] + B[i];
```

Figure 2.1: Addition of 2 vectors in pseudo-C sequential code.

done again for each new application.

One of the main intuitions behind Capsule's model is that it is conceptually easier for programmers to reason in terms of work to distribute, as exemplified by the analogy we developed in Chapter 1. We thus settled on an interface with which a programmer can delimit the regions of code that form a coherent unit. These work units, with proper synchronization, could be individually handed off to a processing unit as a whole.

Concretely, let us illustrate this concept through the simple example of the addition of 2 vectors `A` and `B` of size `n`. The corresponding pseudo-C sequential code is shown in Figure 2.1. The body of the `for` loop at each iteration triggers an independent computation. Thus, it makes sense to indicate that the body is a coherent unit that can be executed on another core/processor for each iteration. The provided primitives to accomplish this change are detailed in the next section.

2.2 Conditional Parallelization

The central concept of Capsule is that of *conditional* or *potential* parallelization. The programmer indicates which portions of a program *may* be executed in parallel. The run-time system decides if a particular portion will *effectively* be executed by another core/processor.

This is an important paradigm shift from the preceding parallel programming environments. The role of the programmer (or the compiler) is to point out all the parallelization opportunities. However, the actual work splitting actions are decoupled from these annotations, contrary to what would happen with processes or threads. Instead, the run-time system, based on these indications, decides how to perform work splitting. It eventually dispatches and schedules the tasks that were effectively created.

The interface to indicate work units comprises two primitives. The first primitive, named the *probe*, is a call to the run-time system to declare that a task may be created at the current code point. On a call, the run-time system uses some algorithm to decide whether it will allow the actual task creation and return its decision to the program. If the answer is positive, the program, if appropriate, prepares the data needed to run the task. It then calls a second primitive, called *divide*. This primitive indicates to the run-time system that the data are ready and that it can dispatch the new task.

The corresponding system functions in our C implementation are called `capsule_probe` and `capsule_divide`. Figure 2.2 (next page) shows the vector addition example with the support code for Capsule. The loop body has been encapsulated in a separate C function. At line 10, the `capsule_probe` function is called with a pointer to the `loop_body` function and returns a pointer to an execution context (`ctxt`). At line 13, the returned pointer

```
    loop_body (A, B, C, i)
    {
      C[i] = A[i] + B[i];
    }
 5
    for (i = 0; i < n; ++ i)
    {
      capsule_ctxt_t * ctxt;
      // Probe
10    capsule_probe (& loop_body, & ctxt);

      // Probe result?
      if (ctxt != NULL)
        // The probe succeeded.  We divide.
15      capsule_divide (ctxt, A, B, C, i);
      else
        // The probe failed.  Execute the body sequentially.
        loop_body (A, B, C, i);
    }
```

Figure 2.2: Addition of 2 vectors in CAPSULE pseudo-C code, naive variant.

is tested. The run-time system returns a non-null execution context pointer if and only if the probe succeeded. In this case, the program calls `capsule_divide` (line 15), which effectively submits a new task to the system. Otherwise, the program has to execute the corresponding piece of work sequentially (line 18).

This example's listing illustrates concretely some important points. First, although not used in this example, it is possible to execute arbitrary code between the calls to `capsule_probe` and `capsule_divide`. Obviously, this code should be composed preferentially of a few instructions that execute relatively quickly, because the call to `capsule_probe` reserves for the new task some resources which stay unused until `capsule_divide` is actually called. For the version of CAPSULE we present here, the resource concretely is one of the architecture's cores[4]. The advantage of such a separation is to save the operations to prepare the data that are to be processed by the new task if its creation is refused.

Second, a probe failure indicates that no tasks can be created at the moment of the call. Still, the probe was called with the intent to execute a particular piece of work, and this work remains to be done. In this case, the program simply executes the work *sequentially*. In the example, this is done by calling the same function `loop_body` (line 18). However, in some cases, it may be more appropriate to complete the work in an alternate way. Parallel algorithms are usually less efficient than optimized sequential variants when executed on a single processor. As an example, knowing that the work is going to be executed locally, at least in part, a function may reuse some already computed results, instead of computing them again to avoid a remote reference. Another example possibility is to dispatch work that is likely not to share data with the current task, whereas it may

[4]We present, in Section 9.1, another scheme in which the resource is a slot in a task queue.

be more profitable to elect another piece of work that shares data with the just-executed one if no additional tasks could be created. Both examples try to reinforce locality of references and to avoid enlarging the set of recently manipulated data.

Third, this example highlights the simplicity of the task paradigm and the ease of use of Capsule probe and divide primitives. The concrete code remains readable and understandable in this implementation based on C, a language which is not especially recognized for its expressiveness. The example also illustrates the power of conditional parallelization. The user doesn't need to worry about the granularity of the declared tasks[5]. Encapsulation of the loop body into a separate function without side-effect is mandated by the model. It is also a good practice transformation because it makes apparent that operations performed in the loop body for distinct iterations are independent[6].

The choice of requiring the programmer to specify the work to be executed in parallel as a parameter to `capsule_probe` instead of `capsule_divide` opens the possibility for a run-time system to take this information into account when making the task creation decision. It may indeed make sense for a distributed implementation to refuse to create a task whose code is not available locally. Another possibility is to match the functional requirements of the code to some specialized units, such as accelerators (DSPs or FPGAs), and to refuse task creation if no units that provide the necessary functionalities are available.

The actual signature of `capsule_divide` is simpler than what is presented here: It cannot take a variable number of arguments. This is the unfortunate consequence of the limited capabilities of the C language for passing arbitrary arguments to a regular function[7]. The arguments for `loop_body` must in reality be stored in a structure allocated for this purpose and a special wrapper function must unpack them before calling `loop_body`. The complete code of the `Quicksort` benchmark's Capsule version on top of C can be consulted in Appendix A. Implementations of the programming model on top of higher level languages would not impose this constraint on the programmer.

2.3 Recursive Work Declaration

The code we presented in Figure 2.2 (facing page) is intuitive but not optimal. All potential work is effectively indicated[8] but it is not dispatched efficiently.

Indeed, before being able to probe for the possibility of executing iteration `n` in a separate task, the initial task must process the iterations from `0` to `n-1`. This is a consequence of the use of a regular `for` loop, which is inherently sequential, as in the original loop

[5]The influence of task granularity in practice depends on several factors such as the run-time system implementation or the underlying hardware architectures. For the implementation detailed in Chapter 3, task granularity's influence is studied in Section 4.2.3.

[6]Provided the functions in the program generally have no side-effects, i.e., don't access global variables.

[7]Tricks can be used to circumvent this limitation but are out of the scope of this discussion.

[8]Except the loop control itself, which contains so little work that it would obviously be useless to try to parallelize it as is.

construction of Figure 2.1 (page 17). If the corresponding iterations are on the program's critical path, the complete execution time will be extended by the launch delay.

Another problem with this approach is that work is declared in small units, which makes the number of divisions to execute the loop work in parallel higher. Although a programmer doesn't know a priori if it is efficient to try to spawn small tasks, and we argue he/she shouldn't care, it is expected that trying to dispatch bigger units of work at once will be more efficient, because this will lead to a lower number of calls to the probe primitive. The overhead of probes and, more importantly, divide operations will be minimized.

These problems can be solved satisfactorily thanks to *recursive work declaration*, a principle which can be stated very simply: *At any given point in time, a task should probe to split the known work that remains to be performed into two parts of approximately equal length.* In particular, this implies that new tasks will perform a probe as soon as they start, which may lead to the creation of more tasks, repeating the process recursively until all resources become busy. This greedy principle leads to the earliest possible dispatching of all units of work. For n units of work, one unit is dispatched after $\log_2(n)$ steps instead of at most n steps, assuming that all probes succeed. Splitting work in approximately equal length at each step avoids to create too many tasks for a given amount of work and is thus a good practice to follow, when task lengths can be evaluated and if the algorithm permits it. It is not necessary that the estimations be very accurate. The reader should notice that this practice doesn't guarantee in itself proper load-balancing. Dynamic variations due to architecture details, such as data residing in a particular cache, or to the used algorithms necessarily make static predictions inaccurate, even when the total amount of work is well-known from the start.

Let us change the original code from Figure 2.2 (page 18) so that the new version obeys the just-mentioned recursive work declaration principle. The revised code is shown in Figure 2.3 (facing page). We remark that, in the case of a `for` loop, application of the recursive work principle amounts to replacing the `for` with a kind of parallel `for` [111]. This new implementation tries to divide the loop in two parts of equal length and to distribute the second part to another task *at each iteration*.

Division is attempted through the `capsule_probe` call at line 22. If the probe succeeds, the right part of the interval (indices `j` to `n`) is dispatched to a new task that will execute the `loop` function on it. The original task will continue to process the left part (indices `i` to `j`). The `continue` statement at line 34 restarts the loop on the the first part, giving an immediate opportunity to split the remaining interval again, if resources permit so. When the probe at line 22 fails, it indicates that a new task can't be created, at least temporarily. Instead of retrying immediately, which would waste resources, the loop proceeds with some useful work, in this case the iteration with index `i`. This is done with the call to `loop_body` at line 39.

Figure 2.3 (facing page) highlights that indicating potential parallelism efficiently through task declarations is not completely trivial, even in the simple case of a perfect loop whose iterations are completely independent. In the common case of a `for` loop, a special `capsule_pfor` construct could be provided by the run-time system. Frequently used

```
loop_body (A, B, C, i)
{
  C[i] = A[i] + B[i];
}

loop (A, B, C, s, n)
{
  i = s;

  while (i < n)
  {
    // The number of elements for the 2nd part
    nb = (n - i) / 2;

    // The start index for the 2nd part
    j = n - nb;

    // Are there elements in the 2nd part?
    if (nb)
    {
      // Yes!  We probe.
      capsule_probe (& loop_body, & ctxt);

      // Probe result?
      if (ctxt != NULL)
      {
        // Execute the 2nd part in another task.
        capsule_divide (ctxt, A, B, C, j, n);

        // Update the upper bound for 1st part.
        n = j;

        // Immediately try to divide the current interval again.
        continue;
      }
    }

    // Execute the loop body with the current value for i.
    loop_body (A, B, C, i)

    // Next iteration in the current interval.
    ++ i;
  } // End of while.
} // End of the loop function.
```

Figure 2.3: Addition of 2 vectors in Capsule pseudo-C code, work-optimal variant.

constructs of sequential programs more generally should have a parallel counterpart in the CAPSULE environment to improve programmers' productivity and the efficiency of the code they produce. Nonetheless, the current implementations do not have such constructs because we concentrated on fundamental concepts on which they can be built afterwards. Although possible in C, with the use of function pointers, introducing a `capsule_pfor` construct would be more appropriate in a higher-level language than C, such as C++, Java or Python.

2.4 Coarse-Grain Task Synchronization

During an execution, a program performs numerous work declarations, some of which are executed in separate tasks created by the run-time system. Often, units of work are not independent: Some of them must wait for the completion of others before they can start. A typical example is regular scientific computations which follow the simple fork/join pattern. A single thread is permanently executing and, at some points, launches other threads to assist in processing a particular region or running an algorithm. Having completed its work share, it waits for the others to finish. Then, it resumes sequential execution.

OpenMP [40, 41] is a popular programming model supporting fork-join style computations[9]. Parallel regions are enclosed within `parallel` and `parallel end` directives. The latter acts as an explicit synchronization point for the worker threads enlisted for the region by the run-time system[10]. Parallel computation thus takes place in a flat delimited section. Nested sections are not supported by all implementations. OpenMP version 3 [41] introduced the `task` directive to explicitly declare tasks. The provided synchronization primitives to manage them are `barrier` and `taskwait`. The first is a regular barrier, i.e., it must be executed by all threads and will block the ones reaching it until all execute the statement. The second specifies a parent-children synchronization: A task wait for its immediate child tasks to complete. In the case of recursive work distribution, performing synchronization with these constructs is difficult. Barriers impose that all threads execute the same sequence of parallel regions, whereas `taskwait` primitives must be issued at each level, which often constraints the possible execution patterns more than necessary.

Figure 2.4 (facing page) shows the execution of a simple fork-join computation with 4 tasks coded with parent-children synchronization. Each line of the Figure represents the life of a single task. Task 1 launches directly or indirectly all the other tasks and then waits for their completion. However, tasks 3 and 4 are launched by task 2, following a computation. Parent-children synchronization prevents task 1 from waiting on them directly. The workaround is to have task 2 wait for tasks 3 and 4 and in turn task 1 wait for task 2, causing the context for task 2 to live longer after the end of its own computation.

[9]OpenMP primitives are passed to the compiler in the form of preprocessor directives for the C/C++ implementation. For that of Fortran, they are signaled with prefixes: `!$OMP` for free source form files, and `!$OMP`, `C$OMP` or `*$OMP` for fixed source form files.

[10]In OpenMP parlance, the threads form a *team* that is created at the start of the section by the thread executing the `parallel` construct. The latter thread is called the *encountering thread*.

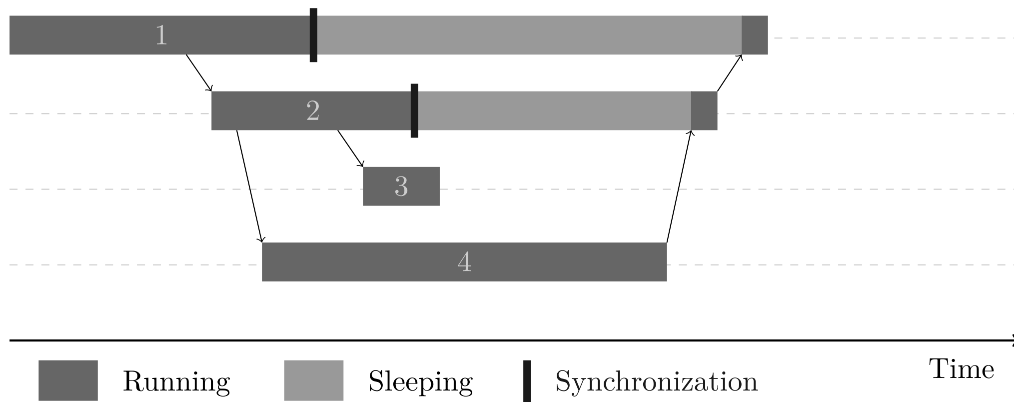

Figure 2.4: A fork-join computation implemented with parent-children synchronization.

Moreover, task 2 is woken up when tasks 3 and 4 have finished just to immediately end and finally wake up task 1. More generally, the parent-children synchronization paradigm has four major drawbacks. First, it requires to preserve contexts of all tasks but leaves in the task tree. Second, a processor/core is transitorily needed to process tasks woken up by synchronization events, even if they are to terminate immediately after. Third, performing synchronization at all levels in the task tree induces a higher global synchronization latency. Four, a task can't spawn tasks executing different algorithms or phases and selectively wait for some of them afterwards.

With the POSIX threads interface [129], a thread can wait for another thread by calling the `pthread_join` function with the identity number of the latter. Only threads whose joinable flag is positioned can be waited for with `pthread_join`[11]. A given thread can be the target of a `pthread_join` only once[12]. Identity numbers may be passed from one thread to another. Semantically, this model is the most flexible: Any thread can wait on any other thread. The associated downside is a considerable complexity for other models than pure fork-join. But the main drawback of this paradigm is the impossibility to wait for a group of threads with a single call. The identity numbers of all the threads to wait for must be stored and their termination has to be checked *sequentially*.

Figure 2.5 (following page) shows the same fork-join execution as Figure 2.4, but implemented with the POSIX threads' synchronization constructs. Task 1 launches indirectly three other tasks and, after performing some computation, waits for them to complete. Assuming that the corresponding three `pthread_join` are performed respectively with tasks 2, 4 and 3 as the thread argument, task 1 is woken up three times. Except for the last one, the wake-ups are spurious. They correspond to a returning `pthread_join` function immediately followed by another call to `pthread_join`. Obviously, if the task terminating

[11]The final draft of the POSIX specification indicates that threads are created joinable by default. Creating them as non-joinable (`pthread_attr_setdetachstate`) or detaching them (making them non-joinable) while they are running is possible and encouraged to free up resources.

[12]This restriction is due to the coupling between the use of join information, including the return value of the joined thread, and their immediate deletion after their first use.

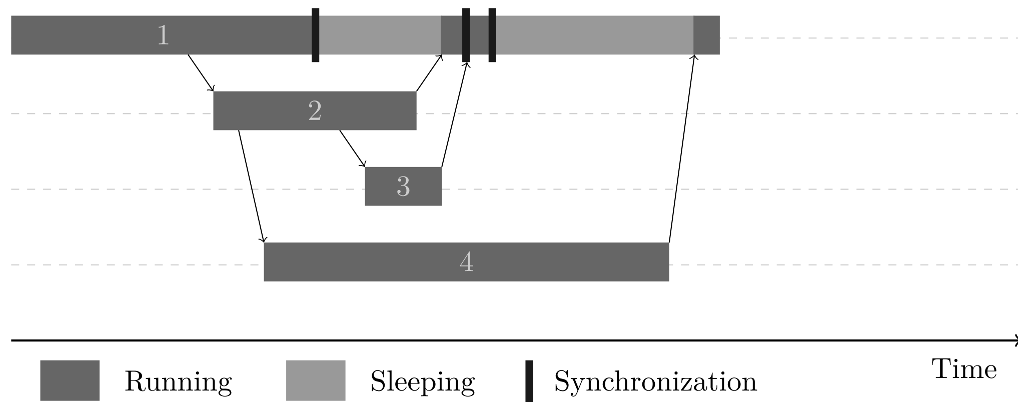

Figure 2.5: A fork-join computation implemented with POSIX threads style synchronization.

last is joined first, task 1 may not have to wake up more than once. In this case, the subsequent `pthread_join` calls will still incur some overhead. More generally, it is often not known in advance which task will finish first. Even in cases where performance models would make it possible to predict the last task to finish, environments like CAPSULE, in which work splitting and task dispatching is performed by the run-time system depending on the current system conditions, eventually make such a prediction useless in practice.

Compared to the fork-join model implementation of OpenMP, the POSIX threads synchronization model has lower wake-up latency, supports selective task waiting and does not require that the context of all non-leaves tasks be held in memory after execution of payload code. No synchronization at intermediary levels is necessary, which has the effect of diminishing the overall execution time. However, the POSIX model suffers from the problem of allocating a processor/core to a synchronizing task temporarily woken up between two calls to `pthread_join`. Moreover, a small amount of data about a task is kept after its end and until synchronization is performed on it. The total amount of synchronization data is thus at worst proportional to the number of tasks to wait for.

To overcome these limitations, CAPSULE features *synchronization groups* that comprise several effectively-created tasks. The intent behind the group concept is threefold. First, using context groups allows to wait for several tasks to complete at once, regardless of the completion order. Second, groups and their relations to tasks must be easy to understand for programmers so as to have practical interest. Third, most of the group management should be performed by the run-time system, involving the programmer only when absolutely necessary, i.e., to indicate program semantics.

CAPSULE groups form a dynamic hierarchy, from the initial group at the top of the group tree, which is automatically created at the start of a CAPSULE program, to the latest created groups at the bottom of the tree, for a given point of the execution. The user doesn't construct groups explicitly by assigning tasks to groups. It is the run-time system that does so automatically thanks to simple annotations added by the programmer according to the following rules. Every running task belongs to a single group, called

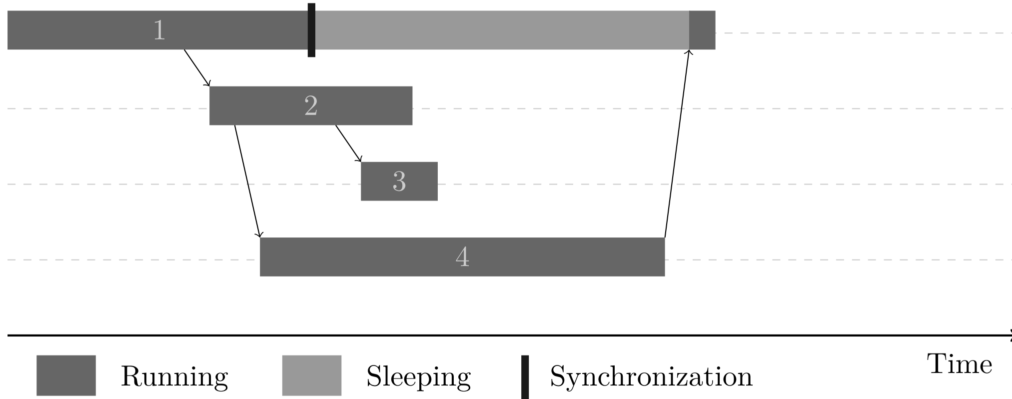

Figure 2.6: A fork-join computation implemented with Capsule's synchronization groups.

its father group or current group. The initial task[13] belongs to the initial group. Newly created tasks are assigned to the group of their parent task, a property called *group inheritance*. Using the `capsule_group_new` primitive[14], a task can create a new group, which automatically becomes a sub-group of the calling task's current group. The task is then itself associated with the newly created group.

The primitive to wait for tasks is called `capsule_group_wait`[15] and relies on the group hierarchy. The calling task is put to sleep until all other tasks in its current group and in all the descendant groups have terminated. Figure 2.6 shows the execution of the same fork-join computation with 4 tasks considered in the previous paragraphs, but here coded with Capsule primitives. There is only a single synchronization call to `capsule_group_wait` performed by task 1. No such calls need to be performed at the lower layers. Task 1 is woken up only when all other tasks in its synchronization group have completed. In the meantime, its context is saved and held by the run-time system. For this example, there was no need for `capsule_group_new`, because all tasks belong to the same computation and task 1 simply needs to wait for all the others. With the group inheritance property, all tasks automatically belong to the same group, the initial group.

In large programs, several mostly independent computations may be going on simultaneously. Some may have been launched directly by the program code, but some others may have been initiated by code from third-party libraries or components, in which case the calling program doesn't know about them. In order to facilitate libraries/components reuse, it is highly desirable that the synchronization operations they perform be invisible to the program using them. For this reason, Capsule provides a third primitive, `capsule_group_quit`, that permits a task to leave its current group and attaches it to the father group of the latter. We shall now detail how this primitive is meant to be employed

[13]The initial task is the sequential code that executes the Capsule run-time system initialization.
[14]In Palatin [201], the equivalent primitive is named `cap_split`.
[15]The primitive called `cap_join` plays a similar role in Palatin [201]. The main difference is that it causes the joining task to leave the current group as soon as it resumes, compromising the composition of several algorithms.

and why it allows, in combination with the other two, to achieve this goal.

Figure 2.7 (facing page) shows a computation that includes the tasks of the previously presented examples with some additions. The initial task is task 0, that launches two independent computations. Tasks 5 to 7 form the first one, whereas tasks 1 to 4, from the previous examples, form the second one. The former is not directly launched by the code from task 0, which ignores its existence, but by a library function or some component method in third-party code that it called. The callee also uses Capsule primitives and tries to launch tasks in response to the call to take advantage of the cores of the underlying architecture. The first computation is composed of a large task, task 5, that launches two other tasks. Task 5 needs the results of tasks 6 and 7 before finishing. Naturally, it should use `capsule_group_wait` to wait for them.

In order to isolate its tasks from those of the main program, the third-party function or component calls `capsule_group_new`, which leads to the creation of a new group, group 1, pictured in orange in Figure 2.7 (facing page), and to the transfer of task 0 to this group. Then, several tasks are launched using `capsule_probe` and `capsule_divide`, as explained in Section 2.2. The call to `capsule_group_wait` by task 5 thus will concern only the tasks that belong to group 1. In particular, all tasks launched prior to group 1's creation are not concerned by this synchronization. However, task 0 now also belongs to group 1. If the third-party function returned to the main code right away, the next tasks that are launched would belong to this same group or to a sub-group of it. The call to `capsule_group_wait` by task 5 would block until all these tasks are finished, which is not the intended control flow of this application.

Thus, just before returning, the third-party function must call `capsule_group_quit`, causing the calling task, task 0, to be assigned to the father group of group 1, which is the initial group. The primitive just transfers the group ownership for a single task. The group hierarchy is left unchanged. In this example, group 1 remains a child of the initial group. Then, the main program continues and finally starts the second computation, without the knowledge that another one was launched before. Whether the latter is still going on or has finished doesn't change the second computation's behavior.

The best practice is to systematically create a new group when launching a self-contained computation. If, in the second computation, task 1 had not needed the results of tasks 2 to 4, it would have been possible to omit the creation of the second group and the corresponding calls to `capsule_group_new` and `capsule_group_quit` surrounding the spawning of task 1. The initial group would have been the container of tasks 1 to 4. The `capsule_group_wait` issued by task 0, used to wait for the end of both computations, would have had the same effect. This optimization is analoguous to tail recursion for recursive function calls. Its use is however discouraged because it makes code reuse more difficult to achieve.

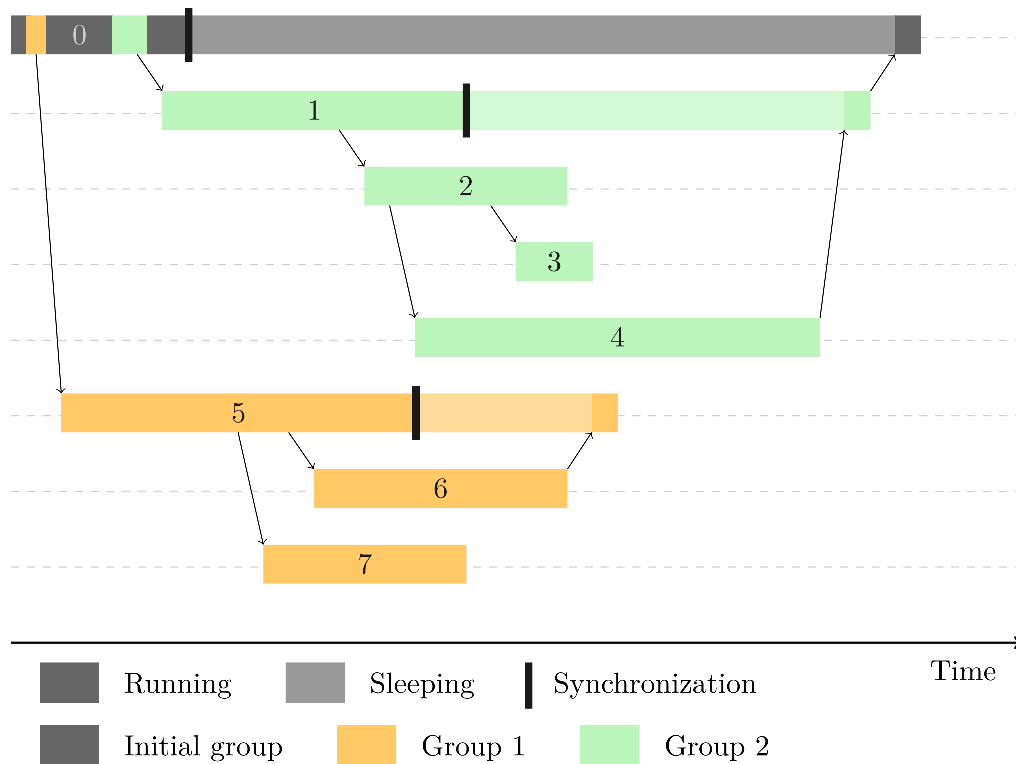

Figure 2.7: A Capsule computation with multiple groups.

2.5 Other Primitives and Abstractions

The previous Sections detailed the five core primitives that are unique to the Capsule programming model. This section presents additional primitives that are of less frequent use, but are still important from a practical point of view. They comprise the primitives for run-time system initialization and shutdown, some primitives providing per-task data storage and abstraction functions for mutual exclusion and other classical synchronization mechanisms, such as conditional variables. Finally, some primitives are used only for benchmarking purposes, to gather statistics for executions or to alter the run-time system's normal functioning in interesting ways.

Two primitives can be used to start the run-time system: `capsule_sys_init` and `capsule_sys_init_warmup`. Only one of them should be called and calling them more than once has no effect. The second performs the same initialization as the first, with additional actions to warm up the run-time system that are useful when benchmarking, as described later in this Section. The conceptual effect of the initialization is to take over the thread calling it, called the *initial thread*, i.e., to have it managed by the run-time system onwards. Also, the execution context at that moment is internally treated as a new task that would have been launched before the initialization. This pre-existing task is called the *initial task*. Finally, an *initial group* is created to contain the initial task.

The primitive to stop the run-time system is called `capsule_sys_destroy`. It must be called by the initial task exclusively. If other tasks than the initial one are still running, `capsule_sys_destroy` first behaves as if `capsule_group_quit` was called enough times to make its current group the initial group, and then as if `capsule_group_wait` was called. The net effect is that the initial task is unscheduled until all other running tasks end. Finally, all initialization operations are undone and the initial thread state is restored.

To facilitate program portability, Capsule provides its own interface to traditional synchronization primitives used in multi-threaded programs. Capsule programs, although using the concepts of potential parallel sections and tasks instead of threads, share with multi-threaded computations the possibility that multiple pieces of code may be executing concurrently. Like them, they must take precautions so that this possibility cannot cause data corruption. As said previously, the choosen interface to ensure proper synchronization is independent of the core Capsule concepts. Currently, locks for mutual exclusion, as well as atomic operations over C integers are provided. The interface may be completed with other synchronization objects, such as conditional variables, or replaced with more integrated mechanisms, like transactional memory [124].

Capsule also provides a simple interface to store and retrieve per-task data. Normally, a task starts with parameters passed by the task that launched it. Consequently, there is conceptually no need for a special per-task data mechanism: The father task simply pass the necessary data to the new task at startup. However, when modifying large programs that already use per-thread data storage, it is often more easy to turn the corresponding function calls to the per-task facilities of Capsule instead of substantially modifying the code to pass some data down to spawned tasks. These primitives are merely a programming convenience for quick conversion of a multi-threaded program to use the Capsule primitives, where the new version tries to create as many tasks as threads of the original version. In this implementation, only a small amount of data may be stored per-task[16]. A destructor may be associated to it and will be automatically called when the task ends[17].

Benchmarking is facilitated by a series of primitives and abstractions. The `capsule_sys_dump_all_stats` primitives outputs to a stream run-time system statistics such as the number of probes and divisions, as well as other numbers specific to the implementation. The `capsule_sys_block` primitive prevents all subsequent probes from succeeding, i.e., no new tasks are ever spawned. The `capsule_sys_unblock` primitive can later reallow probes to succeed. This is useful to test sequential execution of some parts of a Capsule program. Finally, the primitives `capsule_abs_get_ticks` and `capsule_abs_get_time` return the current time in ticks or in some time unit. The precise significance of ticks and the time unit used depend on the underlying architecture. Only relative values, i.e., the difference of numbers obtained through different calls to the primitives, are significant. Programs should use these primitives to measure the execution of program parts instead of the system-provided facilities to preserve program portability.

[16] Currently, a simple pointer.

[17] This interface is simpler than the POSIX threads interface, which can name different pieces of data through keys. This latter possibility usually goes beyond what is actually needed by applications.

Chapter 3

Run-Time System Implementation

This Chapter details a run-time system that supports CAPSULE's programming model as presented in Chapter 2.

Originally, the programming model only consisted of the conditional parallelization concept and an associated primitive to declare a new task, `nthr`[1] [202]. The associated implementation was an 8-way SMT processor modified to support `nthr` [202]. It provided hardware support for locks, but no coarse-grain synchronization primitives; an implicit synchronization was performed at program end.

Later, the programming model was enriched with rudimentary primitives around the concept of synchronization groups. `nthr` was replaced by the two primitives `capsule_probe` and `capsule_divide`[2]. Finally, a software-only run-time system was implemented. This version proved that the conditional parallelization concept could be efficiently implemented purely in software [54].

The current run-time system has been rewritten from scratch for several reasons, of which the foremost was to make it portable by abstracting particular operating system interfaces and machine architectures[3]. The current version is the first to include the synchronization groups primitives presented in Section 2.4. Other reasons for the rewrite included cleaning up the code, removing the remaining deadlocks and group handling bugs and optimizing performance.

Section 3.1 presents the implementation of the task abstraction and the mechanisms to map tasks onto computing resources. Then, in Section 3.2, we detail the actions performed by the run-time system to achieve conditional parallelization, i.e., the decision policy for accepting or refusing to create a new task. Section 3.3 describes the structures and

[1] `nthr` stands for `new thread`, a name which conceptually constrains an implementation to immediately launch a new thread, if the run-time system accepts to execute the task in parallel.

[2] The original names were `capsys_probe` and `capsys_divide`, see Palatin [201].

[3] The previous version was mostly POSIX compliant, but used some Linux's non-portable system calls. It also relied on some atomicity guarantees provided by the x86 architecture that were not isolated from "regular" C code.

algorithms to implement synchronization groups. Finally, Section 3.4 lists the portability abstractions that are used by CAPSULE's core. To port the run-time system to another machine/OS, only adequate implementations of these abstractions must be written, leaving more than 75% of its code lines intact.

3.1 Task Abstraction and Scheduling

Tasks in CAPSULE correspond to some work that can be performed in parallel and that was effectively authorized by the run-time system to be executed on another computational resource. In this C implementation, a work unit is specified as a function, which is its entry point and is passed as an argument to the `capsule_probe` primitive. If the probe succeeds, the program calls the `capsule_divide` primitive with two arguments. The first is a pointer to an opaque context structure that was returned to the program by `capsule_probe`. The second is an argument of type `void *` which will be passed to the task entry point when it starts executing.

Because a new task may not be started immediately, the information about the entry point and its argument must be stored. This is the role of a C structure called a *context*. Such a structure is created each time a probe succeeds and lives until the new task created by the corresponding divide ends. In addition to the above-mentioned information, it also contains a pointer to the current group for the task. The handling of groups is the subject of Section 3.3. Per-task data is also stored in this structure. Finally, it may also contain information about scheduling, as explained later.

Once started, tasks can suspend their own execution because they block on some fine-grain synchronization object, like a lock, or because they invoked a coarse-grain synchronization construct, like `capsule_group_wait`. The run-time system makes the assumption that the programmer follows the good habit of parallel programming that locks must never be held for long. If it is true, switching contexts to execute another task while the previous one sleeps would be more costly than just waiting for the lock to become available. Consequently, we have chosen not to implement context switching in this case. For coarse-grain synchronization, the situation is most often reversed: Waiting for other tasks' completion may be extremely long, and ultimately depends on the amount of work they still have to perform, which is very unlikely to be small. Thus, the run-time system always suspends a task calling `capsule_group_wait`, unless it is the only running task in its group, in which case the primitive returns immediately. This last optimization is valuable because it saves two context switches in the case where the caller's work was the longest of its group, allowing all other tasks to end before it called `capsule_group_wait`.

In the programming model, tasks are scheduled onto computing resources. The current implementation schedules tasks into platform-provided threads. At startup, the run-time system creates a fixed number of threads corresponding to the number of cores/processors in the system, thanks to the programming interface abstraction described in Section 3.4. Currently, the scheduling of these threads onto the actual hardware execution units is left

to the underlying OS or threading library[4]. Each thread is represented in the run-time system by a structure of type thread that serves to hold a pointer to the currently executing context, as long as the thread is processing a user task, and to store the corresponding machine execution context. Both these pieces of information are useful when context switching a task. A description of the context switching API and implementation is described in Section 3.4.

The run-time system employs one of two alternative strategies to schedule tasks onto the threads it created. The strategy is chosen when the run-time system is compiled. The first alternative is to have the run-time system decide which thread is going to handle a particular task. It requires that it maintains a list of its threads and extra information to be able to determine if a given thread is busy or idle. When a probe succeeds, the run-time system tries to find an idle thread to execute the task.

The current implementation maintains two lists of threads: A list of idle threads, available to execute some context/user task, and a list of busy threads. The run-time system chooses the thread at the head of the free list to execute a context. It pushes back a thread that became idle also at the head of this list. The latest idle threads are thus reused in priority[5]. The selected thread is stored as a pointer to its structure in the context structure just before being returned by `capsule_probe`.

When the father task finally calls `capsule_divide`, the thread that executes it transmits the new context to the selected thread by modifying a field of the thread's structure. Then, it signals the selected thread using the function `capsule_pi_thread_wakeup`, which, for this alternative, takes a pointer to the selected thread's structure. Once woken up, the selected thread reads the context to execute in the appropriate field of its structure. More generally, this technique is employed each time a thread needs to inform another thread about an action to perform, even if the latter is not to execute some user context. Such situations are described in the next Sections.

The second alternative is to have the system select the thread to execute the new context. This strategy has the advantage that the underlying OS or threading library knows the system state and in particular the scheduling status of all threads and which execution units they are currently assigned to. This knowledge can help to make a better decision about the particular thread to signal in order to minimize wake-up and scheduling latencies. Compared to the first alternative, the implementation doesn't need to maintain some lists of threads. A simple counter in the run-time system's main structure indicates the current number of idle threads. Another difference is that the run-time system cannot push the new context to the thread that will process it, because most OSes or threading libraries[6] do not usually indicate the thread(s) that they wake up. The selected thread has to pull it from a predefined location.

The naive implementation is to hold a to-be-executed context in a field of the run-time system's main structure. As the number of cores grows, several threads may be issuing

[4]See Section 3.4 for more details on this topic.
[5]This policy was also that of the previous run-time system version.
[6]We are not aware of a single one offering this possibility, even those performing userland thread scheduling.

division requests simultaneously. A single slot in the main structure practically serializes work distribution to idle threads, because a thread cannot place a context in the slot until some idle context retired the previous one to execute it. This has an adverse effect on performance as soon as around 10 cores are used, as can be seen in the experimental results of Section 4.2.1. Augmenting the number of slots reduces the delay during which a thread waits to deposit its task. However, care must be taken to avoid contention on the cache lines where the slots are stored, since the penalty for such cache conflicts grows at least linearly with the number of cores.

A more elaborate solution would be to use a hash function that maps a thread's index to a particular context slot. An idle thread would start the search for a context to execute from the slot returned by the hash function applied to its own index. If this slot was empty, it would then try with the next slot instead and the process would continue until it actually found a context to execute. When a thread would create a new context, it would try to place it in the slot returned by the hash function applied to its index. If the slot was already occupied, it would try with the next slot until finding a free one. This version has been implemented but no experiments have been done with it yet.

One may think of mitigating the serialization effect using an hybrid method that would maintain thread groups and signal one of them when a new context to execute is available. At the OS or threading library's discretion, one of the threads in the group then would handle the signal. This proposal requires that the run-time system track the availability of threads per thread group. It also suffers in part from the same drawback as the first alternative, because only the threads in the selected group are available to the OS or threading library to make an optimal decision. It might be possible to compensate partly for this problem by creating more threads than actual cores/processors or by having threads belonging to multiple thread groups. We have not explored these possibilities.

Circumventing the false sharing caused by automatic hardware cache coherence and its performance impact amounts to finding a distributed algorithm to dispatch the work created after a successful probe. This task is not trivial partly because the current algorithm to make a decision regarding a probe, which is described in the next Section, treat all the available resources identically, i.e., without consideration for their location, the time to access them, the tasks they previously processed, etc. Chapter 9 presents a completely distributed scheme where probe decision and work dispatching are more coupled in this respect.

3.2 Conditional Parallelization

When a thread issues a probe, by calling `capsule_probe`, the programming model states that the run-time system decides whether it is *worthwhile* to create a new task to handle the declared work unit in parallel, as was explained in Section 2.2. It doesn't, however, specify the precise meaning of worthwhile, leaving it to the run-time system implementor to choose some policy adapted to the hardware it targets and the applications that are of interest to him.

The only common constraint faced by all implementations is that *the probe primitive should execute as fast as possible*, because calls to `capsule_probe` may be inserted anywhere in a program, including in innermost loops or critical paths. During a program run, probes may be issued very frequently, whereas the number of divisions will comparatively be very small. The bottom line is that a program's overall execution time is less sensitive to the positive probe code path's execution time than to that of the negative probe and divide code paths [201]. It is thus especially critical that the probe decision algorithm be simple enough to be able to produce a negative response extremely quickly. For computationally-intensive algorithms with fine-grain parallelism, a probe that fails should execute in as low as a few cycles. Comparatively, positive decisions can consume more computing time, since they will be less frequent and most often compensated for by the work that will be executed by another resource.

In the implementation presented in this Chapter, the chosen policy[7] is that a probe is allowed to succeed only if one execution unit is *immediately available* to process the corresponding work unit, which has two advantages. First, the run-time system doesn't need to know which precise execution units are available to make a decision. It is enough to keep a single shared counter indicating the current number of idle threads. If the counter is zero, then no threads are available and the probe fails. Conversely, if the counter is non-zero, some thread is available and the probe can succeed, without having determined at this point which execution unit will actually execute the new task. Second, once a thread has called `capsule_divide`, it has the guarantee that the new task will be started as soon as possible by some idle thread and thus a currently unused execution unit[8]. This property is valuable for embedded systems executing applications that must meet some soft real-time constraint. Additionally, the policy is simple, which makes it relatively straightforward to implement, with a low demand on energy, a critical advantage when choosing a run-time system for an embedded system. Its main drawback is that it requires programs to issue probes frequently to produce good scalability. Free hardware resources sit idle at least until one thread issues a probe, regardless of the amount of available parallelism. The reason is that tasks cannot be declared "in advance" when all resources are busy, as the run-time system doesn't maintain task queues. Another policy involving task queues is presented in Section 9.1.

The policy's implementation is the following. On a call to `capsule_probe`, the run-time system needs to read the shared counter indicating the current number of idle threads to decide if a new task will be created. If the answer is positive, the counter, and possibly other data, will need to be updated to reflect the reservation of a thread that will process the work unit when `capsule_divide` is called. These operations delimit a window of time during which an harmful race condition may occur between two or more tasks performing a probe. If one thread reads the shared counter and the counter is non-zero, it will allow a task creation. If, before decrementing the counter and updating other data, other threads read the shared counter but do not make enough progress to update it, they will find

[7] This policy is the same as that presented in [201].

[8] The time necessary to start the new task depends on the underlying OS and threading library implementations.

it all in the same state, and all these probes will succeed as well, even if they are more numerous than the counter's initial value. Worse, even if the counter is higher than two and only two threads are involved in the race, the shared counter will not be updated correctly, since both will have read the same value, decremented it by one and stored back this same result. The run-time system will later believe that there are more idle execution units than there are in reality, causing interesting bugs.

For this reason, the counter must be read and updated using atomic operations[9] and it must be protected by a lock. However, a lock acquisition can be costly, especially when there is much contention on it. Since the counter is global, the number of threads competing for the lock can be as high as the number of execution units in computationally-intensive pieces of code. Palatin [201] reports a delay of 121 cycles on an Intel Core 2 Duo (2 cores) to perform a locked probe that returns a negative answer.

In order to reduce the probe overhead, `capsule_probe` can read the shared counter holding the number of idle threads with an atomic operation but without acquiring the lock. This value is then used as a hint to optimize the frequent case of a negative response, where no idle threads are available and the shared counter is zero. Performing a non-protected read of a shared variable that can be modified concurrently will sometimes cause the hint value to become outdated before being used by the probing thread.

The first possible case is that threads probing at the same time may not have decremented the counter yet, and the probing thread may see a greater number of idle threads than there actually are. The update of the counter of idle threads must anyway be performed through an atomic fetch-and-add operation or under the lock's protection. Consequently, the thread will notice the discrepancy later in the probe process. If the counter has been changed to 0 in the meantime, the probe will be finally refused, although the lock was acquired. This case is fortunately infrequent because it occurs only if several probes are racing to take over the last available threads. The overwhelming majority of probes are performed whereas all cores are already busy and are refused immediately.

The second possible case is that some thread that has just finished to execute some user task has not incremented the counter yet. The probing thread may thus miss an opportunity to allow a probe to succeed, if it sees the counter as being 0 where a locked access would see a strictly positive value. This situation can occur also with a locked access, although less frequently. It is inherent to the chosen policy, which considers only the resources as they are when the probe is executed[10]. It is not too harmful to miss a probe if the program probes frequently, because the waste of resource is limited to the interval between two probes.

Overall, this speculative technique allows the negative response code path of a probe not to acquire a lock to read the shared counter, at the expense of denying slightly more probes than necessary and increasing the length of the positive response code path by an additional L1 cache access[11]. Using a micro-benchmark that computes the difference

[9] On most architectures, including recent x86 processors, read or write to a word (32 bits) is necessarily atomic. In this case, atomic operations can be replaced by regular memory operations.

[10] Some anticipatory scheme could be used here, and may improve performance for some applications.

[11] The speculative technique introduces an additional counter read without lock. If the counter is not in

between the execution time of a loop executing 100 billions probes that fail and that of the same empty loop, the cost of a failed probe has been evaluated to 2.109 cycles[12] with less than 1‰ of error on an Intel Core 2 Duo processor[13].

The actual cost of a particular probe during a real program execution is extremely hard to predict. It depends on the processor's branch prediction unit's performance, which in turn depends on the algorithm and amount of state it maintains, possibly the processor's pipeline state and finally the outcome of the previous probes by the same thread. It also depends on the contention on the shared counter and the particular cache coherence algorithm and implementation. However, this cost is most often the value returned by the micro-benchmark because most of the probes fail. Branch predictors can easily deal with monotonic sequences of branch outcomes and there is no contention during periods when all probes fail because the counter is not modified.

3.3 Synchronization Groups

Synchronization groups are represented as C structures within the run-time system. As said in Section 3.1, a context holds a pointer to a group structure. By contrast, a group structure doesn't maintain a list of the contexts that belong to it and of its sub-groups. A counter called the *group counter* holds the sum of the number of running contexts belonging to the group and the number of sub-group the group possesses. Another counter serves to store the number of waiting contexts in the group, i.e., how many of them called `capsule_group_wait`. The rest of this section describes the mechanisms and algorithms employed to implement the group semantics, which only require these two counters' presence.

When a new task is spawned by the divide primitive, it is attached to the father group of the spawning task, as a result of group inheritance. From the current context structure, the run-time system retrieves the father group. It then acquires the group lock that prevents concurrent modifications to the group structure. The group counter is incremented to reflect the fact that a new task references it as its father. Finally, the group lock is released.

When a task finishes, before releasing any resources associated to it, its owner group is updated as specified by the following algorithm:

1. The group lock is acquired.

the L1 cache, then it would not have been in the cache either for the first subsequent protected access inside the locked section. Hence a difference of a single access to the L1 cache.

[12] Palatin [201] reports 1 cycle on a Intel Core 2 processor with the old run-time system implementation, which uses a single global variable containing the number of available threads. The new implementation performs an additional indirection, which explains the difference.

[13] For the record, here are some other measurements: 1.872 cycle on a two quad-core Intel Xeon E5430 machine (Hapertown processor, Penryn microarchitecture), 0.931 cycle on a four quad-core AMD Opteron 8380 (Shanghai, K10 microarchitecture). All the numbers are reported with less than 1‰ of error at 99.9% confidence.

2. The group counter is decremented by one. If the counter is non-zero, other tasks belonging to this group or to some descendant group are still running. In this case, the run-time system immediately jumps to step 5. Otherwise, it continues with the next step.

3. The counter of the number of waiting tasks in the group is tested. If it is non-zero, some tasks were waiting for all the running tasks in the group to finish. In this case, the run-time system wakes up as many tasks of them as possible. It respectively decrements the counter of waiting tasks and increments the group counter by the amount of tasks that it actually woke up. Finally, the run-time system proceeds right away with step 5.

4. If the run-time system reaches this step, then both counters in the group structure are zero, meaning that no tasks belong to the group and that the group has no sub-group. Consequently, the group structure can be destroyed. Before that, the pointer to its parent group is copied. The group's disappearance must indeed be signaled to its parent group, because the latter possibly holds tasks waiting for tasks in its sub-groups. To this end, the run-time system first proceeds with step 5. Afterwards, it starts recursively the whole algorithm using the parent group's pointer copy.

5. The group lock is released.

Once the algorithm has terminated, the context structure of the finishing task is destroyed.

Because this algorithm starts as some task ends but just before its resources are released, the run-time system can schedule, at step 3, at least one task to be woken up immediately by making it reuse the previous task's resources. This property guarantees that, each time the algorithm is run, the counter of waiting tasks strictly decreases, unless there are intervening calls to `capsule_group_wait`. Since one task at least was woken up, the algorithm will be run again when it ends. As a result, the counter of waiting tasks eventually reaches 0 and all previously waiting tasks are eventually scheduled. The current implementation, which creates tasks only when some thread is idle, reassigns the thread of the task that ended to one of the waiting tasks, which can proceed immediately. It has the limitation that only one task is allowed to wait for other tasks in a given group, because this feature was not necessary to implement our benchmarks.

A call to `capsule_group_new` retrieves the current group from the current context and then creates a new group which will be a child of the latter. The structure of the now father group is left unchanged: The counter of alive contexts and sub-groups keeps the same value since the current context, that was part of it, is now part of the new sub-group. Apart from the creation of a new structure for the new group, only the current context structure's father group pointer is changed.

3.4 Portability Abstractions

Since the Capsule programming model is general and intended for widespread use, the supporting run-time system must be easy to port to new general-purpose architectures. To this end, all the run-time system's code that is dependent on the machine architecture and on the programming interface, e.g., the OS and threading libraries' APIs, is confined in specific files and implements a single abstraction interface. The core of the run-time system, which implements the Capsule programming model, only uses functions of the abstraction interface. This approach has several benefits when the run-time system is ported to another machine architecture/operating system. First, only approximately 25% of the run-time system code lines[14] need to be changed, instead of the whole run-time system. Second, the core's code lines are left unchanged, which implies less chance to introduce new bugs. The rest of this section is an overview of the abstraction interface and its design choices. It also presents some details about the reference implementation, which runs on top of a POSIX operating system with POSIX threads[15].

The primitives related to concurrency are completely abstracted by the interface, which is roughly a simplified version of the POSIX thread interface [129]. This choice was made because the POSIX thread interface is general, reasonably well-designed and easy to implement with the primitives of other popular systems, e.g., those providing the Win32 API. The `capsule_pi_thread_create` function creates a new thread. Functions to manipulate sleeping locks (mutexes) are `capsule_pi_mtx_init`, `capsule_pi_mtx_destroy`, `capsule_pi_mtx_lock` and `capsule_pi_mtx_unlock`; their name is self-explanatory. Their single argument is a mutex structure of type `capsule_pi_mtx_t` holding the information necessary for the implementation. Similar functions are provided for conditional variables, with the same distinction as in POSIX between waking up a single thread (`capsule_pi_cv_signal`) and all threads (`capsule_pi_cv_broadcast`), and the same requirement to associate a mutex to a conditional variable while waiting for a signal on it.

In the POSIX thread implementation of this interface, `capsule_pi_thread_create` is a wrapper around `pthread_create` with some code to initialize the structure used by the run-time system to represent a system thread. A mutex structure, whose type is `capsule_pi_mtx_t`, simply contains a `pthread_mutex_t` object. The associated functions call their counterparts in the POSIX interface with default attributes. The implementation of other structures and functions of the interface is similar. The POSIX implementation of the interface leaves the scheduling of a task on an actual core to the POSIX thread library and the underlying OS, since the run-time system may schedule a task on a given logical thread, but not some hardware thread directly. Implementing the abstraction interface doesn't however require a threading library. It is possible, for some bare hardware without OS, to code the abstraction functions using assembly language and low-level knowledge of the hardware architecture. The scheduling of the logical threads on the hardware threads is then up to the implementation, which can use for example a simple policy like pinning

[14] Measured on our reference implementation.

[15] Operating systems supporting these interfaces include Linux, the BSDs (NetBSD, FreeBSD, OpenBSD, DragonFlyBSD), SunOS and derivatives (Solaris), and AIX.

each logical thread on a different hardware core.

The run-time system uses per-thread local storage to store a pointer to the structure representing a thread, which is detailed later in this section. The corresponding abstraction functions are `capsule_pi_thread_key_create` and `capsule_pi_thread_key_destroy` to create and destroy keys serving to reference particular thread-local data, and `capsule_pi_thread_data_set` and `capsule_pi_thread_data_get` to set and retrieve data corresponding to a given key. The POSIX implementation of these functions simply calls the POSIX direct counterparts: `pthread_key_create`, `pthread_key_destroy`, `pthread_setspecific` and `pthread_getspecific`. To provide timing facilities to both the run-time system and user programs[16], the function `capsule_pi_get_time` provides a time measurement facility. Its implementation on top of POSIX is a wrapper around the `gettimeofday` system call.

At initialization, the run-time system uses the `capsule_pi_get_num_procs` of the abstraction interface to retrieve the number of processors/cores in the system. In the POSIX implementation, this call is translated into a call to the `sysconf` system call with the `_SC_NPROCESSORS_CONF` argument[17]. `_SC_NPROCESSORS_CONF` indicates to the OS to return the number of processors that were configured. A better argument in the future may be `_SC_NPROCESSORS_ONLN`, indicating the number of processors that are online. However, the run-time system currently doesn't support CPU/core hotplugging and will have to be modified for the new argument to be introduced.

Context switching facilities form another important part of the abstraction interface. They are currently used by the synchronization groups handling code when some task is suspended as the consequence of calling `capsule_group_wait` while other tasks in the group are running. The run-time system switches context to free the waiting task's core in order to provide more resources to tasks making progress. The portable context switching API has been inspired by the UNIX context handling functions, but the latter has been simplified and abstracted.

The `capsule_pi_ex_ctxt_swap` function swaps the current execution context with that stored in an ad-hoc structure. The `capsule_pi_ex_ctxt_make` function serves to create an execution context that, when scheduled, will cause the execution of the specified function. Other parameters include functions to allocate and deallocate a stack for the context. `capsule_pi_ex_ctxt_destroy` destroys an execution context and the allocated stack. These functions are implemented with the UNIX context handling functions (`makecontext`, `swapcontext` and `getcontext`) on POSIX systems[18].

The core implements its own high-level context switching functions on top of the above-mentioned ones. These functions uses the calling thread's structure which, as mentioned in Section 3.1, contains a pointer to the current user context, if the thread is executing one, and a slot to store the machine execution context when a task is suspended as a consequence of a call to `capsule_group_wait`. In the end, these different layers allow

[16]Through `capsule_abs_get_time`, see Section 2.5.

[17]This argument is not standardized by POSIX nor the Single Unix Specification version 3. However, it appears to be implemented in a sensible way in all the major POSIX operating systems.

[18]These functions have been deprecated, but still are implemented on the major systems. They are more efficient than the traditional `longjmp` and `setjmp` families of functions [87].

the core code to create new contexts that can later wait for some task to be created, to save the current context and switch to a new one or to switch to some group's waiting context with a single and simple function call.

In order to improve performance and to be able to implement the conditional parallelization policy, as described in Section 3.2, the run-time system core needs to use atomic instructions, in addition to locks. A specific part of the abstraction interface, called the *machine abstraction interface*, provides functions that realize machine-dependent operations, i.e., those that rely on particular machine and processor architectures[19]. It provides simple atomic read and atomic write operations, as well as compare-and-swap. Instead of implementing the latter in assembly, we relied on the builtin functions provided by the GCC compiler [95] (`__sync_bool_compare_and_swap` and `__sync_val_compare_and_swap`). Portable interfaces are defined to manipulate atomically booleans, integers and pointers, but also 64 bits numbers[20].

Initially, the interface included some functions for a thread to wait for a new user task to execute[21] (`capsule_pi_thread_wait`) or, conversely, to signal that a new user task needed to be handled by a thread (`capsule_pi_thread_wakeup`). Two alternatives for the functions were provided, depending on the chosen strategy to schedule tasks onto the run-time system's threads, as explained in Section 3.1. In the alternative where the run-time system explicitly chooses the thread to handle a particular task, `capsule_pi_thread_wakeup` takes a pointer to the thread structure of the thread to wake up. In the alternative where the task scheduling decision is left to the OS or the architecture, this function takes instead two arguments: The main run-time system structure, through which the tasks pass between creation and take over by a thread, and a structure indicating the reason for the wake-up, either a user task to execute or a signal to shut down.

This situation made the code somewhat complex and had two main drawbacks. First, code specific to some programming interface had to manipulate core run-time system structures, implying that it had to be changed when some core structure or its synchronization protocol was changed. Second, this code had to be duplicated in each programming interface support. Since these pieces of code mostly consisted of elusive synchronization code, one could introduce bugs easily when porting the run-time system to a new programming interface. In the end, we decided to implement the relevant synchronization mechanisms once and for all inside the core. The abstraction interface was changed to include lower-level and simpler mechanisms.

The abstraction interface allows a wide range of implementations for the base concurrency mechanisms, making the run-time system portable over all general-purpose machines. It is even possible to support raw hardware, i.e., machines without operating systems, by using the appropriate C or assembly code to implement simple threading with one thread being assigned and run by one core. A userland threading implementation of the abstraction interface is used in Part III (see Section 13.1.4) as part of the evaluation of

[19] These operations are independent of the particular programming interface, i.e., the OS or threading library.

[20] This interface allows to detect overflows when adding numbers.

[21] Or for some other action triggered by the run-time system itself, such as shutting down.

the concepts presented in Part II.

Chapter 4

Performance Study

This Chapter presents performance results of software run-time system implementations for the CAPSULE environment that were evoked in Chapter 3.

Section 4.1 is a study of performance scalability and variability of regular and irregular programs, with complex control flow and data structures, on multi-core platforms. It compares the performance of common but naive parallel versions of these programs to what can be achieved by porting the program to CAPSULE. It shows that CAPSULE parallelization offers both higher scalability and less execution time variability over data sets. The reduced execution time variability enables to accurately and quickly forecast the performance of computationally-intensive program pieces, a feature especially useful for embedded systems that often run applications with soft real-time constraints. Section 4.1 is an adaptation and refinement of the content of Certner et al. [54].

Section 4.2 is a study of the influence of the run-time system implementation and the underlying architecture on program performance. It first compares the performance of the previous implementation and that of the current one using an example benchmark. For the current version, it gives different sets of results corresponding to some of the scheduling alternatives that were described in Section 3.1. The new version can achieve better performance than the old one, especially for a high number of cores, despite the introduction of several abstractions and some new functionality[1]. The comparison of the scheduling alternatives' results illustrate that better performance could be achieved by leaving some key decisions to the operating system. Finally, the influence of task granularity on performance is briefly studied. Some modifications of the run-time system are proposed to make a program's performance more independent of some specific characteristics of the architecture.

4.1 Performance Scalability and Stability

Until recently, the necessity to predict the execution time of an application was essentially a feature of real-time systems. These systems, in turn, were associated with simple con-

[1] See Chapter 3.

trollers or elementary processors running similarly simple applications. Due to both the increasing performance of embedded systems, and the consumers' taste for rich and complex applications, many programs are now more complex, must achieve high performance and their performance must remain reasonably predictable. They range from soft real-time mobile applications such as GPS navigation software, to consumer electronics or desktop applications such as video compression-decompression, games, rendering software, or even scientific applications such as real-time finite-element modeling for engine control [80].

Two factors determine the predictability of a program execution time: The program workload, depending on the algorithm and the input data set and which is platform-independent, and the program behavior on the architecture. While the impact of the first factor is not trivial, one can correlate in many cases a program execution time on a single core to some restricted set of data characteristics or program parameters. However, the execution time variability on multi-core machines is significantly higher than on single-cores due to thread partitioning, balancing issues and contention. As a result, the second factor is becoming both important and tedious to address. Therefore, just as applications become more complex and the performance of an increasing span of applications must become predictable, the widespread use of multi-cores is making this task even harder. The purpose of this study is to focus on this second factor, and to show that, through a proper parallelization approach, it is possible to get parallel programs with fairly predictable throughput, with neither compromise on speedup nor ease of programming.

In real-time systems, performance prediction is today essentially based on a detailed knowledge of the underlying architecture, and the program behavior on this architecture. It is done using one or a combination of the following approaches:

1. Trying to predict the detailed performance of programs on architectures, such as Absint [228] for simple pipeline architectures and for superscalar architectures [178].

2. Changing the program so that its behavior is better understood and predictable, such as StreamIt [248] for stream processing applications.

3. Changing the architecture so that its behavior is more predictable, such as disabling the cache or replacing it with scratchpads [208], using software instead of hardware cache coherence, VLIW instead of out-of-order execution, etc.

In summary, most approaches rely either on analyzing in details program behavior on architectures or on simplifying programs or architectures. The former is increasingly difficult as architectures have become more complex. The advent of multi-core processors will make it even harder. The latter can have the effect of reducing program performance or limiting the scope of the approach to some domain-specific applications.

We hereby propose a fourth approach for achieving both easier to predict and high performance on multi-core/multi-thread architectures. For complex applications with irregular control flow and/or data structures, one of the main sources of performance variability on these architectures is simply that a variable number of cores are used throughout the program execution. Such poor load balancing can induce both irregular

and poor performance. Our approach is then based on a simple principle: Using real-time conditional task division, we can design parallel programs, and even ones with irregular behavior, so that they try to leave no processor cores unused at any time. That cores are used most of the time makes it easier to deduce performance based on a sample of the execution, for computationally-intensive pieces of code.

This approach requires no knowledge of the architecture, and thus no detailed analysis. It is not tied to any particular way of handling concurrent accesses, thus allowing programmers to use it in combination with, for example, syncrhonization mechanisms that avoid deadlocks, such as the transactional memory approach [124], which has become very popular recently. It can also adapt to varying architectures and scales well with the number of cores. It can be applied efficiently and independently to any computationally-intensive part of a program. Finally, it relies on a all-software implementation which is very efficient on a large range of existing processors and architectures.

The approach is based on the CAPSULE programming environment, which is described in Chapter 2, and a shared-memory implementation following the principles of Chapter 3. Like Cilk [34], or to a lesser extent Charm++ [150], CAPSULE implements parallelization through the intuitive operation of splitting/dividing an encapsulated task in two. The main difference is that CAPSULE's division is *conditional* upon available hardware resources. This distinguishing feature allows the run-time system to better match task granularity to available resources and to considerably reduce the overhead associated with very small tasks.

Using this dynamic parallelization approach, the execution time of irregular programs is both lower and much more stable than statically parallelized programs, as we will show in Section 4.1.1 and Section 4.1.4. As a result, these implementations lend themselves well to performance prediction, which can be achieved through iterative execution time sampling. A program is executed on one or several data sets. The resulting execution time statistics are then used to predict program throughput. This methodology is detailed in Section 4.1.3. Variabilility results as well as example performance predictions are presented in Section 4.1.4.

One can observe that the behavior of programs parallelized using conditional division is even more complex than the behavior of statically parallelized programs, since it depends on run-time conditions. However, we show that their performance is more stable and thus more predictable thanks to dynamic adaptation. This observation means that accepting to relax control and prediction of the detailed program behavior can paradoxically result into better prediction accuracy and improved performance.

4.1.1 Motivating Example

Consider the example of the `Quicksort` sorting algorithm, where an array is recursively split into two sub-arrays according to a pivot element. An intuitive but naive approach for a 2-way parallelization of `Quicksort` is to perform the recursive sorting on the first two sub-arrays concurrently. By repeatedly doing so for subsequent sub-arrays, on an N

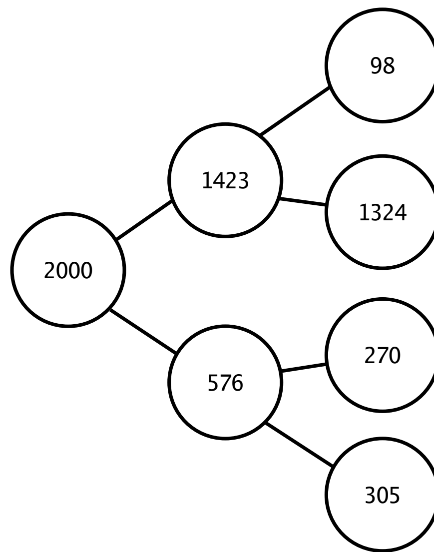

Figure 4.1: Static Parallelization of `Quicksort` on a 4-Core Computer.

cores machine, there are enough sub-arrays for each core after $\log_2(N)$ pivot steps, and all hardware resources are used at that point.

However, this parallelization is static, and the parallel program performance will be quite data set dependent because two sub-arrays can have very different sizes. As a result, the workload of each core can vary considerably from one input array to another. Some cores will finish their work sooner than others, and will be left idle. Figure 4.1 shows an example of workloads for a random array of 2000 elements on a 4 cores machine. The initial array is represented as the node on the left. A node's children represent the sub-arrays to be sorted independently on different cores after one pivot step. The number on a node indicates the sub-array size.

Figure 4.3 (page 46) shows the performance of static parallelization for 1000 random arrays of 1M elements under the "4 cores - static" name. One can observe that the variability of the execution time on multi-cores is much higher than that observed on single-core machines. In short, parallelization increases execution time variability, or conversely decreases program execution time predictability.

Now, let us assume the program is parallelized the same way, i.e., two sub-arrays are treated concurrently, except that this parallelization is done at every pivot step, regardless of the number of cores. The cons are that the task granularity will become exceedingly small after all possible pivot steps, with leaf sub-arrays of one element each, voiding the benefits of parallelization. The pros are that the potential number of tasks, and thus the degree of parallelism, is very large.

In order to get the best of both worlds, this systematic parallelization is in fact

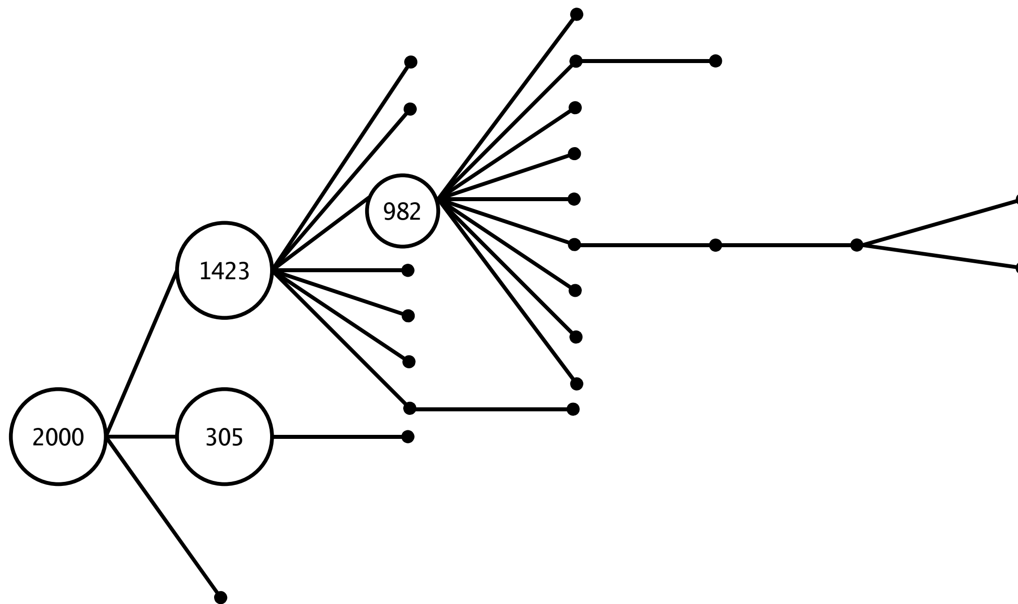

Figure 4.2: Dynamic Parallelization of `Quicksort` on a 4-Core Computer.

performed *conditionally* upon the available cores, according to the CAPSULE programming model and with the chosen run-time system implementation. At every pivot step, if a core is available, the `Quicksort` program will parallelize the treatment of the two resulting sub-arrays. Otherwise, it will sort them sequentially. As a result, cores will almost always be used, but the task granularity does not risk becoming too small, except towards the end of the execution.

The pattern of division depends both upon the data set and the cores' occupancy, possibly even by tasks from other processes. Figure 4.2 shows the execution of `Quicksort` parallelized that way for the same array as in Figure 4.1 (facing page). Each graph node indicates that a division occurred, i.e., one sub-array has been assigned to a core for execution. One can note that more divisions occur on the path with the highest workload (1423) because it takes advantage of the cores freed early by the paths with smaller workloads (305). The irregularity of the graph reflects the difficulty of predicting when each core will become available. Figure 4.3 (next page) shows the performance of the dynamic parallelization for the same data sets as for the static parallelization under the "4 cores - dynamic" name.

The key observations are that the performance of the CAPSULE-parallelized version is better than the statically parallelized version on average and that, over 1000 arrays, it is far more stable than the performance of the statically parallelized version. Let us pick 10 random arrays, compute the average execution time for both versions, and then compare it to the average execution time over 1000 arrays. The error for the statically parallelized version is 6.42%, while the error for the CAPSULE version is 1.21%. In fact, a sample of 3 arrays only is sufficient to achieve an error of 2% or less with the CAPSULE version versus

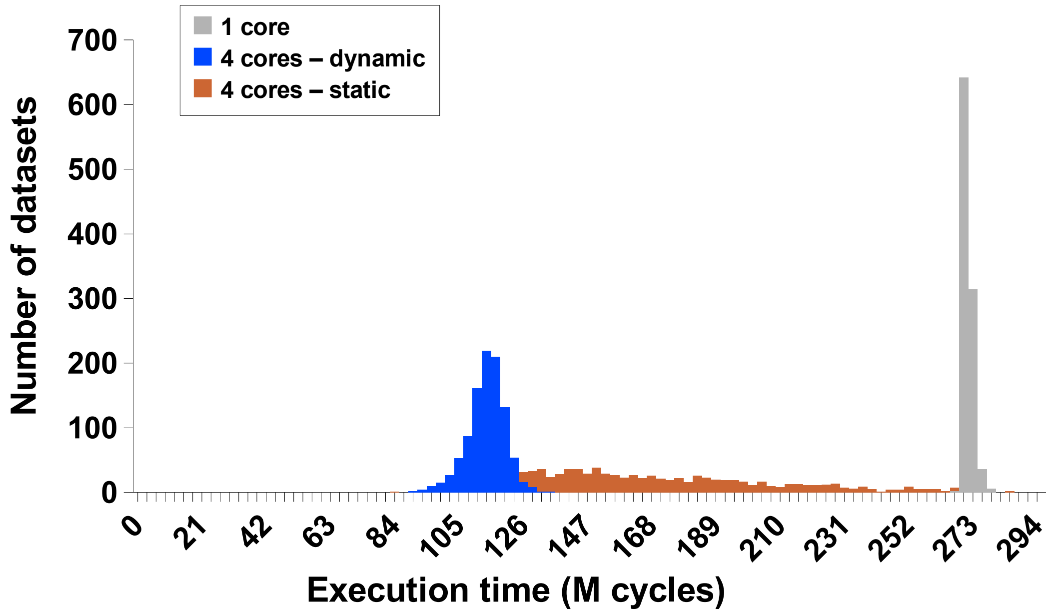

Figure 4.3: Performance of Static vs. Dynamic Parallelization of `Quicksort`.

100 arrays for the statically parallelized version, as shown in Figure 4.4 (facing page).

This example suggests that it is possible to precisely predict the behavior of irregular program parts using a combination of conditional parallelization and execution time sampling.

4.1.2 Benchmarks and Experimental Framework

Unlike for single-cores, there are no widely-accepted general parallel benchmark suites for multi-cores. The SPLASH benchmarks [236, 258] are mostly scientific computing benchmarks, which often have a regular behavior and are not always within the scope of real-time applications. Other benchmark suites, such as ALPBench [170], BioParallel [138] and MineBench [192], are domain-specific, with the first targeted at multimedia applications and the other two at data mining ones. Only recently did the PARSEC benchmark suite [30] appear, including state-of-the-art benchmarks that are more representative of the multi-core era and of modern applications, such as financial pricers, computer-assisted engineering, computer vision, physical simulation and data mining, although it still seems to be slightly biased towards scientific computing.

An influential survey on parallel computing and benchmarking [12] suggests the *dwarves* approach where, instead of large applications, the different characteristics of a parallel machine should be exercised with a mix of targeted benchmarks. To a large extent, we have been following that approach by progressively building a mix of kernels and larger applications, all parallelized using CAPSULE. This suite includes kernels and more

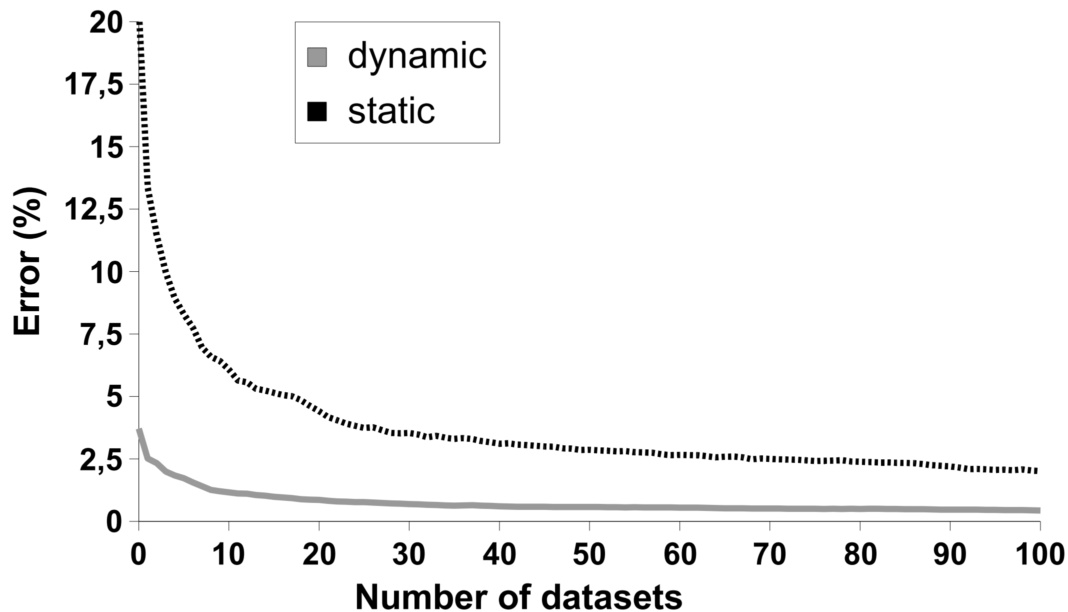

Figure 4.4: Predictability Error, Static vs. Dynamic.

complex applications, with both known regular or irregular behavior. The benchmark list is indicated below, together with a short description of the data sets used.

`MxV` The standard Matrix-Vector multiply kernel. The performance dependency of this algorithm on the input data set is obviously limited to the matrix size. 1000 matrices were randomly generated.

`SpMxV` A Sparse Matrix-Vector multiply kernel, where matrices are expressed in the Harwell-Boeing format. An increasing amount of real-time applications rely on physics modeling (games, engine control, etc.), which can rely upon sparse matrices. 300 matrices were collected from MatrixMarket, a web resource for test matrix collections. 200 matrices were randomly generated using a random sparse matrix generator called Matgen.

`Dijkstra` The graph shortest path algorithm used in network routing or for navigation purposes. 200 graphs were randomly generated with 1000 nodes and 2000 edges.

`Quicksort` The efficient and common sorting algorithm, tested with 1000 arrays of 1M elements.

`Watershed` An image segmentation program used for identification of image areas. The input images sizes ranged from 512×512 to 1536×1024 pixels.

`GZIP` The popular file compression utility. The storage on embedded devices being scarce, compression is commonly used for example on smartphones and PDAs. 500 files of

different content types (audio, video, image, office document, telecom, etc.) were used.

X264 A parallel implementation of an H264/MPEG4 encoder using slicing. Inputs were 134 clips of 100 frames with 640 × 360 pixels size.

The first 4 benchmarks were written from scratch using the CAPSULE API. The number of lines of code, for both sequential and CAPSULE versions, is several hundreds, depending on the benchmark. The `Watershed` code was provided by a company doing some image processing, and was substantially rewritten to exercise the dynamic parallelization of CAPSULE (∼50% more code lines). The `Dijkstra` algorithm was parallelized as described in [202].

The `GZIP` and `X264` benchmarks, by contrast, were converted to CAPSULE by a simple substitution of the POSIX threads calls by corresponding CAPSULE ones. When these original benchmarks are run, they initially create a number of threads corresponding to the number of available physical cores and dispatch fixed workloads to them. Introducing dynamic division would have required to change some of the main algorithms used, which would have led to a major and costly rewrite of these applications.

All experiments were conducted on a bi-dual-core machine equipped with 2 AMD Opteron dies and 4 gigabytes of RAM, half of them being attached to one of the two dies in a cache-coherent non-uniform memory architecture (ccNUMA) architecture. All the benchmarks and their corresponding data sets were run once per step. The first step is a simple run of the original sequential version, when possible, to serve as the reference point when computing speedups. The three other steps employ the benchmark version parallelized with CAPSULE and corresponds to the run-time system being allowed to use 1, 2 or the 4 cores available on the machine, respectively.

The CAPSULE run-time system has also been successfully ported to a 4-core ARM11 MPCore without operating system. Because the latter ARM platform allowed fewer test applications, we chose to perform experiments on the AMD platform.

4.1.3 Iterative Execution Sampling

The CAPSULE run-time system allows tasks to divide until all hardware threads are held busy. This greedy property effectively lets the program spread to all the silicon available, making its throughput much more stable. In the ideal case of programs whose tasks are independent, the total program throughput in the system is the sum of the throughputs of all available hardware threads, if we except the little amount of time spent in the division process. For such programs, performance also scales nearly linearly with the number of hardware threads.

For programs exhibiting contention, the situation is more complex. Giving a quantitative interpretation of the observed throughput can be extremely hard, since it may deviate from the ideal case because of the combined effects of the particular contention patterns, themselves dependent on the cache hierarchy characteristics and behavior. Fortunately,

this doesn't imply that the program throughput is not stable accross data sets. The contention patterns, however complex they may be, may only be a few. More generally, the run-time system's greedy property ensures that the same number of tasks are working concurrently most of the time. Compared to a situation where the number of concurrent tasks varies, this property diminishes the number of possible contention patterns and the execution variability, although the latter may stay at high levels. Contention degrades the attainable scalability from the ideal linear speedup, but only experiments can quantify the effect simply and accurately.

The more stable throughput of CAPSULE-parallelized programs can be leveraged for predicting program performance by sampling executions. For that purpose, we use principles similar to that of iterative compilation [53], which relies on multiple executions of the same program to fine-tune optimizations. In our case, the multiple training executions are used to estimate the average throughput and the throughput relative dispersion of a program across data sets, and then to forecast future execution throughputs.

Each training run, numbered i, uses one distinct input data set, and produces a throughput measure t_i. After n runs, the estimated throughput average $\widehat{\mu_n}$ is:

$$\widehat{\mu_n} = \frac{\sum_{i=1}^{n} t_i}{n}$$

The estimated relative dispersion sequence is, at this point, defined as:

$$\widehat{disp_n} = \frac{\sum_{i=1}^{n} |t_i - \widehat{\mu_n}|}{(n-1)\widehat{\mu_n}}$$

The training phase lasts until the estimated average and dispersion both converge to the actual average and dispersion values. Taking the example of the estimated average, it is useful to introduce the sequence of relative differences of consecutive estimated averages, i.e., at step i, the value:

$$\frac{|\widehat{\mu_{i+1}} - \widehat{\mu_i}|}{\widehat{\mu_i}}$$

since this sequence converges towards 0.

The operator has to stop the process when the estimators have reached the desired precision. Traditionally, the latter is measured thanks to a reference distribution that depends on some assumptions on the distribution of experimental results. The reference distribution allows to derive confidence intervals for a chosen probability of error. The well-known Student distribution [112] is typically used to estimate the mean value of a normal distribution of unknown variance. For our experiments, we have chosen a much simpler empirical approach: We ended the training phase when 5 elements in a row were below a 1% threshold for both relative difference sequences (estimated averages and estimated relative dispersions).

For programs where single-core execution time variability depends on known data set characteristics, e.g., the array size for `Quicksort` or the matrix dimension for `MxV` and

SpMxV, it is possible to predict parallel execution time, not just throughput, assuming all training and target data sets have the same characteristics.

When single-core execution time is a complex function of data sets or if the target data sets is heterogeneous, predictions can usually still be achieved if a meaningful statistical model of execution can be built from experiments. This model usually involves analyzing program characteristics during executions such as the instruction mix, the branch prediction accuracy, cache misses and instruction-level parallelism. The behavior of a particular program-data set pair is closely correlated to some combination of the characteristics that can be determined automatically using principal component analysis and/or clustering [85].

In all cases, the measurements required to do the predictions can be done not only off-line, by repeating execution of a given program on different data sets, but also on-line, if the same piece of code is called several times in turn in the same program, allowing more and more accurate prediction of remaining or future execution times as the program keeps running. In the following Section, we only consider off-line predictions, since on-line ones can be done similarly.

4.1.4 Experimental Results

Performance and Scalability of Division-Based Parallelization

Figure 4.5 (facing page) illustrates the speedups and scalability achieved by the CAPSULE program versions, under the label "dynamic". Even irregular programs, which are typically difficult to parallelize, such as Quicksort or Watershed, can take full advantage of an increasing number of cores.

We observe nearly perfect speedups for 2 cores, with all speedups being above 1.8, except for MxV (1.6) and Dijkstra (0.75). The best speedups are 1.92 for Quicksort, 1.95 for GZIP and 2.23 for X264, the latter being super-linear because of one core warming up some cache for the other core. Dijkstra's performance is highly dependent on the characteristics of the graph it processes but also on run-time conditions, including the number of cores used. With this algorithm, the total workload for a graph indeed depends on the exploration paths followed by the different tasks. Tasks crossing nodes with a lower annotated current distance will stop and those that work on a path with small distances will spawn more tasks to explore nodes from it. Consequently, if a task soon follows a path with small distances, most tasks following other paths will stop early. Since tasks are created based on run-time conditions, namely the availability of some cores, Dijkstra's behavior and performance thus depends on the number of cores and on the task scheduling.

Going to 4 cores, most speedups are around or above 3, except for Dijkstra and Quicksort (1.58 and 2.43 respectively). The latter is penalized by some tasks being created that span too a small input sub-array. Results can be greatly improved if no divisions are requested for sub-arrays under a given threshold. As an example, setting the threshold to 100 elements results in a new speedup of 2.97. This effect will specifically be

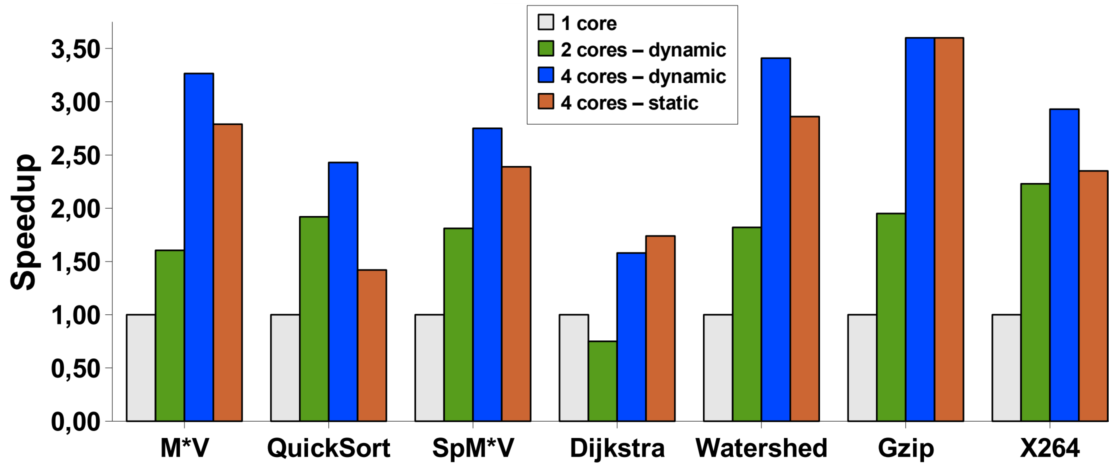

Figure 4.5: Speedup and Scalability.

studied in Section 4.2.3. `SpMxV`'s speedup suffers from memory bandwidth limitation. The Capsule version still performs significantly better than the static one. The performance of large benchmarks, such as `Watershed` (2.86) and `GZIP` (3.6), is excellent.

As can be seen on Figure 4.5, the performance of the programs' Capsule versions are better than those of the naive (static) versions. The difference is particularly important for irregular benchmarks, such as `Quicksort`, `Watershed` and `SpMxV`. `Dijkstra` is an exception here, for reasons explained earlier. `X264` has its performance increased when switching to Capsule, although it doesn't take advantage of conditional parallelization. The reason is that the version we used constantly destroys and recreates threads to process the slices of each frame. The Capsule version creates tasks to reproduce this behavior, but not threads, which are automatically recycled by the run-time system in a pool.

Performance Variability of Parallel Programs

Figure 4.6 (next page) compares the execution time variability of both static and dynamic parallelization. The variability is measured through the relative dispersion, as defined in Section 4.1.3, across all data sets.

The `MxV` and `GZIP` benchmarks exhibit a fairly stable throughput, whereas `Dijkstra`, `Quicksort`, `SpMxV`, `Watershed` and `X264` have an irregular behavior across data sets. One can observe that, for the latter ones, the variability of execution time on multi-core machines is much higher than on single-cores. `X264` is an exception. Its performance also varies a lot on single-cores because frame encoding is dependant on the encoded image's autocorrelation.

Our approach reduces variability significantly and consistently, although it usually remains greater than that observed on single-cores. Together with the increase in performance brought by Capsule, this feature opens up the possibility of trading off performance

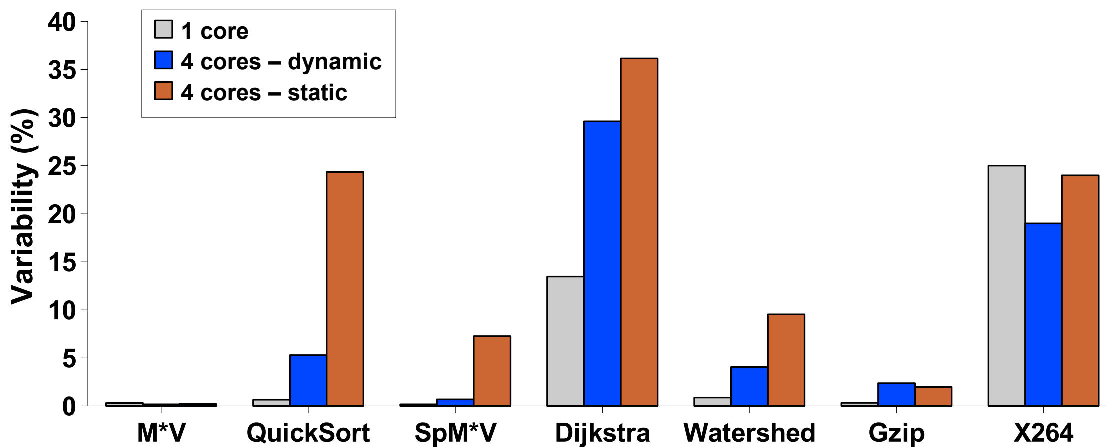

Figure 4.6: Execution Time Variability, Static vs. Dynamic.

variability versus execution time more efficiently. For a given number of cores, the product of both metrics is indeed lower for the CAPSULE-parallelized version.

The reason why GZIP and X264 behave similarly for both parallel versions is due to our implementations, which don't allow them to really benefit from the dynamic division scheme, as explained in Section 4.1.2. Only GZIP performs slightly better in its static version. The difference accounts for the hardly higher overhead of CAPSULE compared to POSIX threads.

Predicting Program Performance

We now illustrate that the greater stability of dynamic division enables a rather accurate estimate of program performance by sampling a few data sets and using the iterative technique proposed in Section 4.1.3.

Figure 4.7 (facing page) gives the number of data sets for which the mean execution time estimation is likely to differ by less than 5% from the real mean execution time. It shows that, for the irregular benchmarks, dynamic division can reach a given mean estimation error with fewer runs.

Figure 4.8 (facing page) illustrates the mean estimation error when the prediction is based on a fixed number of data sets. It shows that the lower dispersion of dynamic division results in a more accurate prediction using the same number of runs.

Dynamic versions bring a huge improvement for benchmarks that exhibit a high variability. The numbers for X264 and GZIP are almost identical for both versions, since the conditional division mechanism is not used. Results for Dijkstra are less spectacular because of the algorithm's peculiarities, but are still significant.

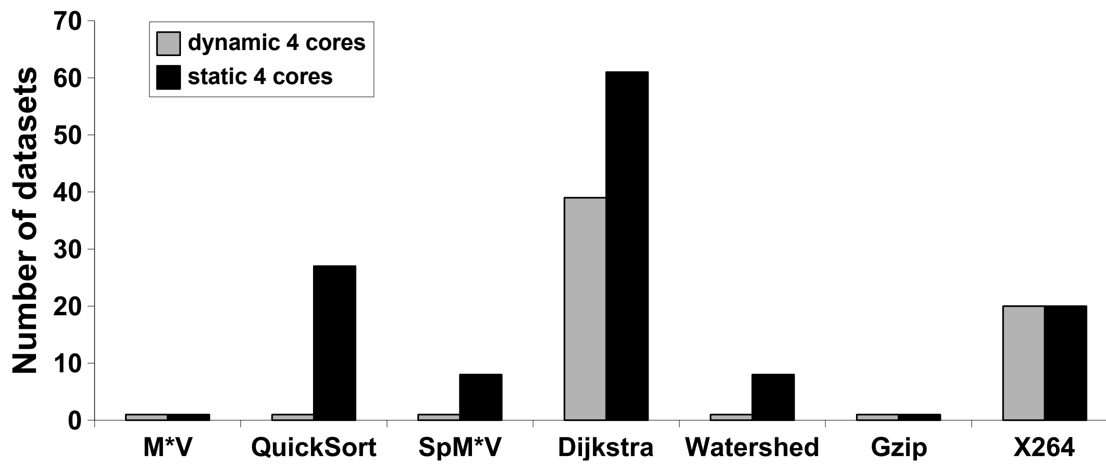

Figure 4.7: Number of Data Sets to Reach a Mean Estimation Error of 5%.

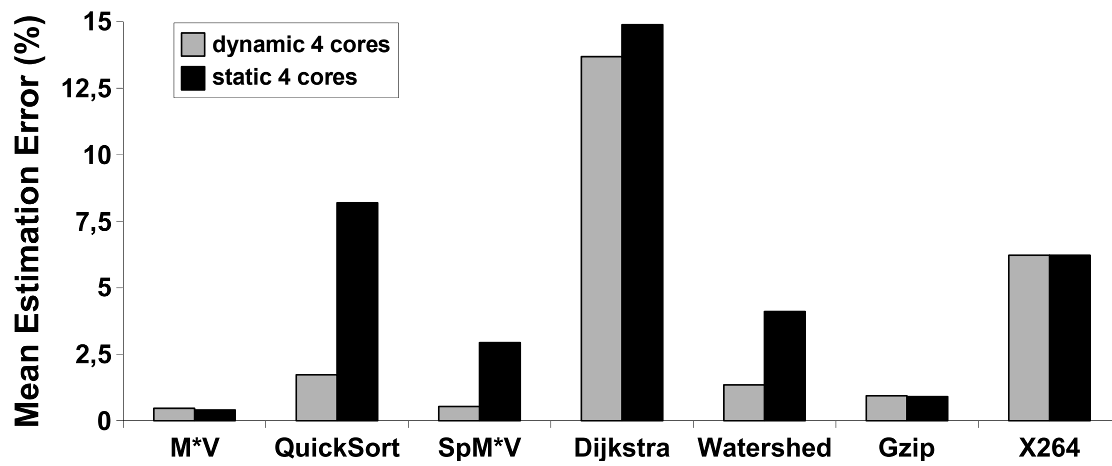

Figure 4.8: Mean Estimation Error After 10 Runs.

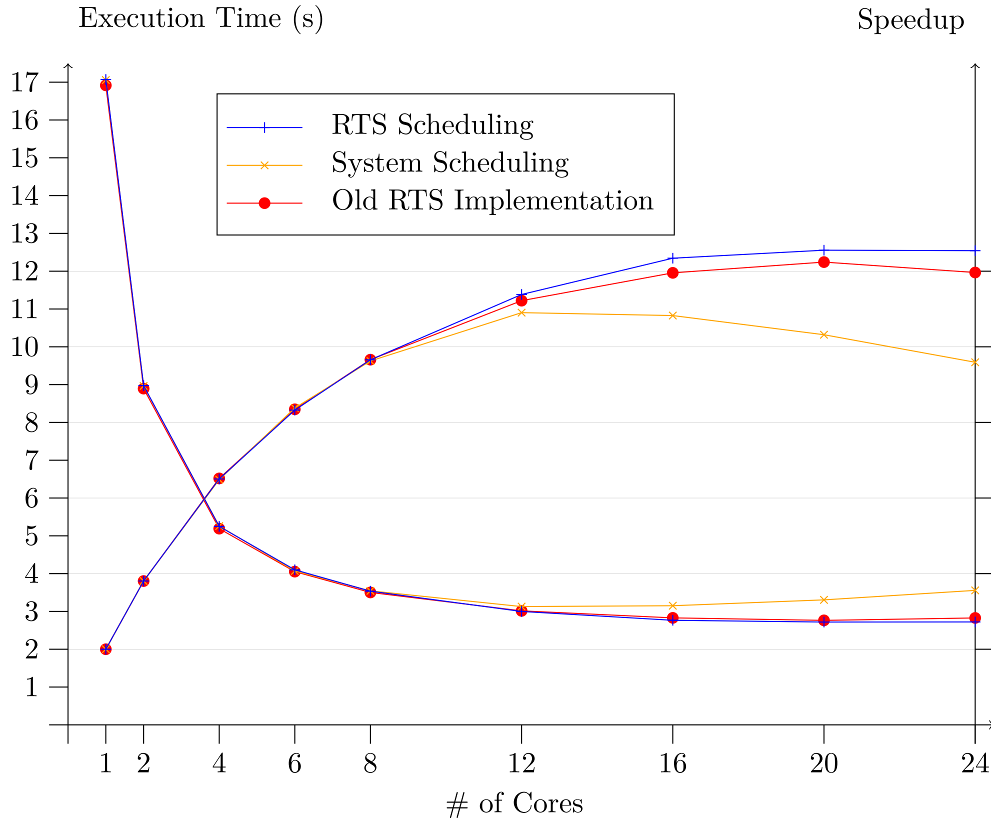

Figure 4.9: `Quicksort` Performance and Scalability For Different Implementations.

4.2 Performance Dependence on the Run-Time Platform

This Section briefly studies some factors by which the run-time platform may influence the attainable performance of CAPSULE programs. Scalability results of an example benchmark are presented on a 24 cores machine.

Section 4.2.1 compares the performance of the old run-time system implementation with the one and two of its scheduling alternatives. Section 4.2.3 shows that performance can vary significantly with task granularity, which indicates a dependency towards the cost of a division on the experimental platform. Some solutions are proposed to mitigate this dependency.

4.2.1 Run-Time System Implementation

Figure 4.9 presents the performance of the `Quicksort` benchmark on 100 random arrays of 10^8 elements run on the old run-time system implementation[2] and on the new run-time system implementation with two alternative scheduling strategies. The first strategy is

[2]See the introduction of Chapter 3.

that the run-time system explicitly selects the thread that last became idle to process a new task, whereas in the second scheduling is left to the OS and the threading library[3]. The first group of globally decreasing curves represent the real execution times for all three variants. The second group, with increasing curves, represent the associated speedups over an execution of each particular variant on a single core. The execution time scale is reported on the left of the graph and the speedup scale on the right.

The machine used for these experiments is a quad Intel Xeon E7450[4] at 2.40GHz, each die comprising 6 cores. It runs a Mandriva OS with a Linux kernel based on 2.6.29. The employed `Quicksort` implementation slightly differs[5] from the one used for the experiments in Section 4.1.4. The main difference is that it doesn't try to divide sub-arrays whose number of elements is smaller than 1000. The rationale for this choice will become apparent below. It would be necessary to run a thorough benchmarking campaign involving other programs to be able to draw quantitatively meaningful conclusions from the comparison. Still, the `Quicksort` graph shows interesting trends that we believe are general.

The different implementations and scheduling methods do not seem to make any significant differences for a low number of cores and when the available parallelism is large. After the i-th pivot step, `Quicksort` tries to generate 2^{i-1} new tasks that different cores/processors can handle independently, provided the sort is relatively balanced and $n \geq 2^i$, where n is the array size. With $n = 10^8$, $\log_2(n) \approx 26.58$, which implies the generation of up to 13 tasks for a given pivot step. This is effectively enough to saturate 1 to around 4 to 8 cores most of the time.

During periods where most cores are busy and probes are frequent, threads finishing a task do not stay idle for long and it is likely that only a few of them are idle concurrently, unless tasks being dispatched are extremely small. This latter possibility has been largely ruled out in this Section's experiments by using the 1000 threshold[6]. For the variant where scheduling is deferred to the OS, this property limits the possible contention on conditional variables and on the single slot the run-time system currently uses to pass a new task to an idle thread.

As the number of core grows beyond 4 to 8, `Quicksort` does not generate enough tasks to maintain all cores busy most of the time, especially if the input array does not produce well-balanced sub-arrays. The amount of time during which the previous property is verified shrinks, causing more contention for the system scheduling variant. Experimentally, this effect shows up starting between 8 to 12 cores: The system scheduling variant does not perform as well as the other variants for 12 cores. Worse, its performance constantly decreases for each measurement between 12 and 24 cores. For the other variants, the performance reaches a plateau at 16 cores, with a slight degradation for the old run-time system implementation.

[3]See Section 3.1 for more details.

[4]This is a Dunnington processor, based on the Penryn microarchitecture, an evolution of Core 2.

[5]The pivot step has been optimized, resulting in a 10% performance improvement over sequential runs (code available in Appendix A). Some threshold prevents the creation of task with small granularity, as detailed in the main text.

[6]Section 4.2.3 demonstrates this claim.

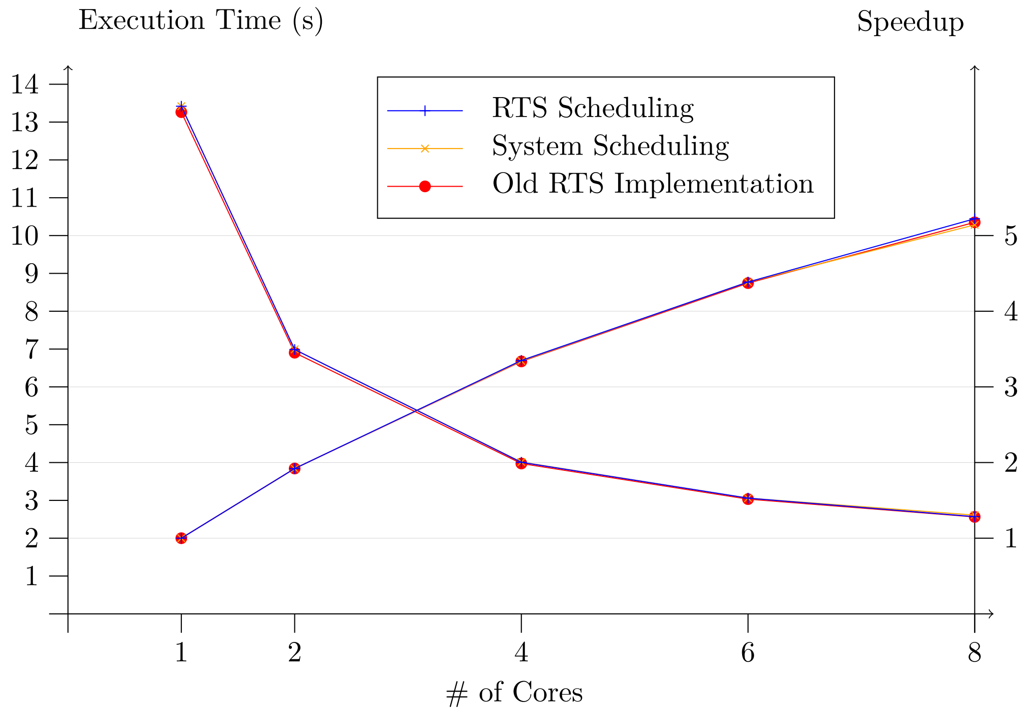

Figure 4.10: `Quicksort` Performance and Scalability on an 8-Core Machine.

For the system scheduling variant, these results demonstrate the need to implement a more sophisticated mechanism than the naive implementation consisting of a single slot shared by all threads to pass new tasks to idle threads. Only with such a mechanism can the results become competitive with the simple scheduling currently performed by the run-time system.

The potential advantage of relying on system components for task scheduling is that these components have a view of the system hardware topology and the threads currently being scheduled. For example, they could use this information to select the thread that is expected to have the lowest wake-up latency, which may be one thread not too far away from the parent task or one that has waited a long time and thus has a higher scheduling priority. Potential benefits may appear only as a larger number of cores are used and when a program is run on a heavy-loaded machine. Section 4.2.3 below discusses other potential improvements, that ultimately have lead us to the distributed implementation presented in Part II.

4.2.2 Hardware Architecture

This Section is a brief study of the influence of the particular architecture on scalability results obtainable with CAPSULE.

Figure 4.10 presents the performance and scalability of `Quicksort` on a different

Platform	Nb of Cores	Microarchitecture	Kernel
Intel Core 2 Duo T7400	2	Merom	F 7.0
2 × Intel Xeon E5430	2×4	Harpertown	L 2.6.29
4 × Intel Xeon E7450	4×6	Dunnington	L 2.6.29

Figure 4.11: Platforms Used for the Micro-Benchmarks.

Nb of Cores	Mean	Standard Deviation	Nb of Samples
2	11921	1442	95
2×4	40250	13437	100
4×6	39922	12722	100

Figure 4.12: Number of Cycles Between a Divide and the Start of a New Task.

machine with two Intel Xeon E5430 processors each composed of two dies with 2 cores. The OS is identical as that mentioned in Section 4.2.1 for the first machine.

The results show that all variants perform similarly up to 8 cores, as in Section 4.2.1. Interestingly, the obtained scalability for 8 cores on this machine (5.22) is higher than that for the 24-core machine (4.83). This difference seems to come from the different core-to-core and core-to-memory latencies and bandwidths of the two platforms.

Figure 4.12 shows the number of cycles[7] elapsed between a divide and the actual start of the new task. The architectures used are reported in Figure 4.11. They include the two that were used to obtain the `Quicksort` results in the previous Section. Measurements were performed by having the initial task probing and launching another task repeatedly. For OSes, L stands for Linux and F for FreeBSD (-STABLE branch). A unit in the number of samples represents 10,000 tests.

The reported time to start a task depends on the architecture as well as on the operating system. The results show that the numbers are very close for the 8-core and 24-core machines, which have processors with similar microarchitecture and the same OS. On the Core 2 Duo, starting a task is nearly 4 times faster, as could be expected. The main reason is the low number of cores and the fact that they are on the same die. As a consequence, the run-time system efficiency is increased. The order of magnitude for all these results is 10,000 cycles. No effect on applications' speedup should therefore be noticeable, except for programs having a critical path composed of many tasks whose individual length is not at least two orders of magnitude greater than 10,000 cycles on average. In the latter case, execution of the critical path will be visibly delayed by this overhead.

Figure 4.13 (following page) presents the duration of a successful probe and divide sequence on the same machines. This duration is the overhead penalty a task suffers

[7] This number has been computed for the variant in which the run-time system selects the thread to handle the new task. When task scheduling is deferred to the system, this overhead is cut approximately by two.

Nb of Cores	Mean	Standard Deviation	Nb of Samples
2	6594	715	95
2×4	15260	4721	100
4×6	14780	4184	100

Figure 4.13: Number of Cycles Spent in a Probe and Divide Sequence.

when spawning another task, including the time to signal some idle thread to handle the new task. Probe and divide also involve simple synchronization operations to update the thread counter. A non-negligible amount of time is dedicated to initializing execution context structures[8]. The results show that the dual-core has much less overhead than the machines with 8 and 24 cores, which is essentially due to the synchronization costs. The overhead is 2 to 3 times less than the interval of time between a divide and the actual task start. We discuss its impact on scalability at the end of Section 4.2.3.

4.2.3 Task Granularity and Other Program Characteristics

In this Section, we provide a study on the influence of task granularity on program's performance. It uses the same `Quicksort` implementation and experimental conditions as in Section 4.2.1. The threshold on the array size is implemented as a simple test inserted before the probe done at the end of each pivot step. It is passed as a parameter when launching the `Quicksort` process.

Figure 4.14 (facing page) presents the performance and scalability results for a variety of thresholds, from 10 to 10^5, on the same 24-core machine as in the previous Section. The convex curves are the execution time ones and the concave curves those reporting speedup. When no thresholds are used (0), scalability peaks at 3.39 for 6 cores and then considerably degrades to the point that, when using 24 cores, the program is faster by only 7% with respect to the single-core run! The trend is the same for a threshold of 10. The scalability peak begins to move to the right with a threshold of 100, where it is around 16 cores with a 5.03 speedup. For a threshold of 1,000, there is no degradation any more. The scalability climbs to 6.27, which is almost reached from 16 cores. It still improves when switching to 10,000 (6.66). Finally, it is 6.82 for 100,000, with that last step showing diminishing returns. The slight shift between the performance curves for 1 core reflects the difference in the number of probes issued and the common probe failure overhead.

Clearly, the problem of task granularity can ruin the scalability of a CAPSULE-parallelized program. To avoid it, programmers could estimate, from the particular algorithm used, how much work remains to be done by a task and, from this, decide whether it is worthwhile to try to divide and create another task. This is exactly the approach we took for `Quicksort`.

[8]Some of these operations could be optimized by reducing the initialization to the essential structure fields instead of zeroing the whole structure.

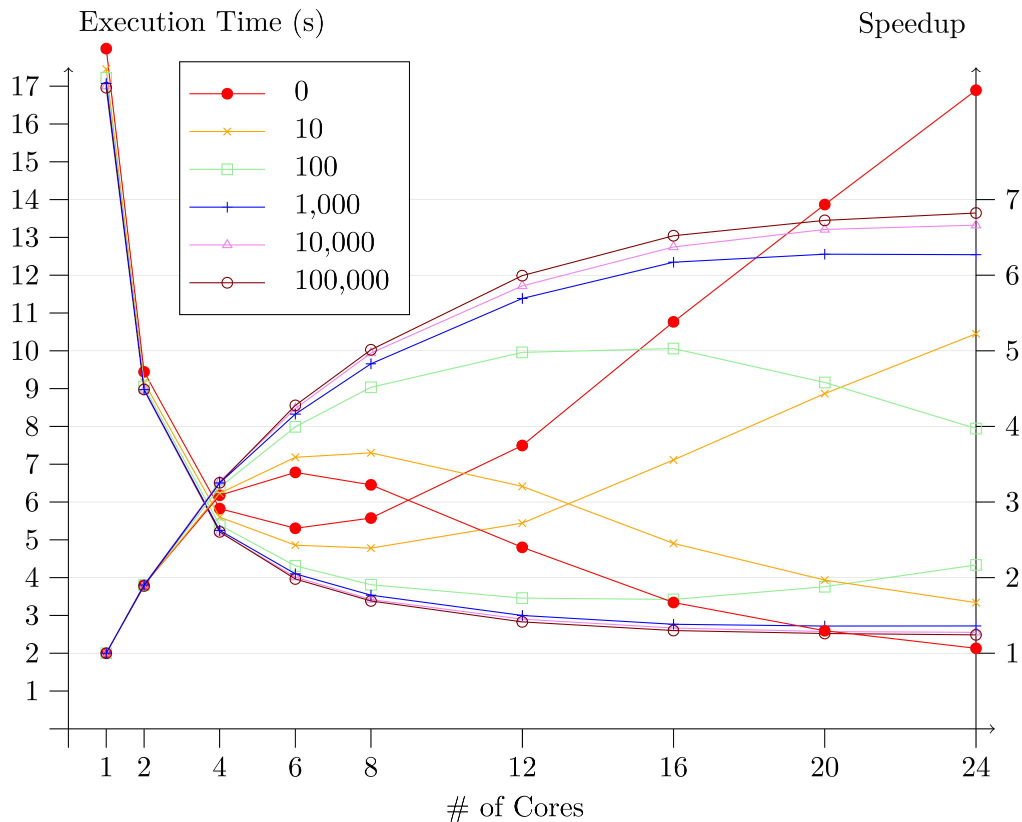

Figure 4.14: `Quicksort` Performance With Several Task Thresholds.

Unfortunately, this approach has several drawbacks. First, the amount of work at a given point in an algorithm may not be known with certainty. In `Dijkstra`, for example, the workload highly depends on the graph structure and the weight attached to its edges. As the algorithm runs, the remaining work also depends on the browsed paths so far, and thus ultimately on run-time conditions. The situation is even worse for complex programs involving lots of components. Expecting a programmer to indicate the work amount together with a task declaration is unrealistic[9], not even mentioning the programming effort involved, except in simple controllable cases.

Second, spawning a particular task may be beneficial on a given architecture, whereas it may degrade performance on another one. The most obvious trade-off depends on the total cost of probing and spawning a new task, from the father task's point of view. If a task requires more execution time to spawn a new task than to execute the corresponding work sequentially, dividing is obviously a performance loss. This may happen for very

[9]Some researchers may be able to produce (semi-)automatic methods in some cases, such as the ones evoked in Section 4.1.3. However, we have not seen any method that we believe will ever be lightweight enough to be applied at run time, which is necessary when workload evaluation for a program largely relies on input data characteristics not known at compile time.

small tasks spawned within innermost loops or on very small data structures, such as a small sub-array for `Quicksort`. Programs with mostly mid-sized or coarse-grain tasks do not run into this problem, since the division cost observed on several recent architectures is around 10,000 cycles[10]. When they do, they would temporarily throttle the spawning of other such tasks by the simple fact that they occupy some hardware resource.

The essential difficulty is that this trade-off is machine-dependent. In practice, this would mean that a programmer would have to change the threshold to run a program efficiently on a different architecture. Although this is mitigated by the fact that, as can be seen on Figure 4.14 (previous page), there may be a relatively large range of threshold values for which scalability is good and doesn't vary much (1,000 to 100,000 for `Quicksort` on this particular machine[11]), we think that more of the burden should and could be handled by the run-time system.

To this end, we propose some possible mechanisms that should be evaluated as a future work. The first mechanism, that was exposed in [202] for the hardware implementation of CAPSULE, is to monitor the rate at which created tasks terminate. If this rate is high, it is likely that lots of small task are being created. The hardware consequently blocks divisions during a small amount of time when the rate is higher than some threshold[12]. This approach has the advantage that it is completely application-independent and rather simple to implement. Unfortunately, it is rather radical and may also block long tasks that happen to compete for task creation with very small tasks. This is the reason why it is not beneficial for all algorithms[13].

A more focused approach is to try to optimize the efficiency of a division, which can be measured as the ratio of the amount of time spent working on a user task over the amount of time that was lost by the father task in the division process. The machine-dependent trade-off mentioned above can be expressed easily in terms of this division's efficiency ratio: It has to be greater than 1 so as not to degrade performance. Going further, the implementation should try to maximize this ratio, which once again leads us to the problem of evaluating the workload of a given task. An approximation is to use the task's depth in the task tree as an estimation of the task's size. It is used, for example, by Cilk when it has to decide which task to steal, in an attempt to steal larger tasks preferentially [36, 37] [14]. Some priority could be given to tasks closed to the root of the task tree by reserving some hardware resources to them.

As Figure 4.14 (preceding page) shows, a growing number of cores leads to even more performance degradation, because more small tasks can be spawned and dispatched at once. Following up on the previous idea, it would be beneficial to limit the number of child tasks a single task can spawn to some threshold smaller than the total number of all available hardware contexts. In the same vein, the run-time system could monitor the life

[10] See Section 4.2.2 and Figure 4.13 (page 58).

[11] Although not shown here, results are very similar on other machines we have access to.

[12] In [202], the threshold was $n/2$ terminations during the last 128 cycles, where n is the number of hardware contexts.

[13] [202] shows that such benchmarks as `Perceptron` or `LZW` benefit from this approach, whereas it achieves only mixed results for `Quicksort`.

[14] A thorough description of Cilk is given in Section 5.2.2.

time of some tasks and throttle the next tasks to be spawned by the same father based on these statistics[15]. Currently, the CAPSULE run-time system doesn't keep track of child to parent relationships; it would have to be adapted to do so.

Although not its primary goal, the work presented in Chapter 9 alleviates part of the shortcomings presented here by providing a scheme that allows load-balancing to be performed over a very large number of cores, while largely limiting the tasks that can be spawned by a single core on common 2D meshes networks.

[15]Section 5.2.2 mentions a recently-proposed approach following a similar principle.

Chapter 5

Related Work

5.1 Data Parallel Environments

5.1.1 Languages

High Performance Fortran (HPF) [155] is an evolution of Fortran 90 to add support for parallel loops, through the introduction of the `DOPARALLEL` statement and the `INDEPENDENT` directive. It assumes a SPMD (Single Program, Multiple Data) model: The same program is executed on all processors. Particular processors can alter the control flow and execute different pieces of code by testing their processor number.

Split-C [158] is a set of minimalist modifications of the C language for parallel programming and follows the SPMD model as well. However, the design intent is to stay close to the computer: Most of the semantics are provided by the programmer. Thus, he can precisely specify which kind of parallelism to exploit, without relying on the compiler or run-time system's ability to discover independent tasks or optimize specific patterns of communications. The compiler is confined to the usual code generation task.

Addressing is global, but there is a distinction between local and remote memory areas. A new kind of pointers, named *global pointers*, is added to the C language to allow to reference objects potentially stored in another processor's local memory area. Global pointers support the same operations (dereference, arithmetic) as regular pointers, which point to local objects only. Distributed memory is supported as well, as described in Chapter 10.

Unfortunately, only global barriers are available for coarse-grain synchronization, which actually limits the parallelism that can be expressed. Split-C, which hasn't evolved since its inception, is not usable as is for complex applications that exercize simultaneous algorithms or phases using several of these operations at the same time. Some of its ideas are however reusable, particularly in terms of data management, as will be discussed in Chapter 10.

5.1.2 STAPL

The Standard Template Adaptive Parallel Library (STAPL) [10, 211] is a C++ library whose aim is to provide easy to use parallel implementations of standard containers and algorithms. It is based upon and compatible with the C++ Standard Template Library (STL) [240], a library whose paradigm is to separate data structure implementations from that of algorithms. This approach has the benefit of providing a large body of common code operating on a wide class of structures through genericity (concretely, C++ templates). It supposedly increases programmer's productivity thanks to code uniformity and reuse. STL's major components are the following:

- Containers, data structures that hold data, such as lists, queues, arrays or hash maps.
- Algorithms, such as sort, search, insertion or comparison of elements or structures of them.
- Iterators, that bridge containers and algorithms. Iterators are an abstraction of regular C++ pointers. They yield a container's element when they are dereferenced[1]. Several methods on a container produce an iterator with which one can browse the container's elements. Depending on the iterator class (input, output, forward, bi-directional or random access), the pattern of accesses is more or less constrained. Algorithms are implemented only in terms of iterator classes and the methods they provide, independently of the implementation of a container and its associated iterators.

STAPL tries to preserve the same philosophy and approach while introducing parallel processing. It keeps the previously mentioned components, except that it substitutes *ranges* for iterators. STAPL's central assumption is that parallel processing requires that elements of a container be accessed randomly to a certain extent. Ranges are the incarnation of this assumption. They represent subsets of elements that can be decomposed into smaller ranges, with the latter being randomly accessible. This definition is recursive until indivisible ranges are reached, which is determined through a selectable strategy, often a user-specified compile-time constant indicating a range's smallest possible size. A parallel algorithm operates on a range and distributes a partition of the sub-ranges to the available processors. The recursive structure of ranges allows the automatic production of recursive parallelism.

In order to avoid computing ranges before each parallel algorithm application, an inherently sequential computation, STAPL parallel containers, such as lists, internally maintain a range object at all time. As elements are added to or removed from a container, its range is updated by modifying the corresponding sub-ranges. After a number of changes, the range's partition into sub-ranges may become unbalanced, which has a performance impact on parallel algorithms because work distribution will not be even.

[1] Provided they are valid. Invalid iterators sometimes serve to mark the end of a structure or the end of a browse of a structure.

For this reason, when the number of changes exceeds a certain threshold, a container computes a new range structure.

Another feature of STAPL, inspired from other more specific frameworks like Spiral [260] or FFTW [97], is to adaptively select the implementation with best performance for a given algorithm class. At library installation, a set of synthetic test programs are run on several input data sets with different characteristics. This phase serves to establish a mapping between data attributes, such as data size, correlation, etc., and implementation parameters to the observed execution time. The map is specific to the architecture that executes the tests. At run-time, simple tests and/or lightweight instrumentation quickly determine the data attributes and, thanks to the map, the algorithm and associated parameters that will yield the best performance. This framework has been applied to the parallel reduction problem [264], and to integer sorting and matrix multiplication [249]. Each problem requires a specific combination of appropriate data attributes and mapping interpolation.

In a recent STAPL evolution [47], algorithms now operate on views, a concept that permits a container to present different interfaces to the program using it. As an example, a matrix can present an interface simulating row-major storage, another one corresponding to a column-major organization and a third one that linearizes it into a vector. Ranges keep their name but are considerably generalized as task graphs whose edges are the task dependencies. A task is composed of a work function and a container. The work function receives a view of the associated container. The run-time system may substitute the view used when calling a high-level function with another one for performance reasons.

Although STAPL is well-adapted for programs using traditional algorithms on common data structures, it doesn't seem to suit batch parallelism that spans several data structures. In this case, it is necessary to code a specialized container to parallelize the computation. Experiments with STAPL exhibit good scalability results for simple operations like sorting, finding an element or computing an inner product. It is unclear how much performance can be gained by replacing STL calls with that of STAPL in complex programs.

5.2 Asynchronous Function Calls

5.2.1 Cool

Cool [57] is an extension of C++ providing asynchronous function calls. Asynchronous functions are syntactically distinguished from regular ones by the use of the `parallel` keyword. Mutual exclusion is provided through an improved monitor paradigm. Regular monitors [126] allow at most one method of an object to execute at a given time. In Cool, methods of an object can be tagged with the `mutex` or `nomutex` keywords, depending on whether they need exclusive access to the object. Several methods tagged with `nomutex` can concurrently operate on the object. `mutex` methods are exclusive with respect to all other tagged methods. The semantic is thus similar to that of a read/write lock. Non-tagged methods simply ignore the object's associated mutex. They can always execute, regardless of any tagged methods executing concurrently. Initially, Cool provided *futures*, variables

that get the corresponding asynchronous function calls's return values when they complete. They were removed to avoid complex corner cases. Instead, a parallel function invocation returns a condition variable on which the caller can wait.

Cool also introduces the `waitfor` construct, by which some piece of code inside a function can wait for the completion of all parallel function calls started inside it. `waitfor` blocks can be nested, but they can't span several functions, because they are syntactical construcs. CAPSULE groups, in contrast, are used through function calls and are thus not restricted to particular language scopes. Nested groups are more efficient than nested `waitfor` blocks. In addition, every task in a group, i.e., not only the initial task, can wait for the other tasks to complete, which has several advantages, as was pointed out in Section 2.4.

5.2.2 Cilk

Cilk is a parallel extension to the C language enabling asynchronous function calls, called tasks[2].

In its original form [34], Cilk allowed child tasks to communicate their results to the parent task's successor through explicit continuation passing. Programming was later facilitated by the introduction of call-return semantics in Cilk-2 [140], with the `spawn` keyword, creating new child tasks, and the `sync` keyword, waiting for child tasks to complete. Data still had to be passed explicitly between procedures, which was both impractical and slow for large shared structures. Cilk-3 [140, 210] thus introduced a software implemented shared-memory system with a relaxed consistency model, which we discuss in Section 10.1.2. Cilk-4 [140] added *inlets*, small procedures that are automatically invoked as each child task returns, in order to process its result as soon as possible, without waiting for the other tasks. It also introduced `abort` primitives to stop the execution of all current children of a procedure, enabling proper coding of speculative computations. However, this later feature was rather immature, since some important task abortion corner cases, such as memory deallocation, were not dealt with.

With Cilk, tasks are scheduled by a hierarchical randomized work stealing scheduler [36, 37]. Each processor maintains a set of runnable tasks from which, if not empty, it selects the next task to run. When there are no more local tasks, the processor randomly chooses a *victim* processor and requests a task from it. If the victim currently holds no tasks, another victim is chosen. Inside the runnable tasks set, tasks are sorted by their level, which corresponds to their depth in the spawned tasks tree. All child tasks of a level n task thus have level $n + 1$.

When selecting a local task to run, a processor first consider tasks with the highest level. On the other hand, when stealing tasks from another processor, it first considers tasks that have the lowest level. The rationale for the first rule is to immediately process new children in order to minimize the number of alive ready tasks on a given processor, which reduces space consumption, and to increase locality if most child tasks work on

[2] The original papers use the *thread* term, but this usage collides with the common definition of thread, so we replace it by a more appropriate term in this description.

subsets of the seed task's data, which is typically the case for most divide-and-conquer algorithms. The rationale for the second rule is to preferentially steal larger amounts of work in order to minimize the number of steals and amortize their cost. It is indeed very likely that there will be many more tasks with a high level than low level ones which represent larger program portions and consequently more work to do.

In the above-mentioned articles, it is shown that this on-line scheduling algorithm produces a parallel execution time and a maximal amount of space consumed by alive tasks that are within a constant factor to the optimal bounds, provided that the considered computations are *fully strict*. Strict computations assume that dependencies between tasks always go from a task to one of its descendants in the spawning tree. Fully strict computations narrows this class to computations in which tasks can depend on some value produced by their direct children only. The latter class typically includes simple divide-and-conquer or dataflow computations, but not more general computations using producer/consumer schemes or mutual exclusion. Tasks in Cilk are not allowed to block and consequently cannot compete for a lock. A very recent development of Cilk, called Nabbit [6], allows to schedule computation graphs with arbitrary dependencies optimally, even if the dependencies are discovered at run-time. However, the optimal space and time bounds do not hold when the creation of a task depends on the outcome of some computation.

Optimal scheduling is a hard problem that, without very specific restrictions, is NP-complete [220], even in the ideal case where one would know in advance all tasks of a computation, their execution time and the graph of dependencies. Fortunately, several techniques to devise quite good schedules in the latter case with polynomial complexity have been known for long [113, 114, 218]. One of them is a simple greedy approach, in which a processor that becomes idle immediately grabs any task whose dependencies have previously completed. This greedy strategy yields an execution time within a factor of two to the optimal. Interestingly, greedy strategies also perform well for on-line scheduling, where the amount of work per tasks is not precisely known and tasks can be created dynamically. This property is the profound reason of the near-optimality of Cilk's work stealing scheduler [35]. This property is thus enjoyed also by the CAPSULE run-time system provided probes are frequent enough, as explained in Section 3.2.

An important drawback of the Cilk approach, also present in most task-based environments, is the influence of task grain size on raw performance and scalability. Creating a new task and scheduling it on a potentially different processor adds an overhead. For very small tasks, this overhead may cancel the expected benefit of parallelization, even if task creation is optimized by preallocating stacks and/or creating support structures lazily [108, 184]. Lots of computations, and in particular divide-and-conquer ones, have their performance improve drastically with a scheme that blocks task spawning for very deep (high level in Cilk terminology) tasks. We gave such an example in Section 4.2.3 with the `Quicksort` benchmark. A recent paper [81] proposes to instrument the run-time system so that it can evaluate dynamically the grain size of the tasks created at a given level. It then uses this information to inline some tasks instead of launching them in parallel. The corresponding article also highlights Cilk's poor performance on some benchmarks without cutoff to

avoid spawning very small tasks. Compared to Cilk, CAPSULE mitigates the task grain size issue by limiting spawning depending on core/processor occupancy. Experiments nonetheless show that more performance could be obtained with an additional limiting scheme.

5.2.3 Thread Building Blocks

Thread Building Blocks [133] (TBB) is a parallel library developed by Intel whose aim is to improve program performance by using threads. Programmers express parallelism as tasks rather than threads, as with Cilk or CAPSULE. They can also use some traditional data containers whose implementations are parallel at a fine-grain level and use atomic operations instead of locks. These containers resemble the containers provided by the C++ standard library in that they are generic: They are C++ templates parameterized by arbitrary types. Tasks are mapped to threads by a non-preemptive userland scheduler which is modeled after that of Cilk. In particular, it schedules tasks by work stealing into a fixed number of threads. Details about the Cilk scheduler can be found in Section 5.2.2.

For general `for` and `while` loops, TBB provides parallel versions, called `parallel_for` and `parallel_do`. These constructs take iterators[3] that form a *range* and a *body object* as parameters. The body object is a functor whose function contains the loop body. The run-time system automatically partitions a parallel loop into tasks that execute a subset of the iterations, according to a user-specified grain size or by delegating the split decision to a *partitioner*. Since a partitioner requires some knowledge about the internal library implementation, it is not possible for a programmer to define its own. Instead, it has to use one of the three that are supplied by the TBB library.

The simple partitioner recursively splits iteration ranges until indivisible ones are reached or they are all smaller than the grain size. The automatic partitioner begins like the simple partitioner by recursively splitting ranges until a number of ranges proportional to the number of threads maintained by the scheduler are created. A range is then split further only when it is stolen by a processor. Sub-ranges are thus created only when doing so is necessary to balance the load, which avoids creating very small tasks from the start and has the effect of automatically choosing a suitable grain size.

The affinity partitioner is useful when a loop is executed several times or different loops work on the same data set. When an instance of it is used for the first time, it works like the automatic partitioner but additionally remembers which processor treats a given range. When this instance is used again later, it divides the loop in the same set of ranges and dispatches them to the same processors that handled them in the first run. This increases cache reuse by scheduling computations to the processors that already hold the data, provided they fit into the respective caches. The memory and performance cost of remembering scheduling decisions is however unclear and may introduce unacceptable overhead.

Partitioners serve to mitigate the grain size selection problem, that also plagues Cilk, in the case of regular loops. Research is still on-going on this topic. A recent proposal,

[3] These iterators are modeled after the C++ standard library iterators.

lazy binary splitting [252], delays splitting as long as the local task queues are not empty. This technique is in essence Capsule's conditional division mechanism applied to work stealing in the context of parallel `for` loops.

For more general computation patterns, like recursive or irregular computations, programmers can directly use tasks and register them to the TBB scheduler. Explicit continuation passing, as in the early versions of Cilk, is even possible. To ease parallel programming, TBB provides concurrent versions of traditional containers such as hash maps, vectors and queues, whose implementations use lock-free techniques to reduce contention.

5.3 Parallel Semantics Through Shared-Objects

5.3.1 Orca

Orca is a language for implementing coarse-grain parallel applications on loosely coupled distributed systems. Parallelism in Orca is explicit, since a program starts as a single process and additional processes can be created by using the `fork` directive. The programmer must specify the number of the processor that will handle the new process. Processes are never moved automatically by the system, which implies that the Orca run-time system doesn't do load-balancing on its own.

Processes interact through *data-objects*, instances of abstract data types specified through interfaces, which are sets of operations on an object. Orca takes advantage of this common object-oriented pattern to provide the essential semantics a parallel program needs. Mutual exclusion, for instance, is enforced by the run-time system at a data-object granularity. Indeed, method invocations on a data-object are automatically serialized, as in monitors [126]. Conditional synchronization is implicitly achieved by the use of *guard conditions* in object methods. Guards are conditions on the method parameters and the object's internal state that will block the calling process if they are all false. Once some guards switch to true, a single one among these is chosen non-deterministically and the process resumes by executing the code associated with it. For ease of understanding and programming, Orca's policy is that single operations on a single object are executed *indivisibly*, whereas sequences of operations are not. Despite this property, Orca's model is flexible because the programmer chooses the granularity of objects and because it allows lock facilities to be built upon it to guarantee that a sequence of method invocations on possibly different objects is performed indivisibly.

As a process executes an object's method, it may block either because the method's guards are all false or as a consequence of invoking another method on a sub-object. In the latter case, blocking can occur either on the new method's guards or because of automatic mutual exclusion enforcement on the new method's object. Making single calls indivisible is then threatened in the following two cases. First, evaluating guard conditions can cause side effects, leading to the process suspending after having modified some object. Second, the process can be blocked on guard conditions of a method called on an inner object. Allowing another process to operate on the object while the first is blocked would violate

the indivisibility principle. However, not permitting it will deadlock the entire system, since, because of mutual exclusion, no other processes will be able to modify the state of the object the initial process is blocked on and no guard conditions will ever become true.

The generic solution used by Orca is to copy an object's content when a process invokes a method on it and have the process make modifications on this private new copy. If the process can run the method to completion without blocking, its private copy replaces the initial object's public state. If it can't, the private copy is discarded and the method invocation is started again later. Moreover, really copying an object, which can be expensive, is not necessary in the following frequent two cases. First, Orca enforces that guard conditions can only be defined and evaluated at a method invocation's start, before the calling process can even change the object's state. Except when a guard condition itself includes a side effect, which can be detected at compilation time, there is no need to copy the object when guards are evaluated and may block the calling process. Second, a process executing a method on an object without sub-objects won't block after mutual exclusion has been enforced and eventual guards evaluated. Copying the object content is thus not necessary in this case either.

Orca's features related to distributed systems and memory consistency are described in Section 10.4.3.

5.3.2 Charm++

Charm++ [150] is a parallel programming language based on C++ that supports communications via typed messages. It allows dynamic creation of parallel work. Charm++ derives from Charm [146, 147], an environment with a programming language consisting of C with extensions and a support run-time system called Chare [89]. Compared to the original language, Charm++ allows to use most of the object-oriented syntax of C++, with the additional benefit that describing chares is much more natural with it (see below). Apart from this syntactic sugar, functionality is identical.

Charm runs on top of shared-memory machines as well as distributed ones. In this Section, we describe some common features and their implementation for shared-memory machines. In order to be portable to distributed-memory architectures, programs should not use C or C++ shared data structures, but rather simple message-passing and *information sharing abstractions*, like read-only data, accumulators, monotonic variables or distributed tables, that are provided by the environment and managed by the run-time system. These special data structures, as well as the mechanisms specific to distributed-memory machines, are described in Section 10.4.4.

Charm is message-driven: All computations start at the reception of a message. Remote requests consist in sending a request message asynchronously and possibly continuing to perform useful work. As the corresponding response message comes back, and provided no computation is still on-going, its processing starts. Potential next steps are then carried on. Launching a new computation or exchanging data with another processor is thus always non-blocking. This has the important benefit of hiding the latencies of these operations. Multiple requests can be outstanding and are processed in the order in which

the responses arrive. A message's content is typed and can contain any data. It occupies a contiguous block of memory, as C structures. On shared-memory machines, messages are stored in global queues that are described below.

Chares are objects representing parallel processes. They comprise private data and functions, as regular objects, but also *entry points*, special methods whose goal is to handle incoming messages. An entry point has a single argument that is a typed message content. Every message in the system comprises some typed data content, an identificator that designates the destination chare and the name of an entry point to be executed by it. The content type of this entry point must be compatible with that of the message payload. However, the entry point to be called is specified by the sender, not determined by the receiver from the content type. An entry point executes atomically, i.e., a single message is processed at a time by a chare. Entry points thus do not need to synchronize. Synchronous requests can be split over into two distinct entry points. The first sends the request and the second is called when the request comes back.

Branch-office chares (BOCs) are objects that have a representative chare, called a *branch*, on every processor. In addition to what regular chares can hold, they can export public methods. Any chare can call a branch-office chare's public method, which results in a regular synchronous function call to the branch residing on the caller's processor, facilitating the support of regular and data-parallel computations. A branch of a BOC can communicate with any other branch of the same or another BOC, since branches are also regular chares. BOCs are especially useful in a distributed-memory context, as explained in Section 10.4.4.

Programs create chares but do not know which processor will execute entry points on the chare. Messages are directed at chares through a chare identificator (ID), which is independent of the chare's location. Similarly, each abstract data type has a separate ID space and objects of a type have a unique ID in the corresponding space.

On shared-memory architectures, when all processors are busy, incoming messages go into one of 3 global message queues. The first queue contains messages for chares, the second messages for BOCs, and the third new chares messages. There is no need to implement a particular load-balancing strategy. When a processor finishes the execution of an entry point following reception of a message, it grabs a message in the first non-empty queue. On distributed-memory machines, there can be no global queues, and different load-balancing strategies have to be implemented. Also, messages have to be treated to remain meaningful in the context of multiple address spaces. All the Charm mechanisms specific to distributed architectures are described in Section 10.4.4.

The way a processor selects the next message to process from a queue can have a significant impact on application performance. The Charm library provides 3 different strategies. If none of these strategies suits a particular application, the latter can choose to install and use its own instead. The first strategy is the LIFO strategy, also called depth-first strategy, in which the latest created or incoming messages are processed first. The second strategy is to process messages in FIFO order, or breadth-first. It has the advantage of usually generating many more messages, especially at the root of a computation. The load-balancing strategy therefore can dispatch bigger work sets earlier.

A disadvantage is that it consumes much more memory than LIFO. A third strategy combines depth-first and breadth-first. The latter is used when the load is low, in an attempt to dispatch the greatest amount of work to other processors, whereas the former is used when the load is high, in order to take advantage of locality. Finally, the run-time system also provides priority-based queuing strategies. Priorities are assigned by the application and consist of either an integer or a bit vector, in which case the vectors are lexicographically ordered.

Charm++ programs are decomposed into modules, which can be compiled separately. A module contains chare, BOC and regular C++ definitions, and an explicit interface declaration listing the names that are visible from other modules. The toolchain translates programs into pure C++ and then compiles and links them with the run-time system's libraries. The special keywords include `chare` to define chares approximately as regular C++ classes, `message` to define a new message content type and `entry` to define entry points in chares.

Chapter 6

Conclusion and Future Work

In this Part, we presented Capsule, an expressive programming model enabling programmers to concentrate on indicating potential parallelism and task dependencies, while taking the responsibility of deciding when work splitting is worthy and eventually dispatching tasks to hardware execution units. Constructs were conceived with the aim of being simple to apprehend, yet powerful, and to be as platform-independent as possible. In particular, programs can be written once and be run unmodified on machines with a varying number of cores.

A support software-only run-time system was presented, implementing a simple work splitting decision policy especially suited to embedded systems. We showed that programs using this run-time system can achieve at the same time a high scalability and have their execution time variability reduced compared to traditional parallelization approaches, on up to 4 cores. Some performance numbers for up to 24 cores were presented, as well as a study on the impact of the run-time environment on the achievable scalability.

This last study leaves a number of interesting questions to be answered. Program scalability, for more than 10 cores, shows a high dependency towards task granularity. We believe that programmers should be able to declare most, if not all, available work units without worrying about their granularity. The philosophy behind this claim is to preserve program portability for current high-end and future architectures, a very important advantage when doing parallel programming. We started to propose a number of schemes to solve this problem that should be evaluated and possibly completed as a future work. Part II, which describes the adaptation of Capsule to distributed-memory architectures, presents a new distributed work splitting and dispatching algorithm that also goes a step further on this road.

Part II

Distributed Architecture Support in Capsule

Chapter 7

Towards Distributed Architectures

As we mentioned in the general introduction of this manuscript, the fast growth in processing power has come mainly from frequency raises until 2005. Each year saw an increase of 50% in computing power per chip on average since the 90s. During the same period, memory bandwidth has progressed at a rate of 35% per year [181]. Memory latency, as seen by processors, has increased from 0.3 cycles in 1980 to 220 cycles in 2005 [122]. From 2007 to 2009, the DRAM read/write cycle time has diminished from 3 to 2 ns. It is expected to lower by approximately 11% each 18 months until 2019 [136], i.e., much more slowly than the expected frequency increase of 23% for the same period[1]. Consequently, the gap between memory and processor in terms of latency and bandwidth, which has increased exponentially in the past years, is going to continue on this trend for the next decade. This problem has been known under the denomination of *memory wall* [181].

Today, to maintain a steady increase in performance, the main chip manufacturers have been putting more and more cores on chips. This move has been facilitated by the decreasing cost of transistors, partly thanks to improved lithography and miniaturization. General-purpose processors with up to tens of cores [135, 254] are currently being shipped. High-end graphic chips already contain the same number of “macro-cores”, each of which is made up of tens of SIMD execution units [13, 196]. Cores generally share the same physical package, which has obvious benefits on communication costs. In designs where they also share the same interconnect to memory, they are putting even more pressure on the whole memory subsystem. The exponential progression of the number of cores per die, with 1.4× more cores expected at each new generation [136] for the coming decade, will thus continue to widen the performance gap between the processors and the memory for these designs. We are currently at a point where the performance of a substantial range of programs is already limited by that of memory.

In the meantime, some researchers have been proposing schemes to reduce the band-

[1] This has been extrapolated conservatively from the following data found in [136]: 1.4× higher frequency at each processor generation, with a generation approximately every 24 months or more.

width usage of processors. A study of the impact of these mechanisms, based on an analytical model of cache misses[2], suggests that combining DRAM caches[3], 3D-stacked caches, cache and link compression and smaller cache lines may allow enough bandwidth scaling for the next 4 generations [120]. However, most of these techniques have an adverse effect on latency. Other techniques, such as scheduling together processes whose bandwidth expectations are largely within the total available hardware bandwidth [261], while improving the situation, do not have enough potential to bridge the gap alone.

For these reasons, architecture designers will eventually have to systematically turn to distributed-memory architectures. In fact, AMD has been shipping processors with the circuitry to arrange several of them in a cache-coherent non-uniform memory architecture (ccNUMA) since at least 2005 (Athlon64 and Opteron). More recently, Intel has adopted on-chip memory controllers with the introduction of the Nehalem microarchitecture. The precise arrangement of future multi-core and many-core platforms is hard to anticipate. It is likely that the number of memory controllers and pins and the chosen memory organization (completely distributed, clustered, hierarchical, etc.) will vary. Nonetheless, their common denominator will be the fact that data at different addresses may not be stored at the same place and may not be accessed with the same latency and/or bandwidth.

In order not to disrupt traditional programming paradigms, the practical solution that has been adopted to exploit these architectures so far is simply to hide the distributed nature of memory through *global addressing*. With global addressing, programs access some piece of data through its unique memory address, which has the same meaning and relevance regardless of the particular core executing the memory instruction. Common languages assuming a shared-memory underlying architecture, which form the overwhelming majority of those that are effectively used in the industry, can readily be used without any particular adaptation, as well as the tool chains to process them.

However, programs running on distributed-memory architectures with global addressing can experience varying data access latency and bandwidth when using different addresses. In these architectures, the memory space in physically different banks or memory chips is usually mapped to contiguous regions of some common address space. This mapping is most of the time programmable and set up by the operating system. Nonetheless, user programs are most often not aware of the chosen partitioning[4]. They use classical memory management primitives, such as the `malloc` and `free` functions from the C standard library [137], that do not allow to pass usage information to the system libraries and the OS[5]. Practical memory management thus adds another layer of complexity when it comes

[2]This model assumes that the number of cache misses for a given program diminishes with increasing cache sizes as a power law. Experimentally, the exponent has been oberved to lie somewhere between 0.3 and 0.7, depending on the application, with an average of 0.5 [120]. These experiments confirm the empirical $\sqrt{2}$ rule stating that the number of cache misses is divided by $\sqrt{2}$ as the cache size is doubled.

[3]Instead of today's SRAM caches.

[4]This remark is a simple factual observation. We are not arguing that programs should be aware of such details. On the contrary, we advise against any change that would seriously impair program portability over different architectures and OSes.

[5]POSIX defines the `posix_madvise` system call for an application to be able to declare its intended use of memory areas. It has been inspired by the `madvise` system call from 4.4BSD. However, there

to predicting the cost of a particular data access.

Previous work [167, 193] has mostly tackled the problem of statically distributing data to be processed by loops with affine access patterns. A recent evolution makes the compiler produce symbolic expressions that can then serve at run time to allocate memory regions on the node that is likely to access them the most [171]. The essential problem of these approaches, besides the fact that they are limited to affine loops, is that they have no or limited ability to adapt their policy depending on the run-time program behavior. In the context of a dynamic and conditional tasking environment like Capsule, where task creation in fine depends on the run-time system and run-time conditions, it is almost impossible[6] to predict where to store/move data that should be accessed by some task, because the lifecycle of tasks itself and their scheduling is run-time-dependent.

We thus need innovative adaptive approaches that can supplement existing techniques in order to meet the grand challenge of supporting distributed architectures in an efficient and mostly transparent way. One of the service they should provide is to bring data close to its users. The programmer should also be relieved from the greatest difficulties of concurrent data accesses. The work presented in this Part is a modest step towards these goals. We nonetheless believe that it is a good candidate as a foundation to achieve these goals through future incremental improvements and, as such, may pave the way for decisive progress in these areas.

Chapter 8 proposes a new model to express and compose data structures independently of the physical location of their elements in a distributed-memory machine. The main contributions of this model are the following. First, despite its large applicability, it retains the spirit of the classical programming interface presented by most sequential and object languages. Programmers are thus already familiar with the concepts exposed, the only novelty being that the latter are more abstract than their original counterparts. Second, this model makes the links between elements within the same structure or from different structures visible to the run-time system. The latter can thus, without necessarily involving the program, take advantage of this information to choose the better location for data and automatically move and/or replicate them to the nodes using them. It can also perform *data-structure-aware prefetching* along the exposed links. Third, the access to some element's data is performed through primitives that automatically ensure a simple synchronization of accesses. More complex synchronization paradigms can be built from this basic synchronization operation, which relieves the user from part of the fine-grain atomicity management.

Chapter 9 then presents a new implementation of conditional parallelization and work dispatching that is *completely distributed and local*, which introduces two benefits. First, there are no central components that would impair scalability any more, as could be the case with the scheme previously presented in Section 3.2. Second, this proposal reduces

are no portable flags to indicate data and thread affinity. Solaris has been known to provide the `MADV_ACCESS_LWP` and `MADV_ACCESS_MANY` flags. Linux uses a new and non-portable system call named `mbind`.

[6] We will be glad to offer some Champagne bottle to anyone who is able to prove us wrong on this in a fairly general manner!

the performance dependency on task granularity, since the number of probes that can be accepted is now independent of the number of cores in the architecture.

Finally, Chapter 10 details a large set of previous proposals to implement global addressing on top of distributed machines or clusters, usually referred to as *distributed-shared memory* environments. It contains a comprehensive survey of memory consistency models and implementations, whose aim is triple: To deepen the description of some distributed-shared memory proposals, that are closely tied to some models, to situate our current consistency model between the known alternatives and to provide some inspiration for future work. It also reviews both distributed support for SPMD and task-based languages as well as languages based on distributed objects.

Chapter 8

Distributed Data Structures

One of the main programming difficulties of distributed-memory and, more largely, of distributed architectures, is to manage data location explicitly. In architectures providing global addressing, whether physically distributed or not, a program can reference an object or structure simply by its storage address. This possibility is the consequence of two main properties.

The first property is the fact that the same address refers to the same physical storage unit on all processors[1], by the very definition of global addressing. The most basic hardware that supports it is a shared-memory architecture where all processors access the same memory banks through a common bus. Other incarnations include various architectures with physically distributed memory that partition the address space so that each different memory is accessible through a distinct portion of it. Hiding the costs of remote accesses is then done either by using the traditional cache hierarchy also found on uniprocessors, as in cache-coherent non-uniform memory architectures (ccNUMA), or by replicating memory regions in different memories, as in variants of cache-only memory architectures (COMA) [117, 169, 221].

The second property, which one may not think of at first, is architecture-independent and comes from traditional language implementations. It is that a structure or an object is stored in memory contiguously. The compiler most of the time is the one that arranges data or fields in the allocated memory area, according to some standards or implementation details. A simple pointer[2] is then enough to reference the structure. The produced code performs the appropriate pointer arithmetic to access some data and/or fields[3]. Within structures or objects, fields are usually referred to by name before compilation. Within arrays, some numerical index, which may have been computed at run time, is used.

[1] This property actually does not come from the architecture alone. It also has to be supported by the run-time system. As an example, operating systems allowing multiple threads per process must ensure that a physical page or segment is mapped at the same virtual address on all processors executing the process' threads. For this reason, saying that an architecture supports global addressing is a language misuse. But it is a common one and we stick to it in the rest of this document.

[2] Together with a reference offset in the structure, which is implicitly 0 in practically all implementations.

[3] This implementation style has been used for years and is itself a significant abstraction from specifying hard addresses at compile time.

In order to be sufficiently transparent and expressive, we believe that distributed-memory support should retain the spirit of both properties, which we find more important than the possibility of accessing an object thanks to some unique storage address, a property that it is undesirable to maintain in future systems, in large part because it hides or imposes object location. To prop up this claim, we will now provide a brief answer to the following question: What are the assumptions behind these properties that simplify programming?

By the first property, programmers are able to transmit between threads or tasks the address of a given object, with the intent to allow them to access the same content and share the modifications to it. In other words, the advantage of this property is twofold. First, programmers do not have to manage explicitly the data transmissions from one memory to another one or different copies of the same data[4]. In addition to the performance benefits it can bring in lots of situations, this feature is a productivity advantage since the programmer doesn't need to provide the transmission code, which consists in possibly marshalling/unmarshalling the data, determining their destination and performing proper synchronization[5]. This advantage has been confirmed in an experimental study of parallel programming by non-expert programmers, which compares the number of lines of code necessary to parallelize a sequential code with MPI and OpenMP [127].

Second, since the programmer does not have to manage explicitly multiple data copies or versions, contrary to message-passing, he is relieved from maintaining *data consistency* for a non-negligible part. The remaining part of this duty then practically consists in describing which groups of memory accesses have to be performed atomically, so as to read a coherent view of an object or to update it in a meaningful way.

This description is itself dependent on the *memory consistency* enforced by the architecture, also called its *memory model*. It consists in a reasonably complete specification about the possible dependencies of memory accesses accross the same or different processors[6]. A detailed survey of memory consistency models is provided in Section 10.2. A reasonable but simplifying summary for it is that hardware usually provides strong guarantees that are intuitive to programmers used to sequential programming[7]. However, for performance reasons, the hardware often requires annotations by the programmer to implement this model.

Memory consistency alone is not enough to enforce data consistency, because the groups of accesses that must be performed atomically are essentially dependent on a particular program's semantics. In shared-memory architectures, and more generally for those that provide global addressing, the atomicity of multiple accesses at once is generally enforced

[4]On shared-memory architectures, these actions are performed by the hardware, principally by the memory hierarchy.

[5]Here, synchronization consists in knowing when some messages have to be sent and when a piece of code has to wait for a message, or conversely which code is run in response to a message arrival.

[6]The word "processors" can be changed to "cores" or "threads" in this context.

[7]These strong guarantees form a consistency model, called *sequential consistency*, that preserves the type of reasoning that can be applied to sequential programs. See Section 10.2.1 for more details.

through mutual exclusion, by using locks[8] or, more recently, transactional memory[9].

The second property, i.e., the fact that objects, structures and arrays are stored in a contiguous memory area, allows to access any fields or any particular element through index arithmetic given a single reference to any part of the object. This reference typically takes the form of a pointer to the start of the entity in the context of global addressing architectures. Although pointer arithmetic has its drawbacks[10], it is especially useful for highly-structured data, such as arrays or matrices. Maintaining explicit links between the different elements in such structures, e.g., by having each element point to its neighbors, would cause a huge space overhead and would increase execution time dramatically for a lot of algorithms, due to the impossibility of accessing elements randomly in constant time[11].

We present in Section 8.1 a new data structure model that relieves the programmer from managing data location explicitly. Like implementations of global addressing, it allows references to objects or structures to be transmitted between processors, which can then access them under a strong memory consistency model. Traditional addressing modes are preserved within objects. However, the addresses finally used by a program as the last step to access an object are not global nor persistent. This model is purely data-centric. By contrast with object-oriented languages, it defines objects as logical units of data, such as a C structures or arrays. Objects do not have methods defined on them and they do not perform any access control at compilation or at run time[12]. Mutual exclusion between tasks trying to access the same object concurrently is automatically enforced.

Section 8.2 contains a possible set of implementation principles for the previous abstractions. These principles enable a run-time system that implements them to move data at its discretion transparently for the user program. Several data management policies can be implemented on top of them. We give the example of a simple policy, where data are moved to tasks that request them. An experimental all-software implementation of these mechanisms is detailed. Simulation results are deferred to Part III, which details a novel many-core simulator we developed for the purpose of evaluating our data structure model and implementation on large scale machines. We also indicate which subset of the mechanisms could be implemented in hardware to improve performance and suggest an appropriate design.

Finally, Section 8.3 discusses the current status, possible improvements and future directions. First, providing different management policies that can be applied per object has the potential to increase performance in highly contended applications. Second, an

[8]The implementations of mutual exclusion now all rely on special hardware support in the form of two-way (or more) atomic instructions, for performane reasons, although they are not required from a theoretical point of view. See also footnote 2 on page 11.

[9]For more details and references about transactional memory, see Section 1.2.2.

[10]The major of them is perhaps the ability to access data through dirty tricks, some of which are: Rogue casts without type checking, memory scans or violations of layered structures.

[11]Another important cause of performance degradation is the higher memory occupation, which can cause more cache misses.

[12]But it is of course possible to implement such mechanisms on top of our proposal.

automatic prefetcher can be implemented thanks to links being transparent to the run-time system, as proposed in Section 8.2. We evoke a number of possible policies that should be interesting to investigate in this context.

8.1 Data Structure Model

Essentially, computer memories allow to store and retrieve data by specifying their address. With such a storage facility, the first step to flexibility was to eventually abandon the naive approach of manipulating data directly through hardcoded locations in favor of dynamic references through variable pointers. With a pointer, a program can designate a particular byte of storage. A pointer itself can be stored as an integer in a processor register or a memory area. The latter can then be used as a fixed name whose target can change over time.

Because pointers match the intuitive perception of tie or relation between ideas well, and because the data to represent an idea most of the time cannot fit into a single byte, programmers have tended to group multiple bytes around the pointed one, most of the time after it. As a consequence, coherent pieces of data have naturally been organized by blocks. Pointers can be embedded within data blocks as any other data, leading to a collection of blocks, also called structures[13], possibly linked with other blocks, thereby forming a graph.

To illustrate these considerations concretely, let's take the simple example of a single-linked list representation. In plain C, a structure type is declared using the `struct` keyword to represent an element of the list. This structure contains both the data that an element is supposed to hold and a pointer to a potential next element in the list. An object of this type is stored as a contiguous memory block[14]. An additional pointer usually serves to point to the list's first element (the head of the list). It is most often not stored in an object of the list elements' type. The `NULL` pointer value can be used either for the head pointer or for the next element pointer in an element's structure to indicate that the list is empty or that the considered element is the last element in the list (the tail of the list). In higher level languages, a list may be presented as a front-end object masking the actual elements. Even in this case do the latter follow the same structure [165].

The C structure type to represent a list element may contain several fields if the element's data are split into several pieces[15]. The C compiler translates accesses to a

[13]The C `struct` syntactic construct serves to declare objects represented as memory structures, in our sense of this expression. These concepts are nonetheless ontologically different. C and Fortran arrays are also implemented as memory structures.

[14]The C standard [137, Section 6.2.6.1] mandates that all objects be stored as a contiguous sequence of bytes. Padding is commonly added between the fields of a C structure to enforce field alignment.

[15]This arises typically when different algorithms need to access different pieces of data, or even simply when a programmer makes this distinction to follow his particular data view. A good programming practice should however be to have a single field to store an element's data, itself being of a composite type with several fields. It helps to clearly distinguish the structure imposed by the list data structure itself and the type to represent for each element the suitable data for a particular problem, with obvious software engineering benefits.

particular C structure field into assembly instructions that use the pointer to the structure as a base address to which they add the field's offset. The latter is determined at compile time based on the number and arrangement of fields according to some standards[16] and/or implementation details.

The data model we present uses abstractions of the concepts of block or structure and pointer. Most of the operations traditionally associated to these concepts are part of the model's programming interface. By contrast, it imposes no particular implementations and does not require addresses to be valid globally. It also introduces two important novelties. First, it treats differently standalone data and pointers when they are part of a block. Second, it enables automatic synchronization of accesses to a single object/block. Section 8.1.1 describes the concepts more formally and Section 8.1.2 presents the programming interface.

8.1.1 Concepts

We present in this Section the central concepts of the data structure model we propose, which are *links*, *cells* and *handles*. With them, we strive to provide an intuitive and easily manageable data model for programmers, while allowing implementations not to provide global addressing.

These concepts do not replace traditional ones and the associated implementations in existing programming languages. They rather complement them and coexist with them. The rationale for this choice is to have programmers use them for data that are to be shared between tasks, whereas they will continue to use traditional concepts and their implementation for data that are private to a task.

Links and Handles

Links are the abstraction that generalizes the notion of pointer. A link is indeed an opaque reference to a particular object or data block, which is called a *cell* in the model. As a pointer, it can be *dereferenced* to access the content of the target cell. It can also be transmitted from a task to another, by passing it as a direct argument to a new task or by ensuring it is stored in a shared cell examined later by the other task.

Like pointers, links are persistent: They can be stored and retrieved at a later point without affecting the program behavior. However, a pointer reused subsequently may not be valid any more or may even point to a different object[17]. By contrast, a link is guaranteed to remain valid and always point to the same object, unless its target is explicitly changed. As a consequence, an object lives as long as there are links referencing

[16]This kind of specification is usually standardized in some document describing the ABI to use for an architecture and/or an OS.

[17]This can happen if an object's memory is freed and later reused for another object, in languages where memory management has to be performed manually, such as C or C++.

it[18]. It is thus of foremost importance that the compiler or the programmer destroy links explicitly when they have no semantical purpose any more, so that memory can be freed. We do not currently provide support for weak references, i.e., references whose existence does not maintain the target object alive by itself, for a reason that will become apparent later. Nonetheless, the concepts presented in this section are powerful enough to allow implementations of weak references on top of them.

Dereferencing a pointer gives access to the pointed structure. However, there are no explicit operations to indicate the end of all accesses to the structure. This is the reason why it is not possible, in the general case, to automatically manage consistency structure-wise for blocks accessed through regular pointers. The situation is different in our model. Links similarly have to be dereferenced to access their target cell, but the programmer or the compiler must also indicate when a set of accesses have been performed. They do so through *handles*, local objects that eventually give access to the cell they represent through local addresses.

A handle is created when dereferencing a link and lives until explicit destruction by the program. The creation and destruction events for handles serve to indicate to the run-time environment when it has to make the content of a given cell available and how to delimit accesses for data consistency purposes. All accesses performed with the same handle, and thus by the same task, appear as a single atomic *macro-access*. Macro-accesses by multiple tasks are performed according to a strong consistency model. We settled for sequential consistency[19], because it is simple to program and to implement.

Overall, the memory consistency enforced by our model is a refinement of release consistency that is similar to a constrained scope consistency[20], with acquires and releases corresponding to creation and destruction of handles and a different scope being associated to each cell. This model might be relaxed in the future, if new experimentations show that a weaker model may bring significant performance improvements comparatively to the extra burden put on the programmer. However, we expect that other models will only bring modest performance enhancements over optimized implementations of the consistency we propose.

Cells

A *cell* is an abstraction of the concept of block or structure. It is the elementary and only data container in the model. Two different cells represent disjoint data sets, which prohibits overlapping or recursive data constructions at this level.

Cells logically comprise two *sections*. The first one is a section containing *pure* or *self-standing data*, i.e., data without any outside references to data stored in some regular

[18] We may later distinguish an object, as manipulated by the program, from its storage, and allow objects to be logically "destroyed" even if links are still pointing to it. However, if the implementation described in Section 8.2 is used as the basis for another implementation supporting this model change, a small storage area will still have to survive the logical destruction as long as some links to the object exist, precisely to indicate that the cell is no more valid.

[19] See Section 10.2.1.

[20] See Section 10.2.7 and Section 10.2.9 respectively.

structures or in another cell. It can be used by the programmer as if it was a contiguous memory region over which he has complete control. Programs can access some random piece of data in this section through indexed addressing, i.e., by giving the offset of the wanted region's starting byte. They also have to specify the region's length, which is normally implicit when manipulating object references in an object-oriented language.

The indexed addressing semantic of data sections allows to use them as arrays, matrices or hash tables, for which random access to any element is required. Specifying an offset inside a data section is semantically analogous to some pointer arithmetic, where an index would be added to the data section structure's base pointer. This is the only kind of pointer arithmetic that has been transposed into the model; no other such operations are allowed.

We acknowledge that, beyond semantics, arrays and hash tables are used for the performance benefits they bring in with current memory subsystems and language implementations. These data structures implicitly allow to perform an access to any of their elements in constant time. Our model requires that implementations conforming to it will preserve this property, but we do not make a more precise statement in order to preserve some flexibility for implementations.

The second section of a cell is the *links section*. As its name suggests, its purpose is to store links towards other cells. Each link within it can be accessed by its index and then dereferenced as explained above. As the other entities of the model, links are opaque. It logically follows that the byte content of the links section of a cell is also not directly accessible by the program. The main reason for this requirement and for a separate link section is to allow the run-time environment to be able to interpret a link without involving the user program in any way. With such a property, automatic prefetching can be implemented, as proposed in Section 8.3.

The size of both sections of a cell is chosen at cell creation. Once created, the sections cannot be resized. Forbidding cell resizing avoids making implementations unnecessarily complex. When a cell becomes too small for the intended data, a new cell has to be allocated again and the relevant data copied into it, as in C, C++ or Java. High level languages, their compiler or an extra layer of run-time system could perform this additional work automatically in lieu of the programmer. Because whole arrays may be stored inside a single cell, cells will considerably vary in size for most programs. This justifies the above-mentioned requirement of specifying the length of the region to access in a data section. The run-time environment is required to provide access to at least the requested region. It may internally give access or transfer a larger data area, for performance or convenience reasons, at the implementation's discretion.

8.1.2 Programming Interface

The programming interface for the data structure model presented above is described below as a collection of functions in the C language. As with the Capsule API in Chapter 2, this choice serves to illustrate how a program can concretely operate on the model's concepts with a simple and widely-known language. Other incarnations in other

languages, or at a hardware level such as the instruction set or even the memory subsystem, are possible and do not affect the semantics of the model. Actually, we expect that such new incarnations, e.g., in higher-level languages, will be necessary for the interface to be used by programmers in practice, as they can remove the need for most explicit calls to some of the presented operations. The code of our `Dijkstra` benchmark is reproduced in Appendix B as an example of use for the C incarnation of the interface.

The programming interface can be split into two categories: The primitives that create or operate on handles and those that create or operate on standalone links. Most notably, a program never manipulates cells directly. It has to use handles, local structures that represent a local access to a cell, as explained in Section 8.1.1. In the presented incarnation for the C language, the opacity of the model's objects is achieved by systematically using pointers to C structures that have an incomplete type[21]. To lighten the description, we will omit this implementation detail in the rest of this section and indicate directly the type of the objects passed as arguments or returned by the primitives.

Handles

Functions operating on the local structure representing a handle are prefixed with `capsule_hdl_`, whereas those that use a handle with the intent to manipulate the cell behind it are prefixed with `capsule_cell_`. Both kinds of functions always take a handle as their first argument.

Cells are created through the `capsule_cell_create_cell` primitive. The expected arguments are the size of the data section in bytes and the number of links in the links section. Once created, the new cell's data section content is unspecified. All the links in the links section are initialized to a special link value indicating that the link is invalid. We will call this value `NULL`, by analogy with the null pointer of C[22]. The primitive returns a handle on the newly created cell. Returning a handle instead of a link was preferred because we expect that, in most cases, the programmer will want or need to fill the new cell with data immediately. Two auxiliary functions serve to request the size of the two sections of a cell.

A cell's links section is accessed through four primitives. `capsule_cell_get_link` takes an index and returns a copy of the link at this index, unless the index is out of range, in which case an error is signaled. `capsule_cell_set_link_from_link` takes an index and a link and copies the passed link into the one at the given index in the links section. As a special case, passing a null pointer as the link to copy sets the cell's link to `NULL`. Similarly to the previous primitive, `capsule_cell_set_link_from_hdl` serves to set a link from a handle, by implicitly constructing a link to the cell represented by the handle.

Accessing the data section is done with two matching primitives. First, `capsule_cell_give_data_access` declares that a task wants to access a specific area of the data section. It takes as parameters the start offset and the length of the area. It returns a local pointer to the area, unless the parameters would imply that part of it is not contained in the

[21] I.e., the structures have been declared without specifying their fields.
[22] Whether `NULL` designates a null pointer or the special link value should be clear from the context.

section according to its size, in which case an error is signaled. The returned pointer is valid until the companion primitive `capsule_cell_revoke_data_access` is called with the same parameters, to indicate that the area is not needed any more. Only a single region of the data section may be accessed at a time by a given task[23]. Between the calls to these two primitives, the program accesses the requested data area as if it was a local structure thanks to the returned pointer. The run-time environment is responsible for mapping and unmapping the data region as a contiguous block of memory, even if it internally stores the data section as several fragments and the region overlaps two or more of them.

A single primitive, `capsule_hdl_release`, considers the passed handle for itself, although it may also have a profound impact on the cell behind the handle. By calling it, a program indicates to the run-time system that it does not need the passed handle any more and that it can be freed. The precise significance of this action is that the cell will not be accessed in the very near future again by the calling task. Conversely, we highlight that an alive handle represent the willingness to perform accesses to the cell immediately. A handle should not be kept if the task is not currently and actually using the cell. Releasing a handle may cause the cell to be garbage collected, if no more links are pointing to it. Long-term references to cells must be done exclusively through links.

Links

Contrary to handles, there are no constraints on the life duration of links, which can span the whole program execution. Another essential difference with handles is that the target of a link can be changed. This process decomposes itself into a removal of a reference on the old target cell and the addition of a reference to the new target cell. When a cell has its last reference removed, it is not reachable any more and is eventually garbage collected. Functions operating on links are prefixed with `capsule_link_` and they all take a link as their first argument.

A link can be created from a handle thanks to `capsule_link_create_from_hdl`. Calling this primitive semantically indicates that the current task intends to keep a long-living reference[24] on the cell represented by the handle and being currently manipulated. It is expected that this primitive will be systematically called after a cell creation on its handle.

A link can also be created from another link with `capsule_link_create_copy`. The last way of creating links is a bit peculiar since it does not allocate itself the necessary storage for a link, but rather relies on the program to provide it. This allows, in C, C++ and similar languages, to embed a link inside an already existing structure or object, for locality of reference reasons.

Links are released using `capsule_link_release`, which indicates that the passed link will not be used any more. This primitive has the semantical effect to destroy a reference to the link's target cell.

[23] We have not seen any practical problem with this choice up to now, but we do not exclude to relax this constraint if it proves to limit performance significantly in some situations.

[24] Other such references may be already held by the same or other tasks.

With `capsule_link_is_valid`, a program can test if a specific link is `NULL` or really points to some cell. Valid links can be dereferenced with `capsule_link_deref`[25], one of the most important primitive of the model. It returns a handle on the target cell. It can be seen as the reciprocal of the `capsule_link_create_from_hdl`. Nonetheless, we emphasize once again that links and handles have different meanings and expected life.

8.2 Implementation Traits

In this Section, we present the important characteristics of the class of implementations we propose. They come from a prototype implementation of a run-time system supporting the data structure model. The purposes of this prototype are:

- To illustrate that there exists at least one possible implementation to support the concepts of Section 8.1.1 with the properties announced at the beginning of this Chapter.
- To evaluate the performance potential of the whole run-time environment, including the concepts, the run-time system and the hardware architecture.

We have chosen to code the prototype in plain C and to integrate it within the run-time system[26] described in Chapter 3. This choice reduces the potential complexity of the implementation and makes experimentations with the different mechanisms manageable in a reasonable time frame. Since there is currently no consensus on how a many-core architecture should be structured and how many cores it can incorporate, the run-time system should adapt to various hardware designs and should be able to run on top of a flexible many-core simulator. We have developed such a simulator as part of this thesis, which is described in Part III.

Although all the mechanisms described below are implemented in software, the timings of some operations inside the simulator can be altered to make them appear as if dedicated hardware components were taking care of them. The only general assumption we make for the rest of this Section is that each memory bank is local to at least one processor, i.e., that the latter can access the memory content without sending requests through complex components such as network controllers on a chip or other processors. It does not affect the validity and generality of our approach significantly.

In this Section, we cover the most important traits of the proposed implementation. They were elaborated with the constraint that significant parts of them should be implementable in hardware instead of software. Their description thus remains slightly abstract in the sense that it does not always specify some particular way to concretize them. However, it provides enough high-level information to devise an implementation in a mostly straightforward way. Section 8.2.1 describes the current object management policy, i.e., how cells are stored and how access requests are fulfilled. Section 8.2.2 details

[25] `NULL` links can also be used as an argument of this primitive, which will return an error in this case.

[26] Also are included the modifications to be introduced in Chapter 9, which distribute the probe and work dispatch operations.

the mechanisms used to reference cells accross different address spaces. Section 8.2.3 proposes a possible storage structure for cells. Finally, Section 8.2.4 evokes hardware support for some mechanisms.

8.2.1 Object Management Policy

For the first implementation, we have settled on a relatively simple object management policy. When a task tries to access an object, the run-time system automatically moves its copy into a memory that is directly addressable by the task. There is at any given time a single copy of a cell in the whole system. It has to be acquired and, if not local, transmitted at each access.

Consequently, concurrent accesses must be serialized. It is not specified which components are responsible for performing this task. With no or slight modifications to existing architectures, a memory controller, a network interface controller or a processor could handle the serialization by arbitrating accesses on receiving requests. This latter possibility is the one implemented in the current software version. Since experiments are done inside a simulator, the performance trends of the other schemes can still be predicted reasonably, by choosing realistic ranges of costs for the various operations performed by dedicated hardware.

With the considered architecture designs, among all possible accesses, those from local processors/cores and those from remote ones come from different paths, and thus can reach memory concurrently. Consequently, even local accesses must be protected, because they may be involved in two scenarios with races. In the first, some remote data request may come in and trigger a cell move, finally invalidating the content currently accessed by a local processor. In the second, the remote data request may be serviced first, and then some cell modifications may be performed locally that may not be transmitted to the requesting processor. In the current implementation, each cell comprises a lock[27]. It is acquired when a core starts a sequence of memory operations on the cell and released when it is done with it. This locking cycle is delimited in the current implementation by the creation and destruction of a handle on the target cell, e.g., by calls to the `capsule_link_deref` and `capsule_hdl_release` primitives, which are described in Section 8.1.2.

Using locks to serialize cell access has an unfortunate consequence: Cyclic access patterns may deadlock the system. For example, consider a core that currently has locked a cell C_a and tries to access another one, C_b, which is stored in the local memory of another processor. If a core of this second processor has already locked C_b and tries at the same time to access C_a, both cores will wait indefinitely for the other core to release its lock. The chosen data model implementation can also introduce additional deadlock cases of its own, which a program must remain independent from.

Our current solution is to prevent a task from manipulating more than one handle at a time, effectively adding a constraint to the data model programming interface, as exposed in Section 8.1.2. We have not yet decided whether this restriction should be integrated

[27]If this choice proves to be unacceptable in terms of memory consumption in some contexts, it can be replaced by other schemes, such as using a single lock for a group of cells, at the expense of scalability.

into the official programming interface. We do not consider it as been impeding, because it can be worked around by a higher-level programming layer or a compiler. The current specification indeed does not specify any data consistency dependencies for concurrent accesses to different cells by a single task. The compiler or run-time system can thus treat them as if they did not overlap, by dereferencing cells sequentially in the order of handle destruction.

We will briefly discuss possible extensions to the current object management policy in Section 8.3.

8.2.2 Object References

In the proposed data structure model, references are opaque and manipulated as links by programs. We shall discuss here one possible implementation of links, which relies on two features. First, we use a very simple way to effectively reference objects stored in another address space. Second, we describe *proxies* and associated mechanisms, by which the final target location of a link may be changed without altering directly the reference data stored into the local structure that represents it, whether it was created directly by the program or is part of a cell's links section. The combination of both properties allows the run-time system to move cells transparently to the program, whose links remain valid after the move.

Global Addresses

Internally, cells are referenced by their storage location. The rationale for this choice is twofold. First, it is the most decentralized way to reference an object, at least theoretically. Reaching an object's content is then just a matter of directly contacting the component holding the data, which has to happen anyway, without involving any other intermediate component that could become a bottleneck. Second, using global and unique identifiers for objects instead of storage addresses would create additional problems. Among them, the foremost are probably to be able to attribute a unique identifier to an object in a decentralized way[28] and then to be able to track the location of an object and retrieve it from its identifier[29].

Since an address is relative to a particular address space, a scheme providing global and unambiguous addresses for all physical locations must be used. For this purpose, the run-time system attributes a different number to each address space. It then simply forms a global address for a location by appending the number of the address space to the local

[28] Identifiers could be attributed in a distributed way by distinct "authorities", each assigning them from its own reserved range of identifiers, all ranges being disjoint. This is actually what is done implicitly for addresses by our scheme.

[29] Early on, we considered using translation tables from identifiers to addresses, but they are also a central structure that will likely serialize execution because of numerous memory accesses. This idea nonetheless inspired us a proposal for a hardware TLB-like component, described in Section 8.2.4.

address within that address space[30]. We advise not to use any tricks that save part of the addressable space of a processor to store the address space number of a global address. Address space is already a scarce resource on lots of architectures and it is hard to predict how many different address spaces may be used in future distributed architectures[31].

Links are thus primarily represented as global addresses. Obviously, this property alone does not allow transparent cell moves, because they have the effect of changing the global address of a cell. One possible solution would be to keep on each cell a list of reverse links indicating which other cells have links pointing to it[32]. When a cell is moved, the run-time system could walk that list and signal an update to the relevant cells. This solution has three important drawbacks. First, reverse links will add a considerable memory overhead. More importantly, they will force the run-time system to manage containers of variable size for cells, in order to be able to store an unspecified and potentially high number of reverse links. Second, setting a link to a new target will incur a communication to the memory nodes where the old and the new target cells are stored, in order to update the corresponding reverse link of both cells. Third, when a cell holding a link is moved, the corresponding reverse link has to be updated. We have settled for a completely different solution, which is described below.

Proxies

Instead of trying to change the content of a link representation when the target cell is moved, we look at the problem in another way, and more precisely by the opposite end. We leave at the cell's initial location a special kind of cell, which we call a *proxy*, indicating that the original content has moved elsewhere. This is analogous to a web page redirection, where the page at the initial URL forces a browser to load another page, or, more trivially, to what mail forwarding services do.

Figure 8.1 (next page) shows a general cell situation, with some links pointing to a cell C, which itself points to other cells. For clarity, the latter cells are stored in address spaces not represented on the picture, but could as well reside in those drawn. Figure 8.2 (following page) shows the situation after the run-time system has moved cell C to another address space. A proxy has replaced the cell at its former location. Proxies are special

[30] Although we generally assume that address space are disjoint, i.e., that a physical location can be designated only by a single pair of address space and local address, this is not a requirement. The mechanisms we present in this Chapter also work even if some memory regions are mapped at different addresses on a single or several processors.

[31] This kind of error has already been made a surprisingly high number of times since the beginning of computers, and especially in the PC architecture. An example is the limitation introduced by Intel on the early x86 processors, which were able to address only 1 Mb although using 2 16-bit registers for that purpose in an ankward way (segmentation). The DOS limit of 640 kb is also famous among early PC adopters. We are relatively confident that the industry will make this mistake again, albeit under another form!

[32] In addition to links, the mentioned list should also contain references to standalone links that should also be updated. Since those may be contained for example in a core's hardware registers, the updating may be a complex process, although a solution à la Shasta for downgrades in the context of SMP clusters could be used (see Section 10.3.5).

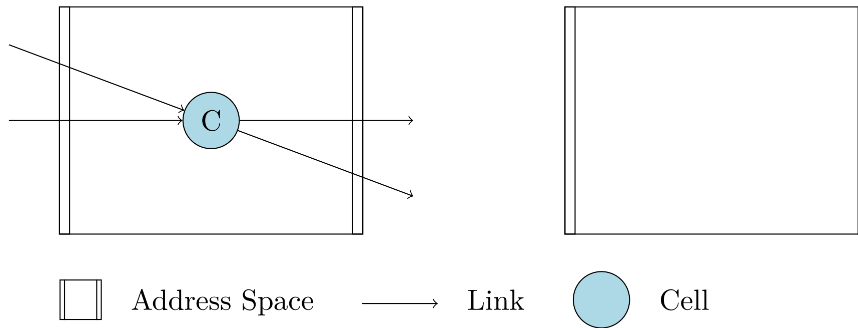

Figure 8.1: Situation of a Cell Before a Move.

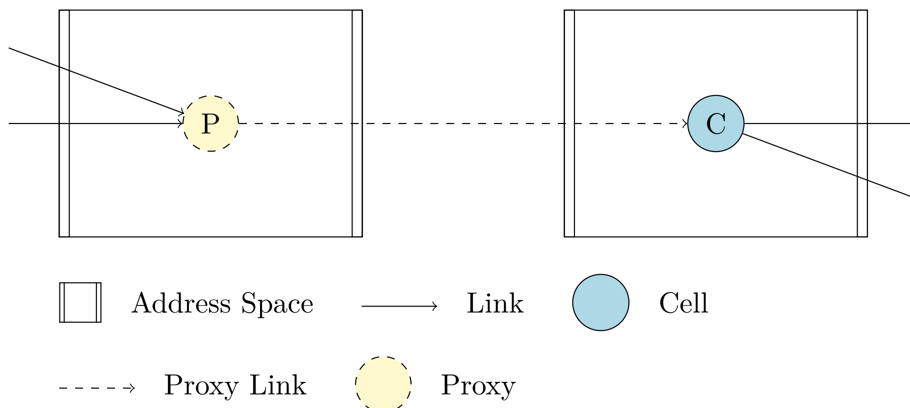

Figure 8.2: Situation of a Cell After a Move.

implementation cells that only hold a single link, called the *proxy link*. They do not have a data nor a links section, contrary to regular cells.

When a link pointing to C is dereferenced, the system follows the global address stored in the link, as usual. Some network interface or core tied to the address space that formerly held C is contacted and the proxy link retrieved instead of the actual cell content. Then, the proxy link is followed by having the contacted network interface or the core send a new request to some component tied to the proxy link's target address space. This request is formed on behalf of the initiating component and must contain the location of the initial request so that, when a reply is sent to the initiating component, the latter can match it with one of its requests properly. This solution is preferable to sending to the initiating component a negative reply with the new location to look at, leaving to it to send the new request. Indeed, this scheme would cause an additional message to be exchanged and may considerably increase latency, particularly if the address space containing the proxy is far from the path between the initiating component and the address space that holds C's content.

Every cell move leads to the creation of a new proxy. A cell and its proxies form a graph. It is easy to show by induction that:

- This graph is acyclic.
- That every proxy is the target of exactly one other proxy in the graph, except for one of them, which is the target of regular links only.
- Starting from any proxy in the set, following the proxy links eventually leads to the cell content.

Thus, this graph is a chain, which we call the *proxy chain*. Some cell content request will be forwarded along its proxies until the address space holding the content is contacted. Some component tied to the latter will then issue a reply message, holding the cell content. Once a cell has moved lots of times, dereferencing a link to it will necessitate to follow a long chain of proxies, increasing latency and creating more traffic in the network. Although a chain of proxy is acyclic, it may pass several times by the same address space with distinct proxies. This situation will typically occur for cells frequently accessed by multiple tasks and for which lots of different link copies are used.

Without reverse links on cells, it is generally not possible to directly update the links pointing to them. We have settled on a *lazy update* approach to reduce chains of proxies. When the initiating address space receives the reply with the content of some cell, it matches it to the initial request thanks to the global address stored in the link that was dereferenced. Then, the global address of this latter link is updated to the new location of the cell. With the policy of having a single cell copy in the whole system, the new location is local and is necessarily the most up-to-date[33]. In other words, each link is updated only when it is dereferenced. Other links logically pointing to the same cell will not be changed as part of this update. Consequently, the number of proxies to be traversed by a dereferenced link is at worst the number of the interleaved accesses since the last access through the same link by tasks running on different address spaces or using different link copies.

Starting from this simple mechanism, we see a large design-space of possible improvements and trade-offs to be explored. As an example, the run-time system can take action to limit the length of a proxy chain. As it dereferences a link, it can keep track of the number of proxies traversed. If that number is above a threshold, the run-time system, having updated the dereferenced link, can follow the chain again and update all the proxies to point to the new location.

In a radically different approach, the run-time system could use a single proxy for each cell and would not have to update any links. Rather, the first cell move would create its unique proxy and the cell would keep a single reverse link[34] to its proxy, updating it for any subsequent move. The trade-off with this approach is that the worst case is improved, with no more than one intermediate cell before the seeked content, but the best access case is worsened, since once a cell has moved, any cores have to go through

[33] Schemes allowing multiple copies of a single cell can exhibit a similar property, if carefully designed.

[34] The main difficulty to implement reverse links with the mechanisms presented before was indeed the variable and potentially unbounded number of reverse links, since the model does not place a limit on the number of links a single cell may be the target of.

one proxy for each access and this proxy has a fixed location. This last scheme is very similar to that used in home-based variants of distributed-shared memory environments[35], with the property that the "home", in our scheme the unique proxy, would be determined dynamically on first access. It would be interesting to compare such a scheme to the first one we proposed above.

The use of proxies and the absence of reverse links are important reasons why we introduced in the data model the requirement that links always point to valid cells, beyond correctness and improved error detection. Proxies are completely transparent to programs, which thus cannot be involved in their management in any way. Cell moves create additional proxies, consuming memory that cannot be reclaimed by the program trigerring the moves. In order to know when their memory can be reclaimed, the run-time system must track references on cells and proxies. Proxies, as regular cells, maintain a reference count. Each time a proxy is traversed, as part of a link dereference, its reference count is decremented by one, since the link that lead to it will be updated to the cell's new location. Thus, subsequently dereferencing the same link will not cause this proxy to be examined any more. Proxy links count as a regular link for reference counting purposes. Collecting proxies is ultimately necessary to be able to reclaim regular cells.

8.2.3 Cell Structuration

All cells need to store a common subset of information, including a reference count and the cell type, regular or proxy[36]. Moreover, cells are referenced by their location and the run-time system does not know a priori the type of the cell it is going to find at a particular place. For these reasons, all cells have part of their content structured in the same way, i.e., having exactly the same in-memory format. This part is called the *header* of a cell. It is a contiguous memory block and is stored at a cell's global address.

In the current implementation, when a link to a cell is dereferenced, the cell is moved to the requestor. Its header and data content are recreated in the destination address space, with the intent that the now local content can be manipulated directly by the requesting task. We have leaned to an efficient implementation, which in particular tries to avoid memory allocations when possible. To this end, a cell header can also serve as an handle. After a dereference and the cell move, the run-time system returns to the requesting task a pointer to a handle that is in reality a pointer to the cell header. Beside the cell type and its reference count, cell headers include a lock to ensure mutual exclusion when accessing a cell. This lock is thus taken and released without any indirections, as a handle on the corresponding cell is respectively handed to the program and later released by it.

Finally, a cell header contains a pointer to its payload data, which depends on the cell type. For proxies, the payload data consists of a memory block containing a single link

[35] The most important distributed-shared memory environments are described in Section 10.3. See in particular Section 10.3.1 about Ivy and its distributed management of page location and Section 10.3.4 about some other such environments, notably the home-based lazy release consistency approach (HLRC).

[36] In the future, more cell types may be used, in particular to handle cells containaing a large random-access data structure, such as arrays.

structure to store the proxy link. As explained in Section 8.1.1, a regular cell comprises two separate sections: The data section and the links section. The payload data for a regular cell is a small metadata structure containing the following information:

- The size in bytes of the data section.
- A pointer to a memory block containing the whole data section.
- The number of links in the link section.
- A pointer to a memory block containing all the links of the cell stored contiguously.

and the two payload memory blocks it references.

When a cell is transmitted between two address spaces, only the invariant and non-address-space-dependent parts of the header and the payload data are sent. More precisely, the lock in a header structure is not transmitted, nor the pointers to the payload memory blocks, which are meaningless in the destination address space. Instead, a new lock is created, the two payload memory blocks are allocated and the pointer slots are filled with their addresses. The content of both the data and links section is transmitted as is, without being interpreted nor translated. This implies that the current implementation is restricted to machines where all processors share the same data organization and representation rules[37].

At the start of a cell move, the run-time system grabs the lock in the header of the original cell copy. Then, it transmits the cell payload, as explained previously. Just after it has finished the transmission, it substitutes the original regular cell by a proxy without releasing the lock on the cell in between, in order to prevent intervening accesses. This substitution conserves the header as a local structure but leads to the deallocation of the other blocks composing a regular cell and the allocation of a proxy payload block to hold the proxy link. Finally, the run-time system releases the lock, authorizing again accesses to the cell. Holding the lock for the duration of the whole process is a safe way, albeit coarse, to enforce the model's data consistency. Other more fine-grain schemes may be considered in the future to improve the access latency of large cells.

Creating a regular cell involves 4 block allocations: One for the header, one for the metadata structure and one for each cell section. The rationale for splitting the total amount of memory required to store a cell, beyond proxy substitution that imposes it at least for the header, is both to facilitate memory allocation and to allow not too complex hardware implementations. Of the 4 blocks mentioned above, the header and the metadata structures always have the same size for any cell. They can thus be allocated from separate pools. They may also be allocated from special memory areas, as described in Section 8.2.4.

Having a separate memory region dedicated to the storage of cell headers in practice avoids a high level of fragmentation, not only at cell creation or deletion, but more

[37] Contrary to what the reader might think, only minor run-time system modifications and, optionally, some compiler support, would make this limitation disappear.

importantly when a regular cell is turned into a proxy. If a single memory block was allocated for a whole regular cell, a substitution to a proxy would cause the space used by its payload data parts to be wasted, because it is not possible with traditional allocators to free a sub-region of an area allocated at once[38]. Even if it was, the headers of proxies would remain as small allocated regions separated by the payload size of former cells, considerably fragmenting memory.

8.2.4 Hardware Support

We mention in this Section ideas about some possible hardware support for the data structure model. We have sketched the associated proposals with the constraint to keep them reasonably realistic. To this end, they are often similar to already existing mechanisms with different purposes. Further work might prove that some of them are actually required to produce an implementation on many-core machines that can obtain excellent performance for a vast majority of applications. We point out that none of these ideas were actually implemented during the thesis and that this section should be viewed as experimental. Our goal when devising these schemes was to roughly estimate the time that some operations in the model implementation would take in the best case, i.e., with proper but tractable hardware support. These estimations have been used in our simulations, as explained in Section 8.3.

A major drawback of the chosen cell structure presented in Section 8.2.3 is the number of indirections actually required to access data inside a cell. Starting from a pointer to a cell handle, which is concretely the cell header, the program and the run-time system need to follow 3 pointers to reach one of the cell's sections: The initial pointer to read the cell header, the payload pointer to read the metadata structure and the pointer for one of the sections to access the requested data or links. The main purpose of using distributed-memory architectures is to reduce contention on memory, which is too limited both in latency and bandwidth to allow a very large scalability. Thus, it does not seem a priori annoying that their benefit comes at the expense of increased latency for local accesses. However, tripling the latency to local memory may lead to unacceptable performance for programs whose shared data is not or moderately contended. Purely local accesses, i.e., those that are not done through the data structure API but operate on local structures, are obviously not affected by this problem.

The solution we propose to alleviate this increased latency is to introduce special hardware instructions for accessing a cell and a dedicated TLB-like component doing translations from a cell address to the local address of a given section. Accesses to a region of the data section or a link in the links section would be performed through special load and store instructions. They would take an handle address as the argument indicating the cell on which to operate. As a preliminary step, the program would still have to

[38]We note that this possibility is offered by the POSIX interface through the `mmap` and `munmap` system calls, but at the granularity of a page. Unfortunately, the traditional user interface for memory management [137] does not include this possibility.

dereference a link (`capsule_link_deref`) and indicate which regions of the sections it would be interested in (`capsule_cell_give_data_access` and possibly other new primitives).

When executed, the load and store instructions would simultaneously send the handle address to the memory hierarchy, as is common, but also to a new TLB-like component, called the Data Translation Buffer (DTB). This buffer would map the handle address, which we assume to be virtual, to the virtual addresses of both sections. If the handle address is not present in the DTB, a core has to wait for a reply from the memory hierarchy for the header structure. The process is then repeated for the metadata structure. Finally, the addresses for both sections are returned and an entry is created in the DTB. All these steps also have the effect of bringing the cell header and the metadata structure in the first level cache. If the handle address is present in the DTB, a core uses the relevant section pointer from the DTB to compute the local address where the data that it wants access to are stored. It then directly sends it to the memory hierarchy and the TLB.

A core dereferences a link to access the content of some cell. When it does so, according to the current policy, the cell content is transferred to an address space locally accessible by the core. We do not describe which kind of hardware support is necessary for this operation and assume some common implementation, such as a DMA engine like that of the Cell [207, 214]. Except perhaps when prefetched, the requested data are expected to be used nearly immediately by the requesting core. We assume that, after a cell transfer, the core that issued the request has the cell header and metadata blocks in one of its cache, in L2 or in L3. For small cells, e.g., up to some threshold and depending on the current cache use, we assume that the content of both cell sections is also copied into some cache in the hierarchy. Filling the caches as soon as possible before an access will improve data access latency. From this simple idea, there are numerous possible actual mechanisms, with the usual trade-off between cache occupancy and reduced short-term access latency.

It is desirable that a remote data request be processed without interrupting some processor tied to the address space where the requested cell is stored. To this end, it appears necessary that cell headers can be accessed directly by a network interface associated to the address space. This could be achieved by allocating headers in special memory regions. These regions would be delimited by the values of special processor registers and would receive a particular caching treatment. More precisely, the network interface and the local processors must be able to properly operate on cell header locks. By contrast, the requesting core would be interrupted or restarted if stalled when data are incoming. Allowing it to continue execution nevertheless in this case would require at least that the allocation of a new header be performed in advance, i.e., before issuing the data request. This address would be transmitted in the request and reply messages, so that the network interface receiving the reply could use it directly to store the content of the new header. However, it would be necessary also to pre-allocate all the cell's blocks. This requires to know in advance what the lengths of both sections are, which can be achieved by having proxies retain this information, at the expense of enlarging the memory footprint of the model.

Operation	Duration (in Cycles)
Cell Header Allocation	1
Cell Creation	3
Proxy Substitution	1
Handle Creation or Destruction	1
Data Access Request or Revocation	1
Link Content Access	1
Link Local-or-Remote Test	1

Figure 8.3: Cost of Specific Operations of the Data Model.

8.3 Status and Future Directions

Section 8.1 has presented a new data model that allows to write programs for distributed-memory architectures with simple concepts similar to that used in shared-memory. Moreover, Section 8.2 has detailed the most important characteristics of a class of implementations for the associated run-time system. It has also suggested some ideas of hardware support to enhance the performance of the latter. We now give a brief status on the current implementation but also on the data model itself. We discuss their current focus, list their main holes and indicate how they may evolve in the future.

A software run-time support having the characteristics described in Section 8.2 has been coded. It has been integrated in a many-core simulator to evaluate the performance potential of the approach before undertaking a complex but more accurate implementation and simulation experiments. The simulator and the results of these experiments are presented in Part III.

Figure 8.3 presents the quantitative assumptions for data structure operations that were used in the experiments. The durations attributed to cell creation, proxy substitution and header allocations may seem overly short at first. Actually, they assume that some regions are pre-allocated in advance and/or that hints are available that make a majority of allocations succeed very fast. The cost of pre-allocation or hint maintenance is thus amortized over a large number of creation events. The other operations are access operations and their cost is compatible with the proposals of Section 8.2.4. These costs do not include that of actual cache or memory accesses, which are factored in through annotations in the simulator.

For ease of programming, we intended that our data model would uniformly treat all kind of data, including very small and very large units. The chosen programming interface reflects this intent. This commitment and the current implementation for it may lead to a number of performance problems.

For small objects, performance is very much dependent on the overhead of the data structure implementation, which itself depends on the cost of its particular operations, i.e., allocation, mutual exclusion, proxy substitution, garbage collection, and the communication costs of the architecture, i.e., the time it takes for some amount of data to

be exchanged by a pair of cores. Comparatively to parallel architectures of the past, multi-core processors have lower communication costs thanks to networks-on-chip which can function at very high frequencies and use up a larger fraction of the available silicon, raising bandwidth and reducing latency. Implementing a distributed-memory support with fine-grain objects, which may have been regarded as infeasible in the previous decade, now appears to be possible with good performance, with the tricks previously presented. This conclusion is supported by the results presented in Chapter 14.

At the other end of the cell size spectrum, the current implementation does not deal with large cells in any particular way. Large objects are essentially arrays and sometimes hash tables, since simple objects are small and other object containers are implemented as a collection of small objects linked with pointers. The significant amount of data they contain increases the probability that a high number of tasks may access to them. But tasks are generally interested in retrieving only a small part of the array's content, e.g., one or several elements. The chosen management policy is thus inadapted for such objects, since it causes the whole content to be transferred at each access, taking a time proportional to the cell's size, instead of that of the useful regions.

We deliberately chose not to deal with this situation in this first work. One reason is that our initial focus was irregular data structures, since there already exists an abundant literature on array data distribution for distributed machines, albeit mostly concerning static work distribution. Another reason is that we considered that an implementation of the data structure model should remain relatively simple and mostly application-agnostic, in order to allow an efficient hardware support and be generally applicable. Array distribution has to be performed differently for each application/algorithm to bring any performance benefits. Allowing to specify and implement a customizable and potentially complex data distribution while retaining fast operation in the simplest cases is not a trivial task and appears to be a research subject on its own. The important guidelines to explore this path, based on our experience, would be the following:

- The programming interface should allow complete customization of distribution for applications that need it, even if it already provides several common distribution patterns, e.g., for arrays with 1 or 2 dimensions. There will always be applications for which the default support has an adverse effect on performance.
- The interface and the compiler should work to ease the task of the programmer, possibly by providing him with policies requiring a very complex implementation. On the contrary, at the run-time system level, only simple but general enough mechanisms should be implemented. The compiler or libraries should bridge the gap between them.

Even for small objects, different object management policies than that presented in Section 8.2.1 would be beneficial. As an example, mostly-read variables would benefit from the run-time system allowing multiple copies of them in the system. As most accesses are reads, they do not need particular synchronization operations. On writes, the run-time system would need to ensure that other tasks will henceforth work on the new value, as specified by the data consistency model, a potentially costly but infrequent operation.

A study conducted as part of the development of the Munin distributed-shared memory environement indicates that several object uses are enough to cover the access patterns for almost all shared data. The performance of parallel applications can be improved significantly by choosing an appropriate protocol for each object use. For a detailed discussion about the access patterns of shared data and the associated protocols, which allow multiple copies of data to live in the system concurrently, please see Section 10.3 on distributed-shared memory environments, in particular Section 10.3.2 on Munin, and Section 10.4 about distributed objects environments.

Using opaque links whose format is known by the run-time system has the promising potential application to prefetch irregular data structures more efficiently to hide latency. At any point in time, a task is manipulating several cells and holds a number of standalone links. By automatically following some links among the latter and those stored in the currently dereferenced cells, the run-time system could prefetch some among those that are within the current scope of a task, i.e., those that are reachable by this task. We have just started the implementation of a simple policy that tries to prefetch the links of all manipulated cells up to a configurable breadth threshold and apply the procedure recursively up to a configurable depth threshold. The performance of this scheme will depend on the order in which semantical links are indexed and thus stored in links sections. For the future, it would be interesting to investigate schemes where the frequence of accesses to links are monitored individually and could serve to indicate the “hot” links that are likely to be dereferenced first and should thus be prefetched in priority. The compiler or, as a last resort, the user could also indicate whether some objects are strongly tied to help the run-time system make a good choice. The difficulty of this study will be to find an acceptable balance between the overhead of the monitoring and the benefits it brings.

The current interface and implementation of the data model are formulated in the C language. With higher-level ones or with special compiler support, cells could be dereferenced automatically around accesses to their content. The programmer could even use links with the same syntax as pointers. Some static analysis at compile time would group accesses performed in a row and dereference the cell only once for those. The compiler could also work around the current limitation of having to dereference a single cell at a time for data consistency purposes. In a computation that needs data from multiple cells, a cell that is accessed at different steps has to be dereferenced and released multiple times. The current data consistency model allows the content of this cell to be changed by other tasks between the different access requests. If the compiler dereferences cells implicitly for the programmer, it should also specify a consistency model not based on handle operations that lets the programmer know which guarantees to expect. A reasonable contract would be to guarantee that a cell content cannot be modified by other tasks between multiple accesses that are considered grouped, for a meaning of “grouped” to be defined precisely, e.g., for all accesses within a lexical scope. Such a model can be enforced by copying all data that must remain coherent at the first cell access in a row. This task could also be performed by the compiler.

Chapter 9

Distributed Work Management

In Part I, Chapter 3, we presented a run-time system implementation suited to shared-memory architectures with a moderate number of cores. In particular, Section 3.1 showed different techniques used to map user tasks onto execution units that assume a flat underlying architecture with uniform memory access and communication costs between cores. In Section 3.2, we detailed the chosen conditional parallelization policy, which essentially allowed probes to succeed if an execution unit is immediately available to process the new task. The associated implementation is based on a single shared counter indicating the number of currently available execution units.

Although this design provides excellent performance for a low number of cores, it may not scale on distributed architectures with more complex topologies. In such architectures, the choice of the execution unit to execute a task will increasingly impact performance. Also, the counter of available execution units will tend to become a botteleneck. Even in the steady state of an application where most probe requests are denied and the counter accessed only for reading without synchronization, access latency will increase with the number of cores, augmenting the duration of a denied probe. Locked accesses to update the counter will furthermore cause expensive cache coherency traffic, penalizing small tasks and degrading overall performance.

For these reasons, we developed a new class of mechanisms where the conditional division policy and task dispatching are both distributed and local. They only involve components directly attached to a given execution unit or its immediate neighbors, thus implicitly being adapted to the architecture topology. Global load-balancing is achieved thanks to local control rules. The proposed schemes have been implemented. One of them has been used for the many-core experiments performed in Part III. The parameters used in this evaluation will be detailed in Section 13.2.6. In this Chapter, we explain and discuss the mechanisms we propose to reach a global work balance. In the context of many-core architectures, they must be highly scalable, which we achieve by constraining them to be entirely distributed and local.

This work was at its inception and then for a large part afterwards a joint effort with Zheng Li. We focused more on the algorithmic setting and Zheng Li investigated possible hardware implementations of it. The particular algorithm he employed can be found in Li

et al. [173], a paper we recently published, and his PhD Thesis [172]. It slightly differs with the mechanisms presented here on several respects, namely the measure of local activity used for load-balancing and some details about the migration protocol related to the preservation of locality. The Li et al. [173] article contains a comparison of this scheme to a central scheme, showing that it is more scalable and offers significantly better performance starting from 16 cores.

Section 9.1 details the proposed new probe granting policy. It is based on task queues of fixed size assigned to groups of execution units. Section 9.2 considers possible designs for a distributed and local load-balancing scheme. In Section 9.3, we expose the chosen work load-balancing policy, and notably the algorithm that decides whether tasks should be migrated to neighbors. Section 9.4 details how the different components involved in load-balancing interact and the task migration protocol.

9.1 Probe Policy and Task Queues

As mentioned in Section 3.2, the constraint imposed by the programming model is that a probe must be an extremely fast primitive, at least in the case when a task creation is finally denied. It allows the system to be able to take advantage of very fine-grain parallelism. In a distributed architecture, maintaining this property can be achieved by taking decisions locally, in order to avoid remote access latencies and interactions with many other components.

To this end, we assign a bounded *task queue* to each execution unit in the architecture. A queue contains tasks waiting to be executed that are currently assigned to the execution units associated with it. When some execution unit finishes its current task (or some system or maintenance work), it will grab a new task from the queue. In our proposal, the only requirements for task queues is that it must be possible to insert or remove a task at both ends. One end is called the *local end*; it is where an execution unit tries to fetch some waiting task to process it. We will describe in more details the interaction of the task queue with other components in the next Sections. We emphasize that the introduction of task queues is an important shift of paradigm from the implementation that was presented in Section 3.2. Indeed, tasks may not be started right away after their creation. This may have for example an influence on the soft real-time properties exhibited by an implementation[1].

In the context of homogeneous architectures, where we assume that all cores have the same computing power and general-purpose capabilities, all the task queues have the same size, i.e., the same number of task slots. For other types of distributed architectures, each task queue may be sized differently, based on the expected use of the associated execution unit(s). In hierarchical architectures, where a single memory and/or network

[1] Section 4.1 shows that the implementation presented in Part I lowers execution time variability for multi-core programs. Allowing task creation even when all execution units are already busy may alter this property. We have not investigated this possible consequence experimentally. A priori, we do not expect that it will be significant enough to alter the conclusion that CAPSULE-parallelized programs are more stable than programs parallelized naively.

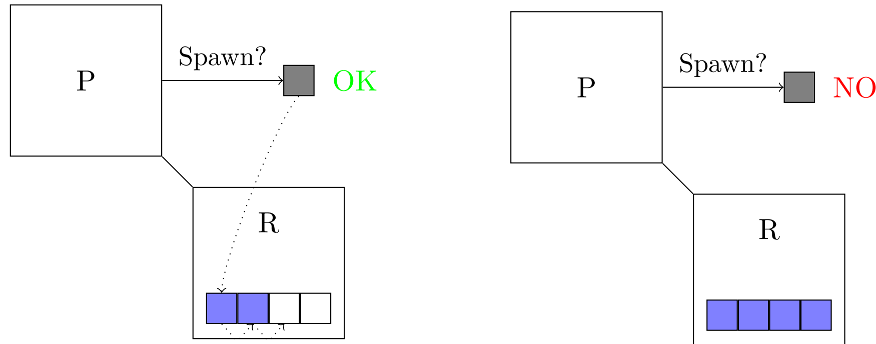

Figure 9.1: Probe Policy Example.

controller may be shared by several cores or other execution units, a single task queue may be used for each such group. In such a case, it could comprise the total number of slots that would be assigned separately to each unit or some lower number. In the rest of this Chapter, we consider only the example of homogeneous architectures and talk about the size of a task queue in general as being the common size of all task queues in the architecture, for simplification and without loss of generality. We will discuss the choice of a task queue size in the following Section.

To perform a probe or divide primitive as mandated by a program, an execution unit only needs to access its associated task queue, which we also call its local task queue. During a probe, it simply examines the number of free slots in its local queue. As long as this number is greater than some predefined threshold, it allows probes to succeed. Figure 9.1 illustrates the resulting policy with a threshold of 1 and a task queue size of 4.

To be more concrete, let us note q_{size} the size of task queues and q the number of occupied slots for a particular queue. If we note q_{free} the number of free slots in this queue, we have the elementary relation $q + q_{\text{free}} = q_{\text{size}}$. We introduce $q_{\text{max}}^{\text{local}}$, a threshold on a queue's occupancy above which new divisions are denied. Obviously, we have $1 \leq q_{\text{max}}^{\text{local}} \leq q_{\text{size}}$ for this parameter to make any sense. The condition for authorizing new divisions that was formulated above thus can be written mathematically as

$$q_{\text{free}} > q_{\text{size}} - q_{\text{max}}^{\text{local}} \quad .$$

A successful probe implies the reservation of one slot in the queue that will accommodate the task created by the subsequent divide, thus decrementing q by one. More precisely, the access to the task queue is an atomic operation that checks the queue's occupancy and simultaneously reserves a slot if the condition above is true. Atomicity is necessary to prevent concurrent accesses, not only from other execution units that may share the same task queue, but also from the component which is responsible for load balancing. The implementation of the divide primitive is then straightforward: The run-time system just puts the new task in the reserved slot.

As we did in Section 8.2 when describing a class of implementations to support our data structure model, we will not specify a precise implementation of task queues beyond the

requirements already exposed above, in order to allow variations in concrete achievements. The task queue can be implemented purely in software, as a double-linked list. This case requires special mechanisms to handle incoming tasks, such as interrupts, and more generally to perform load-balancing. It also requires some trampoline code that will call tasks to execute them and will try to fetch a new task each time it regains control. Hardware implementations are also possible since the task queues are bounded. We will briefly discuss one possible class of such implementations as part of the next Section.

So far, we have explained how execution units put the tasks they create in their local task queue and how those are fetched by a local execution unit that has just finished to execute another task. But, to create parallelism, i.e., to have several tasks executed concurrently, tasks from a queue must also be able to migrate to farther cores. We discuss some such mechanisms in the following Section. Compared to the policy exposed in Section 2.2, the one presented above is less independent with respect to work dispatching. The queue occupancy determines whether a probe can succeed, and it is dependent on the local work creation but also on how tasks are migrated from/to the local task queue. In other words, the outcome of probes also depends on the precise actions performed for load-balancing and their consequences on the task queue, by contrast with our earlier proposal.

9.2 Design Considerations

With the above-mentioned policy, task queues are gradually filled with tasks spawned locally. Enabling efficient parallel execution commands that tasks be spread to all the available execution units in the network. This is the purpose of the class of dynamic load-balancing schemes that we present in this Section.

9.2.1 Classical Strategies

There exists a substantial litterature on task scheduling and dynamic work dispatching for regular multi-processors and cluster-like architectures. Some of the oldest and most well-known techniques include an early work stealing proposal in the context of functional languages [46], the drafting algorithm [194], which also makes lightly loaded nodes pull tasks from busy processors, the gradient model [174] that implicitly computes a gradient surface which is used by heavily loaded nodes to send tasks to lightly loaded ones, some refinements of these [177], the receiver or sender initiated diffusions (RID, SID) [1, 257] or the dimension exchange method (DEM) [70].

These strategies differ on several respects. They can be classified along the components that initiate load-balancing in push, pull or hybrid models. In the first case, idle or lightly loaded nodes are the initiators and try to acquire work from loaded ones. In the second, loaded nodes push extra work to lightly loaded ones. In the third, both sets of nodes can initiate the transfer of work, depending on the conditions. A second distinguishing feature of the load-balancing strategies is how they estimate the load for each processor, and in particular if this assessment is based on a local or global mechanism. Local

evaluations can influence considerably the migration policies, potentially accentuating the behavioral differences between pull and push schemes. A third axis of comparison is the work migration algorithm, and in particular if it distributes work locally, i.e., to neighbors and other close processors, or globally, i.e., to any processors in the network.

The classical strategies we evoked are not really well suited to supporting a highly variable number of tasks, some of which are potentially very fine-grain, a situation that we aim to handle efficiently. Part of these strategies indeed trigger actual load-balancing only when some thresholds are met. As an example, the gradient model [174] typically classifies nodes into lightly loaded nodes, normally loaded and heavily loaded ones. Only the latter can push some of their pending work towards lightly loaded nodes. Depending on the load measure and the thresholds used, processors that consume work at higher speed, because they execute short tasks, benefit from locality effects between tasks and/or have a higher processing power, may not get new work quickly enough. The main cause is that the algorithm does not globally balance work in advance. In some situations, work in local excess may not even reach some processors that are already idle, if during its migration it reaches a normally loaded processor whose classification does not subsequently change.

Taking another example, the SID method [257] tries to attribute part of the local load excess of a processor to deficient neighbors, in proportion to the difference of their current load with the load average of the neighborhood. It assumes that excess load can be split into a wide range of proportions. This method apparently ensures a fast convergence of neighbor loads to the local average with a few messages. However, if the situations of the neighbors change rapidly and simultaneously, processing the new load indications sequentially can temporarily lead to a higher load imbalance. After a neighbor processor indicated that its load raised, the average load calculation will lead to sending work to some processors. In the meantime, one of the latter may have sent a notification to indicate that it is now highly loaded as well. But since its notification was not processed, it can be still seen as lightly loaded. As a result, it may have received a large range of work, loading it even more. Of course, this discrepancy will eventually be corrected, but introduces unnecessary latency and network traffic. The RID method similarly suffers from this kind of problem.

To diminish and control more finely their impact on the overall information traffic and contention on hardware components, we have chosen to focus on distributed and local schemes. They must be able to cope with extremely quick changes in local loads, which can happen especially with irregular programs parallelized at a fine level. For this reason, we favor those that do not require much information nor many computations to balance the load at a low level and that minimize the latency to propagate up-to-date statistics. It is also desirable to permanently balance the currently available work to be able to reduce global disparities more quickly. Moreover, the scheme has to manipulate tasks as black boxes which it cannot split on its own initiative and whose total amount of work is unknown. The drawback associated to all these features is that a potentially much higher number of messages will have to be exchanged to ensure quick adaptation.

9.2.2 Push or Pull?

The class of schemes we propose is based on the push paradigm, for task dispatching and also for updates of load statistics. This may seem surprising at first, as currently popular schemes such as Cilk or TBB [133] use work stealing, a technique in which idle processors try to pull pending tasks from other processors at random. If work stealing has shown excellent performance for a moderate number of processors, it becomes increasingly inefficient as their number rises [68], essentially because the probability to request work from a processor whose task set is empty also augments.

Another drawback shared by all pull methods is their higher latency. Transferring work at least requires a round trip, i.e., two messages, where the same transfer in a push model requires a single message. It can be argued that with some pull models, and in particular work stealing, a pull that succeeds moves a task to a place where it is immediately needed, whereas tasks may sometimes be pushed to processors already having work instead of idle cores. This reasoning, however, implicitly makes two assumptions that we will argue are not suitable for fine-grain parallelization and many-core architectures.

First, it supposes that pulls will most often succeed. As said previously, this becomes less verified as the number of cores grows. Also, such a policy amounts to favor phases where almost all cores are already busy. Some of the most influential characteristics of a parallelization approach performance-wise include the handling of “take-off” and reduction phases, where work is gradually spread to the network and then reduced to produce results. These phases indeed serialize the execution and thus should be optimized in priority. With push models, processors can dispatch work as soon as it appears with a single message when they cannot handle it, i.e., without any trial and error cycles as in pull models.

Second, some pull models, including work stealing ones, begin to perform load-balancing only when a core becomes idle. A commonly reported reason behind this choice [174] is to limit the number of messages exchanged to balance work[2]. This position has been argued in situations in which exchanging messages is costly, e.g., when executing parallel programs on clusters of workstations. However, with integration of cores on the same chip, exchanging messages has become considerably cheaper. Moreover, we will show with our scheme that, if load is permanently balanced, including when all cores are busy, cores finishing a task can get a new one nearly as fast in push models as in pull ones.

As a complement to this Section, we have detailed the scheduling strategy of Cilk in Section 5.2.2 and that of TBB in Section 5.2.3.

9.3 Load-Balancing Policy

As the load measure for a group of execution units, we could simply use the occupancy of the associated task queue and perform load-balancing guided by such statistics. In

[2]Some authors even consider that, as long as all processors are busy, load-balancing is simply not necessary, which we dismiss later in the main text.

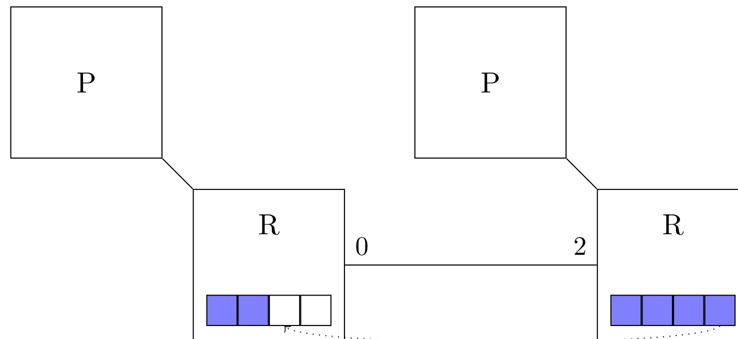

Figure 9.2: Example of a Task Migration Decision.

our schemes, we rather use the amount of work that can potentially be accepted by a group of execution units, i.e., the number of free slots in task queues. Both approaches are equivalent if all task queues have the same size. The latter seems however more adaptable to heterogeneous or polymorphic architectures. If some cores can process tasks quicker than others, they should have a larger task queue and should on average keep more pending tasks in them, since they may consume them at a higher rate.

Each group of execution units maintains proxies for the number of free slots in the task queues of its neighbor groups, i.e., local values reflecting the latest free slots updates received from them. Task migration decisions by a group are taken solely based on the values of its proxies. The algorithm that perform them is thus conceptually decoupled from the mechanisms used to update the proxies and to transfer tasks. Nonetheless, it is their interplay that determines the overall scheme efficiency.

9.3.1 Migration Decision Algorithm

Each time the occupancy of the task queue of a group changes or on receiving proxy updates from neighbor groups, some component of the group executes the decision algorithm. This component keeps a reference to the neighbor group having the higher number of free slots. The reference has to be recomputed for each new update, but this operation can be considerably optimized. If the new update contains a number of free slots higher than that of the currently referenced neighbor, then the neighbor in the update message becomes the new reference. If the new update message contains a lower number, then the reference must be updated only if the referenced neighbor is the one that sent the update. Indeed, in this case, the maximum may be realized by another neighbor. It is the only moment when an examination of all the proxies is necessary to find the new maximum.

Once the reference is up-to-date, its number of free slots is compared with that of the local task queue. If the difference is strictly greater than a predefined threshold, noted q_{diffmin}, a task will be pushed from the local queue to the reference neighbor. Figure 9.2 illustrates the policy with two cores each having its own task queue. The values of proxies are indicated over the link between both routers, which hold the task queues and execute locally the decision algorithm. The threshold is assumed to be one in this example. The

right router executes the algorithm and decides that it should send a task to the router on the left, since it has no free slots in its queue whereas the other has two empty slots. The difference between both numbers is two, which is effectively strictly greater than the chosen threshold. Note that, with a higher threshold, no tasks would be migrated and the free slots would remain slightly unbalanced.

The chosen threshold establishes a trade-off between the responsiveness of the scheme, i.e., how quickly it reacts to load imbalance, which increases if the threshold diminishes, and the amount of tasks and messages exchanged for load-balancing, which diminishes if the threshold is raised. It must be greater or equal to 1, in order to avoid a *ping-pong* effect. If the number of free slots for two neighbors differs by one and the threshold is set to 0, then the one with one more free slot will receive a task from the other. The situation then becomes the mirror of the initial one, triggering again a task migration which will send the task back. The process then continues until some other task is consumed or produced by one of the processors.

This effect can subsequently augment the amount of messages exchanged for load-balancing if not carefully controled. It might however prove useful in order to work around a possible limitation of the current scheme on large networks. As noted above, the task queues may stay slightly unbalanced. An important such case is if all tasks are created in the same network area. The queues in this area may be full because of task creation, but queues at distance one will have at least q_{diffmin} more free slots. More generally, queues at distance d will have at least $d \cdot q_{\text{diffmin}}$ more free slots. Consequently, tasks created in the area can never reach queues at a distance d verifying:

$$d \geq \frac{q_{\text{size}}}{q_{\text{diffmin}}} \quad ,$$

noting q_{size} the size of the queue as in Section 9.1. We call this phenomenon the *radius effect*. The greatest integer that is strictly lower than $q_{\text{size}}/q_{\text{diffmin}}$ is called the *horizon*, i.e., the maximum distance to which a task can send a child task, without the help of other task creators.

To give some quantitative idea of the radius effect, let us consider a 4-connected two dimension mesh and some processor P of it running a task that is the only creator of other tasks. We will assume that P is not close to the edges of the mesh to facilitate distance counting. Let P_d be some processor located at distance d from P. Given its relative position to P by a couple of coordinates (x, y), we have $|x| + |y| = d$. From that relation, we deduce that the number of such processors at distance d when $d \geq 1$ is $4d$, and finally that the number of processors at distance less or equal to h, some arbitrary horizon, is

$$1 + \sum_{d=1}^{h} 4d = 1 + 2h(h+1) \quad .$$

With a task queue of size q_{size} equal to 4, a difference threshold q_{diffmin} of 1, the horizon h is 3 and the number of reachable task queues is 25. This last number is pessimistic, since it is obtained by considering that a single task is the source of all other tasks, which is never the case in practice. As soon as tasks at the edge of the initial task creation

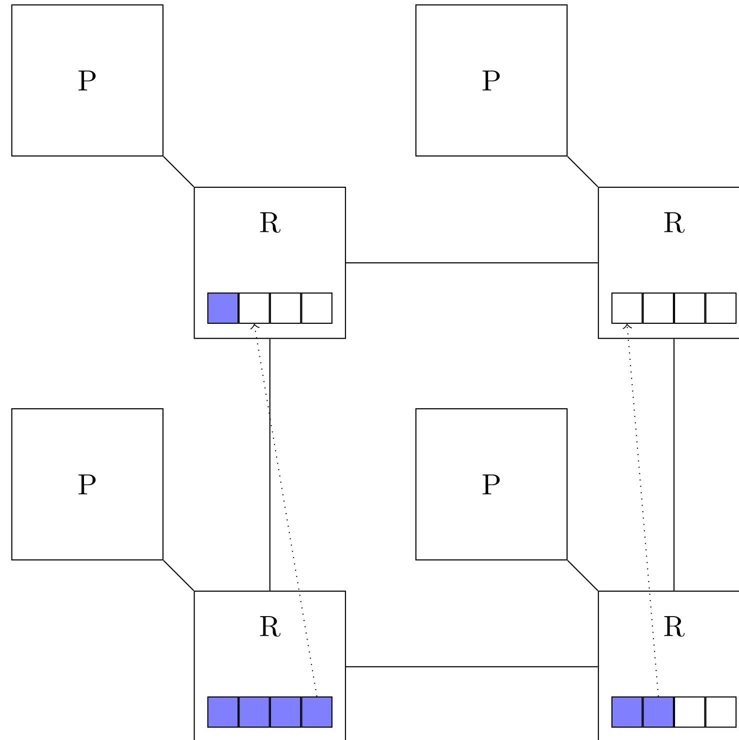

Figure 9.3: Global Load-Balancing Example: Initial Situation.

area in turn create tasks, they extend the horizon by the same number. After 1 such extension, 85 cores can be reached, and after 4 extensions, 481 cores. So the radius effect will probably not cause any problem up to several hundreds of cores, for not too small programs. We have however not investigated how it practically influences the simulations on 1024 cores performed in Chapter 14. It would be interesting to conduct such a study as part of future work.

9.3.2 From Local Decisions to Global Balance

If we except the peculiar situation involved in the radius effect, the local decisions taken by the algorithm make tasks spread gradually to all cores in the network and create a global work balance during periods where task creation and consumption is low. The demonstration of such a property is straightforward and similar to the ones that have been done for the classical diffusive schemes. We refer the reader to Cybenko [70] for an in-depth analysis of such schemes with formal proofs. We intuitively illustrate the global balance property through the example of a simple 2×2 2D mesh network, with q_{size} equal to 4 and q_{diffmin} to 1. Figure 9.3 shows the initial unbalanced situation and the first migrations computed by the algorithm. Notice that the task queue in the lower left corner does not push a task to its neighbor on the right, but rather to the neighbor above, because the number of free slots is higher in it. Figure 9.4 (next page) shows the

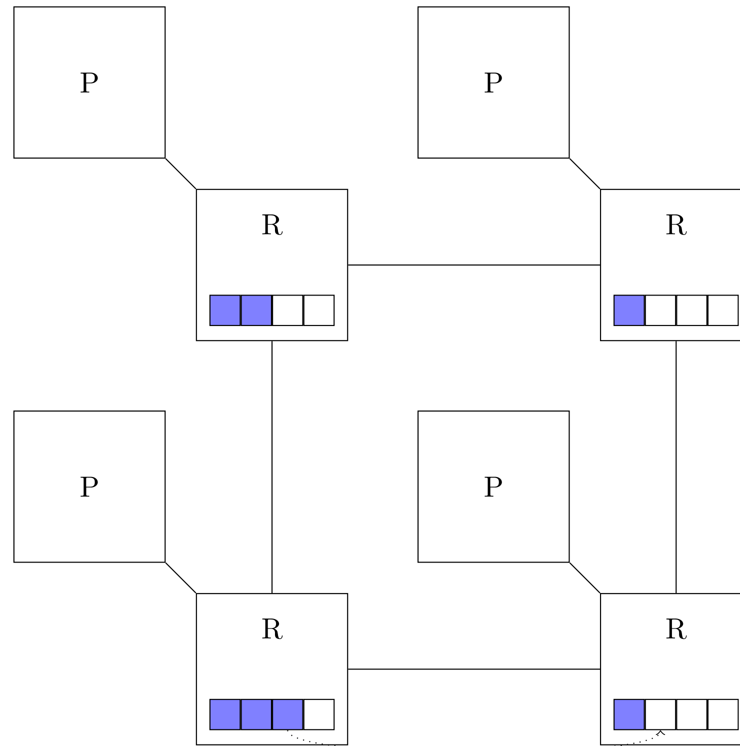

Figure 9.4: Global Load-Balancing Example: After One Step.

situation after the algorithm was executed once on each router concurrently and the next computed migrations. Figure 9.5 (facing page) shows the final situation, where all task queues hold 2 tasks, except the task queue at the upper right corner which has only one task.

The rates of creation and consumption of tasks determine the variance of work distribution throughout the network, in conjunction with the rate at which the load-balancing algorithm is run and can effectively migrate tasks. Good decisions also requires up-to-date information, i.e., frequent proxy updates. For this reason, an execution unit group sends an update of all the statistics that represent its current work load situation to all its neighbors as soon as one of them changes. These statistics comprise the number of free task slots in the queue and the number of idle local execution units. Each reception of an update message both triggers the update of the corresponding proxies and a run of the decision algorithm. In order to reduce the number of generated messages, a task queue sends through a single message the task it wants to push and a situation update to the elected neighbor.

With proper hardware support, for example in routers, an iteration of the load-balancing algorithm can last only a few cycles, which is faster than the time a core needs to create a single task. Without hardware support, i.e., if cores themselves have to run the algorithm, they can do so at each task creation, which enforces the same property.

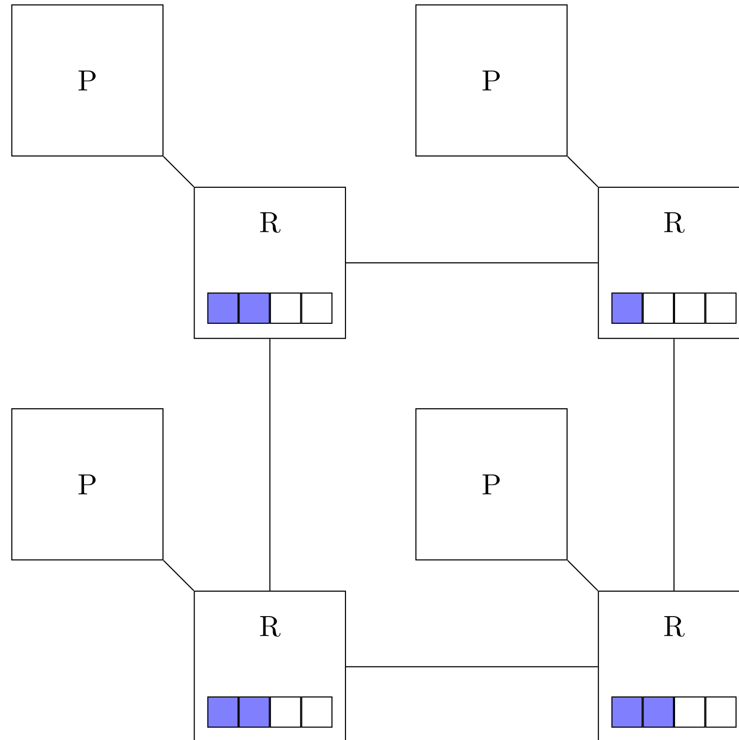

Figure 9.5: Global Load-Balancing Example: Final Situation.

Consequently, we have not considered extensions of our algorithm where several tasks would be pushed at once instead of only one in case of load imbalance. We have already argued in Section 9.2 that, in a rapidly changing environment, doing so may have an adverse effect on the bandwidth used by the scheme and the latency introduced before task startup. In the same line, we also have not considered delaying proxy updates or running the decision algorithm once out of several updates, because the benefits regarding bandwidth, and also task creation time if load-balancing is performed in software, probably will not outweigh the potential loss of performance due to reduced parallelism and a slower adaptation to program behavior.

9.4 Migration Protocol and Interactions With Task Queues

9.4.1 Concurrent Task Migrations

During some period of an execution where tasks are exchanged at a high rate, some queues may receive more tasks to be executed than can be simultaneously dispatched to the local execution units. Consequently, some execution units may temporarily stay idle although there are enough tasks in the queue to give work to all of them. The group of execution units may then be wrongly considered as having less available task slots, potentially inhibiting tasks from being pushed to it, or worse, causing some of its tasks to

be pushed to other groups. To correct this problem, the number of idle execution units in the group is maintained in the task queue, is communicated in update messages and stored into proxies, along with the number of free slots. The sum of both numbers is actually used in the comparison algorithm, practically considering idle cores as free slots.

However, without further precaution, it may happen that, based on this sum, the algorithm pushes a task to a group that has a higher sum but for which there are actually no free slots in the task queue. Since pushed tasks must transit by task queues, such a possibility must not occur. For this reason, the number of free slots and the number of idle execution units are both transmitted and stored as representing the state of a group. No tasks are ever pushed to a task queue that is full, even when the comparison of sums alone would hint for it.

Similarly, since migration decisions are taken based on proxy values, it may happen that the algorithm decides to push a task to some neighbor group although in the meantime some tasks have arrived in its task queue. If, as a result, the latter is full, the pushed task cannot be accepted. We emphasize that this problem is completely independent of the method or frequency of group statistics updates, since there will always be a delay between a state change in a task queue and the propagation of information about it[3]. In addition to this case which is mandatory to handle, a task may be pushed while the number of free slots in the destination queue has so diminished that the algorithm would have decided not to send the task if it had had this information. If the difference threshold q_{diffmin} is low, the receiving task queue may even try to push the task back immediately after reception.

To handle these situations, task queues can refuse tasks pushed to them. They respond to a push message with either a positive or a negative answer. While waiting for an acknowledgement, a task queue keeps the task it tries to push in one of its slots. From the point of view of the other neighbors, the status of its execution units group is unchanged. It is only after having received a positive acknowledgement that a task queue can actually free the corresponding slot. Negative acknowledgements trigger a new run of the algorithm. For this reason, they hold an update of the task queue situation, so that the algorithm uses the latest provided hint and doesn't try to push again the same task to the same queue.

Because tasks being pushed do not immediately free their slots, if the migration algorithm produces a decision based solely on the number of free slots, it may try to push too many tasks. To avoid this problem, the number of local free slots is biased by adding to it the number of tasks that the queue is trying to push, as if they had effectively been pushed. This technique ensures that there will never be more in-flight pushed tasks than the number of tasks to actually push, i.e., no tasks will be simultaneously pushed to two or more task queues. If pushes are all accepted, load-balancing is locally completed. If they are not, the algorithm will run again and determine the new best neighbors to push

[3]It is possible to avoid this problem by forbidding other task queue updates while some neighbor is interacting with the current group. However, this would require a stateful protocol and would impair concurrency. A symmetrical problem is present in pull models, since a task request to a neighbor may be denied if its queue became empty without the requestor being notified early enough.

the tasks to or stop if the neighbors' situation evolution has balanced the load with the local task queue.

9.4.2 Preserving Locality

Tasks form a tree in which siblings and parents and their children often manipulate related sets of data. Since maintaining locality is a critical necessity in modern architectures to attain high performance, it is desirable to execute all these tasks on the same or relatively close execution units. If related tasks are executed on the same execution unit, they may share data through its cache. Running them on close execution units may allow to share data in a cache of higher level, if the distributed architecture is clustered with respect to the cache hierarchy. In any case, they will be able to communicate with a lower latency than if they were very far apart, involving less communication links and thus less overall bandwidth.

As mentioned in Section 9.1, task queues have two ends. The *local end* is the one where tasks are fetched from or conversely added to by the local execution units. The other end is the *global end*. It is where incoming tasks that have been accepted are stored. It is also where the algorithm grabs a task when it has decided to migrate one to another task queue. In the Figures we presented previously in this Chapter, the local end is on the left of task queues, whereas the global end is on the right. For example, in Figure 9.1 (page 105), spawned task are placed in a slot at the local end on the left, shifting the other tasks towards the right[4]. By constrast, in Figure 9.2 (page 109), migrated tasks are taken from and migrated to global ends.

With this simple policy, tasks created by a single processor are processed in LIFO order, i.e., the most recently created tasks are the ones that will be executed next, if they are not migrated to other execution units groups. The tasks that are migrated are those that have been created the earliest and thus are less likely to share data with the latest created ones, if the temporal locality principle is verified. When a task is migrated to an empty task queue, it will be executed immediately. Conversely, if the destination task queue already has lots of tasks, it won't be executed until they, and all their potential descendants, have terminated, unless it migrates again.

There is thus a potentially sharp turn from attraction to repulsion towards migrating tasks. Some tasks from a branch in the task tree may stay in the same group area, whereas other may be forced to migrate far away. In cases where all the task queues are near saturation, tasks towards the global end of queues may migrate several times, heading towards less loaded queues but not being executed if the queue fills up again once they have arrived. Although we do not expect this situation to happen very frequently, it can randomize the final destination of a task and increase its migration delay and the related startup latency.

Another possible policy is to migrate tasks from the global end of the origin queue to the local end of the destination one. It is the policy employed in Li et al. [173]. With

[4]This move is conceptual. It would not really happen in an efficient implementation of task queues, e.g., a circular buffer with pointers to the global and local ends.

it, tasks creating other tasks at a high rate will progressively push away the tasks in the neighbor queues, whereas the previous policy would make child tasks migrate to less loaded areas in the network, even if they are very far apart. By putting migrated tasks into queues at the local end, i.e., closer to the execution units, it tends to limit the number of migrations per task. On the other hand, it also diminishes locality of execution. We have not experimented yet with this policy on our simulator. A comparison of the behavior of several programs should help to understand the impact of such a policy change. Some hybrid schemes may be able to trade off locality for better work repartition more finely and adaptively.

Chapter 10

Related Work

10.1 SPMD and Task-Based Languages

10.1.1 SPMD Languages

High Performance Fortran (HPF) [155], based on previous proposals around the Fortran programming language, supports array and matrix element distribution accross local memories along any dimensions. It also allows to specify which elements of different arrays should reside in the same memory bank.

Split-C [158] distinguishes local and global pointers. The latter are declared using the `global` type qualifier. They are implemented on distributed architectures by simply storing, along with a local address, a processor number that identifies the address space in which the address is meaningful. Our links similarly store a core number and a local address. Local pointers can also be converted to global pointers in Split-C. This is somehow different than CAPSULE's approach, where cells must be created explicitly and the local data have to be copied into it, because the run-time system stores cell data in memory with a different layout than local data.

Array or matrix elements can be stored accross different local memories according to common patterns, as in HPF, but the layout description differs. The *spreader* operator (::) separates the dimensions that are cyclically spread on all processors from dimensions whose elements are distributed locally. For example, `X[n]::[m]` creates a matrix `X` with `n` rows of `m` columns, the first row being stored on the first processor, the second on the next processor, and so on up to P, the number of available processors. Row $P + 1$ is stored on the first processor again, and so on. Special types of global pointers, declared with the `spread` type qualifier, can index data in the processor dimension: Incrementing them returns the element stored at the same local address in the next processor's memory, wrapping around to the first processor if necessary. As for HPF, it is impossible for the programmer to specify the location of each array element individually, besides grouping consecutive elements in one dimension[1].

[1]Not necessarily a canonical one.

Split-C introduces new data operators. The *split-phase* operator (`:=`) allows one to access (read or write) a shared variable[2] asynchronously. The execution thus doesn't block, and the programmer can specify several instructions that will be executed while the shared access is pending completion, masking the cost of the communication. When the result of the shared access is needed, the programmer calls the `sync` function, that waits for all pending accesses to complete. Some functions allowing split-phase bulk data transfers are also provided.

The *signaling store* operator (`:-`) behaves as a shared store specified with the split-phase operator, except that the store is signaled to the processor owning the region it references. The program can wait for the completion of a signaling store by calling the `all_store_sync` or the `store_sync` functions, the latter taking in argument the number of bytes to receive in the local region before continuing execution. There is however no support to discriminate an individual split-phase assignment or signaling store.

As a future work, we intend to assess whether we can improve CAPSULE's data structures handling's performance by introducing Split-C's split-phase data retrieval and comparing its performance with intelligent prefetching. Signaling stores could also allow to write data into a cell without moving it to the local processor first, reducing the operation's latency and the traffic over the network.

Unified Parallel C (UPC) is an extension of C inspired by Split-C and AC [48]. It keeps Split-C's distinction between local and global variables, except that the `global` qualifier is replaced by the `shared` one. By contrast, there is no more distinction between spread and global pointers and the spreader operator disappears. Instead, all shared arrays are distributed by blocks over all processors, with a default block size of 1, as in AC. For example, `shared double M[3][2*THREADS];` declares a matrix `M` consisting of `THREADS` matrices of size 3×2. `M[0][0]` and `M[0][THREADS]` designate the two elements of row 0 of the 3×2 matrix stored on the first processor. Other block sizes can be specified at the right of the `shared` qualifier, as in `shared [2] int N[4][3]`, which declares a matrix whose elements are spread in blocks of size 2. The first two elements of its row 0 are stored on the first processor, the last element of row 0 and the first of row 1 are stored on the next one, and so on.

The two split-phase assignment operators `:=` and `:-` of Split-C also disappear. As AC, UPC relies on the compiler to move data fetch requests earlier than the location in the instruction flow where the data are actually used in order to reduce latency. To enable this, the programmer can choose between two memory consistency models. The *strict* consistency is simply sequential consistency (see Section 10.2.1). The *relaxed* consistency chosen is local consistency (see Section 10.2.10). The consistency model choice is a default choice specified at the top of a source file. It can be overriden for particular variables by using the `strict` or `relaxed` type qualifiers, or for particular statements through a `#pragma` preprocessor directive.

[2] The *global* adjective is used in Krishnamurthy et al. [158] to qualify shared variables.

10.1.2 Cilk

The Cilk programming language and run-time system was presented in Section 5.2.2. In this section, we describe its adaptation to distributed architectures, namely the shared memory model it implements on top of distributed-memory machines and the changes to the work stealing scheduler.

Cilk, in its third incarnation [140, 210], introduced distributed-shared memory support, with a memory consistency model called *dag consistency*. This model intents to enforce that, for two comparable tasks A and B, if $A < B$ (i.e., if A is specified to terminate before B begins), then all memory writes by A are visible by B. Incomparable tasks performing writes, on the other hand, may not see each other's writes. As an example, child computations launched concurrently with the `spawn` directive can modify a common store area but may or may not see each others' modifications. The parent task most often calls the `sync` directive to wait for its child tasks' completion. Only at the time it regains control are all writes by the children guaranteed to have been performed.

To be more precise, a procedure is composed of several tasks[3], each of which is the continuation of a previous one (except the first task). The reason for this organization is that the Cilk scheduler doesn't allow a task to block when scheduled. Continuations serve to wait for data produced by another task, and are activated when the data are available. Fully strict computations restrict the possible data dependencies to those from a child to a task in its parent procedure, i.e., a continuation of its parent task. The task partial ordering of a fully strict computation is the transitive closure of a "precedes" relation in which a task precedes all its child tasks and these child tasks all precede the continuation of their father task. Children of the same task are not comparable by it.

Two definitions of dag consistency appeared in Joerg [140] and Blumofe et al. [39]. They essentially differ on how values are propagated between tasks that are not comparable. The first definition leads to a model which is the same as location consistency, which we discuss in Section 10.2.10. The second definition is stricter. It eliminates the corner cases of location consistency in which some old values can reappear later. This new definition, however, leads to a memory model that is not constructible [96]. Since computations are dynamic, a model implementation must be able to produce values returned by reads as a program executes. Constructibility is the property that the values assigned to reads for a computation could be assigned to the same reads in a super-computation containing at least the same dependencies and tasks as the original one. This property implies that a machine that is discovering new tasks and dependencies of a computation graph can attribute values to new reads coherently with the model and the values already attributed to performed reads. The weakest memory model, as advocated in Frigo [96], is a model in which accesses to a given location are serialized in an order compatible with program order. This model[4] is identical to the slow memory model presented in Section 10.2.6.

[3]We say task where Cilk's authors would say thread, as in Section 5.2.2.

[4]Frigo calls it location consistency, even knowing that this name collides with the historical location consistency model, described in Section 10.2.10. Although his reasons are defendable, we chose not to use this term since there was already a name for this model (slow memory), which he was apparently not aware of.

The first implementation of dag consistency, called DAGGER [140], has each processor maintain a page cache. When a task accesses a word whose page is not in the cache, the processor executing it sends a page fetch request to the processor that holds the parent of the seed task on the first processor. The seed task is the root task of the tasks that a processor holds. Except for the initial processor, it is a task that was stolen from another processor. In fully strict computations, a continuation of a task cannot be stolen until all the children of its previous task return, because it is schedulable only when it has received the children's data. This property guarantees that the father of a seed task stays on the same processor until the latter finishes. If the processor holding the parent task of a seed task doesn't have the page in its cache, the process is repeated recursively with the seed task of this processor. It stops when the page is found in some cache or the request comes to the initial processor, which directs it to a backing store.

A second implementation of dag consistency, called BACKER [38, 210], use frequent flushes of local page caches to the backing store. When a processor needs some data not in its cache, it obtains a copy of it directly from the backing store. Let us consider a dependency $T_1 \to T_2$, with processor P_1 executing task T_1 and processor P_2 executing task T_2. When T_1 finishes, P_1 reconciles its cache with the backing store, i.e., it stores back modified pages in its cache into the backing store. When T_2 starts, P_2 reconciles its cache with the backing store, and then empties its cache. This algorithm is not distributed and lazy like DAGGER, but it has the advantage that it works also for computations that are not strict. Page content is actually transferred through differences, as in TreadMarks (see Section 10.3.3).

The Cilk scheduler runs on multi-processor machines. It has also been adapted to a cluster of multi-processors. The work stealing strategy is modified to favor processors on the same cluster node as the stealing processor over remote processors. The ratio of the probability to steal a task from a local core over that of the remote core case is constant and set to a parameter α. Experimentally, α can be chosen to be equal to the ratio of the average remote steal latency over the local steal one, in order to divide by 2 the rate of remote steals which balance work at the beginning and end of a computation. Such a value gives linear speedup for very simple benchmarks [210].

10.2 Memory Consistency

10.2.1 Sequential Consistency

When multiprocessor computers appeared, where several processors could interact with a shared set of memory modules, programmability of such systems was studied. Lamport was the first to recognize that prior work on the behavior of programs on a multiprocessor system assumed a simple global ordering property in such systems, which he called *sequential consistency*. He formalized this notion by giving the following definition in its original paper [162]:

> The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

```
a = 0; b = 0; c = 0;
```

Initialization code.

Process 1.	Process 2.	Process 3.
`a = 1;` `b = 2;`	`if (b == 2)` `  c = 3;`	`print (c);` `print (a);`

Figure 10.1: Example Program to Illustrate Sequential Consistency.

Sequential consistency allows programmers to reason about their programs in an intuitive way, as if every execution had taken place on a single processor. Let us consider the example program of Figure 10.1, written in pseudo C. It is composed of a common initialization sequence, at the top of the figure, and 3 processes executing the 3 different instruction groups at the bottom. Each process is executed on a distinct processor. The `print` function, as expected, prints the value of the passed variable. Not counting the `print` statements, each instruction is a simple memory access, except the first instruction executed by process 2, which is also a test whose outcome determines whether the second instruction is executed.

We restrict ourselves to the executions in which the printed value for `c` is 3. We now reason "intuitively" on the program to determine the possible printed values for `a`. Because `c` starts with the value `0`, the statement `c = 3` must have been executed, implying that `b == 2` was true. Since `b` was also `0` at startup, the statement `b = 2` was executed. `a = 1`, the preceding statement in program order, was executed before. Finally, since `print (a)` happens after `print (c)`, the printed value for `a` is `1`. In this reasoning, we implicitely assumed that an address designates a single memory location and that memory accesses happen atomically at a given point in time. Sequential consistency ensures that each execution has the same outcome than an execution in which these assumptions are true.

Lamport's concise definition is operational for systems where access to memory is centralized, such as systems with a single globally shared memory, where a sufficient condition is that all requests arriving to memory be handled in FIFO order[5]. It is much more difficult to devise a practical scheme to obtain a global sequential ordering in systems where some data can be stored at multiple places simultaneously, which are the majority of systems in use nowadays. For performance reasons, cache hierarchies are designed to hold copies of data that usually reside in memory. Processors also hold outstanding requests to the memory hierarchy in prefetch or store buffers in order to hide the latency it introduces. This situation creates more possibilities that distinct processors may see different sequences of values for shared variables. As we will see in Section 10.2.3, practical constraints to enforce sequential consistency considerably hinder potential performance.

[5]Even in the case of multiple memory banks.

Sequential consistency is considered the strongest practical memory model[6]. For this reason, all the subsequent memory models presented here, which are all weaker than it, are called *relaxed* memory models.

10.2.2 Strong Ordering

A later paper by Dubois et al. [79] tried to determine sufficient conditions on how memory accesses must be performed to implement sequential consistency when data are buffered. To this end, it provided synthetized and new definitions regarding individual memory accesses that we hereby summarize.

A processor *initiates* a memory access when it forms it in its pipeline and then possibly stores it in one of its local private buffers. It then *issues* the access when the request leaves the processor and enters the memory hierarchy or the interconnection between processors. A read memory access by a processor P_1 is considered *performed* with respect to a processor P_2 if subsequent stores at the same address by P_2 can't affect anymore the result of the read access by P_1. Conversely, a write memory access by P_1 is performed with respect to P_2 when subsequent loads by P_2 at the same address will return the value stored by P_1 or by another processor that performed a store after P_1. Finally, a memory access by a processor is simply said to be performed if it is performed with respect to all other processors in the system.

Based on these definitions, Dubois et al. [79] presented *strong ordering*, a memory model equivalent to the following conditions:

1. All accesses to shared data by a given processor must be issued and performed in program order.

2. When a processor P_2 loads a value that was stored by processor P_1, all memory accesses performed with respect to P_1 up to the issuing of the store must have been performed with respect to P_2.

Condition 1 merely states that the intruction flow order must be respected and obviously implies the last part of the sequential consistency definition. Condition 2 imposes a constraint on how distinct processors can interact through memory. Roughly, a processor reading a value subsequently “sees” all accesses “preceding” the store that wrote it. The intuitive intent behind condition 2 is to allow reasonable inferences about whether some writes are visible to a given processor, by establishing transitivity between some accesses.

To illustrate the usefulness of this condition, we consider once again an execution of the example program of Figure 10.1 (preceding page) that prints `3` as `c`'s value. For a moment, let us assume that only strong ordering's condition 1 is in force, i.e., memory accesses are

[6]Sequential consistency is weaker than linearizability [123, 125] for objects that are the usual abstraction of memory cells, i.e., for which a read operation returns the value of the last write applied to the object. This is because linearizability does not allow to reorder operations that are timely disjoint. In general, however, processors don't know when instructions from other processors are issued and performed. Strictly speaking, for truly distributed systems, because of relativistic effects, linearizability is not applicable.

issued in program order. By the same reasoning as in Section 10.2.1, `a = 1` was executed before `print (c)`. However, the new value of `a` may not have reached processor P_3 before the execution of `print (a)` because of wiring delays, network contention or the new value being stored temporarily in a store buffer or a cache. Consequently, when processor P_3 executes `print (a)`, the returned value may be `0`, in spite of `print (a)` being executed after `print (c)`.

If we add condition 2 to the picture, `print (a)` can't print `0`. Indeed, `a = 1` was performed with respect to processor P_1 before `b = 2`, which in turn was performed before `b == 2` and `c = 3` with respect to processor P_2. By condition 2, `a = 1` has been perfomed with respect to processor P_2 when `b` is read. Similarly, P_3's `print (c)` reads `c`, which was set P_2's `c = 3`. Since `a = 1` is visible from P_2 when `c = 3` is executed, condition 2 says that `a = 1` is visible to P_3 when `c` is read. In conclusion, `print (a)` must print 1.

In the original article, it is stated that:

> It follows from logical considerations on the timing of events in a multiprocessor that a coherent system with a strong ordering of events is sequentially consistent.
>
> Condition 1 constrains the ordering of accesses on global data to be in program order. The only way that a processor I can affect another processor K is by I modifying a global variable, X, and by K subsequently reading the value. Condition 2 guarantees that all global accesses issued and observed by I before the issuance of the STORE request "happened before" all global accesses issued and observed by K after the LOAD request is performed.

Unfortunately, as was later discovered, this statement is false: The strong ordering conditions do not necessarily imply sequential consistency. To exhibit a counter-example, we reproduce the example program used in Adve and Hill [3] in Figure 10.2 (next page). This example also proves that concurrent consistency [227], which is very close to strong ordering, is not equivalent to sequential consistency either.

We consider an execution of it that yields the following values: `x = 1`, `y = 0` and `z = 0`. `x = 1` implies that `a = 1` was performed with respect to processor P_2. `y = 0` implies that `b = 1` has not been performed with respect to P_2 at this point. `z = 0` implies that `a = 1` has not been performed with respect to P_3 at this point. Overall, this execution doesn't violate the strong ordering conditions.

At the same time, it is not sequentially consistent. Let us proceed by contradiction and assume it is. For a given execution, we consider an equivalent execution in which all processor operations are totally ordered. In the latter, we can note A_n^i the i-th memory access by processor n. We also note $\prec$ the total strict ordering over memory events. As before, `x = 1` implies that $A_1^1 \prec A_2^1$. The difference is that, this time, `y = 0` implies that $A_3^1 \not\prec A_2^2$ and thus $A_2^2 \prec A_3^1$, because of the total ordering assumed by sequential consistency. Similarly, `z = 0` implies that $A_1^1 \not\prec A_3^2$ and thus $A_3^2 \prec A_1^1$. With $A_1^1 \prec A_2^1$ and $A_3^2 \prec A_1^1$, we deduce $A_3^2 \prec A_2^1$. By program order, we also have $A_3^1 \prec A_3^2$ and $A_2^1 \prec A_2^2$. By transitivity, we thus have $A_3^1 \prec A_2^2$, which contradicts `y = 0`.

This counter-example exists because of the nature of condition 2, which only adds an ordering constraint if a causal ordering between a write and a read at the same address already exists. In the case of a read by processor P_i not influenced by a given write

```
a = 0; b = 0;
x = 0; y = 0; z = 0;
```

Initialization code.

```
a = 1;
```

Process 1.

```
x = a;
y = b;
```

Process 2.

```
b = 1;
z = a;
```

Process 3.

Figure 10.2: Example Program to Highlight that Strong Ordering is Weaker than Sequential Consistency.

by processor P_j, the write was necessarily not performed with respect to P_i, but the conditions don't say whether the write is already performed with respect to another processor P_k. Such a situation can prevent a total ordering from existing, if the write is effectively performed with respect to P_k and another write issued and performed by P_i is not immediately performed with respect to P_k, as shown in the counter-example.

Strong ordering is said in Adve and Hill [3] to be a sufficient condition for most practical purposes, and in particular well-written programs that access shared data using usual synchronization primitives[7]. In fully general cases, however, it makes it harder for programmers to devise whether a particular execution is authorized.

10.2.3 Practical Sufficient Conditions for Sequential Consistency

As mentioned in Section 10.2.1, sequential consistency is not easy to enforce when multiple copies of data exist in a system. Practical rules to realize it were devised in Scheurich and Dubois [226], as part of an effort originally aimed at converting strong ordering's condition 2 to a more easily implementable constraint. This latter condition indeed poses the following question: How does a computer system ensure that operations performed with respect to a processor P_n are also performed with respect to another processor P_o, as soon as P_o performs a read returning a value stored by P_n?

The proposal to get around condition 2's generality is to replace it with two restricted conditions:

1. When a processor issues a store, it waits until the stored value is performed with respect to all other processors before issuing subsequent instructions.

2. When a processor issues a read, it waits until the read value is returned and the related store has been performed with respect to all other processors.

These conditions obviously imply strong ordering's condition 2. They are also easier to implement since each processor initiates a simple verification at each access, instead of having to monitor the performance of several issued accesses to be able to wait for their completion on an external read request. Moreover, they actually imply sequential consistency.

[7] We think that this statement may be false and might investigate its veracity in a future work.

Unfortunately, they also make it hard to implement, and sometimes even completely inhibit, most critical hardware optimizations performed in uniprocessors, such as prefetching or store queuing. Similarly, some compiler optimizations, such as instruction reordering, are precluded. To alleviate this problem, researchers have investigated two directions. The first is to investigate less restrictive rules that still guarantee sequential consistency but allow more optimizations to be performed. An example of a more elaborate set of conditions can be found in Adve and Hill [4]. The second is to define consistency models that allows violation of sequential consistency while still being programmable. This second path has given birth to lots of relaxed memory consistency frameworks since then.

The conditions above have been used in Scheurich and Dubois [226] to prove that several cache coherence protocols, including the classical MSI protocol [110, 244], make data accesses sequentially consistent.

10.2.4 Weak Ordering

Dubois et al. [79], which defined strong ordering, also proposed a relaxed memory consistency model, called *weak ordering*. It establishes a distinction between variables that do not control concurrent execution, and those that protect shared and writable variables or serve to synchronize the control flow of processes/threads. The former are called *global data* and the latter *synchronization variables* in the paper. When a processor accesses a variable, it must be able to somehow infer its type, so that it can perform the access differently.

The rules for weak ordering are:

1. Accesses to synchronization variables are strongly ordered.
2. Before issuing an access to a synchronization variable, a processor ensures that all previous global data accesses have been performed.
3. Before issuing an access to global data, a processor ensures that a previous access to a synchronization variable has been performed.

Compared to strong ordering, weak ordering allows global data accesses to be performed out-of-order, while still allowing to reason about synchronization or critical sections in the same way. Hardware mechanisms to hide memory access latencies, such as prefetch or store buffers, can thus be used for regions that do not require synchronization, which form a large majority of regions in most programs. More precisely, weak ordering restricts the use of such optimizations only at entry and end of critical sections.

The Adve and Hill [3] paper proposed a different definition of weak ordering, formulated as a contract between software and hardware, rather than specifying constraints the hardware must adhere to. This different point of view has the benefit that the programming model is clearer to the programmer. A drawback is that it is not obvious which hardware optimizations are allowed and how they can be practically implemented.

The new definition involves a synchronization model, and is stated as follows:

> A system is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all executions of a program that obey the synchronization model.

The chosen synchronization model essentially dictates which data dependencies are to be signaled to the hardware and when. When a program obeys the model, the system guarantees to the programmer that it will appear sequentially consistent to it, which eases reasoning on the possible program outcomes.

A simple synchronization model is the *data-race free model* [3, 5]. In this model, conflicting accesses are defined as accesses to a same shared variable by different processors that are not both reads. The model essentially states that all conflicting accesses must be ordered with the help of intervening synchronization accesses, in any execution of the program on an idealized architecture in which program order is followed and all memory accesses are atomic. This property allows to define, for each such execution, the last write corresponding to a read. Weak ordering is then showed to be equivalent to the property that, for each execution, there exists a corresponding execution on the idealized architecture that contains the same reads and for which the value of each read in the initial execution is the same as the value of the read's last write in the corresponding execution.

An implementation of this new weak ordering is proposed in the same papers [3, 5]. It consists in the following rules for the hardware:

1. Intra-processor program dependencies are preserved.
2. Writes to a given location are observed in the same order by all processors.
3. Synchronization operations to a given location are totally ordered[8].
4. No new accesses are issued by a processor until all the previous (in program order) synchronization accesses have been performed.
5. Once a synchronization access has been performed by a processor, another synchronization access at the same location by another processor will not commit until all accesses prior to the first synchronization access are performed.

That an access has committed means for a read to have determined its return value and for a write to have its value potentially observable by another processor. We point out that rule 2 is not really necessary. It only serves to simplify some proofs in the mentioned papers. Furthermore, it is not implied by the original definition, nor by the new abstract definition.

In the papers, it is claimed that systems conforming to the original definition of weak ordering also comply to the new one. Actually, as mentioned in the proofs provided by these papers, the original weak ordering definition has to be modified for this assertion to hold. More precisely, in condition 1, strong ordering must be replaced by sequential consistency. In the subsequent sections, weak ordering designates the new definition associated with the data-race-free model for programs.

[8]Equivalently, they appear to be performed atomically.

10.2.5 Processor Consistency

Processor consistency is a relaxed consistency model that was introduced independently in Lipton and Sandberg [175], under the name *Pipelined RAM (PRAM) consistency*, and in Goodman [109], which gives a more abstract definition. The underlying intent is to allow processors not to wait for stores to be globally performed before they can continue executing. This property indeed allows an implementation where writes are communicated to other processors asynchronously through buffers and an interconnect where point-to-point messages for a given couple of processors are guaranteed to arrive in the order in which they are sent[9].

Weak ordering requires that the programmer insert synchronization accesses for an execution to appear sequentially consistent. Processor consistency enjoys the same property, for properly defined synchronization operations, but is stronger outside and inside critical sections. Indeed, without intervening synchronizing accesses, writes by the same processor may be issued and performed out-of-order in weak ordering, without violating intra-processor dependencies, which is generally not allowed with processor consistency. Processor consistency allows to implement more algorithms using data races rather than explicit synchronization.

A system is processor consistent if any execution of any program is equivalent to an execution of the program in which all memory accesses by a processor P_i are performed in program order with respect to any other processor P_j[10]. Note that this is substantially different than imposing that an access by P_i must be performed before P_i can issue the next access in program order. Indeed, at some point, a processor P_j may observe several accesses issued by P_i, but another processor P_k may not have observed any accesses by P_i yet, which is not possible if P_i must wait for each access to be performed globally before issuing the next one.

An hardware implementation verifying processor consistency is proposed. It relies on each processor not having more than a single write pending, i.e., issued but not yet performed. This implementation in reality imposes a stronger ordering than processor consistency, since, in it, all writes by a processor are serialized, i.e., a new write can't be issued before the preceding one has been performed. By contrast, it does not impose more constraints than implied by the definition for reads.

With processor consistency, reads are allowed to pass previous non-dependent writes by the same processor, i.e., reads following a write at a different location in program order may be issued and performed before the write. When performed, this transformation, which is likely to improve execution performance, suppresses some possible program outcomes, which can make it harder to detect that a program is not correct with regard to processor consistency. Because of this property, processor consistency is not stronger than strong ordering[11].

[9] Hence the name Pipelined RAM.

[10] The original formulation in Goodman [109] differs. Our formulation leverages the vocabulary introduced by Dubois et al.. It is strictly equivalent to the original.

[11] We developed a proof that strong ordering is actually stronger than processor consistency that may be published in a future work.

Another processor consistency definition is given in Gharachorloo et al. [103], which results in a much stronger model, in which accesses may not be issued before some other accesses are performed. This model serializes reads and writes from the issuing processor's point of view, except that reads are allowed to pass previous non-dependent writes. Moreover, the paper generally assumes that writes to a given memory location are always serialized, i.e., are observed by all processors in the same order, which is not implied by the original definition. Unless otherwise stated, when we mention processor consistency in the next sections, we refer to the first and weaker definition.

Another paper [8] claims that Goodman included serialization of accesses at a given location in the original definition. The main presented argument is that he said processor consistency to be "weaker than strong ordering, but stronger than weak ordering", and that weak ordering in this sentence designates cache consistency. We argue that this confusion results from the introduction of the unnecessary location serialization condition by Adve and Hill, as explained in Section 10.2.4, rather than from Goodman's original intent. Cache consistency and weak ordering, as presented in this thesis, are not comparable. Moreover, the concise abstract definition found in Goodman [109] simply does not mention this requirement. Finally, the proposed hardware implementation concerns processors only, not memory, and it doesn't maintain by itself location serializability. For these reasons, we do not consider the location serialization requirement to be part of processor consistency[12].

Processor consistency usually corresponds, with some variations, to the implemented memory consistency of lots of processors/architectures, including recent ones [219], at least theoretically.

10.2.6 Slow and Causal Memories

Slow memory and *causal memory* were introduced as relaxed memory models in Hutto and Ahamad [128]. Causal memory was later formalized in Ahamad et al. [7].

Slow memory is weaker than processor consistency because it imposes that only the accesses to the same location by a given processor are observed in program order[13] by all other processors. Mutual exclusion and atomic updates can be implemented, but at the cost of complexity and performance degradation.

Causal memory, as processor consistency, allows different processors to observe different interleavings of events. However, it is stronger than it since these interleavings must obey causality. Causality is defined [7] as a partial order that respects both program order on each processor and the precedence induced by a write being observed by a read on another processor. Causal memory is thus equivalent to strong ordering, as defined in Section 10.2.2. Two algorithms to implement causal memory on top of a message

[12] It is claimed in Ahamad et al. [8] that the fact that Goodman would have "reinvented" PRAM is unlikely. This same article presents an example attributed to Goodman that would invalidate our position. However, it doesn't appear in Goodman [109], nor in any other articles from the same author.

[13] The requirement of program order as the common observed order is not stated explicitly in the paper, but implied since only physical memory models are considered. All operations are implicitly issued and performed in program order.

passing infrastructure are proposed. The simple algorithm works with reliable message delivery and the assumption that processes never fail. The robust alternative works even if messages are not reliably delivered or if a limited number of processes stop communicating.

10.2.7 Release Consistency

Release consistency was introduced in Gharachorloo et al. [103]. It is a refinement of weak ordering in which synchronization accesses are further divided into *acquire* and *release* accesses.

A program is considered properly labeled if, for two conflicting accesses u and v, as defined in Section 10.2.4, with at least one of them being a non-synchronizing access, for any sequentially consistent execution of the program, there exists a release access r and an acquire access a such that $u \prec r \prec a \prec v$ or $v \prec r \prec a \prec u$, with $\prec$ being a total ordering of operations implied by sequential consistency. In other words, conflicting accesses must be preceded by an acquire access and followed by a release access.

With this notion, the conditions for release consistency are the following:

1. No regular memory operations are allowed to be issued until all previous acquire operations have been performed.

2. A release access is issued only when all previous regular operations have been performed.

3. Synchronizing accesses must obey some strong consistency. In Gharachorloo et al. [103], either sequential consistency or processor consistency[14] is suggested.

Each different choice of consistency in condition 3 defines a different version of release consistency. When sequential consistency is chosen, release consistency is often noted RC_{sc} or simply RC. When processor consistency[14] is chosen, it is noted RC_{pc}.

For properly labeled programs, release consistency is equivalent to weak ordering and to sequential consistency. As with weak ordering, all traditional processor and compiler optimizations can be used outside of and inside critical sections. An additional benefit of release consistency is that it allows more instruction overlap than weak ordering at critical section boundaries. Condition 1 ensures that all other processors are aware that a process is in a critical section before seing any effects of the inner instructions. Contrary to weak ordering, it does not impose that instructions preceding the acquire operation must be performed before the acquire is issued, which is semantically not necessary. Condition 2 enables a similar additional overlap property.

10.2.8 Entry Consistency

Entry consistency [26] continues the trend of weak ordering and release consistency. As with these models, entry consistency distinguishes the acquire and release synchronization

[14]This refers to the second definition of processor consistency, as explained in Section 10.2.5.

accesses to be able to enforce strong consistency at critical boundaries only. But it goes further than these previous proposals in two directions.

First, entry consistency associates to a critical section and its related acquire and release operations the set of shared variables that are to be accessed inside it. This additional information allow a release operation to be issued even if some previous memory accesses have not been performed yet, provided these accesses do not concern the shared data locations guarded by the critical section. Similarly, acquire operations do not need to delay the issuance of subsequent accesses to not-guarded data. Another benefit is that no systematic strong consistency has to be imposed on acquire/release operations for critical sections that access disjoint data sets. A whole critical section can even execute while other acquires are pending, which is beneficial when several threads share the same die.

Second, entry consistency can distinguish exclusive and shared accesses to the data guarded by a critical section. Shared acquires are used for read-only accesses of the guarded data, whereas exclusive acquires are used for read/write accesses. As usual, shared and exclusive accesses are incompatible. When several processors want to enter a critical section to read the same data set, synchronization variables and the regular data are replicated to let them proceed concurrently.

It may be the case that the data to be accessed inside a critical section is not known at acquire time. Indexed memory accesses whose index is also computed inside the critical section are particularly problematic. They require that data can be associated to synchronization variables dynamically and even while the critical section is executing. Consequently, the run-time system has to provide functions the program can call to establish such an association. Performing these calls at run-time has a performance cost. Moreover, they are cases that are hard for a compiler to insert these annotations automatically, putting more work on the programmers' side.

A further optimization is not to wait for guarded data accesses to be performed before issuing a release. The purpose of a release operation is to mark the end of the critical sections and of the guarded data accesses. The next access to the same guarded data will occur inside another critical section, i.e., after another acquire operation. It is thus enough to ensure that the accesses of the first critical section are performed with respect to the processor executing the second one before the second acquire can proceed[15]. Implementing this optimization requires that the architecture keep track of such accesses and send them at acquisition. An implementation of release consistency can also benefit from it in a similar way, as demonstrated by TreadMarks, which is described in Section 10.3.3.

The implementation of entry consistency in the Midway[16] run-time system [26] is that a single processor owns a given synchronization variable at any time. Shared access requests are handled by simply replicating the variable and the guarded data. Exclusive access requires a transfer of ownership towards the processor that is about to enter the critical section. Each processor maintains a best guess for the owner of all synchronization variables. Each time the ownership is tranferred from a processor P_i to a processor P_j, both P_i and P_j update their corresponding best guesses. Other processors don't change

[15] This optimization is probably the reason for the name "entry consistency".

[16] See also Section 10.3.4.

their guess about the owner. When they issue an acquire request, that request is forwarded to their current best guess, which in turn relays it to its own. The process recursively continues until the request eventually reaches the current owner. We use the same idea to implement link dereferencing by following the referenced proxy chain to find the pointed node, as explained in Section 8.2.2. This mechanism is also very similar to what the Ivy distributed-shared memory system does, as described in Section 10.3.1.

10.2.9 Scope Consistency

Scope consistency [130, 132] is a memory consistency model that tries to keep the benefits of entry consistency without imposing on the programmer the burden to associate to a synchronization variable the data it guards.

All memory references are associated to one or several *scopes* as follows. At run-time, a scope's *session* starts with an annotated instruction[17] referencing the scope and ends with another. All memory accesses issued after the start and before the end of a session are informally considered to form part of the scope. By definition, all writes issued during the session are performed with respect to the scope when the session ends.

With these definition, the conditions for scope consistency are the following:

1. Before a processor P can open a new session for a scope S, all writes performed with respect to S must be performed with respect to P.

2. A memory access issued by processor P is not allowed to perform before all the previous session openings in program order.

In other words, all modifications performed during previous sessions that closed are guaranteed to be visible to a processor when it opens a session for the same scope.

Scope consistency was designed to suit synchronization through critical sections without requiring program modifications. Lock acquisitions and releases are natural candidates to form the start and end of scopes. A new scope is thus associated to each lock object. The code for acquiring and releasing a lock is augmented with calls to the run-time system to declare the start and end of the associated scope. Along with an ability of detecting writes, this information allow the run-time system to associate writes to a scope and to know when these writes are to be considered performed with respect to it.

Programs working correctly with release consistency do not, however, work correctly under scope consistency, despite the fact that critical sections are used to protect access to shared data. A side effect of release consistency is indeed to guarantee that all accesses issued before a critical section are performed with respect to all processors when the critical section ends. This is not the case with scope consistency. Only data modified inside a critical section are transferred when another processor enters it.

Producer/consumer schemes will typically fail without intervention from the programmer. In their case, data are often prepared ahead of the critical section, which then consists

[17] Annonated instruction is an abstract term and refers to a special hardware instruction or a run-time system primitive, such as a system call or a library function call.

solely of an update of some pointer to that data. There are two alternative solutions to this problem. First, the critical section can be enlarged to contain data preparation, at the expense of increasing the locking duration and augmenting lock contention. Second, another scope has to be created. Data preparation must take place within a session associated to this new scope and data reading in another one. For this reason, explicit annotations must be provided by the run-time system.

An implementation of scope consistency with hardware support is considered in Iftode et al. [132] and is summarized in Section 10.3.4.

10.2.10 Location Consistency

Location consistency was introduced in Gao and Sarkar [99] and later completed [100, 101]. It essentially allows a read to a given memory location to return one of the "most recently" written values.

An abstract location consistent execution is a concurrent execution of processes in which the instructions of each process are assumed to be issued and performed sequentially. Each memory location is associated to a *partially ordered multiset* (*pomset*)[18]. When a process writes into a location, the write is inserted into the associated pomset so that all elements of the set that were inserted by the same processor are ordered before it. Synchronization operations are considered inserted by all the processors participating in the corresponding synchronization event. A pomset includes only explicit or implicit operations on its associated memory location.

A read to a given location may return one value from the location's value set, which is constructed from the current content of the location's pomset as follows. Among the writes in the pomset performed by the processor issuing the read, only the last one is allowed to be returned. If, before the read, accesses were synchronized, other writes by other processors may be considered having been performed before the read. In this case, the last ones of them[19] can have their values returned as well. Finally, any value set by a write that is not comparable to the read and was not performed by the reading processor is also part of the value set.

A location consistent system is a system in which any execution of any program is equivalent to an abstract location consistent execution, in the sense that the operations in both executions are the same, and the result of each read in the original execution is part of the value set of the memory location at the corresponding read in the abstract location consistent execution.

The intent behind location consistency is to allow more efficient hardware implementations and compiler optimizations than in other memory models by dropping the memory coherence constraint that all writes to a single location are serialized, i.e., all processors

[18]The example developed in Gao and Sarkar [100] is not consistent with the fact that pomsets are partially ordered. The described update procedure for a pomset omits to mention transitive closure. The interested reader should instead consult one of the other two papers [99, 101].

[19]A last write w to a read r precedes the read ($w \prec r$) and there are no other intervening writes between them ($w' \prec r \Rightarrow w' = w \vee w \not\prec w'$).

oberve them in the same order. As explained in Section 10.2.5, processor consistency is another such memory model with the same intent that also drops this requirement.

Location consistency adds to processor consistency the description of how common synchronizing operations must behave. For regular reads and writes, location consistency is weaker than processor consistency. Reads by a processor may not return a sequence of values compatible with the write order on another processor, since, in a pomset, a read by a processor is never considered preceded by any write by other processors, except in the case of intervening synchronization operations. Consequently, such a read can return indifferently any written value by other processors, including older values. Location consistency even allows to return older values after newer ones.

A new cache protocol implementation, called the *location consistency protocol*, that makes a multi-processor system obey the location consistency requirements, is proposed in Gao and Sarkar [101]. Each cache line has one of the usual three following states: Invalid, clean (or shared) and dirty. A write-back policy is employed. Read operations hit in the cache if the corresponding cache line is in the clean state, as in other cache protocols, but also if it is in the dirty state. Indeed, location consistency always allows a read to return the result of the previous write to the location on the same processor. Write operations always take place in the initiating processor's cache. At acquire on memory locations, the corresponding cache lines, if clean, are marked invalid in order to force the next read accesses to go to the main memory. By contrast, the cache lines that are dirty are not flushed and their values will satisfy subsequent reads. At release, dirty lines are written back to memory and marked clean.

In this protocol, local modifications have priority over remote ones, except those occuring in a critical section. The latter are written back at release and the corresponding line is dropped at acquire. This property ensures that a value written at a location will be communicated to a processor subsequently entering a critical section to access the same location. It also allows to safely conform with location consistency, because a read has to be satisfied with a value coming from a remote write if there are no previous writes on the initiating processor.

We nonetheless point out that location consistency never obliges an implementation to propagate a location's value from a remote processor to a processor trying to read the location if the latter previously wrote to it. A consequence is that some values produced on one processor may never reach any other processors. Moreover, at most one value is stored by each processor in its cache. Returning earlier values after late ones is thus not possible. The proposed implementation is thus in reality much stronger than location consistency. Because location consistency is extremely permissive, stronger models, such as processor consistency, are rather used in practice.

As a side note, the original paper [99] claims that the vocabulary introduced by Dubois et al., and thus any memory model defined in terms of it, implicitly assumes the existence of a global clock for events. As should be clear from the description and the examples presented in Section 10.2.2, only partial orderings for the different processors are actually assumed, and they always exist for any physical processor system.

10.2.11 Total Store, Partial Store and Relaxed Memory Order

The *total store* (TSO), *partial store* (PSO) and *relaxed memory* (RMO) orders were introduced with the SPARC architecture [238, 255]. All processors of this architecture must implement at least total store order. The other consistency models are optionally implemented and must be explicitly activated.

All these models share the following properties. First, stores are always performed atomically, i.e., they appear to be performed in the same order for all processors, irrespectively of their memory location. Second, loads are never reordered with following dependent instructions, which read the destination register of the load or are conditionally executed depending on the load value. Third, for consistency, loads and stores that precede in program order a write to the same location are always performed before it. Loads on a processor can be satisfied with the value coming from the last store in memory order or from the latest local store in program order, as long as this doesn't violate the first property.

With total store order, the total order of stores must be compatible with program order. Loads are blocking and ordered with respect to previous loads in program order. They may not be reordered with subsequent stores[20]. Atomic read/write operations follow the rules for both stores and loads. Total store order is stronger than processor consistency[21]. With it, all stores are ordered and their order is a superset of the program order. Processor consistency, in its strongest version, just requires that the stores to a single memory location be serialized. In both models, loads are allowed to pass earlier writes. Total store order is also incomparable with strong ordering.

In the partial store order, the total order of stores is not necessarily compatible with program order. Partial store order is thus not stronger than processor consistency[22]. Load accesses still obey the same restrictions. Finally, relaxed memory order only imposes the above-mentioned restrictions common to all these three orders. With it, loads may be reordered with other loads.

Total store order recently regained some interest after Intel published a revision of the memory consistency model available on x86 processors. Starting from this document and by observing actual processor behavior, which is not completely consistent with the specification, some researchers have formalized a memory model called x86-TSO [200] that, according to them, reflects the actual intent and implementation of the x86 processors memory model.

10.2.12 Local Consistency

Local consistency [121] is the weakest possible memory model that is compatible with program order. It simply states that, from the point of view of a given processor, all local

[20]This requirement was not explicitly specified in an earlier version of the SPARC architecture manual [238]. But it is assumed by some of the example implementation code for usual synchronization primitives given in this document. The ambiguity was removed in the next version [255].

[21]First or second definition, see Section 10.2.5.

[22]Partial store order and the weakest version of processor consistency are actually incomparable.

memory operations appear to have been executed in program order. Interaction between processors is not constrained at all.

This model allows maximum sequential performance, because it is compatible with all optimizations that can be applied to uniprocessors. It is however useless without special synchronization operations that can offer minimum guarantees about the propagation of values between two processors.

10.3 Distributed-Shared Memory

The aim of *distributed-shared memory* sytems (DSM) is to present to programmers the familiar abstract interface of shared-memory systems, even on hardware where memory is physically distributed. The message-passing paradigm, which can unleash most of the performance of such architectures, is indeed widely recognized as being difficult to program because it is low-level. It requires the programmer to explicitly send data from one process to another. The programmer must elaborate the complete program's data flow and the program must keep track of the location of shared data, except for some restricted classes of computation, which the producer/consumer scheme is a common example of.

Even today, with multi-core being mainstream, lots of programmers are still unaware that memory is physically distributed under the hood in high-performance systems [2], because these systems still present the usual shared-memory-like interface. To avoid confusion, we advocate the use of the more general term of *global addressing* for this abstract interface, and keep the terms shared and distributed to qualify the physical repartition of memory.

There are several possible classifications of distributed-shared memory along their design, including the presented memory consistency model and the sharing structuration and granularity, and along their implementation, such as the data location and access, the coherency protocol and the implementation level (hardware or software) of all the mechanisms involved.

So many distributed-shared memory systems were proposed since the mid 80s that it is impossible to detail them all here. We have nonetheless tried to mention some of the most influential ones among those that combine mostly regular hardware and software management, since they favor portability and flexibility, two of our goals in building the Capsule approach.

10.3.1 Ivy

Ivy was the first proposal for a distributed-shared memory system based on virtual memory that would allow applications to benefit from locality of reference, even though the underlying memory is physically distributed. In other words, it uses local memory as a cache for the whole virtual memory shared by the participating machines. Ivy manages memory at the granularity of a page size, which enables it to rely on the usual virtual

memory mechanisms[23] available on single processor machines [75]. It sets the hardware page permissions so as to trigger page faults when it needs to perform an action at some memory access.

Other systems before it, such as CM* [242, 243] or Butterfly [21, 166], did global addressing by merely directing memory requests to the nodes holding the data, according to some static partitioning of the physical address space. Programs, which were using virtual memory, couldn't determine in advance if their accesses were going to be local or remote. Remote accesses were 5 times more costly than local ones, and severely degraded as the number of processors trying to access the same memory bank increased.

Ivy is targeted at loosely-coupled multi-processor machines, where communications have different characteristics than in tightly-coupled ones. Latency is much higher, because sending a message involves the virtual memory and networking layers of the OS. It largely dominates the cost of message transmission. At that time, sending a message of a thousand bytes didn't take significantly more time than for a few bytes. Page size was on the order of a thousand bytes, which also didn't cause too much contention.

Overall, data accesses are managed through page ownership. Only the current page owner can write to a page in order to maintain memory consistency[24]. When a processor issues a write, it has to acquire ownership of the corresponding page through a protocol before the write can perform. As long as processors are only reading the page, several copies of it can exist simultaneously. The system then must ensure that nodes are not using outdated page copies after the page has been modified. Ivy maintains consistency at each memory write[24].

Traditional algorithms to maintain consistency are divided into two main classes: *Invalidation* and *update*. Schemes of the invalidation class inform processors holding data copies that their copy has become invalid. On message reception, they simply discard it. Schemes of the update class send messages to all nodes holding a copy so that they can bring them up-to-date. The former class can be viewed as a lazy form of the latter, because new copies are not automatically obtained and requests to the node holding the original data will have to be issued later, usually at the next access to the same data.

With write update protocols, the simplest approach is to have writes trigger a page fault, so that the OS can detect and update page copies at a single write granularity. Obviously, a page fault per write is extremely costly and doesn't allow a process writing often to benefit from spatial or temporal locality. Ivy doesn't propose another mechanism for update and considers only invalidation protocols.

Shared data are accessed through regular memory operations. Initially, a page is mapped at one processing node. When a read access to a page not present is made, the hardware and the OS handle the resulting page fault. They ensure first that the page is not mapped read/write at another processing node, demoting an existing mapping to read-only. Then, they map the page read-only at the initiating node. Finally, execution is restarted

[23] Such distributed-shared memory systems are sometimes qualified as *shared virtual memory* systems (SVM).

[24] These design choices actually enforce a strong memory consistency, called sequential consistency, described in Section 10.2.1.

at the faulty instruction. Once the page is mapped read-only, every subsequent read access is serviced locally. A write access is handled in a similar way. If the page holding the data is not mapped or mapped read-only, the OS invalidates all other mappings of the pages on the other processing nodes. Then, the page is mapped locally with read/write access. Finally, the faulty write is restarted.

At page fault, Ivy must be able to locate the existing page copies to obtain a new copy and, in the case of a write access, to invalidate them. To this end, it explores several strategies to maintain ownership and copy information.

The first is the use of a centralized manager that runs on a predetermined node. It keeps track of the owner of every page, as well as the location of all copies (the *copy set*). When a node faults because of a read, it contacts the manager to obtain a copy of it. The manager, who knows the owner, locks the page entry and send a request to the current owner on behalf of the node, asking for a new copy. The owner sends a copy and sets its access right to read-only. The faulting node acknowledges reception of the copy to the manager, which updates the copy set for the page and release the page entry lock. The write fault case is similar, except that the manager, before requesting a copy of the page on behalf of the faulting node, invalidates all other copies. In this mechanism, the central manager synchronizes both the access to the owner information but also the access to the owner's copy. The manager can avoid to wait for the acknowledgement message by delegating serialization for page access to the owner. In this case, its availability is increased and it can process more requests per time unit. Concurrent requests for a page have to be queued at the owner. Still, the manager eventually becomes a bottleneck as the number of nodes increases.

The second strategy is to distribute the management of pages. It consists in simply mapping statically each virtual page to a processing node that acts as a manager for it. This mapping is realized through a hashing function. When a page fault occurs, the node uses this function to compute the managing node for it. The rest of the protocol is the same as in the centralized case. Experiments indicate improvements over the centralized manager when a parallel program exhibits a lot of page faults.

The third strategy is to have no managers. Each node knows if it is the owner of a given page. At page fault, the request for a copy or ownership of a page is broadcasted to all processors. Only the owner responds with a page copy, and possibly the copy set in case of ownership transfer. This strategy is nonetheless very costly. Experimentally, applications that cause a large number of page faults perform poorly. The systematic broadcasts incur a substantial performance penalty for as low as four processing nodes.

A variant of this strategy is to avoid broadcasts by having each processing node maintain a probable owner field for each page. Initially, those fields point to the true owner of the page[25]. At each page fault, the processor requesting a copy or ownership of a page sends a message to its probable owner. If the receiving node is not the real owner, it forwards the message to its current probable owner for the page. In any case,

[25]For example, at machine startup, pages can statically be assigned an owner. An alternative is to assign the mapping at program start, depending on its characteristics.

the receiving node updates its probable owner for the page to the requesting processor. The message eventually reaches the real owner.

For a given page, a tree can be constructed whose vertices are the processing nodes and whose root is the current page owner. An edge in the tree between vertices i and j indicates that node i's probable owner for the page is j. At startup, all nodes point to the true owner. By induction, it is easy to see that modifications to the tree keep it a tree whose root is always the current owner.

This tree allows to establish a mapping between this last Ivy's scheme and a class of set union algorithms [245]. These algorithms work on sets that are represented as trees. The vertices are the set's elements. The root vertex is an element used as a representative of the set. The defined operations on sets are the unite operation, which merges two sets starting from one element of each set, and the find operation, which returns the name of the set an element belongs to. During find operations, the path from the starting element to the root is traversed.

In order to optimize subsequent finds on the same elements or for elements on this find path, an algorithm can perform *compaction* of the path, by making an element point to an ancestor of its father. This has the effect of shrinking the path to follow to reach the root node. Another class of strategies are the *reversal* ones [245], by which the start node of a find operation is raised in the upper levels of the tree. With *reversal of type zero*, the start node becomes the new root of the tree. This variant corresponds exactly to the Ivy's scheme. The bounds given in Tarjan and van Leeuwen [245] on the number of nodes on find paths thus apply. In particular, the worst-case number of messages exchanged for m page faults on the same page is proportional to $n + m\log(n)$, where n is the number of processing nodes. A number of other optimizations presented in this paper could be applied to Ivy, changing the trade-off between the cost of retrieving a page in the short term, which essentially determines the data access latency, and the average cost of accesses in the long term.

We use a similar technique in CAPSULE to find the real location of a cell and transfer its content when dereferencing a link. It was described in Section 8.2.2. A difference with our current technique is that the traversed proxies are not updated to point to the new owner. This is not however a conceptual limitation. Some of the graph techniques presented in Tarjan and van Leeuwen [245] may be applied to CAPSULE's data structure management, as part of a future work.

CAPSULE has several advantages over Ivy. First, it uses objects of variable sizes, in contrast with Ivy's fixed size and coarse-grained virtual pages. This eliminates false sharing and the costs of transferring potentially unrelated data structures. Second, CAPSULE is aware of the links between the structures, which has important benefits, as was explained in Section 8.3.

10.3.2 Munin

Munin [24, 49–51] is an object-based distributed-shared memory system implementing the release consistency memory model[26]. It allows different consistency protocols to be used for individual objects[27] of a program to enforce this model. The programmer introduces hints on objects describing the expected access pattern. The compiler passes them to the run-time system, which uses them to select appropriate replication, migration and update strategies per object.

Munin supports 8 types of object use [24] that cover the majority of object access patterns in shared-memory programs [25]. An additional type serves as the default for objects that do not fit in the other categories. Write-once objects, also called read-only objects, are written only during initialization and read the rest of the time. Private objects are accessed by a single thread. Write-many objects are frequently modified by multiple threads between synchronization points. Result objects are special write-many objects whose content are updated by multiple threads operating on disjoint areas. They are read only when all the data have been collected. Synchronization objects are used to give threads a temporary exclusive access to some data. Migratory objects are accessed by multiple threads in batch, i.e., each thread performs a potentially long uninterruptible sequence of accesses before another thread accesses the object. They are associated to a synchronization object. Producer/consumer objects are written by one thread and read by several other threads, with the sets of threads accessing remaining stable. Read-mostly objects are hardly ever written. Finally, general read/write objects, also simply called conventional, are objects that could not be classified in one of the other categories. They account only for a small percentage of all data accesses.

According to these hints, the run-time system associates to each object several flags that indicate how it should be handled. A set of 7 flags was initially used [50], some of which were not independent, rendering the implementation quite complex to comprehend and maintain. Experiences also showed that such flexibility was not really necessary. The flags were later removed in favor of 5 different protocols [49, 51].

The *conventional* protocol ensures that, when a processor wants to write to an object, it is the owner of the sole copy of the object in the whole system. It does so by invalidating other copies before performing a write, in a manner similar to the Ivy strategy using probable owners. The *read-only* protocol handles a read-only object by simply replicating it on each node that tries to read data from it. Any write access after initialization triggers a run-time error. The *migratory* protocol ensures that there is a single object copy in the system. When a processing node accesses a migratory object, the object content is sent to it and the local copy is invalidated. Further accesses by the same node thus do not produce invalidation/update messages. The *synchronization* protocol handles synchronization operations on locks, barriers and conditional variables. It sends appropriate messages as soon as an operation begins, in an attempt to minimize synchronization

[26] See Section 10.2.7. The actual model is a bit stronger, but programs are not supposed to rely on it.

[27] In this context, objects are contiguous regions of memory, not "full-fledged" objects of object-oriented languages.

latency. Finally, the *write-shared* protocol handles the variables of the write-many, result, producer/consumer and read-mostly types. It is described in more details below, along with some optimizations applied to the migratory protocol.

All protocols are implemented in software to allow flexibility. Munin's run-time system needs to be informed of memory accesses to maintain object consistency. To this end, as in Ivy, it relies on the traditional hardware virtual memory facilities provided by the underlying hardware. The main consequence is that consistency is in reality maintained per page. Objects with different consistency requirements are allocated in different pages. If the OS running Munin doesn't allow user applications to register custom fault handlers, it has to be modified to properly call the run-time system.

The aim of the write-shared protocol is to reduce the number of exchanged messages and false sharing. The latter phenomenon is especially problematic for software page-based distributed-shared memory systems. Software fault handling has a much higher cost than misses in hardware caches. Communication latency between nodes is much higher than in tightly-coupled architectures. Changes have to be detected at a lower granularity than pages, which requires software processing, or whole pages have to be transferred. Write-shared mitigates these problems thanks to concurrent writers support, write buffering, compressed updates and timeouts to invalidate not-used copies.

At access to an object that is not present, a fault occurs and the corresponding page is requested from any node that is known to have a copy. When the copy is received, the page is mapped read-only and the access is resumed. In the case of a write, the whole page is copied into a *twin* page before being mapped read/write. This twin page holds the initial page content. It is used both to service requests for copies of the page and when updating other copies with the current page content. In the latter case, only the differences between the current page and its twin are transmitted. The object is also put in a *delayed update queue*.

Subsequent writes to the object are normal memory operations that directly modify the current page version. The latter thus acts as a buffer for modifications. When a release synchronization operation is performed, the node walks the delayed update queue to encode the differences between the twin and current pages for each object. Encoding directly takes place in the twin page to save space and time. Compressed updates are then sent to node holding object copies, which uses them to bring the page up-to-date. If a receiving node has a twin page for the object, because the local processes wrote to it concurrently, it can detect data races by performing a 3-way diff with the twin, its local page version and the received version.

To locate copies of an object, each processing node maintains an *object directory* of all shared objects. An entry in this table corresponds to a single object. It includes a lock to serialize concurrent accesses, the start and length of the memory area the object resides in, the owner for the object, the copy set, which contains the nodes that are probably holding a copy, the protocol used to maintain release consistency and some run-time statistics.

With the write-shared protocol, the copy set on a node doesn't necessarily contain all the nodes holding a copy. When a page copy is requested, the node servicing it adds the requesting node to its own copy set and send it along with the copy. The receiving

node merges the received copy set into its own. By induction, it is easy to see that the enumeration of all copies of an object can be performed by asking all nodes in the local copy set to return their own copy sets, then merging them into the local copy set, and continuing this procedure as long as some nodes in the local copy set have not been contacted. As an optimization, pages are directly mapped read/write at write and encoding is not performed when the owner node is the only node to hold a copy of an object (its copy set is empty or contains only a single reference to it).

The owner field on a node indicates an object's probable owner. The real owner can be found by following the chain of probable owners, as for the conventional protocol. The main difference is that the role of the owner in the write-shared protocol is to guarantee that at least one copy of the object is still held in memory. This copy is called the *copy of last resort*. Ownership hardly changes: In particular, a node trying to write to an object doesn't need to become its owner; it simply gets a copy of it.

In order to reduce the time spent in updating multiple object copies, an update timeout mechanism causes nodes to drop an object copy when receiving an update message, if the following conditions are met:

- The copy must not have been accessed since the previous update.
- It must not have been accessed for a long enough time period. A configurable threshold serves to evaluate this condition.
- It must not have any pending local modifications.
- If the node is the current object owner, it must first ensure that another node can take ownership.

When a node drops a copy, it informs the updating processor and sent it its copy set. The updating node merges the received copy set into its own and drops the other node from it. The dropping node also sets its owner field to the updating node. If it was the previous owner, the updating node becomes the new owner and changes its own owner field to point to itself. This is the only ownership transfer case.

The migratory protocol features two optimizations. First, a node that requests the associated pages to write into them can map them read/write as soon as they are received and doesn't need to create twin copies for them. This is because there is a single writer to the object at any given point in time: The object owner. Second, a simplified version of the freezing mechanism [91] prevents a just-acquired page to be requested by another node during a fixed time period (10 to 100 ms).

Experiments with 7 benchmarks showed that Munin can reach performance within 20% of the equivalent message-passing versions on workstations running the System V kernel [51] and connected with an Ethernet network. For programs relying on a work queue, which would reach only 55% to 65% of this performance, alternative implementations were developed, using new function shipping capabilities. Performing remote procedure calls in this case is a large improvement because only elements pushed or retired from the queue need to be transferred, whereas with shared objects, the whole queue has to

be transferred before a processor can operate on it. The new implementations reach performance within 5% of that of message-passing versions. These examples shows that mixing distributed-shared memory with message-passing is possible and effective. It also highlights that an efficient sharing granularity is sometimes smaller than that of the manipulated objects.

10.3.3 TreadMarks

TreadMarks [9, 82, 151–153] is a distributed-shared memory designed to run on commodity workstations on top of regular Unix operating systems. Its main contribution over Munin is the introduction of a lazy implementation of release consistency, called *lazy release consistency*. Another innovation is the use of new Unix facilities allowing page access rights and page faults to be handled from user space.

In release consistency, as explained in Section 10.2.7, accesses issued inside a critical section must be performed at release. The intent of this disposition is to ensure that subsequent accesses to a critical section using the same lock will see the previous modifications. A strict approach to release consistency leads to updating/invalidating the modified data copies at release. However, only the first processing node that will successfully acquire the lock afterwards really needs these modifications. When executing the critical section, it will in turn probably modify the same data, thus generating new update/invalidation messages in addition to the previous ones. Other processing nodes holding data copies will receive and process all these messages, thus seeing all intermediary object versions, even if they only need the latest versions when effectively acquiring a lock.

Lazy release consistency [151] consists in sending update/invalidation messages only to the node that will subsequently acquire the lock. No remote operations are performed at release. The acquiring processor sends a lock request to the current lock owner. The latter responds with a lock grant as well as a list of *write-notices*, indicating that some pages have been modified since the previous acquire. These write-notices do not necessarily contain the data modifications, which may be sent at a later time, depending on whether the chosen consistency policy is update or invalidation.

Program instructions executed on a processor are grouped into intervals. A new interval starts at each synchronization operation (acquire or release). In order to determine which modifications have to be applied, each processor maintains a *vector timestamp*. This vector contains, for all processors in the architecture, the number of the last interval whose modifications have been incorporated locally. In the lock request message, along with lock information, a processor transmits its vector timestamp. The receiving processor compares this vector to its own to determine which intervals have not been propagated to the requesting processor yet. If the interval for processor P_i in the requesting processor's timestamp vector is m and if it is n in the receiving processor's one, and $m < n$, then the modifications held by P_i in all intervals from $m + 1$ to n are transmitted to the requesting processor.

TreadMarks uses the virtual memory facilities provided by the hardware and the OS. As done in the write-shared protocol of Munin, it uses twin pages to store the original

content of a page to be able to generate a list of modifications from the current page version. However, compressed diffs are not computed at release as in Munin. A processor defers their creation until the modifications are actually requested by another processor or if new modifications for the page are received locally. Until they are generated, local modifications continue to accumulate in the current page version, including those taking place after a release. This technique reduces the number of diffs to create, which saves computing resources, lowers release latency and eliminates some messages. It is called *lazy diffing* [152].

TreadMarks uses a single consistency protocol supporting multiple writers for all data. This protocol is very similar to the write-shared protocol of Munin. The main difference is that it uses distributed management of pages, as described in Section 10.3.1, instead of the probable owner technique of Ivy and Munin. Each lock or barrier also has a corresponding manager by which all requests to it pass. The manager keeps track of the current lock owner and arbitrates between multiple lock requests.

Two consistency protocols can alternatively be selected: *Lazy invalidate* and *lazy hybrid* [82, 153]. In the first one, diffs are not sent at lock request with the write-notices. The node that is granted the lock invalidates its copies of the pages that are referenced in the write-notices. Subsequent accesses trigger page faults, which lead to diff requests being sent to the processor mentioned in the corresponding write-notices. Write-notices for a single page are always kept in order of creation and the corresponding diffs are applied in this order.

In the second protocol, some diffs are speculatively generated and transmitted along with write-notices in lock grant messages. Processors maintain a probable copy set for each page. Initially, the copy set contains only the page manager. As diff requests are received, the initiating processors are added into the copy sets of the relevant pages. Also, the processors designated in received write-notices are added to copy sets. Lock grant messages include diffs for pages whose copy set contain the requesting processor. This mechanism favors executions were the same pages are repeatedly modified inside critical sections by the same processors.

For each interval in which a page is modified, a diff is created. Diffs for a single page accumulate and are transmitted on demand to processors that access the page. Diff compaction, by which several diffs for a page would be merged in a single one, generally cannot be performed. Each processor keeps a copy of a page it has accessed once, even when the page has been invalidated by a write-notice, to be able to later apply diffs instead of requesting the whole page content to be transferred. Diffs can only be merged when it is known that other processors that will request them didn't modify some pages concurrently. Compaction is thus not implemented because it is complex and would have other drawbacks [153]. The consequence is that diffs tend to pile up for a heavily modified page. They are reclaimed at barriers through garbage collection.

Experimental results with 8 benchmarks show that lazy release consistency implemented with a lazy invalidate protocol outperforms an eager invalidate protocol, where invalidates for copies of modified pages are sent at a release operation. In this latter protocol, when copies to be invalidated are found modified, a diff is created and sent back

to the invalidating processor, which incorporates the diff in its own copy. Eager invalidate generally performs better than eager update [82], where pages are updated rather than being invalidated. For the record, Munin's write-shared protocol is an adaptive protocol combining the advantages of eager invalidate and eager update. Improvements brought by lazy invalidate over eager invalidate vary depending on the application: The speedup of most of the benchmarks improve by 20%, whereas for some in which objects exhibit a migratory access pattern, lazy invalidation performs worse. Lazy hybrid improves performance over lazy invalidate, but only by a few percents on average.

10.3.4 Other DSMs with Relaxed Consistency

Midway [26, 28, 265] is an environment providing entry consistency semantics and a global addressing view of distributed-memory machines. Entry consistency was described in detail in Section 10.2.8. Midway is composed of a compiler accepting common languages augmented with some keywords and function calls, and a run-time system responsible for thread management, synchronization operations and memory consistency.

Annotations serve to distinguish synchronization variables and shared data. Each shared variable must be associated to at least one synchronization variable supposed to protect it through run-time system calls. Since synchronization primitives are usually provided through a library, the latter can easily be changed to include the relevant annotations. The only indications that must still be included per program are the synchronization primitive/shared data associations. As discussed in Section 10.2.8, this is an additional requirement compared to distributed-shared memory systems implementing release consistency. In order to allow a graceful transition from data-race-free programs to programs annotated for entry consistency, Midway also supports release consistency or processor consistency [28].

Automatic update release consistency (AURC) [131] is an implementation of lazy release consistency with hardware support. The network interface at each processor keeps a list of several local virtual memory pages. It also has a mapping between each local page in this list to a single remote page. It uses this information by snooping on the system bus to detect writes to local pages in its list and to propagate them automatically to the corresponding remote pages. A prerequisite is that the processors access pages with remote mappings in write-through mode, causing the writes to appear on the system bus.

In order to maintain consistency, the processor holding pages that receive automatic updates must be able to know when updates have been completed. To this end, the network delivers messages in the order in which they are sent. A release synchronization operation triggers a flush of all local updates. Every page includes a flush timestamp vector similar to the timestamp vector used in lazy release consistency (see Section 10.3.3) which tracks the latest modifications by every processor to the page that have been sent through automatic updates. The flush timestamp vector is changed after data modifications are performed on the page, so that the receiving processor see its new version after the data updates. The receiving processor compares the flush timestamp vector on its copy of the

page with its regular lock timestamp vector, which indicates which version of the data should be accessed according to release memory consistency. When both agree for a page, it knows that the updates have been received.

When less than two copies of a page exist in the system, the network interface is configured so that every update to a page is automatically propagated to the other page. With the fine-grain detection and automatic updating of data, AURC doesn't need to hold twin pages or compute diffs. When more than two copies exist, one processor is declared the owner and all other copies are configured to send automatic updates to its copy. In this second case, faults at non-owner nodes trigger a page fetch. Experiments show that AURC reduces the number of page faults, particularly in the case of pages that are shared by two processors only, for which there are no page faults. Also, the cost of a page fault is generally reduced, because no diffs have to be computed and then applied. The gain can be even more significant for applications in which diffs for a page tend to be numerous because of multiple successive updates, in which case transferring them quickly becomes as costly as a whole page transfer. Experiments show that AURC increases the network traffic, but reduces contention on the bus between a processor, its network interface and its local memory.

An implementation of scope consistency is proposed in Iftode et al. [132]. It uses the AURC techniques. The main difference is that, with scope consistency, only writes performed with respect to a scope have to be performed when a new session is opened[28]. This is achieved by scopes maintaining an *incarnation number*, which is the number of sessions that were opened and closed for a scope. A processor also keeps track of the pages modified during each local session and the incarnation number of the last session it executed. At acquire, this number is communicated to the previous lock owner, which sends back the list of the pages modified in sessions having a greater incarnation number. In other words, only the pages modified in sessions that occured later than the last session on the processor are invalidated.

The home-based lazy release consistency protocol (HLRC) [267] uses the same update policy as AURC, but implements it purely in software. Updates are sent to the home node for a page at release through diff generation. As soon as the diff is received and applied at the home node, it is discarded. The advantages of HLRC over LRC is that diffs are not retained, implying that no garbage collection is necessary. A processor acquiring a lock also doesn't need to communicate with potentially multiple processors as page faults occur in the critical section.

10.3.5 Fine-Grain Coherency

A later extension to Midway [265] introduced software write detection through compiler and run-time system support, instead of relying on the virtual memory facilities. This latter approach incurs the overhead of page fault and twin page creation at first write, and the time to scan both the twin and the current version of a page to produce diffs, which consumes memory bandwith. It is thus efficient only if it can be amortized thanks

[28]See Section 10.2.1 for a description of scope consistency and the associated terminology.

to a high number of writes before a synchronization operation. Software write detection's overhead is significantly lower [246, 265], but takes effect for each write. Therefore, the software approach allows for a more efficient support for most programs with fine-grain synchronization.

Dirty bits are associated to shared data and are modified at write thanks to instructions emitted by the compiler. Memory is organized into regions of varying size and different sharing granularity. The smallest unit of granularity is called the *software cache line*. The first page of a region contains special routines that know the sharing granularity and can update the proper dirty bit at write. The routines for private regions do nothing and immediately return to the caller.

When another processor is granted a lock, it is communicated the modification's logical timestamps of all software cache lines associated with the lock, along with the current logical timestamp of the last processor holding the lock. It then changes its own logical timestamp to the maximum of its old value and the newly received one. Processors always keep the logical timestamp of the software cache lines they currently hold and transmit them with lock requests. This enables them to receive only pieces of data that have been modified at a later timestamp than the copy they hold.

Blizzard [229] is a distributed-shared memory implementation on top of the Tempest interface [212]. The goal of Tempest is to provide a stable interface for fast user-level communications and for controlling the semantics and performance of virtual memory. Virtual memory mappings as well as fine-grain access control are possible with this interface. Blizzard maintains sequential consistency per block of 32 bytes. Three versions were presented. The first, Blizzard-S, is an all-software implementation. It inserts access control code before each shared-memory access into program executables. The virtual address of access, appropriately masked, is looked up into a state table. Depending on the access type (read or write) and the state of the accessed block, a particular handler is invoked. The size of the coherency block can be changed but only at compilation. Blizzard-E is a hardware implementation that uses the ECC correction bits of the CM-5 as well as MMU support. Blizzard-ES is an hybrid version of Blizzard-E and Blizzard-S. Blizzard-S is typically slower than Blizzard-E and Blizzard-ES in experiments by 27% to 108%.

Shasta [224, 225] is a software distributed-shared memory system with two levels of granularity. Lines are indivisible units, whose length is configurable at compile-time and is typically 64 to 128 bytes. Blocks are coherency units composed of multiple lines. An application can define several shared virtual memory ranges with varying block size values, allowing it to tune the coherency unit to particular data sizes and access patterns. Coherency is maintained through a directory-based invalidation protocol.

A *state table* on all processors indicate the sharing state of each line among the invalid, shared and exclusive possibilities. All lines in a block have the same sharing state. The fixed size of lines makes it easy to compute the relevant entries in the state table and in the directory given a line's address. Each virtual page is assigned a home node. All processors maintain a directory holding sharing information about the lines that are in the pages they manage. A directory entry for a line contains the current owner, which is

the processor that last issued an exclusive access on the line, and a bit mask indicating which processors currently have a copy of the line.

The main contribution of Shasta is a number of optimizations to implement efficient load and store miss checks in software. The first optimization is to move the code computing the address of a state table entry before the instructions that use it. These instructions can even be mixed with the regular program instructions that precede the memory access.

Second, the flag technique allows most loads to valid lines to complete without even accessing the corresponding state table entry. Longwords for locations that are invalid are set to a special value. The load check code simply loads the value and, if it is different from the special value, allows the program to proceed with it. If the value is equal to the special one, the state table entry is accessed to verify whether the line is really invalid. For an appropriately chosen special value[29], the first check allows the program to continue right away most of the time. This optimizes the common case of a valid line, because no cache misses are suffered to access the state table.

Third, store checks can be optimized by accessing an exclusive table, which indicates with a single bit per line whether a line is valid. This table is partially redundant with the state table. Its advantage is that a line state is represented with one bit. More line states can thus be held in a single hardware cache line, lowering the number of misses compared to a regular state table entry access for the program parts exhibiting spatial locality.

Four, checks for stores and loads to an address range whose size is smaller than a line can be batched ahead of a code block, which then allows the normal memory operations in the block to be compiled without checks.

The result of these optimizations is to reduce the check overhead compared to the original non-instrumented executable from 83% to 21% on average [225]. Further experiments [224] show that the empirical detection of migratory patterns is not beneficial for 64-bytes lines.

Shasta has been extended to support cluster of SMP workstations [223]. The potential benefits of such a particular support are faster communication between processors on a same node, intra-node data sharing in hardware, and the elimination of some remote requests (data fetched by one processor is available to the other processors on the node). The main difficulty is to allow both for fast inline checks and multiple concurrent accesses by processors on a single node. A memory operation and its preceding check are not atomic, and making them appear so thanks to a lock would cause a large overhead for operations that are expected to be satisfied with local data the majority of the time.

This problem manifests itself for state downgrades, where a local copy is invalidated because of a write request. A processor having performed a check and preparing to read a valid value may read an incorrect one if it is invalidated in the meantime and the flag technique is used. A solution is to have downgrade operations post messages to the processor holding copies. The receiving processors treat these messages at some special program points, as they do for the regular messages exchanged in the original

[29]The value should not be zero nor any small integers.

Shasta system. This processing never happens between a memory operation and its check, because the compiler doesn't insert such code at these places. Processors finally acknowledge these messages, allowing the initiating processor to know that the downgrade is complete.

Two optimizations allow to reduce the number of downgrade messages to send and contention on the state table. The latter is indeed shared and entries in it are accessed with a lock associated to the block a line being looked up belongs to. First, each processor on a node maintains a private state table, that maintains the same state as a processor would in the original Shasta. Even if another processor in the node accessed a line, it can still be marked invalid for a given processor. On a memory access to such a line, a miss will occur, trigerring a check of the shared state table. The processor will notice that a copy exists locally, and will update its private table accordingly. A processor servicing a request that requires some downgrade can consult the private tables in order to determine if another processor accessed a line and considers it as been local. Second, when executing protocol code, a processor sets a flag to indicate it is not executing user code. With proper synchronization, other processors can directly modify its private state table instead of sending a downgrade message.

10.4 Distributed Objects

Distributed objects are also environments aiming at abstracting programs from the potentially distributed nature of the underlying hardware. They achieve this goal by conferring a central role in data management and distribution on objects. In other words, they perform a similar job as distributed-shared memory systems, but with coherency units linked to the manipulated object sizes. Operations on data are not viewed at the level of read or write accesses to memory words but rather at the higher and more abstract level of methods performing arbitrary operations on objects.

Some usual advantages of these approaches are an adaptive granularity better suited to the application data, an easier integration to existing languages, and especially object-oriented languages riding the popular trend of encapsulation. The common potential drawbacks are the need to modify old applications written in imperative or functional languages and the fact, as demonstrated by Munin or Shasta, that objects are not always the appropriate coherency unit to obtain good performance.

We also group in this category a proposal that is not based on full-fledged objects but rather memory regions, which we consider to be some kind of precursors to objects, much as structures in C form the basis for the C++'s object implementation.

10.4.1 Emerald

Emerald [32, 143] features distributed objects implementing abstract interfaces. Objects are defined by a unique name, a set of operations, a representation (the data stored in the object, including references to other objects), and an optional process that, if present,

executes independently and in parallel with invocations to the object. To ensure mutual exclusion, shared variables and operations on objects are declared inside a monitor.

Objects reside at a single location at a time, except for immutable objects that may be copied and held by several nodes. Emerald transparently supports remote invocations and object movement. Arguments to method invocations can be passed with the call-by-move strategy, which moves them to the node servicing the invocation, avoiding a subsequent remote reference from this node. Objects are moved back when the call returns.

Each node maintains an *object table* mapping object names, known as *unique ids*, to a local object descriptor. This descriptor contains a *forwarding address* and a local data area pointer. When the object is moved to another node, the local area pointer is set to `NULL` and the forwarding address indicates the new node the object resides in. Objects use local pointers to inner objects and to the descriptors of other objects. As a node moves, the inner objects are moved and all pointers in the object have to be updated. Pointers to inner objects must point to their new storage area and pointers to object descriptors must be updated to local descriptors on the new node. The compiler generates a *template* for each concrete object's data area indicating which parts are references to other objects. The migration code uses it to perform pointer translation as an object moves.

Emerald doesn't try to hide distribution from the programmer. Some applications may benefit from tuning data distribution, while still using transparent remote invocations. To this end, Emerald provides several location primitives. `Locate` determines on which node an object resides. `Fix` imposes a new fixed location for an object. `Unfix` allows an object to move again after a `Fix`. `Move` moves an object to a new location. To designate a location, the program can pass as an argument another object to stand for its current location, or an explicit node object. Node objects are abstraction of the underlying machine, and need not reflect exactly the hardware. All objects on an abstract node must share the same address space.

Another interesting aspect of Emerald is that, contrary to previous languages, it proposes a single set of syntaxic constructs to define and use objects, whatever their use may be. The compiler chooses a proper implementation depending on code analysis and attributes, such as immutability. As an example, objects determined to be local are accessed directly through regular pointers and do not have object descriptors.

10.4.2 Amber

The aim of Amber [62] is to explore the possibility of using loosely-coupled multi-processors as a single large-scale machine by providing language support for concurrency and distribution. As Emerald, it does not try to hide the object distribution, considering that the best distribution policy is application-specific. It is built on top of a subset of C++ and is a descendant of the PRESTO framework [27]. It provides a set of classes to manage threading, synchronization and distribution.

Threads are managed through thread objects having `Start` and `Join` methods. Threading is designed to be inexpensive, allowing applications to create as many threads as logically needed, regardless of the number of processors of the underlying architecture.

Threads are scheduled by the default Amber's scheduler, which supports time-slicing and priorities. An application can install instead its own scheduler if it has peculiar needs. In the end, these user-declared threads are executed into threads of the underlying OS.

Amber's data placement primitives are similar to that of Emerald. A difference is that Amber doesn't provide call-by-move semantics. Object placement is never performed by the run-time system; it has to be specified by the program. Amber also avoids doing pointer translation by ensuring that the address space of each thread is the same on all nodes. When objects are moved, they reside on the new node at the same virtual address as in the original node. Each node starts with a private heap region from which it allocates local objects. Private heap can be extended by asking for more address space from a central manager. This choice allows to avoid complex compiler or run-time system support and facilitates process migration[30].

Although the address of an object is valid on all nodes, the object content may not be locally present. A processor can check this condition by using the object descriptor that is the first field of a record actually stored at the object's address. The record contains the descriptor and room for the object's representation. The descriptor includes a flag to indicate if the object content is local. If it is, the object content is stored in the representation area. If it is not, the area stores instead a forwarding address, whose purpose is to indicate the probable node hosting the object[31].

Remote method invocation actually moves the invoking thread to the node holding the object. The implementation is to transfer the stack frame representing the invocation. For multi-processor nodes, the check of the object descriptor's locality flag and a subsequent local access are not atomic and may be interleaved with an incoming request of object migration serviced on another processor. Instead of using a lock to protect access to the descriptor and the object, Amber favors the local access case by just performing a simple read and avoiding expensive synchronization for this code path. It preempts and reschedules all threads on a node receiving a move request, making them check again for local availability of the object associated to their current stack frame.

10.4.3 Orca

Orca [16, 18] was designed from the ground up to support coarse-grain parallelism on loosely coupled distributed systems. It presents to programs a shared passive *data-object* model, as described in Section 5.3.1, but the data-objects are in reality distributed over the underlying architecture. With the goal to improve performance through *object replication*, several object management policies have been implemented and evaluated in Orca's run-time system [19]. The design space for such an approach includes the mechanism that preserves consistency when writes occur, the protocol chosen to implement it and the number and location of object copies.

[30] The main difficulty is to translate pointers possibly stored in registers, which are not a priori distinguishable from data.

[31] See the description of forwarding addresses in Emerald in Section 10.4.1.

Replicating an object speeds up read accesses for nodes that hold a copy because the content is readily available in local memory and no communication needs to take place at access. When the object is modified, the system must make sure that the other nodes don't keep accessing older inconsistent copies[32]. As explained in Section 10.3.1, two schemes are possible to ensure consistency: Invalidation or update. The former scheme has the benefit that invalidation messages are short compared to update messages, making write accesses faster. On the other hand, it increases latency and network traffic for subsequent read accesses.

Contrary to the Ivy distributed-shared memory system[33], for which a simple update scheme would require a page fault to be generated at each write operation, Orca implicitly groups accesses performed by a single method because of the indivisibility principle[34]. It thus updates an object only after a method modifying it finished, i.e., after all its potentially numerous writes were issued. Moreover, the update can be done by transferring the object's content only instead of a whole page, or the arguments to the method that modifies the object, in which case the update is performed by executing the method remotely.

Replicating objects is done for performance reasons and is thus also dependent on the protocol chosen to implement a particular update scheme and the underlying hardware facilities. Several alternatives are studied [19], based on whether efficient broadcast hardware support is present. It is shown that, in the case where only point-to-point RPC is available, the cost of invalidating/updating N copies of an object grows linearly with N. Partial replication to the nodes where processes really access the object limits this growth in practice. For the invalidation scheme, new copies are created automatically at each access. For the update scheme, the node holding a primary copy maintains statistics on the read/write ratio of each other node and decides to assign a copy to a node when its ratio exceeds a predefined threshold. Nodes holding secondary copies also maintain local statistics of the number of reads and writes and the run-time system uses them to decide if a local copy should be dropped. Writes are always transmitted to the node holding the primary copy. Migration of the primary copy is merely evoked, but no implementation details are provided. It is unclear if it has been implemented in this case.

When a reliable and efficient hardware broadcast is available, it is shown that invalidation or update messages can be broadcasted in constant time, at least up to 10 processors. Unless some nodes continuously write into some objects without reading them, which may be the case in a strict producer/consumer pattern, the updating scheme leads to less messages exchanged than the invalidation scheme. Both schemes perform a broadcast for the update/invalidation messages, but only the latter causes nodes that perform a subsequent read access to issue a point-to-point RPC to the node holding the primary copy. For this reason, in the broadcast hardware support case, only the update scheme is considered in the Orca literature. The chosen policy is to simply replicate all objects on

[32]This strong requirement enforces sequentially consistent accesses to objects (Section 10.2.1). Other consistency models are possible (Section 10.2).

[33]See Section 10.3.1.

[34]See Section 5.3.1.

all available nodes. Processes perform read accesses directly on their local copy. Write accesses, on the other hand, lead to the broadcast of the object's new content to all other nodes. For objects that are accessed by few remote processors, this strategy induces a write overhead that could be avoid by disabling replication. In the same case, both the write overhead and the message processing delays on the receiving processors could be lowered by using partial replication and performing RPCs instead of broadcasts. As an example, a broadcast costs 3 to 4 times a single RPC message on the experimental hardware used in Bal et al. [19].

A later paper [17] considers an enhanced distribution scheme to be used in conjunction with efficient hardware broadcasting. The compiler determines the pattern of access to a given object by each process. This static analysis is facilitated by the fact that the Orca language doesn't include pointers, global variables or `goto` statements. From a pattern, the compiler computes the numbers of read and write accesses to an object per process and includes them in extra parameters of `fork` statements. When a fork occurs, the run-time system broadcasts the corresponding event to all processors, allowing them to permanently maintain the same current estimated numbers of reads and writes by each processor to each shared object. From these numbers, the run-time system takes the object management decisions. Because all processors have the same view of the global situation and use the same algorithm, they all come to the same decisions and perform them independently.

The decision-making algorithm works as follows. Let us call R_i the total number of read accesses and W_i the number of write accesses to be performed by processor P_i on a given object O. After each fork, if one of the R_i or W_i for all i changes, the run-time system considers again its current management policy for object O. The two considered policies for O are to replicate it on all nodes or to have it in a single node's memory, which must be determined. In both cases, the number of messages to be exchanged according to the R_i and W_i is computed. For the full replication case, reads do not generate messages but each write causes a broadcast. In total, the run-time system will send $\sum_i W_i$ broadcasts. For the single copy case, all reads and writes generate a RPC message, except when issued on the processor owning the object. For all j in turn, P_j is considered the owner and the number of exchanged RPC messages, $\sum_{i \neq j} R_i + W_i$, is computed. The run-time system finally finds the minimum of all the sums weighted by the cost of the message type used (broadcast or RPC), which determines the policy to use.

This decision-making process has a relatively low overhead in the experiments for two reasons. First, all processors come up to the same decision with simple operations on the statistics. Second, Orca is targeted at coarse grain parallelism, in which `fork` statements are infrequent. In contrast, CAPSULE aims to exploit all the available parallelism in an application, including fine grain one, and is designed to run on many-core architectures. The number of operations Orca's run-time system has to compute before making a decision linearly increases with the number of processors. A single broadcast message also becomes more costly with a large number of processors. In this context, a distributed approach will probably be more appropriate. Finally, with a large number of tasks being created

dynamically during a program's life, Orca needs to assess frequently its object management decisions. For all these reasons, Orca's mechanisms will probably have to be adapted to bring performance improvements in CAPSULE.

Orca enforces a strong memory consistency assuming that all memory accesses are serialized. This is implemented in the run-time system by using *reliable broadcasting*, which guarantees that all broadcasted messages are delivered in the same order to all nodes. Thus, all processes see the exact same sequence of values for all objects, although they may not see a particular set of values for different objects at exactly the same time. Orca was initially developed [18] on top of the Amoeba operating system, which provides reliable broadcast through group communication [144] on top of hardware broadcast using the FLIP protocol [145]. It was then reimplemented in a layered fashion [20], with the run-time system using a lower layer, called Panda [29], whose role is to provide a machine-independent interface to multi-tasking, RPC and group communication. This new software architecture made the Orca run-time system more portable, since adapting Orca to a new machine amounts to porting Panda. Panda itself requires only bare message-passing and multi-tasking facilities, which are provided by the underlying OS or libraries, but can also take advantage of more sophisticated communication mechanisms, such as hardware support for broadcast and/or reliable communications, where they are available. Several algorithms to implement reliable broadcasting on top of mere RPC can be found in Bal et al. [18].

10.4.4 Charm++

Charm++ is an object-oriented programming environment able to run applications indifferently on shared-memory and distributed-memory architectures thanks to a single programming model. Its base characteristics and the implementation of its run-time system in the context of shared-memory machines were presented in Section 5.3.2. In this Section, we detail the additional mechanisms needed to support distributed-memory architectures.

Charm's message-passing makes a program portable to both shared-memory and distributed-memory architectures. Its message-driven specificity is especially important for the latter, because it hides the even higher latencies they introduce.

Chares can interact with BOCs with a regular function call instead of having to send a message to them. Besides SPMD-style programs, BOCs are useful to implement distributed data structures. Users call regular functions locally and the distributed management of the structure is transparently done by the branches through messages. Branches need not even communicate with other branches, in which case the BOC serves to provide uniform local services. In the end, this is equivalent to duplicating several pieces of code on all processors. Finally, distributed services can be implemented using BOCs, such as the Charm load-balancing system modules themselves.

In theory, message-passing is enough to write any parallel program, but the resulting language is inexpressive for some common cases, for which shared-memory programs can use traditional data structures. Charm proposes different information sharing abstractions

to the programmer through *abstract data types*. They provide a well-defined interface which the programs must use to valuate and manipulate the associated objects. These interfaces include read-only or write-once variables, distributed tables, accumulators and monotonics. All these objects are implemented through BOCs on distributed-memory machines.

Read-only variables are initialized at application startup and only read afterwards. Write-once ones differ in that a program can initialize them during the execution at any time. A distributed table is a hash table mapping a record containing an integer key to untyped data. Its content is potentially distributed on several processors. It provides 3 methods: `Insert`, `Delete` and `Find`, which are self-explanatory. All these methods are asynchronous. A call to `Find`, in addition to the searched record, must also specify a chare and an entry point, to which the result will be sent as a message. Accumulators are pieces of data that support a single operation that is associative and commutative, which permits the local accumulation and graduate propagation of values on distributed-memory machines. Monotonics are special accumulator cases in which updates are idempotent (applying an update several times gives the same result as applying it once) and for which the value is increasing or decreasing consistently at each update. For example, this type of variables is useful for branch-and-bound computations which store the best result so far.

Chares and objects can be distinguished through identificators (IDs). On shared-memory architectures, the ID is the address at which the object is stored. Charm's implementation for distributed-memory machines is the usual couple composed of a processor number and the address of the designated chare or object in the processor's address space. Another implementation would be needed to support object movement.

Because, in distributed-memory, there can be no global message queues, the run-time system has to perform load-balancing. Messages to chares, in the Charm implementation, are sent to and handled by the processor holding the chare, and chares never move once assigned to a processor. Consequently, the load-balancing performed by Charm concerns only new chares messages. The run-time system chooses the processor to which a chare is dispatched as soon as it is created, and before it may start to receive messages. The load on a processor or a node is evaluated by the number of messages waiting to be processed in its message queue [148]. The Charm library provides several load-balancing strategies. A user may choose among one of these strategies or may provide its own, better suited to a particular application.

The random strategy sends new chares to processors randomly. In the contracting within neighborhood (CWN) strategy [148], processors piggyback their current load on messages exchanged with their neighbors. A chare creation message is dispatched to the least loaded neighbor node, and the process continues until it reaches a node which has the minimal load in its immediate neighborhood. Two parameters influence this gradual message transmission. The radius specifies the maximum number of hops for a given message. When this number is reached, the node processing the message must accept and enqueue it. The horizon, by contrast, specifies the minimum number of hops a message has to travel. This technique allows a message to eventually reach a less loaded node than

the current one, even if the latter has a locally minimal node. Once a node accepts a chare creation message, the message is enqueued locally and doesn't move further. Adaptive CWN (ACWN) [235] is a refinement, inspired by the Gradient Model [174], in which the effect of the horizon parameter is mitigated. A new parameter, the minimum load, permits to classify nodes as idle or busy nodes. Idle nodes never try to dispatch a message to neighbors if the latter are all classified as busy.

For distributed-memory architectures, nodes that do not share some common memory must physically send a message to communicate. Messages are passive entities that contain application-specific data. Since they may reference local data through pointers which are not valid accross nodes, they have to undergo a special treatment when sent. Charm permits the application to associate packing and unpacking methods to a message type. The packing method is automatically called when a message is going to be sent. Its goal is to replace the initial content with self-standing data that include all the content the original message references. The node receiving the message calls the unpacking method on it to reconstruct the appropriate data structures. This approach has two main benefits. The first is that the application tunes message packing/unpacking according to its needs, without the run-time system imposing a particular structure. The second is that packing and unpacking are called by the run-time system only if the destination chare resides on a different node than the sender. This property optimizes message sending by avoiding unnecessary marshalling when the run-time system has decided that both objects should reside on the same node. This is useful for clustered architectures or multi-processors of multi-cores in which all cores of a die share the same memory.

10.4.5 CRL

CRL [141, 142] is an all-software distributed-shared memory implemented as a library. It requires no special support from the hardware, the OS or the compiler and is intended to be language independent. Shared access is provided through *regions*, arbitrarily sized contiguous areas of memory. A particular region is identified using a unique global region identifier. Once a region has been created using the `rgn_create` primitive, a process has to map it into its address space to use it, with the `rgn_map` primitive. It also has to specify whether it is going to perform only reads or reads and writes to the area, using `rgn_start_read` or `rgn_start_write`. It must then inform the run-time system when the operations have all been performed by using the corresponding `rgn_end_read` and `rgn_end_write` primitives. Finally, regions can be unmapped through `rgn_unmap`.

A given region may not be mapped at the same address for all processes or for a single process as it repeatedly maps and unmaps it. References to shared data thus must include the region's identifier and can't consist of a pointer alone. The run-time system implicitly synchronizes concurrent accesses to a region: A read/write access by a process is exclusive to all other accesses, whereas read-only accesses can be granted simultaneously to several processes. The programmer thus doesn't use explicit synchronization objects, but rather implicitly uses an equivalent of a read/write lock associated to a given region.

Most of the characteristics of CRL are also found in Capsule's distributed-memory

support. The latter requires the programmer to explicitly call the run-time system to signal that a cell is going to be accessed. Subsequent calls must then be made to state which kind of access the program wants to perform. Synchronization is also performed automatically, based on the primitives' call sequence.

However, the distributed-memory support of CAPSULE goes farther than CRL's memory regions with the central concepts of cells and links, which introduces important benefits. First, memory management is entirely performed by the run-time system automatically. CAPSULE applications do not need to map or unmap a region or a cell. It is instead the run-time system that is responsible for deciding how many copies of a cell exist and where they are stored. Second, the CAPSULE distributed-memory support provides links as objects to reference cells. The programmer doesn't need to explicitly refer to data by using an extra global identifier; he simply uses a link[35] instead. Third, there is no notion of "home node" for a cell in CAPSULE, thanks to the proxy mechanism (see Section 8.2.2). Data access is completely decentralized and doesn't arbitrarily cause contention on nodes that have created most cells and/or the most accessed ones. Four, as discussed in Section 8.3, knowledge of the relations between cells can be leveraged to improve performance.

CRL features some global operations that have not been implemented in CAPSULE for now, such as reduction on floating-point values. However, such features barely introduce a performance advantage [141]. Also, it only supports global barriers, which are enough for benchmark programming but are not in the case of real applications. CAPSULE's groups are much more expressive because they allow to implement partial barriers, enabling several parallel algorithms to operate concurrently (see Section 2.4). Finally, CRL has no support for tasks and thus doesn't support dynamic and recursive parallelism, whereas CAPSULE's distributed-memory support is integrated with the base tasking primitives.

CRL's performance is comparable to the available hardware-implemented distributed-shared memory at that time (1995), for benchmarks that have a low to moderate ratio of memory accesses over computations. As an example, Barnes-Hut was the benchmark with the highest ratio (436 cycles of useful work per access, consisting in a 100 bytes region on average). Some experiments were conducted on an Alewife machine with a simple round trip costing about 500 cycles and the bandwidth being around 18 megabytes per second. They gave for Barnes-Hut with 4096 bodies a speedup with CRL of 14.5 on 32 processors, which was 12% lower than the speedup obtained with Alewife's hardware distributed-memory support activated.

To put these old results into perspective, let us recall that today's multi-core machines have a bandwidth on the order of 10 to 100 gigabytes per second and data access latencies varying from 1 to 100 nanoseconds [185, 230]. Both latency and bandwidth have considerably improved, but the latter quicker than the former. We intend to promote the use of CAPSULE for all kind of general-purpose applications, in which the ratio of memory accesses over computations is likely to be much higher.

[35] Unless a particular field or data member inside a cell is the reference's target, in which case its offset in the cell must be stored.

Part III

SiMany: Simulating Many-Core Architectures

Chapter 11

The Need for Many-Core Simulation

With the aim to assess how effectively our run-time system can handle data on distributed-memory architectures, based on the principles and mechanisms exposed in Part II, the need for a simulator arose for several reasons. First, the currently available distributed-memory architectures hide their distributed nature to programs thanks to global addressing. Second, although many company and academic roadmaps [134, 250] had been mentionning up to hundreds of cores per chip, there were, at that time, no publicly available simulators able to sustain such a number of execution units while keeping simulation time manageable on commodity hardware. Using direct execution on a cluster of machines wasn't a solution either, since inter-machine links' latencies are considerably higher than that of inter-core links. Finally, hardware variability is a highly desirable feature in order to study whether the mechanisms can adapt to a wide range of topologies.

Besides programming model validation, simulating many-core machines has become in the past years a major challenge to the whole hardware community as well. Using cycle-level simulators to help designing chips, as designers have done for decades, is simply impractical for many-core architectures. The fastest software-based simulators [45] indeed introduce a slowdown on the order of 10^4 to 10^5 over native execution. A recent contribution [66] improved the situation (1000× slowdown) by delegating part of the simulation (the timing model) to some specialized hardware (FPGAs). Even in the optimistic case where this approach would become feasible for hundreds of cores, the slowdown for simulating them would still be of the order of 10^5. A single second of execution thus would translate into almost 28 hours of simulation and a minute into 2 months!

Because cycle-level simulation of large single-core core architectures was already considered slow enough to hinder the design process, the community has come up with a number of efficient solutions to speed up simulation for single-cores. SMARTS [259] and SimPoint [234] demonstrated that sampling can improve simulation speed by a factor of 10^3, while producing errors varying from 1% to 10%. SimFlex [256] brought significant performance for multi-cores running server-type workloads, where all threads operate

independently, with 10^4 to 10^5 speedups. Statistical simulation and synthetic benchmarking [84, 195, 199], which recreate a similarly-behaving but much smaller trace or benchmark, are also successful alternatives for single-cores. They bring speedups up to 10^5 with an error varying from 5% to 10%.

However, there are no sampling nor statistical simulation techniques mature enough for general parallel workloads and many-cores yet. Earlier proposals require to simulate an exponentially growing number of program *co-phases* [31, 154], which reduces the attainable speedups (less than 10^2 for 2 cores). Recently, Namkung et al. [191] showed that relaxing similarity constraints when clustering samples can practically mitigates this growth up to 16 cores. Perelman et al. [205] improved the speedup to 10^4 for 4 cores by clustering phases accross threads, with errors of a few percents.

Using a different approach, Penry et al. [204] have successfully leveraged modular simulation to parallelize multi-core simulators on a host CMP, achieving super-linear speedups thanks to shared caches and careful scheduling of simulation phases. But the provided speed relief over detailed single-core simulation is bounded by twice the number of available cores on a chip at design time, i.e., 8 to 16 today. More recently, Genbrugge et al. [102] applied interval analysis, which consists of a mechanistic model of superscalar processors able to produce performance numbers without having to model individual pipeline stages, to multi-core simulation. This approach currently yields a speedup of 10 over traditional cycle-accurate simulation, with a mean error rate of 5% for 8 cores.

Although the problem of simulating hundreds of processors has been a fixture of large-scale parallel machines design for many years, only a few existing simulators are able to sustain hundreds to thousands of cores with practical simulation time. As an example, the design of the IBM BlueGene/L supercomputer family between 1999 and 2004 essentially involved units and small sub-system tests, simulation being used only to functionally validate the development tools and the system-level software [56].

Recently, the COTSon team at HP labs proposed an approach where a trace-driven simulator is fed with thread instruction streams computed by a single-core full system simulator [186]. They present very fast speedups for 1024-core simulation, but only consider an idealized architecture with a perfect memory hierarchy, i.e., without any interconnect, caches nor distribution of memory banks. Most of the other recent approaches are parallelized discrete-event simulators with varying levels of detail. They allow some events to be committed out of virtual time order, trading accuracy for speed. SlackSim [63] is a cycle-level simulator that allows individual cores to progress at different paces in a controlled manner. Notably, it proposes a *bounded slack* scheme where cores can run ahead of the current global time by at most a fixed amount of cycles. Graphite [182], a higher-level simulator, dynamically instrument executables and then run them natively. It explores additional *lax synchronization* schemes, and in particular a distributed one, LaxP2P, in which one core's progress is periodically checked against another randomly chosen core.

In this Part, we propose *SiMany*, a new discrete-event simulator for many-core architectures supporting modern task-based programming models, like Cilk [34] or TBB [133]. It improves on the previous simulation approaches in three main directions. First, it

introduces a novel synchronization technique, *spatial synchronization*, in order to approximately reproduce the concurrency of interactions between threads/tasks and the hardware components that the simulated program would experiment on a real many-core machine. This scheme is *distributed* and, contrary to previous ones, purely *local*. A core is allowed to make progress ahead of its topological neighbors in the interconnection network by at most a fixed time drift, which involves only nearby information. Cores can be simulated without interruption during longer phases than in schemes where they have to check their progress against a unique global window. Both these properties have the effect of improving the locality of accesses, which ultimately speeds up the simulation.

Second, SiMany pushes further the current trend of raising the level of abstraction in simulators. This trend was recently illustrated by interval simulation [102] and Graphite [182]. It is also widespread in the embedded systems domain, where systems-on-chip (SoC) typically comprise tens of off-the-shelf IP blocks. To cover the resulting large design-space, the initial stages of a SoC design process are mainly concerned with capturing the interactions between the IP blocks and the interconnects (bus or networks-on-chip), using simple, and thus potentially inaccurate, block models. Approaches of this kind have been successfully used to choose components, evaluate performance and verify general operation of SoC in early design stages [104, 156, 222]. SiMany includes simple models for caches and cores, decreasing the time required to simulate these components. It is highly configurable and can explore a wide range of designs, such as shared-memory and distributed-memory architectures and arbitrary network topologies. All operations that do not require interactions are directly executed.

Third, we evaluate the relevance of the scalability results reported by SiMany by comparing them to those obtained with a cycle-accurate simulator [14] up to 64 cores. We show that the main performance trends are successfully captured and that SiMany can be used to forecast scalability and performance variations resulting from coarse-grain architecture changes. To the best of our knowledge, this quantitative assessment is the first of its kind for an abstract simulator featuring a loose synchronization mechanism.

We show that SiMany allows to explore a large range of hardware designs and evaluate software implementations 10^3 times faster than existing flexible approaches for 1024-core architectures, while yielding comparable error rates than sampling and statistical simulation for a low number of cores. Chapter 12 presents the main time modeling and synchronization concepts. Chapter 13 details the implementation of SiMany. Chapter 14 first presents the experimental methodology. It then compares SiMany's results to those of a cycle-accurate simulator up to 64 cores, showing that the main performance trends are successfully captured. Finally, it presents the performance results for different architecture classes, such as polymorphic architectures, in which cores differ in computing power, and clustered network patterns. We also investigate the accuracy/speed trade-off related to time drift control. Chapter 15 relates this work to the simulation field and Chapter 16 wraps up this many-core simulation Part. The content of Chapter 12 and Chapter 14 will appear in Certner et al. [55].

Chapter 12

Virtual Timing

The implementation of the notion of time in an abstract simulator must realize a delicate trade-off between accuracy and simulation speed. On one hand, an accurate simulation of hardware components involves more processing power and increases the number of synchronizations, resulting in a much longer simulation time. On the other hand, too little timing information or infrequent synchronizations would make the simulator too inaccurate to obtain meaningful results for design-space exploration and software development.

Moreover, the method for providing timing information must be generic and systematic enough that an architect can vary the design parameters and still obtain reliable performance information. In the end, the simulator must be able to produce an execution time for the program run on the simulator. We call this reported execution time the *virtual time* of a program, as it is a computed execution time, by contrast with the native execution time of the simulator itself running on the host machine to simulate the program.

Section 12.1 presents virtual timing and our spatial synchronization scheme. Section 12.2 highlights how parallel programs are correctly simulated in spite of some events being processed out of order.

12.1 Principles

12.1.1 Timing Annotations

Virtual time is computed by the simulator from the timing information that has to be provided for each instruction block. These blocks are not necessarily basic blocks, i.e., blocks where the control flow is purely sequential. They rather are pieces of code performing local computations and control flow changes, i.e., blocks of instructions that are directly executed by a local CPU. The user is given complete freedom about the timings assigned to these blocks. The simulator handles the timing of any operations that trigger interaction with other components than the local CPU. When simulating a shared-memory architecture, this implies that memory accesses will be timed by the simulator itself. Similarly, when simulating a distributed-memory architecture, the simulator times the

messages exchanged between the different components, based on the components' location and the network model.

A systematic way to assign timings to blocks is to profile a native run of the program on a single-core architecture or to execute it in an accurate simulator. This technique may introduce a bias for pieces of code that will be executed in parallel on the simulator but are not during the profile run. In the latter case, the execution of all pieces of code shares the same hardware components, such as caches or branch predictors, which decreases or conversely increases their execution time compared to the former case where some pieces execute on different processors. Architectural events, as reported for example by hardware performance counters, could be used to refine execution time to reduce this bias.

Another possibility is to use a processor model to compute the timings by summing the values it assigns to each individual instruction. This may be less accurate with respect to the microarchitecture, but will allow to avoid the bias. However, interval analysis showed that modeling all pipeline stages is not necessary to obtain accurate simulation results [102]. Moreover, taking into account microarchitectural details is less important for many-core than for single-core simulation, as the experimental results will show in Section 14.2.

Finally, timings can also simply be attributed by hand. Additionally, the timing annotations can be computed by the simulated program itself. This is useful, for example, to allow to attribute approximate timings to coarse program parts at once with very low overhead, depending on the data being processed, or to model branch prediction in a probabilistic way. Most blocks, though, are usually statically timed[1].

12.1.2 Distributed Timing

In our simulation framework, each simulated core, as well as each modeled hardware component, maintains its own private virtual time when it is active. All these virtual times would always be the same if they were permanently kept perfectly synchronized[2]. But frequent synchronizations have a cost and are not even necessary for independent events. For this reason, our simulator updates virtual times in a purely *distributed* fashion.

On one hand, each active component *independently* increases its virtual time depending on the operation it is performing. A core increases its virtual time in accordance with the timing annotations of the code block it is executing. A network interface processing a message increases its virtual time by the amount necessary to process the header and then the payload chunks.

On the other hand, when a component interacts with another, inactive one, it *propagates* its virtual time to it as it wakes it up. Since interactions conceptually occur in the simulator through *messages*, the latter are stamped with the virtual time of the component

[1] An illustration of annotations can be found in Appendix B.

[2] Actually, virtual times are updated discretely according to block annotations and the simulator's timing parameters. So, even in the case of perfect synchronization, virtual times of cores may differ at worst by $B_{\max} - 1$, where $B_{\max}$ is the greatest virtual time assigned to a block.

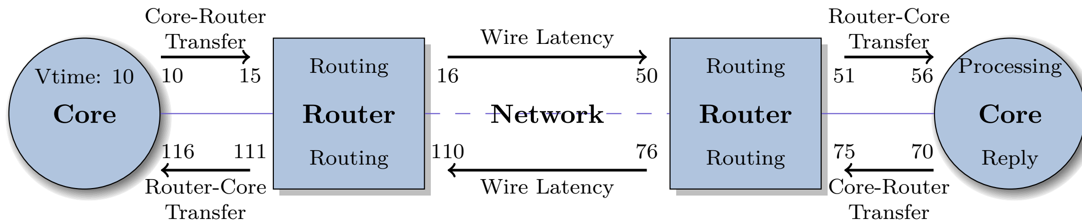

Figure 12.1: Messages' virtual time incrementation.

that created them at the moment the message was created. A network interface that was idle takes the virtual time that a message is stamped with as it begins processing it.

As an example, let us examine a full memory access round trip, presented on Figure 12.1. Each memory request is initially stamped with the initiator core's virtual time and is increased by a specific delay as it traverses the architecture's communication components (CPU to network interface links, routers, wires, etc.). The reply message is dated with the request time augmented with a local processing time corresponding to the data retrieval from the memory banks. When the initiator core finally starts processing the reply, its own local time is updated to that of the reply.

To summarize, the sum of all delays induced by all the components traversed is added to a core's virtual time in case of interaction.

12.1.3 Distributed Spatial Synchronization

Messages themselves are not a reliable enough form of synchronization: Two sets of cores may not communicate for a long time period, resulting in a possibly excessive time drift. On the other hand, a systematic global synchronization of all cores would be very costly. Thus, cores synchronize with their neighbors only, as specified by the interconnect/network topology, a feature we call *spatial synchronization*. Upon each modification of its virtual time, the core sends a *virtual time update message* to its immediate neighbors[3]. The latter then update their view of the neighbor's virtual time, which we call its *proxy*. Note that these update messages are *control messages*: They have no architectural existence and they only serve to implement the simulation.

Figure 12.2 (next page) shows such an example for a core belonging to a 2D mesh that is advancing its virtual time from 20 to 33. The number at each core's cardinal point indicates the core's proxy for the corresponding neighbor. The bottom part of the Figure represents the core and its neighbors in their final states, i.e., after the update messages have been processed and the proxies updated.

If a core's virtual time is greater than the time of its most late neighbor by a user-chosen constant D, then the core stalls its execution. As soon as this neighbor catches up with it, thus lowering the time drift under D, the core can resume its execution. Figure 12.3 (page 167) shows an example of how an active core that is making progress (the one on

[3] Actually, not so many messages are exchanged, as explained in Section 13.2.4.

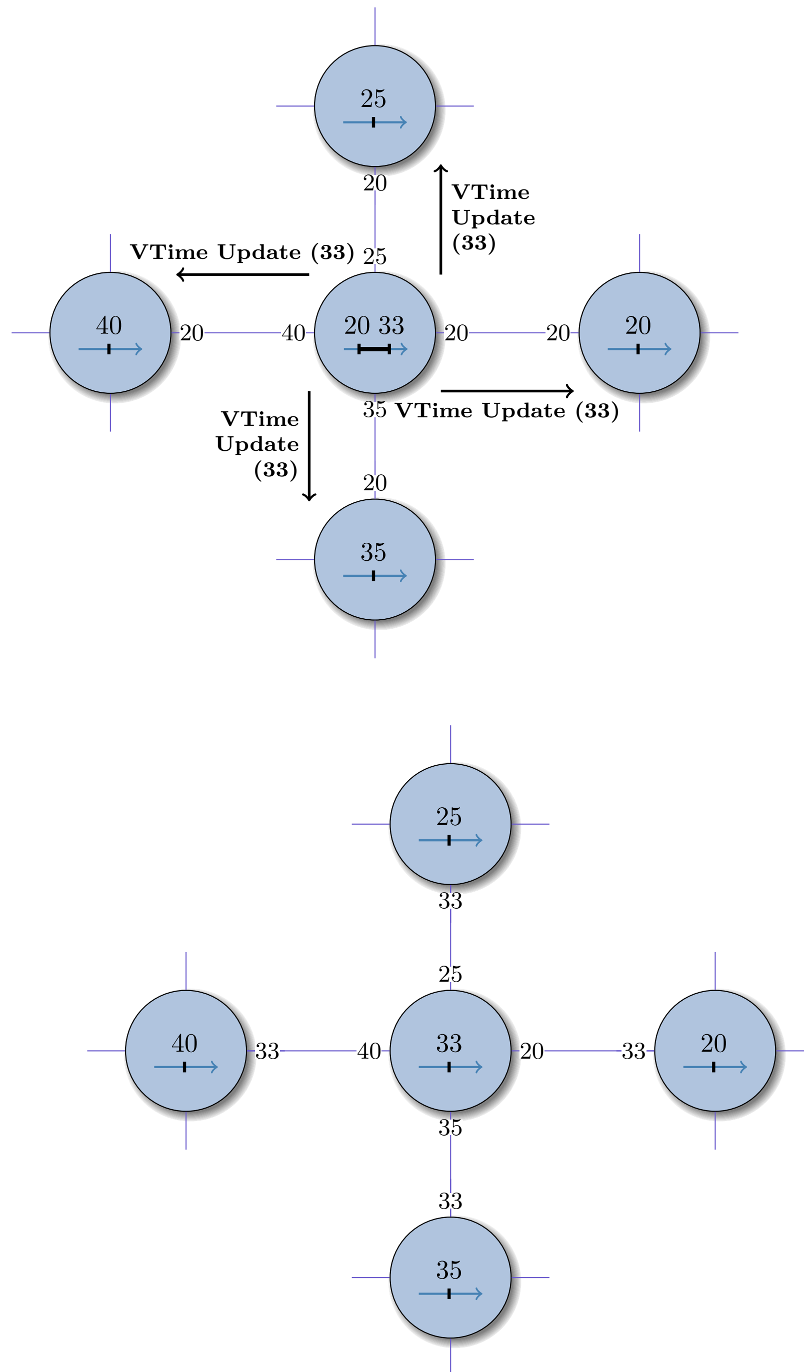

Figure 12.2: Virtual time updates.

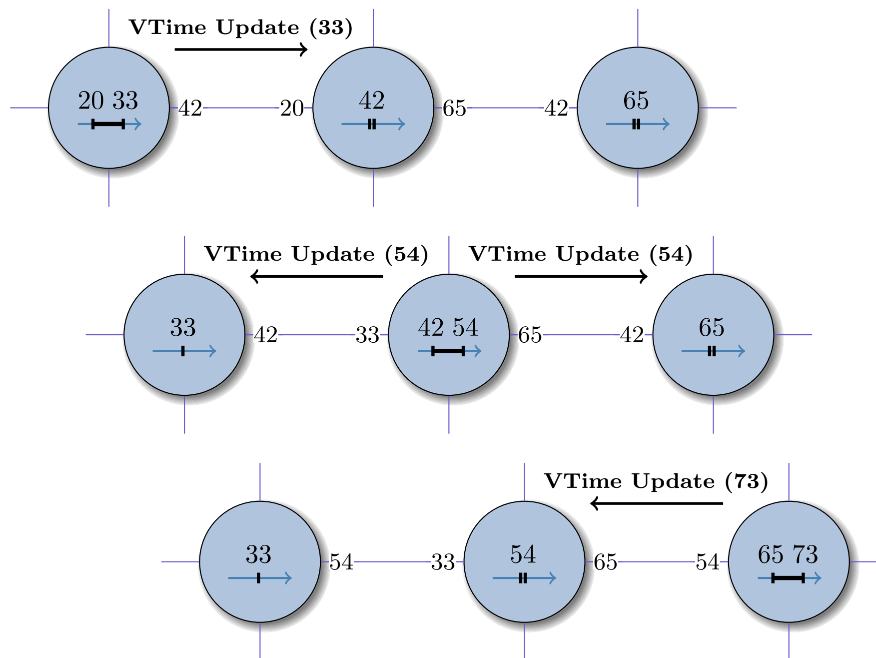

Figure 12.3: Spatial synchronization example.

the left) gradually wakes up cores that were waiting for it (D is 20). The three cores and their simulation states are represented at three successive points in time, from top to bottom.

In other words, cores are allowed to advance to differing virtual times, but they are not allowed to drift apart from their neighbors by D or more[4]. Note that this local bound guarantee immediately implies a global bound on the drift between any two cores that cannot exceed the diameter[5] of the network graph times D.

In the same way that sampling consists in trading some accuracy for speed, allowing a time drift reduces the number of synchronizations and thus speeds up the simulation, at the expense of some inaccuracy. Indeed, drifts open up the possibility that two messages be received and processed in the opposite order of their virtual time of reception. With $D = 20$, consider the example where two cores A and B, having times 50 and 20, want to send a message to a common neighbor core C. Having a common neighbor, A and B are allowed to drift apart up to 40 ($2 \times D$) and thus can be simulated concurrently at this point. If the simulator processes A, then B and finally C, A's message can reach C before B's, although its timestamp when arriving at C will be 55 whereas B's message's

[4]In all generality, the bound is the maximum of $D + B_{\max}$ and $D + I_{\max}$, where $B_{\max}$ is the greatest virtual time assigned to a block and $I_{\max}$ is the maximum cost of an interaction timed by the simulator. Unsurprisingly, the simulation's accuracy also depends on the level of detail of the program's annotations and that of the simulated hardware mechanisms. In the main text, we will keep mentioning D as the local bound instead, for simplicity. This substitution does not change the validity of our remarks and results, except quantitatively.

[5]The diameter is the largest topological distance between two cores of the network.

timestamp will be 25, assuming a common 5-cycle latency to reach a neighbor.

When the simulator processes two messages out of order, it biases their processing's completion time by the same amount of virtual time, which is at most the time needed to process the earliest message, but in opposite directions. However, we recall that a message represents a data access; a huge number of them are thus actually exchanged, with out-of-order executions occurring randomly. In the end, as the experimental results will show in Section 14.2, these errors statistically filter out and do not engage in any visible snowball effects. Section 13.2.3 also presents mechanisms to enforce bandwidth and concurrency limitations in spite of out-of-order processing.

Therefore, the maximum local drift simulation parameter D is an accuracy/simulation-speed toggle. The smaller the value of D, the more frequent the synchronizations, the better the accuracy, but the slower the simulation.

12.1.4 Non-Connected Sets of Active Cores

As mentioned before, a core only propagates its own virtual time to its neighbors when it changes. Indeed, regularly sending time messages, as done in cycle-level simulators, would degrade simulation speed. However, it may happen that the set of active cores in the network is non-connected. In that case, time drift control does not spread to the whole network. Local control indeed relies on the neighbors' virtual time information but idle cores don't have a virtual time of their own and don't produce any time update messages. An example of this problem is presented in Figure 12.4 (facing page), where advances of cores in both sets, on the right and left sides, are not propagated through the idle cores. As a result, their time drifts apart by more than the diameter of the network times D.

A solution to overcome this problem is to have idle cores maintain a *shadow* virtual time which is the minimum virtual time of all their neighbors plus D, as if they were executing code and had advanced to the maximum virtual time allowed by the local drift bound before stalling. Like working cores, they only propagate their time to their neighbors when their shadow virtual time changes. Thanks to this technique, all cores within the network remain connected with respect to virtual timing, which finally enforces the expected global time drift.

12.1.5 Time Drift of Dynamically Created Tasks

Many programming models (e.g., Cilk [34] or TBB [133]) provide primitives to spawn tasks dynamically. Such a task is sent to another core, as specified by the simulated run-time system and architecture, by emitting a task creation message that is stamped with the time of the initiating task, as any other messages. While this message travels to its target core in the network, active cores are concurrently making progress depending on the order in which the simulator executes them and on the spatial synchronization mechanism. But, since the latter only takes into account cores, not messages, cores may well drift ahead of the not-yet-running new task's timestamp by much more than the authorized global bound.

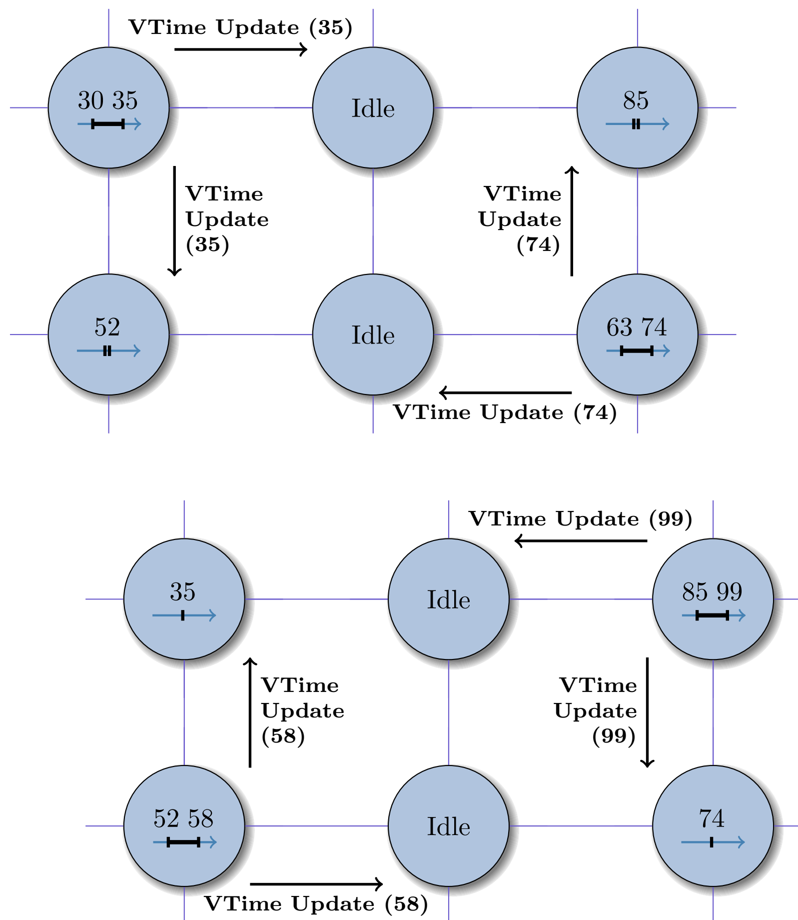

Figure 12.4: The non-connected sets problem.

Figure 12.5 (following page) shows such a situation. The core at the left spawns a new task at virtual time 20, through a task spawn message, but then continues to make progress. Since the other cores in the network are idle, the spatial synchronization mechanism doesn't prevent it from reaching timestamp 90, which is a lead of more than D (here, 20) over the new task's start time.

To keep this drift under control, a core can track the birth times of the tasks it spawned and take them into account when computing its current drift, as if the new tasks had started execution on one of its neighbors. When a core finally starts executing the new task, it sends a control message to the initiating core to inform it that it can discard the corresponding creation date.

Spawned tasks may be queued while waiting for a core to seize and execute them, since no cores may have been attributed to them yet or because all cores may already be busy. Without further precaution, this situation can cause a deadlock. If all cores are busy and happen to depend on a creator task to make progress and terminate, and if this creator task is blocked by spatial synchronization, waiting for one of its child tasks

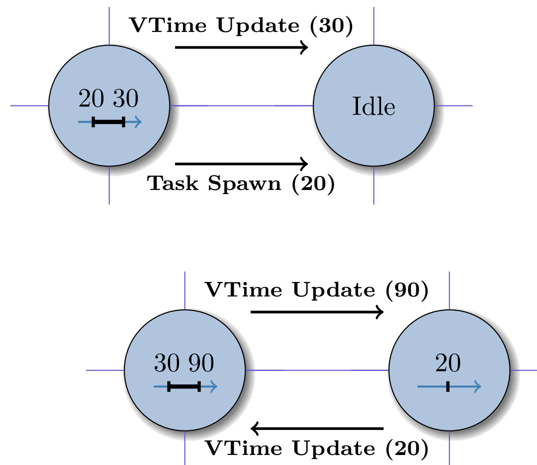

Figure 12.5: Time drift of a dynamically created task.

to begin execution, the child task will never have a chance to start precisely because all cores are busy. The simulator avoids this condition by requiring that the run-time system, which manages the tasks and implement the task queues, signals the core running the creator task as soon as its child tasks have been dispatched to a core or have been enqueued. The core running the creator task then drops the child tasks' birth dates.

The assumption behind this mechanism is that some tasks are enqueued only if all cores are busy. Waiting tasks won't be executed until some running task finishes. When this happens, the simulator calls the run-time system, offering it the opportunity to schedule a new task on the corresponding core. This ensures that waiting tasks will finally be executed. When they are, they will have their start virtual time adjusted to the finish time of the previous task. Thus, while tasks are enqueued, the current virtual time they hold, which was computed from the creator core's time at spawn, becomes no more relevant as an indication of their real start time and thus is not taken into account by spatial synchronization. If the simulator is to be used to model enqueing of tasks even when not all cores are busy, then the queues themselves have to be modeled as separate hardware components and they must participate in spatial synchronization.

12.2 Ensuring Correct Simulation

12.2.1 Program Execution Correctness

In spite of some messages being processed out of order, program execution correctness is preserved. A well-written program's outcome is correct independently of how its threads are scheduled. In shared-memory, in particular, output must be correct for all potential orders in which different threads can acquire a given lock. Only the lock acquisitions within the same thread have to be performed in order so as to avoid deadlocks. In distributed-memory, a program must be prepared to process incoming messages in any

order that is allowed by the programming model/run-time system.

Our network implementation enforces that a core receives all messages coming from another given node in the order the latter sent them. Only messages incoming from different nodes can be processed out of order. This solves the lock order reversal problem, because lock acquisitions are immediately translated into messages, and is compatible with traditional distributed-memory programming models that assume this property, such as MPI [94].

For dataflow programs, message ordering would not even be necessary. In the end, maintaining program execution correctness depends on the programming model and can be implemented by the underlying network and core models independently of virtual timing and its associated mechanisms.

12.2.2 Locks and Critical Sections

A task running on a core can stall at any time because of spatial synchronization. This can cause a deadlock if this task is holding locks or is executing a critical section at the moment of interruption. Indeed, a second task may then try to acquire the same locks or enter the same critical section and will then block until the first one wakes up again and releases its resources. But if this second task is very late, i.e., if its virtual time is far behind that of the first task, spatial synchronization will prevent the first task from making enough progress to release its resources.

Figure 12.6 (next page) illustrates such a situation in the simple case of two neighbors. The core on the right acquired a lock at virtual time 35 and then reached time 45 at which point spatial synchronization suspended it, being ahead of the left node's time (20) by more than D (equal to 20 in this example). The left node then sends a lock request at time 22 and blocks until it receives an acknowledgement. Note that both cores involved in a deadlock need not be neighbors but can be located at distant places in the network.

Avoiding such deadlocks without the help of a global monitor can be done by temporarily allowing the core holding a lock to execute until it has released its resources. Waiving the time synchronization could potentially result in more time drift than the aforementioned global bound. However, locks or critical sections are used precisely to serialize resource access. Thus, contention essentially occurs when attempting to acquire a lock rather than in the ensuing critical section, except in the rare cases of deeply nested locking. As an example, in our dwarf benchmarks, tasks simply do not interact with others while holding a lock. In practice, the additional time drift thus has no effect on performance.

In the rare cases where there can be high contention inside a protected or critical section, typically when atomic instructions are used on a shared variable, the following mechanism, which is completely general, should be implemented instead of the previous one. Every task T that needs to access a resource currently in use by some "holder" task T_r has its virtual time advanced to that of T_r before being put to sleep, waiting for the resource to become available. Then, T_r remembers that T depends on it so that, each time T_r makes progress, the virtual time of T can be updated accordingly.

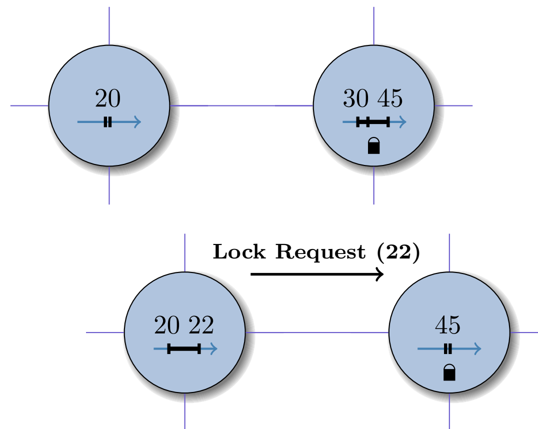

Figure 12.6: Deadlock between two tasks competing for a lock.

This mechanism is directly inspired from what happens inside a real machine: Tasks waiting for a resource interact only with the resource holder and they get the resource at a time greater than when the holder released it. This implies that, between the point when they start to wait and the point where they get the resource, time passes by without them making any other useful actions. The simulator can thus advance them to the current time of the holder task without influencing the behavior of any other tasks or components, because, up to this point in virtual time, the holder task is still owning the resource.

12.2.3 Deadlock Avoidance Proof

The spatial synchronization mechanism doesn't cause by itself any deadlocks in the simulator's execution because, at any given point in time, the task/core with lower virtual time can always make progress, unless the system is deadlocked for another reason. Virtual timing and its associated mechanisms thus don't introduce new deadlock possibilities. This Section presents a proof of this property.

Let us assume that the simulator is deadlocked and let us call T_0 the task with smallest virtual time at this point. In all generality, T_0 was either just about to execute some code, such as a local computation, a task spawn, a regular remote or shared-memory access, or was waiting to receive some message, to acquire some lock or to enter some critical section. But T_0 has the lowest virtual time and is thus late compared to its neighbors. Consequently, the distributed synchronization mechanism doesn't prevent it to execute some code. T_0, which is blocked, is thus necessarily in a waiting state. This means that it is dependent on some action to be performed by another task, which we'll call T_1. To be exhaustive, we should also consider the case where T_0 is dependent on some other component than a CPU. We show how to deal with implementations where this can occur at the end of this Section. Suffice it to say here that they can be handled the same way.

The next part of the proof now depends on how locks and critical sections are handled. If they are handled in the simplest way, T_1 will run until it can release the resource that T_0

needs or until a response to T_0's request can be produced, eventually allowing progress of T_0, unless T_1 needs to acquire another resource or receive a response. Progress is blocked, so we are in the latter case and T_1 depends on another task T_2.

If the more general approach is taken, then T_1 is blocked holding the resource that T_0 needs, either because of spatial synchronization or because it is dependent on another task T_2. In any case, thanks to the rules the simulator abides to, which are described in Section 12.2.2, T_1 has propagated its virtual time to T_0 just before being suspended. T_0 and T_1 thus have the same virtual time, which is minimal. Consequently, T_1 can't have been suspended by spatial synchronization and we are in the case where T_1 depends on another task T_2.

In both cases, T_1 depends on another task T_2 and, in the second case, T_1 has minimum virtual time. We can iterate the previous reasoning $n + 1$ times, where n is the number of cores in the network, to build a dependency chain: $T_0 \rightarrow \ldots \rightarrow T_n$. But, since there are only n cores, there exists i and j in $[0; n]$, with $i < j$ such that $T_i = T_j$. The dependency chain $T_i \rightarrow \ldots \rightarrow T_j = T_i$ shows that there exists a deadlock involving tasks T_k, with $k \in [i; j]$.

We haven't detailed the case where some tasks depend on some actions to be performed by components other than the CPUs in order to make progress. This never happens in our implementation, as will become apparent to the reader of Chapter 13, which explains how the simulator is implemented and, in particular, Section 13.2, which details the network implementation. Indeed, network interfaces are designed to transmit messages immediately and never depend on CPUs to do it. Moreover, the sending of response messages is always triggered by a CPU making progress. In the end, there are only dependencies between CPUs.

Nonetheless, such a case can be dealt with exactly in the same way as for tasks. Within this solution, components have to maintain, at least when they are in a dependency relation with CPUs, a virtual time of their own. They also must take part in spatial synchronization and be subject to the handling of locks, critical sections or other exclusive resources described in Section 12.2.2, as tasks do. Under these conditions, our previous proof will then obviously still hold if we simply replace tasks by more general components.

This definitely establishes that virtual timing and the associated spatial synchronization can be applied to any possible simulator implementations without introducing deadlocks. However, they do not solve any pre-existing ones. The simulated run-time system and programs themselves must be deadlock-free for the simulation to terminate.

Chapter 13

Simulator Implementation

13.1 Implementing an Efficient Simulation

13.1.1 Direct Execution of Computations

Instruction blocks, as defined in Section 12.1.1, are processed by one of the simulated CPUs. Their execution increases the corresponding CPUs' current virtual time by their timing annotations' values. Their only other influence in the simulation process comes from the results of the computation they perform, because these results can change the subsequent control flow. In other words, a minimal correct simulation is performed if and only if blocks' *functional results* are computed. Consequently, and unlike most simulators, SiMany does not perform instruction set (ISA) emulation. It does not even do dynamic binary translation, as used in fast simulators like Graphite [182]. Instead, apart from network simulation and task management, it relies on *direct execution*.

To achieve this, it is currently necessary to modify the programs to be simulated, both to integrate the timing annotations and to explicitly call the programming model's API and/or the simulator's primitives when the program spawns a new task, needs to send some message or wants to access some remote data. To ease the simulation of very large programs or to enable it for closed-source ones, for which recompilation is not possible, these modifications could be applied automatically using one or a combination of the following techniques: Binary code insertion, static binary translation or JIT compilation (a.k.a., dynamic translation, as done for example by QEMU [23]).

13.1.2 Software Architecture

The simulator's main duty, in addition to maintaining a virtual time for all cores, is to manage the simulated CPUs and the other architecture components, such as the network interconnecting them, so as to reproduce in large part the operation of a many-core machine and to report realistic virtual time values. In order to support a large range of architectures and to vary them simply by changing some configuration files, the knowledge of low-level hardware characteristics must be restricted to the simulator core as far as

possible. Programming models' support is then implemented on top of the simulator's low-level primitives that give access to the simulated hardware.

However, user programs need to somehow access some simulator's primitives, when the programming model doesn't provide specific services for some categories of actions. Typically, local memory accesses are directly performed by the underlying architecture and their timings depend on the accessed data's location. Furthermore, multi-cores with global addressing can use one of several different memory consistencies, each leading to different parallel program semantics. Modeling a different semantics set than what is provided by the simulator and/or the host machine requires to abstract memory operations. For these reasons, any data accesses not done through the programming model's run-time system must be done through other functions that ultimately call the simulator primitives. Another example is task/process spawning, for which some programming models provide no support, such as MPI[1] [94]. In this case, the user program must still be able to launch code on remote cores.

The software architecture of the simulator running a program thus comprises 3 layers. The first layer is composed of primitives that give low-level access to the simulated hardware. They allow to launch a piece of code on a core and to exchange messages between cores in the network through network interfaces. This layer is decoupled from the programming model and is thus generic in this respect. It can be extended to include other hardware components than cores and network interfaces. The second layer's purpose is to provide an implementation for the chosen run-time environment that relies on the first layer's API. It is divided into 2 parts: The run-time system support for the chosen programming model's API and some glue functions that expose the computation model *implicitly* assumed by the simulated hardware (e.g., for global addressing architectures, the memory consistency). The third layer is the simulated user program itself. It uses only APIs from the second layer.

The implementation of the first layer itself uses a mix of shared data, function calls and message passing primitives. The latter serves both to transmit user messages, explicitly sent by the program or implicitly generated by memory accesses or calls to some programming model's functions, as well as *control messages*, used internally by the simulator's first layer[2]. Messages are used instead of directly modifying the state of shared objects in order to allow the hardware cores to be simulated in different threads and possibly different processes, thereby enabling a multi-threaded or even distributed simulation. Because of the messages' purpose and nature, a huge number of them are exchanged during a simulation. Thus, the bandwidth and latency of the chosen implementation for communicating are of foremost importance to achieve high-speed simulation.

[1]MPI-2 introduced functions that allow to spawn new MPI tasks, but lots of programs are still using the much simpler MPI 1.2 API and its comparatively lightweight implementations.

[2]Control messages will be described in Section 13.2.4.

Message Size	Sockets	Sockets NB	Named Pipes	Shared-Memory
16	2.13	1.49	4.76	0.27
32	2.19	1.82	4.75	0.25
50	2.16	1.96	4.85	0.26
64	2.18	2.02	4.86	0.28

Figure 13.1: Time to Send a Message (in µs) on an Intel Core 2 Duo T7400, FreeBSD 7.0-STABLE.

Message Size	Sockets	Sockets NB	Named Pipes	Shared-Memory
16	0.60	0.29	1.17	0.13
32	0.61	0.31	0.99	0.15
50	0.61	0.37	1.01	0.21
64	0.62	0.39	1.00	0.23

Figure 13.2: Time to Send a Message (in µs) on a 4×AMD Opteron 8380, Linux 2.6.29.

13.1.3 Overhead of Network Communications and OS

Our initial simulator implementation was distributed: Each core was simulated in a separate process and they were communicating through sockets, with the intent to distribute the cores onto a cluster of machines to speed up the simulation. It turned out that this implementation was extremely slow. As an example, it took it up to 3 hours to simulate a Dijkstra computation on a thousand-node graph onto 4 cores only.

We finally found out that this was not due to network communication bandwidth or latency but to the different inter-process communication (IPC) mechanisms' implementations and to process scheduling latency. We reached this conclusion in part by benchmarking the time to send messages of varying length from one process to another on the same machine using different APIs: Sockets in blocking and non-blocking modes, named pipes and System V shared-memory segments with busy-waiting, using a lock-free implementation of circular buffers. Let us recall the following characteristics of communication between cores on today's available dies in practice [185, 230]: The bandwidth has an order of magnitude that varies from 10 to 100 gigabytes per second and data access latencies vary from 1 to 100 nanoseconds.

Message Size	Sockets	Sockets NB	Named Pipes	Shared-Memory
16	0.82	0.38	1.46	0.08
32	0.83	0.39	1.41	0.09
50	0.85	0.48	1.68	0.11
64	0.85	0.44	1.69	0.10

Figure 13.3: Time to Send a Message (in µs) on a 4×Intel Xeon E7450, Linux 2.6.29.

Figure 13.1 (preceding page), Figure 13.2 (previous page) and Figure 13.3 (preceding page) show the results of this benchmark on different machines and OS. 10^8 messages were sent in a row, with lengths ranging from 16 to 64 bytes, which are representative of the length of messages exchanged during an actual simulation. A variance of around 0.1 µs has been observed in practice, so the presented averages should not be regarded as sharp. Nonetheless, in all runs, the relative execution times for different message sizes and communication techniques are preserved. According to the raw performance numbers for current hardware reported in the previous paragraph, 1 gigabyte should be transferred in approximately 10 to 100 ms at best, with bandwidth dominating latency.

In all experiments, the shared-memory implementation outperforms all the other communication methods. The second best method in each case is non-blocking sockets and it is 2 to 6 times slower depending on the machine and the OS. For regular sockets, the slowdown ranges from 4 to 10. Surprisingly, named pipes by far perform the worst of all methods[3], being around twice slower than sockets. This simple message benchmark shows that using shared-memory speeds up the simulation by almost an order of magnitude at least. In typical simulations of small to moderate benchmarks, billions of cycles are simulated and nearly as many messages are exchanged, with lengths varying from several bytes to up to 1 kB. Moreover, contrary to what happens in the message benchmark, the messages are not all exchanged at once, potentially introducing additional latency (less friendly cache behavior, scheduling latency, etc.). Due to the number of mechanisms involved, we do not present a thorough breakdown of the message benchmark time into raw communication, normal OS scheduling and wake-up events latency, because this would require very complex experiments and an intimate knowledge of large parts of kernel code.

These are not the only costs involved in distributed simulation. We have observed experimentally that, for older versions of Linux (2.6.17) and on FreeBSD, the OS may wait for the next clock tick to wake up a process to handle incoming data. This may delay each message by several milliseconds, translating to hours when millions of messages are exchanged, slowing down the simulation by a 10^4 factor at least. The presented numbers also don't take into account networking delays. Besides the hardware latency, ranging from 10 to 100 µs for Gigabit Ethernet [106], network interrupt coalescing in the OS yields an additional delay, usually in the order of 10 µs [107]. Finally, the protocol used and its tuning are important as well, which the deadlock problem between the Nagle and delayed acks TCP mechanisms [183] is a famous example of.

Thus, simulating some cores on different machines will yield an overhead that, depending on the conditions, translates into seconds to hours of simulation time for our studied benchmarks. For distributed simulation to be fast, it is necessary to pay special attention to all elements that can increase latency or reduce bandwidth compared to shared-memory simulation: The hardware communication layer, the hardware controllers, the network drivers state, protocol tuning, the kernel's communication method's implementation, kernel wake-up latency and general scheduling policy. Each of these elements alone may spoil the benefit of using multiple machines if it introduces too much overhead. This makes distributed simulation's efficiency difficult to predict. Additionally, performance is

[3]This is definitely something that someone interested in OS should look at!

likely to vary greatly as the execution environment changes, leading to concerns about the practical portability of distributed simulation.

13.1.4 Userland Threading and Scheduling

Section 13.1.3 showed the large overhead that distributing simulation would introduce, including when running it all on a single machine. A solution is to completely bypass the kernel by using userland thread multiplexing, avoiding its provided means of communications (sockets and named pipes) and the OS scheduling latency.

Several userland scheduling implementations have been developed over the years, such as the GNU Pth [105] and Marcel [247] threading libraries. They essentially provide transparent userland threading to applications using the Pthread interface by "wrapping" system calls in order to prevent a userland thread having to block in the kernel to stall the whole kernel thread it is executing in. For GNU Pth, this is achieved by using non-blocking calls where possible. For Marcel, enhanced OS activations are used [73]. In both cases, however, IPC is performed by the kernel in the end. We thus reimplemented userland scheduling and the associated communication infrastructure so that no kernel IPC implementations are used.

Each core in the architecture is simulated within a lightweight thread. Context switching is done through the UNIX context functions (`makecontext`, `swapcontext` and `getcontext`) which are available on the most popular UNIX variants, are fast and more easily used than the C standard's `setjmp` and `longjmp` family of functions [87]. These functions still perform a system call to save the current signal mask and to install the new context's one. It would be possible to improve again the current simulator by reimplementing them purely in userland since signals are not part of the programming model exposed to user programs and are neither used by the simulator itself.

Threads communicate thanks to lock-free circular buffers[4]. When a thread wants to read from an empty buffer or write to a full buffer, the buffer functions call the userland scheduler to suspend it, marking it as waiting as well as registering an event function. The event function serves to check whether the suspending condition has been cleared without having to context switch to the thread. In the buffer example, the event function checks whether new data can be read or if buffer space is now available to write into.

The userland scheduler has been kept relatively simple. It knows nothing about the simulator infrastructure that runs on top of it. A userland thread yields control to the scheduler mostly implicitly, i.e., by calling some infrastructure primitive that can't be serviced right away. The scheduler code then runs and selects the next thread to run from a single list containing all userland threads. To ensure fairness, all threads are considered in turn from the list head and are put to the list's end when they release control. For threads that are marked sleeping, their event function is executed to see if they can resume.

The scheduler also maintains a list of real threads executing the lightweight threads.

[4] This is the same implementation as was used in Section 13.1.3's shared-memory benchmarks.

We have conducted preliminay experiments using several real threads. Performance currently drops when using more than one thread because of concurrent accesses to the various scheduler structures. Indeed, in the current implementation, the lightweight threads are kept in a single list and a single real thread can manipulate it at a time. The scheduler will have to be optimized for multi-threading before any conclusion can be made on whether the simulation can be parallelized efficiently in practice.

13.2 Modeling a Network of Cores

13.2.1 Simulated Architecture Overview

The simulator models architectures that are composed of general purpose cores connected through a network. The implementation is flexible enough to model a full range of architectures, from a single die comprising all hardware parts to a core per die with dies connected by an off-chip network.

The network topology can be freely specified in a configuration file as an adjacency matrix that gives the connections between the cores. One network interface is associated to each core and connects it to the network. It is currently not possible to model standalone interfaces or switches or multiple interfaces per core, but they are hardly found in today's and anticipated designs and can be added without much hassle if the need arises. Links between two network interfaces are configurable: They can have individual latency and bandwidth multipliers[5]. This allows to model, for example, a network of clusters, where the links between clusters have higher bandwidth and latency than intra-cluster links.

All the cores are general purposes ones, providing homogeneity over the network, which simplifies the environment running on top of it. However, they can be attributed individual slowdown multipliers to simulate chips with different speeds, e.g., enabling the study of power-constrained architectures having efficient slow cores for regular operation and more powerful cores for infrequent uses of heavier applications. Simulating heterogeneous cores, with different ISAs and/or capabilities, such as specific hardware accelerators, can be done at the run-time system level by restricting the kind of code that may be executed on the cores.

No particular memory models are implied by the simulator. We used it to evaluate realistic distributed-memory and ideal shared-memory architectures, but shared-memory with full cache hierarchies could be simulated as well. There are currently no memory limits imposed on the simulated program at each node, other than the host machine's memory size. An extension introducing memory limits per node will have to be developed to use the simulator in the context of memory-constrained embedded platforms.

No cache models have been implemented yet. Rather, cache effects are reproduced through timing annotations in programs by assuming that the requested data are not in the cache at the start of a routine and are subsequently in the cache for the rest of it. This reproduces a pessimistic cache behavior where all caches are emptied at the simulated

[5] These are integer numbers.

program's routine boundaries[6]. Real cache models can be integrated into the simulator since all memory accesses must ultimately go through the simulator's primitives, as was explained in Section 13.1.2. We did not experiment with this possibility but we expect that the subsequent slowdown will not exceed an order of magnitude.

Finally, although the simulator doesn't simulate a core's ISA nor its microarchitecture in detail, it is possible to reproduce some of the impact they have on performance through timing annotations. The effects of the implementation and number of functional units for a class of instructions can be mimicked by varying the timing attributed to instruction blocks that use them. As an example, we used in our experiments a model where the instructions of a class are attributed a fixed cost and we split instructions into several classes, including floating-point and integer additions, multiplications and divisions. We also introduced annotations to model a branch prediction scheme that succeeds at least 90% of the time and assumes a pipeline depth of 5. More details about the precise microarchitectural parameters used in the experiments are presented in Section 14.1.1. Appendix B presents the example code of the `Dijkstra` benchmark with annotations.

13.2.2 Network Interface Implementation

The network interface (NI) module provides code both for sending and receiving messages, but these cases are handled asymmetrically at execution time. A NI-dedicated lightweight thread serves solely to receive messages. It doesn't maintain a virtual time by itself, but rather takes the one of each message it receives and increments it as it is processing it. Message sending, on the other hand, can happen both in the NI thread, if processing a message itself requires sending another message, or in a thread simulating a core (e.g., when requesting some user data). Consequently, before sending data, a thread must acquire the lock associated to the communication channel it wants to use.

The NI module is designed to roughly reproduce the timing behavior of a real hardware network interface. Messages are divided into header and payload parts. The cost for transmitting a header to/from a core is fixed and the payload part is divided into fixed sized chunks whose processing cost is also fixed. Those costs are counted twice, the first time at message creation and the second time at message reception. While traveling, a message's virtual time is increased by the latency of each traversed hop, irrespective of its length, which corresponds to a simple model of worm-hole routing [71].

Links between two NIs are implemented as two one-way circular buffers of fixed length. The buffers do not know anything about the structure of the messages being transferred. They correspond to a simple model of a physical layer that is concerned only about sending words of data. Besides being more faithful to the real hardware mechanisms, we saw no need to implement an extra lookahead facility for messages, since messages in a given link are always processed in order, which is the same as the messages' virtual time order[7].

[6] The boundaries are specified implicitly through timing annotations and do not necessarily correspond to the programming language's syntaxic structures, such as functions.

[7] Actually, some messages may not be in virtual time order even in one way of a link, because they may have been sent by different threads modeling different hardware components. This is inherent to virtual

A major concern when choosing a network architecture and associated protocols is to avoid deadlocks. Although message transmission is timed as in worm-hole routing, messages are in reality transferred using store-and-forward inside the simulator, which avoids deadlocks caused by message reservation of several links at once. Deadlocks can still occur if a dependency cycle appears between messages only partially transferred because of NI buffer overflow. To prevent them, NIs maintain for each link separate receive and send buffers. Also, the protocols chosen in this implementation ensure that each sent message is always eventually consumed by the destination node before any other messages are created at this node.

The last kind of deadlocks that can occur are deadlocks related to the simulator's software architecture. More precisely, NI threads may themselves generate messages as part of processing some incoming messages. If this message generation can't complete because of a buffer being full, a NI thread becomes blocked and can't service other incoming messages until the condition disappears. Except for a particular interaction with its associated core, described in Section 13.2.5, and for the routing startup algorithm described below in this Section, a NI thread doesn't generate new messages as part of message processing. Because of this property and thanks to how the two special cases are handled, such deadlocks don't occur.

Routing decisions are taken independently by each NI, which simply looks up the next hop corresponding to the message destination in its routing table. Only one possible next hop is stored for each destination and is used all the time, regardless of the adjacent links' traffic. This technique is commonly referred to as *oblivious* routing [231]. The routing table itself is computed at startup using a distributed adaptation of the Roy-Floyd-Warshall [92, 216] all-pairs shortest path algorithm which is described in the next paragraph. The length of a path between 2 cores, as used in this computation, is the number of hops between the cores, regardless of the links' latencies.

Our routing startup algorithm works as follows. A NI coming up broadcasts to its neighbors a message whose aim is to indicate the current best distance to it. NIs receiving the message increase this distance by one hop and compare it to the one currently stored for the origin NI. If the new distance is strictly lower or if it is the first time a message from this origin NI is seen, the new distance is recorded along with the neighbor from which the message came, and the NIs in turn relay the updated message to their neighbors. This algorithm effectively terminates since a finite amount of messages are exchanged. A NI indeed can't broadcast a message to others concerning the distance to another NI more than the diameter of the graph times, because each new broadcasted message embeds a strictly decreasing distance value.

13.2.3 Bandwidth and Concurrency Limits

As messages may arrive out of virtual time order, if they come in from different neighbors, the virtual time assigned to a NI, when it starts to process a new message, may be lower than the end time of processing for the previous message. When this happens, both

timing and loose synchronization, of which spatial synchronization is an incarnation.

messages processing may appear to overlap, in terms of virtual time, if the end time of processing for the new message is greater than the start time of processing for the old one. This practically means that the NI is seen as able to handle concurrently the different messages coming from its neighbors. As will be described in Section 14.1, we only used networks based on 2D meshes, where each core has at most 4 neighbors, implying that each network interface has at most 4 ports, a quite realistic assumption. We propose below a way of limiting concurrent processing for networks in which nodes could have a much higher number of neighbors.

There are at present no general traffic bandwidth limitations implemented but two nodes communicating with each other will see a circuit bandwidth limit, corresponding to the smallest bandwidth in the routing path from one node to the other, regardless of the fact that another pair of communicating nodes may be sharing part of the circuit with them. In practice, this means that traffic congestion is not modeled.

Even if those limitations were not a concern in our experiments, we hereby propose a new and general mechanism, called *historization*, that can serve to implement both NI port and global bandwidth limitations. It consists in keeping information about the number of processed packets during a window of fixed or variable length into the past, at each link and network interface. This history is consulted each time a packet traverses a link or reaches a NI and is used to further delay its processing, according to a given policy.

As an example, a link may first process a packet from times 20 to 40 and may then receive a message with virtual time 15, whose length implies it should be processed in 10 units of times. The NI at the link's end can update the message timestamp so that the end of processing appears to happen at virtual time 45, effectively enforcing the fact that both message processings can't overlap. Other more sophisticated and realistic policies can be implemented within this framework.

A weak form of historization is already used when new tasks arrive out of order at some core to be executed. When some task starts to run, it conveys its time stamp to the core executing it, except if the previous task that ran there terminated at a higher virtual time, in which case the core uses the latter as the time starting point for the new task. Thus, for cores, virtual time is never allowed to go backwards.

One may argue that historization requires an unpractical amount of memory to store potentially long histories. But only the most recent portions of them are really necessary, since no new messages are created with a timestamp lower than the current minimum virtual time. We didn't evaluate the mechanism yet, but we conjecture that it will allow to simulate the most relevant part of the effects discussed above with a much narrower window of time in practice. In any case, the simulator can assess whether it has kept history information corresponding to an incoming message's virtual time, which gives a metric of how much overlapping may be missed when using shorter histories.

Historization introduces further virtual timing bias, since, in a completely ordered execution, the message coming second would have been delayed instead of the first. This kind of bias, however, is completely similar to what is already introduced by out of order processing. We thus expect that it will neither have a major influence on the simulation's

outcome. Again, this will have to be confirmed experimentally.

13.2.4 Control Messages

Among the messages exchanged on the network, some of them only serve to implement the simulation and would not have any counterparts on a concrete many-core machine. They are called *control messages* and are described in this Section.

The most important control message is the virtual time update message, which was already introduced in Section 12.1.3. This message is not routed, but rather processed by each NI. It holds a virtual time value that was reached by the neighbor core that sent it. This value is cached by the receiving NI, to speed up the time limit calculation done by the associated core for the purpose of spatial synchronization.

The simplest solution to ensure that tasks will keep making progress is that each core broadcasts its current time after each incrementation. Incidentally, doing so raises the probability that different cores can be simulated concurrently, because cores are informed of their neighbors' progress as soon as it happens, shifting further their allowed time window in the future. However, the number of exchanged messages would be overkill. As an example, if a timing annotation is introduced on average every 5 cycles, a 100 million cycles execution, corresponding to a small to medium benchmark, would imply that 20 millions time update messages are created, sent, received and processed, which would considerably slow down the simulation.

A necessary and sufficient condition to avoid deadlocks is that a core communicates somehow its just-reached virtual time before going to sleep because of spatial synchronization. Because we currently only run our simulator sequentially in only one system thread, this is the approach we chose. It is implemented by sending a virtual time update message when the thread representing a core relinquishes control, thanks to the possibility of registering context-switch hook functions into the scheduler. Investigating a trade-off between the two previous extreme approaches is left as part of a future study on the simulator parallelization.

The other control message used by the simulator is the *shutdown* message, used to implement the simulator shutdown at end of execution. Shutdown phases are often overlooked although they are critically important for correct and practical implementations. Additionally, they are often not trivial to implement. The main concern in shutting down the simulator is to cleanly signal all threads that they have to exit.

When the simulated application exits, shutdown messages are sent to all NIs through gradual broadcasts to neighbors. But NI threads can't exit as soon as they receive such a message. Indeed, since they are potentially multiple routes between two NIs, several shutdown messages will be sent to the same NI. Shutting down the associated thread would disrupt the links and messages still arriving would crash the simulator. NI threads thus have to wait for shutdown messages from all neighbors before exiting, but they should relay only the first shutdown seen.

Unwanted interactions between control and applicative messages could occur, if the delivery of a control message is required for the simulator to make progress but this delivery

is blocked. This can happen for example if network buffers are exhausted by application messages which are not going to be consumed without delivery of a control message. A general solution to this problem is to have control messages exchanged on a separate virtual circuit, with its own buffers. In the current implementation, this was not necessary because of the guarantee that all messages in receive buffers will eventually be processed without the need to receive any intervening control messages. This guarantee is provided both by the simple handling of locks and critical sections, described in Section 12.2.2, and by the interactions between the network interfaces and cores, described in the next Section.

13.2.5 Network Interface and Core Interactions

A network interface processing a message usually has to interact with its associated core. As an example, a task may be waiting for some data from another task in order to continue execution. In this case, the NI receiving the data must wake-up its associated task. Naturally, it will make the task restart at its own current virtual time stamp, itself computed from the message time stamp at reception (see Section 12.1.2).

The other important particular case where the NI interacts with a core in our implementation is for remote memory access requests. In hardware, such a request would be serviced by a memory controller, in the general case, or a processor cache, for cache-coherent shared-memory. In the meantime, the NI would queue the request and process other messages, until memory sends the reply. Since we do not model separately cores and memory, and because a thread representing a core can't be preempted, we decided that the NI thread would handle the request itself.

This choice creates a situation where both the NI thread and the core thread will compete for some memory locations, introducing new deadlock cases since the NI thread, while waiting for a resource, can't process any incoming messages. In our current implementation, we took care to guarantee that the resource held by the core thread will eventually be released without the need to receive any other message in the meantime. In practice, this added to the programming model the constraint that each core can operate on only one lock-protected memory region at a time. The use of the simple locks and critical section handling (see Section 12.2.2) finally ensured that the held lock will eventually be released without receiving further virtual time update messages.

A more general way to handle such a situation would be to have a dedicated thread for memory management that would wait for the requested memory area to become available. This, in addition with the use of virtual circuits, exposed in the previous Section, would simplify deadlock avoidance, would allow to use the more complex scheme exposed in 12.2.2 and would remove the aforementioned constraint on the programming model. However, this latter rule has the important benefit of avoiding any deadlocks in applications. So it is arguable that it is actually a feature rather than a limitation in the current design.

13.2.6 Programming Model Support

Programming model support in the simulator is located in the run-time system layer (see Section 13.1.2 for the software architecture overview). In the current implementation, support is provided only for the CAPSULE programming model, as described in Part I, enhanced with support for distributed-memory architectures, as presented in Part II. In addition to evaluating our approach, this choice considerably eases programming because there is no need to change the code as the simulated architecture varies.

There are no theoretical obstacles to adding support for simpler distributed-memory programming models, such as some MPI [94] support. However, we expect that programming using such an interface in a way generic enough to support architectural variations, while still retaining decent performance, will be a daunting task. Only MPI 2.0 specifies an API for spawning processes, which is very primitive. Moreover, it doesn't provide portable mechanisms to discover important hardware characteristics, such as the underlying architecture topology. Extensions to it will have to be designed, coded and used. The amount of work required will thus practically be largely superior to the time that was spent to support our integrated programming model.

The CAPSULE run-time system generates messages to drive task dispatching and object movements upon program requests. When a program attempts a task spawn, it calls the `capsule_probe` primitive, triggering a resource check in the core's local task queue. The task propagation mechanism then ensures that tasks are properly balanced in all the cores' task queues. It does so using the `RESOURCE_UPDATE` message, which a core sends to its neighbors to inform them about its current task queue's occupancy any time it changes. Using these messages, cores maintain local proxies to neighbors' occupation status. With the same aim, `RESOURCE_UPDATE` messages piggyback the tasks to be transferred to another core as a result of the chosen policy, based on the sending core's current idea of a neighbor's queue occupancy. Since the latter may have changed concurrently, the transferred context may ultimately be refused. The destination core thus responds to a `RESOURCE_UPDATE` message containing a context with a `CTXT_ACK` or a `CTXT_NACK` message.

A complete description of the load-balancing policies and algorithms used can be found in Chapter 9. In our implementation, task queue size is configurable, as well as the task difference threshold above which local migrations are enacted. The threshold limiting probes is hardcoded to the task queue size, i.e., probes succeed as long as there are free task slots in the task queue. In the experiments performed in Chapter 14, the task queue size was set to 4 and the task difference threshold to 1.

Shared data are stored in cell objects and program access them by dereferencing links, generalized pointers that can reference cells be they stored locally or remotely (see Section 8.1). The run-time system automatically sends messages (`DATA_REQUEST` and `DATA_RESPONSE`) to retrieve remote cell content when requested and locks the cell for the access duration.

Finally, coarse synchronization is expressed thanks to task grouping (see Section 2.4). Each time a task terminates, it decrements the active tasks counter in a cell specific to its

group, potentially generating a remote object access. By calling the `capsule_group_wait` primitive, a task can wait for active tasks in a group to finish. During the wait, its execution context is saved on the current core and resumes when the latter receives a notification (`JOINER_REQUEST`) from the last active task in the group.

Chapter 14

Experimental Evaluation

14.1 Framework and Methodology

14.1.1 Simulator Parameters

Architecture Configuration

All benchmarks are annotated in order to mimick architectures comprising scalar 5-stage pipeline cores with timings and functional units similar to the 32-bit PowerPC 405 CPU[1]. Each core has a private L1 cache with 1-cycle latency, whose policy is simple and pessimistic (see Section 13.2.1). The instructions in the ISA are grouped by classes, including unconditional branches, conditional ones, common integer arithmetic, integer multiply, simple floating-point arithmetic, floating-point multiply and floating-point divide. All the instructions within a given class share a single time value. Branch prediction is handled specially. Where its outcome is known with certainty at compilation time[2], its effect are included in the timing annotations and a 5-cycle penalty is applied to the mispredicted branch. For the other cases, a probabilistic branch predictor with a 90% success rate is assumed.

For scalability and architecture exploration, we use two different types of architectures. The first type is a shared-memory architecture in which all cores, besides their private L1 cache, access shared-memory banks with a common low latency (10 cycles). The delays induced by cache coherence effects are not taken into account. The purpose of this optimistic architecture model is to study inherent program scalability. This architecture is actually obtained from a distributed-memory architecture, described below, where all costs associated with object movement and run-time system operation, such as the spawning delay, the time to start a task or data transmission between cores, are set to 0.

The second architecture type features distributed-memory without cache coherence. The run-time system handles shared data as described in Section 13.2.6. Cores comprise a L2 cache with 10-cycle latency. The base link traversal latency between two of them is

[1] Floating-point instructions are treated similarly as in PowerPC 750.
[2] This is true, for example, for unconditional branches and loop constructs.

set to 1 cycle and the bandwidth to 128 bytes per cycle. The purpose of this architecture type is to present results using realistic and currently common hardware parameters while anticipating short to mid-term improvements in network bandwidth. Its completely distributed design is one of the possible choices for future many-core architectures to avoid the overhead of systematic hardware cache coherence.

Cycle-Level Parameters

In order to validate the results obtained by SiMany, we compare them to those obtained on a hybrid cycle-level/system-level simulator based on the UNISIM framework [14] up to 64 cores. This simulator models architectures of the shared-memory type described above, except that L1 caches are really modeled, are split into separate instruction and data caches, and that cache coherence effects influence the reported number of cycles.

UNISIM uses a traditional directory-based cache coherence allowing read-only data sharing, whereas SiMany runs the distributed CAPSULE run-time system that allows a data cell to solely reside on a single node at a time, regardless of the concurrent accesses' types. To perform a fairer comparison, SiMany's benchmarks used in the validation are modified versions in which big and mostly read-only data structures, that are shared when the benchmarks are run on UNISIM, are replicated on all nodes of the underlying distributed-memory architecture used for SiMany's runs. Small read/write structures are still handled by the CAPSULE run-time system, which is comparable to the UNISIM mechanism for a small or moderate number of cores, for which contention on the central directory is not a bottleneck.

Architecture Exploration

In addition to uniform 8, 64, 256 and 1024 cores 2D meshes, we simulate our benchmarks on clustered architectures with the same number of cores but split into 4 or 8 clusters. The latency of network links inside a cluster is set to 4 times the base latency. The intra-cluster latency, on the other hand, is set to half the base latency. We also simulate our benchmarks on polymorphic architectures where one core out of two is twice slower than base cores and the other faster by a factor of 3/2. These latter architectures thus have exactly the same cumulated computing power as the uniform ones.

Because the multipliers supported by SiMany are necessarily integer ones (see Section 13.2.1), the base configuration already uses a multiplier value of 3 for the simulated cores and of 2 for the link latency between cores. For the clustered architectures, the link latency multiplier is removed on the intra-cluster links, which effectively halves the latency compared to the base configuration and has the side effect of raising the maximum bandwidth to 256 bytes per cycle. For the polymorphic architectures, the 3/2 factor with respect to cores in the uniform meshes is obtained by using cores with a multiplier of 2. The slower cores simply use a multiplier of 6.

Virtual Timing Parameters

The reference value for the maximum local drift parameter D is 100 cycles and this value is used for all experiments, except when studying the effect of varying D on speed and accuracy, where values of 50, 500 and 1000 are also used.

The run-time system advances virtual time on its own to account for task management. Starting a task on a core has an overhead of 5 cycles, in addition to the time to fetch the task from its slot in the NI's task queue. Tasks are exchanged between cores in 2 cycles, according to neighbors' task queue occupancy information processed into hints by the NI in 1 cycle. A context switch to a joining task resuming execution costs 10 cycles. Probe decisions, based on the local task queue occupancy, are done in 1 cycle.

14.1.2 Benchmarks

Our choice of benchmarks follows the dwarf approach's philosophy advocated by researchers at Berkeley [12], which proposes a set of kernels deemed representative of large classes that encompass current and future parallel programs. Porting a full suite comprising numerous benchmarks to a task-oriented environment and making it support both shared and distributed-memory architectures would have consumed an exorbitant amount of time. We thus decided to focus on benchmarks that include a wide range of computation and communication patterns. Most of them are notoriously difficult to parallelize because of their complex control flow and/or data structures. We argue that, in the context of the advent of many-core in general purpose computing, studying the scalability of irregular benchmarks is as relevant, if not more, as that of niche scientific applications.

We present a parallel version of `Quicksort` adapted to work on lists instead of the traditional array version, in order to avoid copying cells or transferring the complete array to remote processing nodes. Pivot steps are distributed and they gradually construct a binary search tree. In a second step, the sorted list is sequentially reconstructed by traversing this tree in-order which brings back all list nodes on a single processing node's memory. This last operation is necessary only if a program needs to process the sorted list sequentially afterwards. Execution times without this last operation are reported. We used 50 random lists with 100,000 elements for common experiments. To further explore scalability, we additionally tested balanced lists of 1M elements.

A graph's `Connected Components` computation is a common graph algorithm, which is for example used in image processing. Since the graph topology is not known in advance, depth-first searches are launched from most of the nodes, resulting in contention for nodes belonging to the same component, although the conditional division mechanism mitigates this issue. Parallelizing this algorithm has already been studied but solutions were proposed in the context of shared-memory machines with large structures being shared between nodes [64, 118]. We ran our algorithm on 50 random graphs with 1000 nodes and 2000 edges.

We have also parallelized `Dijkstra`'s shortest paths algorithm, used for routing and navigation purposes, as described in [202]. It bears some similarity with the connected

components algorithm except that already explored paths may have to be explored again when reached with a lower value of the current distance computed. On the other hand, a task encountering an already explored path close to the optimal can terminate quickly and free a core so that it can be reused for more interesting paths. Again, sophisticated variants have already been studied [69] but they require frequent global synchronizations, which we wanted to avoid. The code for this benchmark is reproduced in Appendix B. We ran the algorithm on 50 random graphs of 2000 nodes having 3000 edges on average.

`Barnes-Hut` is the well known N-body simulation algorithm. It partitions space into a hierarchical tree. Each internal node in the tree represents the center of mass of all the bodies in the underlying subtree, while the leaf nodes are the bodies. The construction of this tree is the first phase of the algorithm. In the second phase, the force on each body B is computed by traversing the tree starting at the root. If the node is a group of bodies (represented by the center of mass) far away from B, the interaction with each body in the group is approximated by the summarized interaction with the center of mass. Otherwise, the subtrees rooted at that node are traversed. The force on a given body can be computed independently of that of the other bodies, and thus in parallel. The resulting communication patterns are highly irregular [115]. Only the scalability of the second phase is reported, assuming that the built tree has been broadcasted to all cores before it starts. We used 4 data sets with 128 bodies and 4 data sets with 200 bodies.

`SpMxV` is the sparse matrix-vector multiply algorithm. Matrices are specified in a row-oriented format alike to the Harwell-Boeing format. We used 30 matrices coming from a freely available sparse matrix collection [43] and 60 randomly-generated matrices of size $10^6 \times 10^6$, half of them having an average of 50 non-null coefficients per row and the other half 100 of them. For the validation experiments, only the first group of 30 matrices is used because of excessive cycle-level simulation time.

Finally, we use a tree traversal algorithm that updates all objects within an `Octree` structure. This scenario is typically used in gaming or for graphics generation. We ran the experiments with 50 randomly generated octrees of depth 6.

14.2 Experimental Results and Hardware Exploration

14.2.1 Simulator Validation

The virtual time speedups obtained by SiMany and by the UNISIM-based simulator for shared-memory architectures with cache coherence are compared in Figure 14.1 (facing page) for uniform 2D meshes and in Figure 14.2 (facing page) for polymorphic ones. In the legend, CL stands for Cycle-Level and designates the results from the UNISIM-based simulator. VT stands for Virtual Time and indicates results obtained with SiMany. Both axes employ a logarithmic scale, which tends to emphazise scalability differences at the bottom of the graphs. One can see that, for every benchmark, SiMany correctly captures the speedup evolution as the number of cores increases.

SiMany's results are quantitatively close to the reference ones. The geometric mean of the errors with respect to the cycle-level simulator results for uniform meshes are 8.8% for

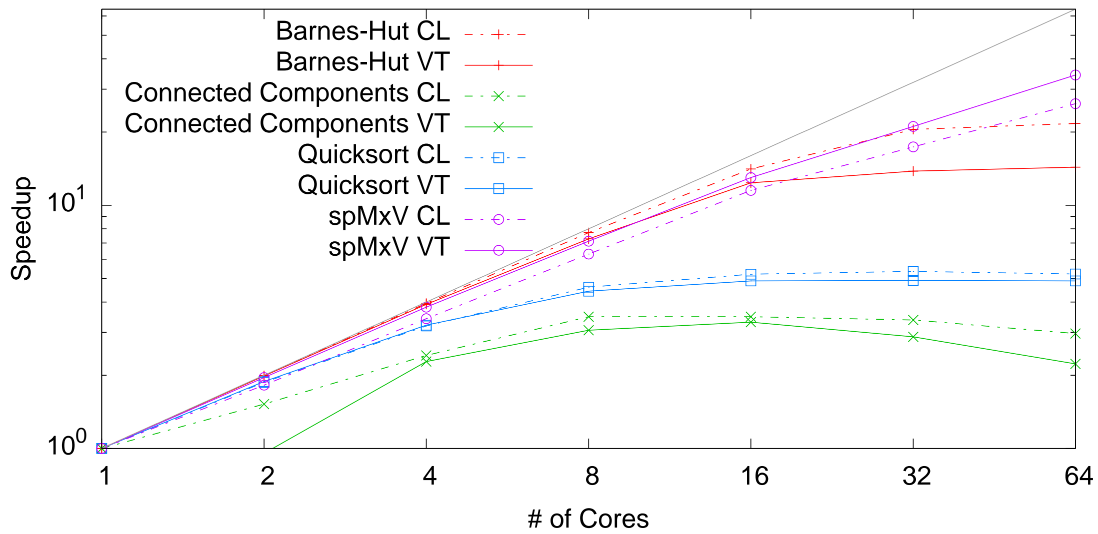

Figure 14.1: Regular 2D Mesh Speedups Cycle-Level Comparison.

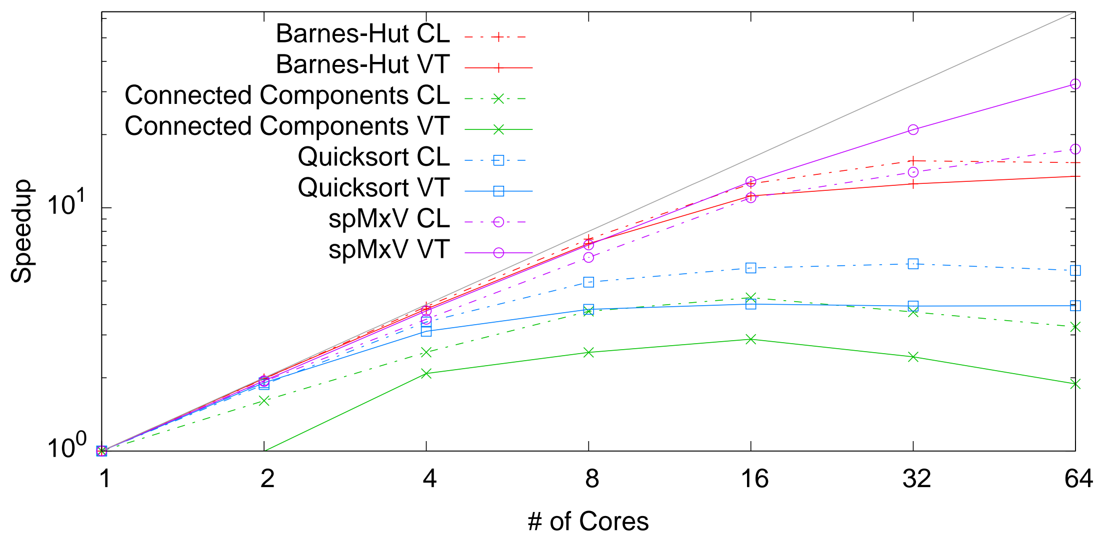

Figure 14.2: Polymorphic 2D Mesh Speedups Cycle-Level Comparison.

16 cores, 18.8% for 32 cores and 22.9% for 64 cores. The error for 16 cores is comparable to that obtained by other simulation techniques, such as interval simulation, sampling or statistical simulation on 2 to 8 cores. For 4 and 8 cores, the error is smaller but close to the 16 cores' one. These results validate our intuition that simulating the smallest details of the microarchitecture is less important to many-core simulation than it is for single-core or dual-core simulation.

For polymorphic meshes, the errors are 22.2% for 16 cores, 30.3% for 32 cores and 33.4% for 64 cores. The higher errors for these meshes are due to slightly different implementations of the polymorphic architectures. In the UNISIM-based simulator, the L1 cache speed is the same for all cores, whereas in SiMany it is proportional to the core speed. For this reason, the cycle-level curves in Figure 14.2 (previous page) are slightly offset upwards compared to those of Figure 14.1 (preceding page), whereas the virtual timing curves practically don't change, since a polymorphic architecture has the same computing power as the uniform one with the same number of cores.

We observe that, for both architectures, the error increases at a much slower pace than the number of cores. More importantly, the trends exhibited by the benchmarks on the reference simulator are fully reproduced in SiMany. For `Barnes-Hut`, the speedup is close to ideal until 16 cores, the point of diminishing return after which the curves start to flatten. For `Connected Components`, scalability reaches a peak at 16 cores and then decreases rapidly, because of the high contention on graph nodes as their tag are changed. `SpMxV` scales well up to 64 cores.

14.2.2 Simulation Speed

Figure 14.3 (facing page) shows the overall simulation time for every benchmark in all architecture configurations, normalized to native execution on a single-core machine. The time required to simulate most of the benchmarks on 1024-core architectures is on the order of 10^4 compared to native execution.

The higher simulation time for `Barnes-Hut` and `Connected Components` are due to the distributed-memory architecture simulations. We recall that, on these architectures, the run-time system is responsible for handling shared data. It actually implements data access as an exclusive operation, requiring data transfer to the core that needs them, whether the access is a read or a write. Algorithms that frequently access shared data will thus cause a high number of messages to be exchanged. On this respect, `Quicksort`, `SpMxV` and `Octree`, which exhibit no or little data dependencies, are much more representative of the simulator intrinsic behavior and performance. A regression shows that the average simulation time increases as a square law with a small coefficient.

SiMany is considerably faster than previous approaches. The best sampling and statistical simulation techniques so far [191, 205] can produce a 10^4 simulation time speedup for 4 cores relative to cycle-level simulation, whose best normalized simulation time is currently around 10^6, giving a net simulation time of 10^2. SiMany simulates our benchmarks on 4 cores with a normalized simulation time of 26, at the expense of some accuracy for this low number of cores. The recently proposed Graphite simulator [182]

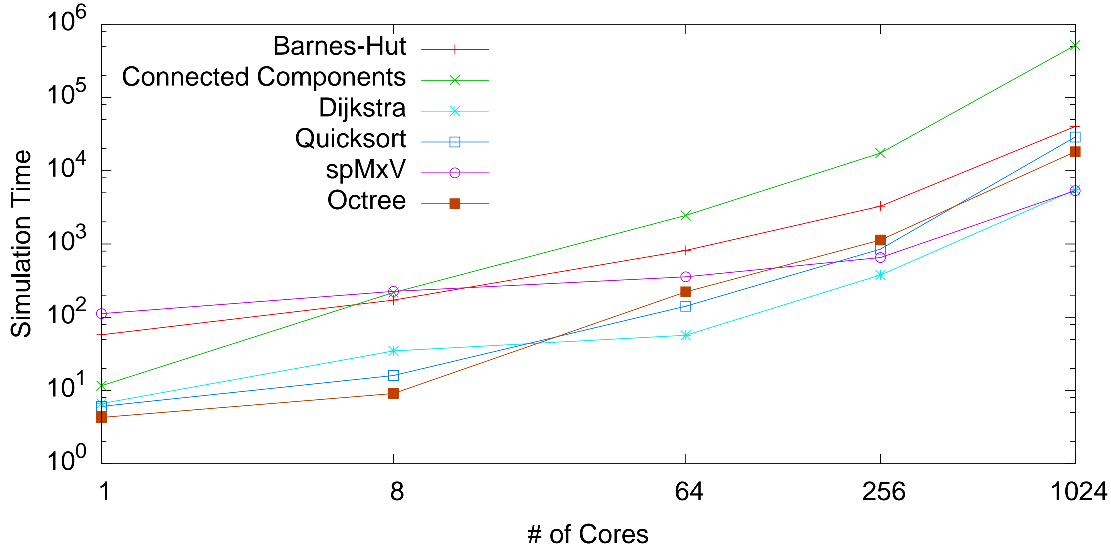

Figure 14.3: Average Normalized Simulation Time.

reaches a normalized simulation time of 1751 in average to simulate a 32-core architecture using 8 host cores, which, scaled to a single-core, gives a simulation time of around 14000. SiMany's corresponding normalized simulation time is 112 with only one host core being used.

14.2.3 Speedups on Regular 2D meshes

Figure 14.4 (following page) presents the benchmark virtual time speedups on our shared-memory architecture type, as explained in Section 14.1.1, whereas Figure 14.5 (page 197) presents results obtained on distributed-memory architectures.

On the optimistic shared-memory architectures, `Dijkstra` performs best and exhibits super-linear speedups (up to 4282 at 256 cores), which is purely due to the algorithm and the data sets used. Indeed, more cores enable more parallel tasks which increases the probability to tag nodes optimally, which, in our algorithm, leads to giving up non-interesting paths quicker. `SpMxV` scales well up to 256 cores, reaching a 153.6 speedup, but then hits a ceiling and doesn't take advantage of more cores, with a speedup of 157.6 at 1024 cores, essentially because of the size of the data sets we used. Performance of `Quicksort` may be surprisingly bad. In fact, the theoretical maximum speedup reachable by `Quicksort` is $\log_2(n)/2$ for balanced arrays of n elements. We used lists of 100,000 elements, in which case the ideal speedup is about 8.30. In practice, we reached no more than 5.72. Figure 14.6 (page 197) complements these results by presenting speedups obtained on a few balanced lists of 10^4 to 10^6 elements. A least-squares regression on these results gives the formula $0.95\log_2(n)/2$ with 2% error. `Barnes-Hut`'s performance tops at 15.1 (256 cores) and `Octree`'s at 6.9 (1024 cores). It is interesting to note that, for most benchmarks, going from 256 to 1024 cores doesn't make a big difference and

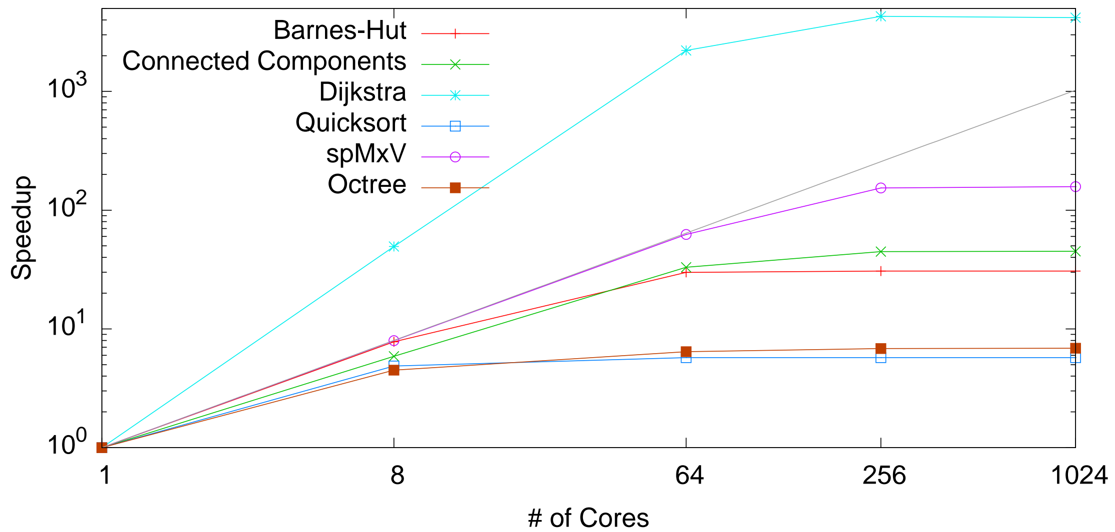

Figure 14.4: Regular 2D Mesh Speedups (Shared-Memory).

sometimes even lowers performance a bit.

On the realistic distributed-memory architectures, `Quicksort`'s and in particular `SpMxV`'s results do not significantly change, because they cause little data movement and no contention on cells. Unsurprisingly, the performance of data-contended benchmarks, such as `Dijkstra` and `Connected Components`, collapses, showing how great these benchmarks' sensitivity to communication costs is. `Connected Components`'s performance actually degrades above 8 cores, despite the run-time system's load-balancing property.

14.2.4 Simulation Time/Accuracy Trade-Off

We hereby study the practical effects of varying the maximum local drift parameter D. In addition to the baseline ($D = 100$), experiments were performed with values of 50, 500 and 1000 for the shared-memory architecture type. As explained in Section 12.1.3, increasing D means relaxing the spatial synchronization, allowing to simulate each task for a longer time window, which increases temporal and spatial locality and potentially causes more messages to be processed out-of-order.

Figure 14.7 (page 198), Figure 14.8 (page 198) and Figure 14.9 (page 199) show the virtual speedup results for D equal to 50, 500 and 1000 respectively. These results are to be compared to those of Figure 14.4, which where obtained with the baseline value of D. Figure 14.14 (page 201) summarizes the average speedup differences to the baseline values. Only the values for 64, 256 and 1024 cores are considered in the average, since they give the interesting part of the benchmarks' scalability profile and exhibit the highest variations.

Similarly, Figure 14.11 (page 200), Figure 14.12 (page 200) and Figure 14.13 (page 201)

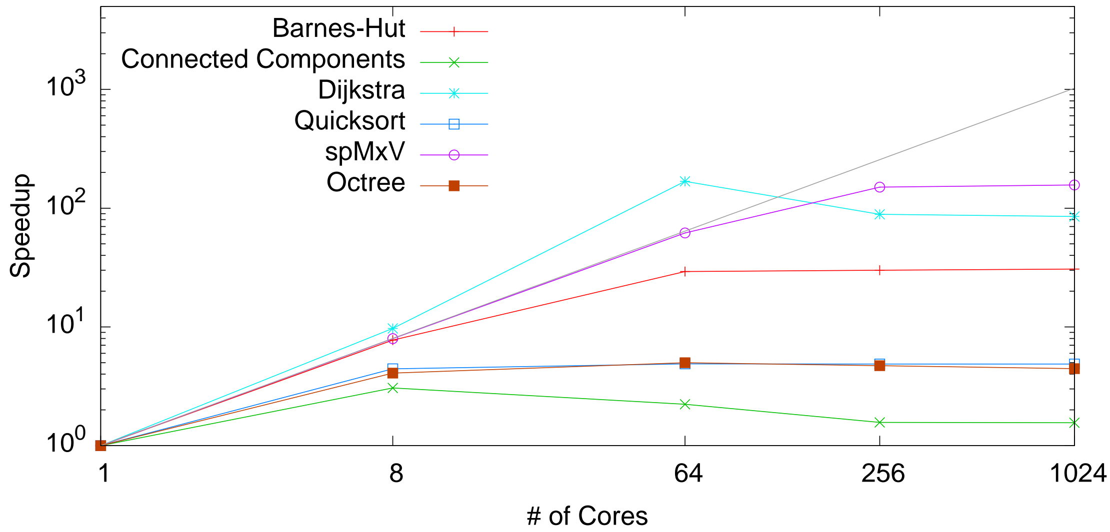

Figure 14.5: Regular 2D Mesh Speedups (Distributed-Memory).

# of elements	10^4	10^5	10^6
Practical Speedup	6.46	7.96	9.26
Theoretical Speedup	6.64	8.30	9.97

Figure 14.6: Speedups for `Quicksort` with Balanced Arrays.

show the normalized simulation time results for D equal to 50, 500 and 1000 respectively. The baseline results are given in Figure 14.10 (page 199). Figure 14.15 (page 201) summarizes the average relative simulation time difference for all values of D with respect to the baseline for all benchmarks (the average is over 64, 256 and 1024 cores).

Lowering D to 50 increases simulation time for all benchmarks (17.1% on average), as expected. Speedup variation is only a few percents for each benchmark. Raising D to 1000, on the other hand, speeds up simulation by an average 2.33 factor (4.04 when considering only 1024-core architectures).

Figure 14.14 (page 201) shows that only `Dijkstra` and `Connected Components`, whose algorithms are highly dependent on the tasks' timings, exhibit speedup variations of more than a few percents. They perform worse because simulation of cores is less intermixed when D is high, which decreases the probability of exploring a good path quickly and thus increases the amount of work to perform. By contrast, the performance of even relatively contended cases (`Barnes-Hut` and `Octree`) doesn't vary much (respectively -2.9% and -1.1% for 1024 cores). Regular benchmarks practically don't exhibit any changes. It is thus interesting to increase D if it is known that the specific characteristics of a given algorithm or application make it barely sensitive to timing variations.

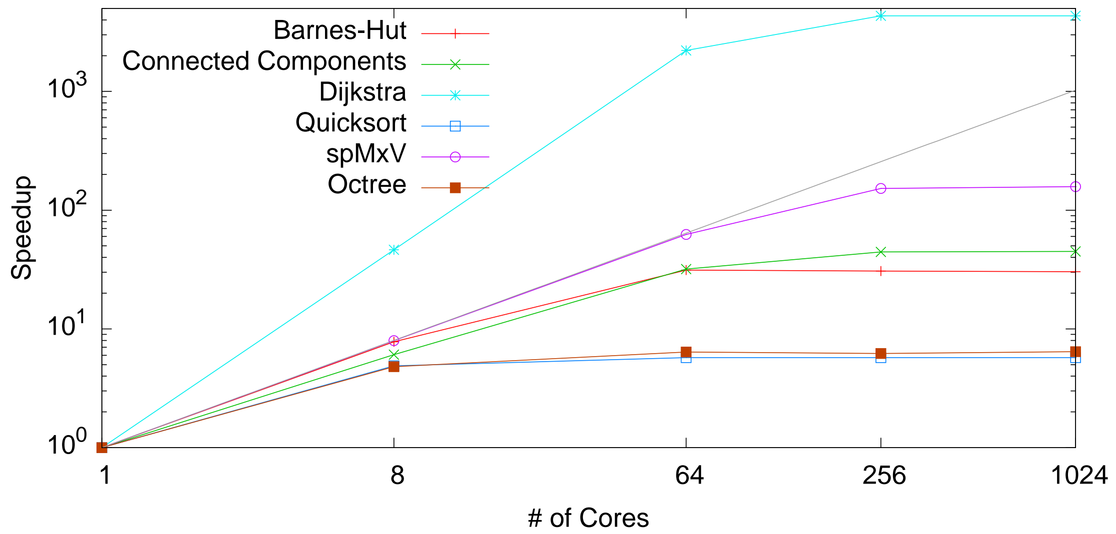

Figure 14.7: Regular 2D Mesh Speedups with $D = 50$ (Shared-Memory).

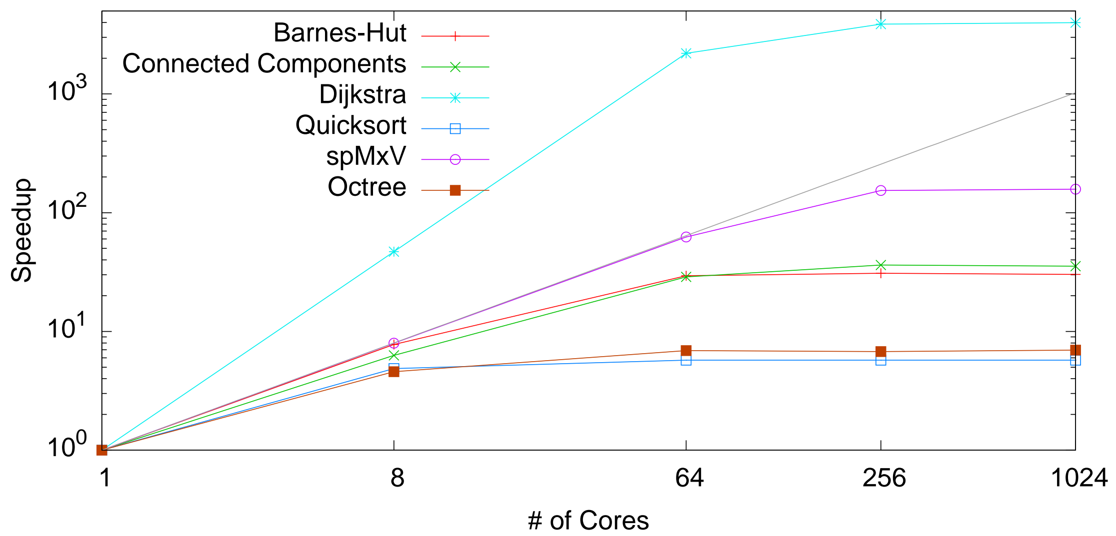

Figure 14.8: Regular 2D Mesh Speedups with $D = 500$ (Shared-Memory).

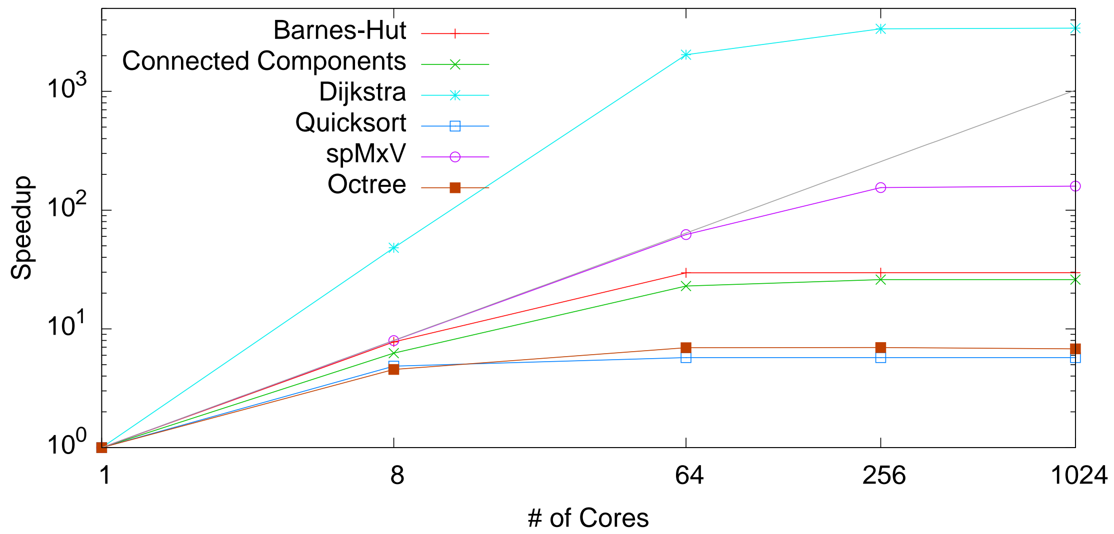

Figure 14.9: Regular 2D Mesh Speedups with $D = 1000$ (Shared-Memory).

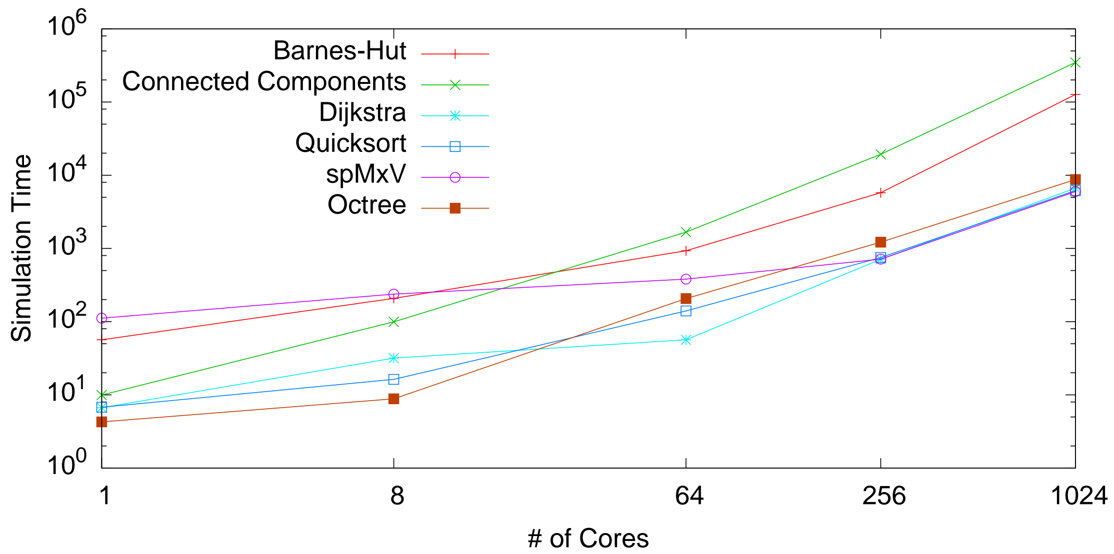

Figure 14.10: Normalized Simulation Time for Regular 2D Meshes with $D = 100$ (Shared-Memory).

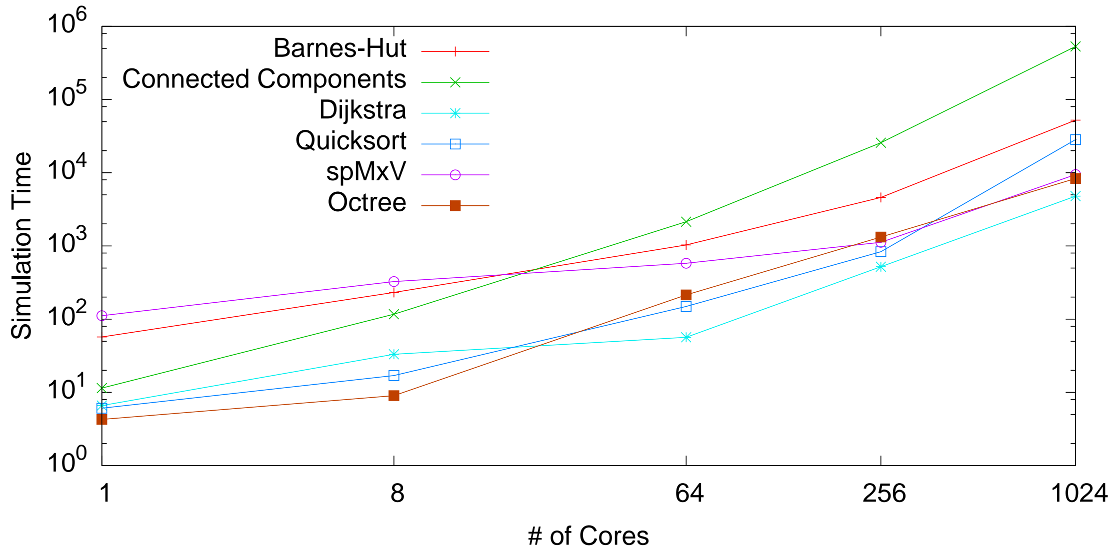

Figure 14.11: Normalized Simulation Time for Regular 2D Meshes with $D = 50$ (Shared-Memory).

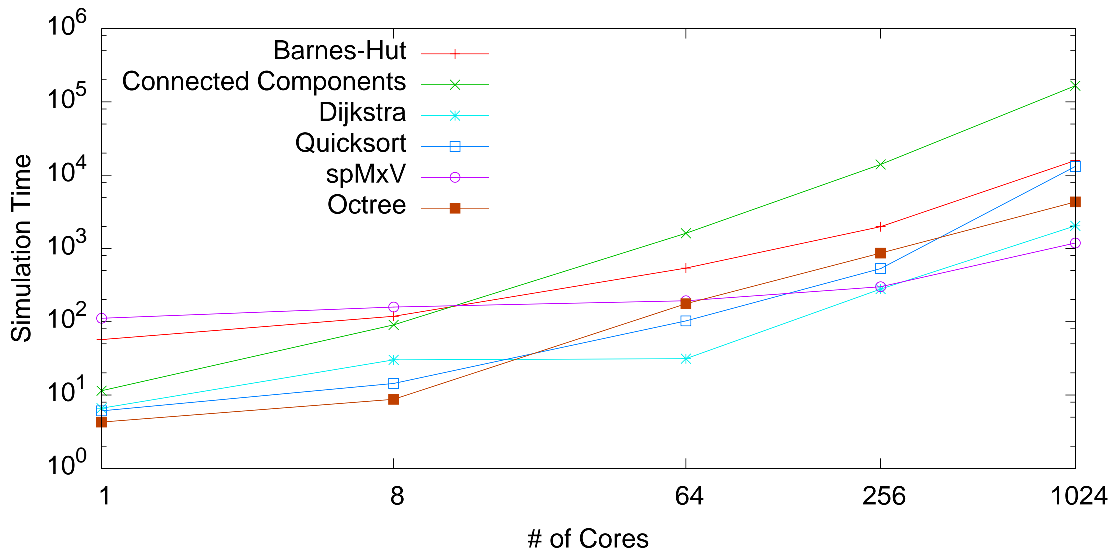

Figure 14.12: Normalized Simulation Time for Regular 2D Meshes with $D = 500$ (Shared-Memory).

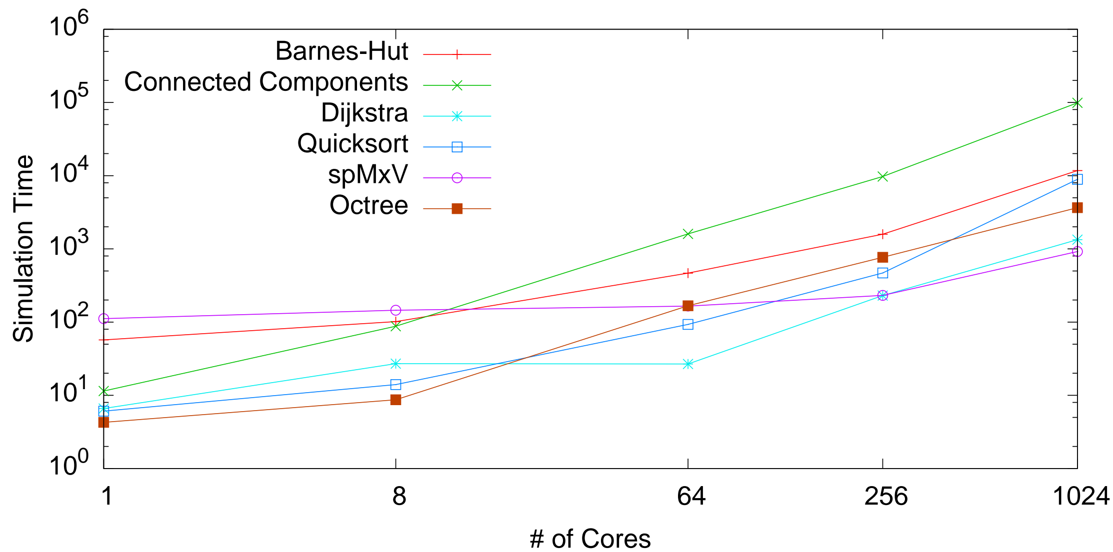

Figure 14.13: Normalized Simulation Time for Regular 2D Meshes with $D = 1000$ (Shared-Memory).

D	Barnes-Hut	Conn. Comp.	Dijkstra	Quicksort	SpMxV	Octree
50	1.2%	−1.6%	1.6%	0 %	−0.2%	−5.4%
500	−0.9%	−17.8%	−5.1%	0 %	0.1%	2.5%
1000	−2.4%	−38.6%	−16.1%	0.1%	0.5%	2.9%

Figure 14.14: Average Virtual Time Speedup Variations with D (Baseline: $D = 100$).

D	Barnes-Hut	Conn. Comp.	Dijkstra	Quicksort	SpMxV	Octree
50	6.9%	22.2%	11.2%	12.3%	55.1%	2.1%
500	−56.2%	−38.3%	−44.3%	−33.8%	−65.2%	−33.4%
1000	−65 %	−54.1%	−56.8%	−46.1%	−72.2%	−40.4%

Figure 14.15: Average Simulation Time Variations with D (Baseline: $D = 100$).

14.2.5 Clustered Architectures

Figure 14.16 (facing page) presents speedup results on a clustered distributed-memory architecture comprising 4 clusters. Intra-cluster latency is twice as low as the one used for our regular 2D mesh experiments, whereas inter-cluster latency is 4 times higher. Data-contended benchmarks' performance varies the most. For low numbers of cores, clusters are small and the inter-cluster latency dominates. Results in this case are better on the regular meshes. This situation reverses as the number of cores grows. The average turning point for all benchmarks is around 78 cores, with however large disparities between them (from 15 for `Barnes-Hut` to 139 for `Connected Components`). Virtual execution speedup on 1024 cores decreases by 28.7% for `Connected Components` and by 25.6% for `Dijkstra`, whereas it practically doesn't change for `Quicksort` (−2.2%) and `SpMxV` (−0.2%). These results are coherent with the fact that, in the former benchmarks, tasks continuously exchange vertex data, whereas in the latter ones, tasks don't communicate much and sensitivity to network latency is low.

Figure 14.17 (facing page) shows the speedup results for the experiments with 8 clusters. Results for 8 cores were not computed. 8 clusters on 8 cores implies that each core is a cluster on its own, with the consequence that all links have the same latency (the inter-cluster one). Therefore, the results would have been similar to those obtained for regular meshes with a different link latency for all links. The speedup variations, with the results for 4 clusters as a reference, are presented in Figure 14.18 (facing page). Compared to the results for 4 clusters, the individual speedups obtained are all lower except for `SpMxV` at 256 cores, whose speedup stays practically the same (+0.1% variation). The variations are lower as the number of cores increases, coherently with the fact that the ratios of inter-cluster links over intra-cluster ones for 4 and 8 clusters are getting closer[3], as shown in Figure 14.19 (page 204). The average variation is significant for `Connected Components` and `Dijkstra`, with −13.1% and −15%. By contrast, `Quicksort` and `SpMxV` have the lowest variations: −1.1% and −0.2%. As for the 4 clusters case, the variations between regular meshes and 8-cluster ones are coherent with the benchmarks' characteristics.

14.2.6 Polymorphic Architectures

Figure 14.20 (page 204) presents results on a polymorphic architecture in which one core out of two is slower and the other faster than in the tested uniform meshes, in a way that preserves the theoretical computing power of the mesh (see Section 14.1.1). The `Dijkstra`'s and `SpMxV`'s performances decrease slightly. The decline is higher for the other benchmarks (−18.8% on average for 256 and 1024 cores). The run-time system, which is not particularly tuned for such architectures, has a harder time at balancing the load because the slower cores can't spawn tasks at the same rate as faster cores.

[3] Their ratio, $(N-2)/(N-3)$, with N being the side size of a square mesh (here, 8, 16 or 32), indeed rapidly converges to 1.

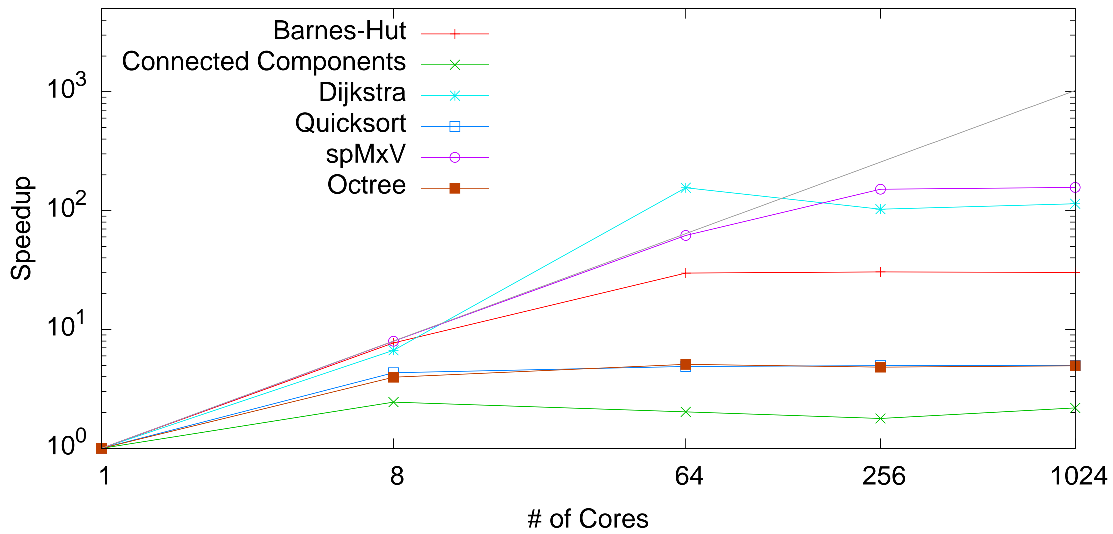

Figure 14.16: Clustered 2D Mesh Speedups with 4 Clusters (Distributed-Memory).

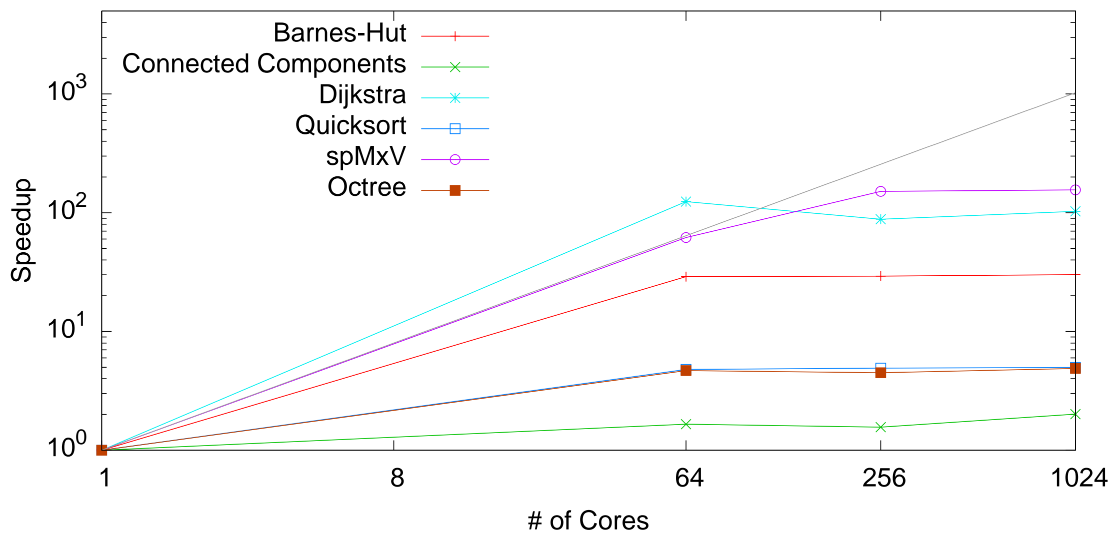

Figure 14.17: Clustered 2D Mesh Speedups with 8 Clusters (Distributed-Memory).

# of Cores	64	256	1024
Speedup Variation	−9%	−6.6%	−3.5%

Figure 14.18: Average Speedup Variations From 4 to 8 Clusters.

# of Cores	64	256	1024
4 Clusters	85.7%	93.3%	96.7%
8 Clusters	71.4%	86.7 %	93.5%

Figure 14.19: Ratios Intra-Cluster Over All Links For 4 and 8 Clusters.

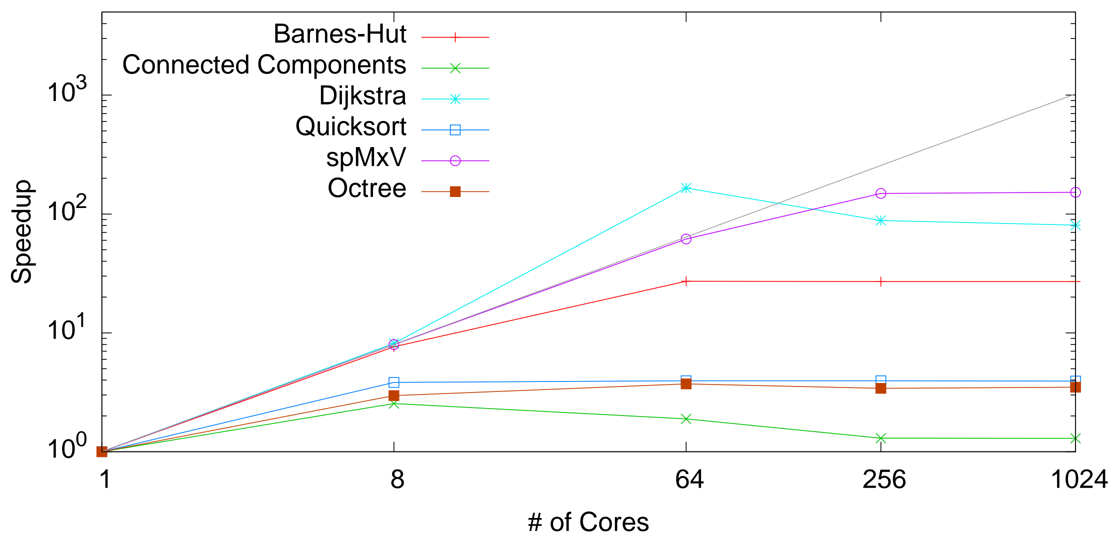

Figure 14.20: Polymorphic 2D Mesh Speedups (Distributed-Memory).

Chapter 15

Related Work

The traditional major problems in simulation are efficiency and accuracy [263]. Efficiency essentially sets the trade-off between the accuracy, the size and complexity of the architecture being simulated and the size of the programs being run on this architecture.

15.1 General Discrete-Events Simulation

The virtual time concept was initially introduced under the denomination of *logical clocks* by Lamport in a seminal article about the ordering of events in a distributed system [164]. This article pointed out that events specified in a distributed program are in essence weakly ordered. Executing the program will yield a total ordering, imposed by the computing system's physical details and clocks. Different executions on different machines, and even sometimes on the same machine, will lead to different total orderings and potentially different outcomes or perceived results. Their only guaranteed common subset is the program's weak order specification, a required property to ensure correct execution. The end of the article deals with the possibility of capturing the global order of a distributed execution by using distributed physical clocks that don't drift too much apart.

Based on this concept, Chandy and Misra introduced *distributed simulation* [58], meant to simulate distributed programs without the need for any global control, which obviously would hinder scalability. But, although they share the same logical clock concept, the intent behind distributed simulation is actually the *opposite* of Lamport's reasoning: The point is not so much in highlighting the different global orderings a distributed program can give spring to than actually *constructing* one of them from the program specification, as a physical machine executing the program would do.

The simulation technique presented in this paper is part of a set of such constructions commonly referred to as *conservative* distributed simulation. Indeed, processes are allowed to make progress to some time t only when they have received all messages that completely determine the process' behavior up to t. Advancing processes may generate new messages that are delivered in order to each other process. The different processes are simulated by independent physical machines between supposed interaction points. The implicitly constructed global ordering of events here depends on the order in which non-blocked

processes are simulated, i.e., on physical characteristics of the distributed machines used to run the simulation.

This technique seems very natural but its implementation is not trivial. While the arrival of a message with a given time stamp t offers the guarantee that the initiator process has progressed at least up to t, because of message ordering, the *absence* of such messages is uninformative and can lead to deadlocks. As an example, let us consider a process P_A with current time t_A that may receive messages from at least 2 processes P_B and P_C. P_B may have sent a message at $t_B > t_A$, but P_A may not start processing it and raise its clock, simply because of the possibility that P_C may send it a message at t_C with $t_A \leq t_C < t_B$, in which case processing first the message from P_B would be wrong. Incoming messages from P_B are thus blocked in the meantime. If this happens to block in turn another process P_D, on which P_C depends to output a message to P_A, then the system is deadlocked. Note that this can happen in a very simple acyclic network. The possibility of such deadlocks is actually tied to the use of bounded-length queues and the conservative simulation rules.

Chandy and Misra proposed several ways to deal with deadlocks. In the original paper, they introduced *NULL messages*, which are sent by a process P_i that is making progress but has no real messages to send to some other processes. NULL messages simply contain the timestamp t_i of P_i, informing the receiving processes of P_i's progress. They can be shown to eventually allow the simulation to continue [58]. There may be, however, a huge number of them exchanged, particularly when processes are idle most of the time.

Another approach is to let deadlocks happen and solve them after the fact rather than trying to prevent them by sending NULL messages. Chandy and Misra also proposed a scheme based on this paradigm. It relies on some adaptation of the Dijkstra-Scholten algorithm [77] to work with CSP-like processes in order to detect that a deadlock occured. It essentially maintains a tree of *engaged* processes, i.e., processes still taking part in the current computation. The root of the tree is the global controller. Only leaves of this tree may become disenganged when they block, causing a message to be sent to their father. When the root of the tree finally receives such a message, it knows that the other processes are all blocked. This algorithm is described in [60].

When the global controller has finally received a signal indicating a deadlock, it wakes up all process to make them exchange information about their real progress and compute the new time to which they can advance. It can be shown [59] that at least one of them will then notice that it can resume execution because it can't be influenced any more by any other processes.

Chandy and Misra later proposed a completely distributed algorithm to detect deadlocks in [61]. One may contemplate to use it instead of the aforementioned global detection algorithm as a first step. But doing so without modifying the second step as well would be vain, since the latter still requires a global trigger. A process noticing the deadlock could trigger clock advances in the set of deadlocked processes it belongs to. As far as we know, nobody has proposed such a scheme yet.

Our approach's virtual time update messages are similar in nature to NULL messages. Their role is to indicate processes' advances to other processes. However, they are the sole

source of such information in our scheme, in which we intend to model a real network and associated interfaces. A message arriving at a node doesn't have the same timestamp as when it was sent. Its timestamp increase depends on the path it took through the network, which may be affected in non-trivial ways by contention and routing algorithms. Thus, the receiving process can't devise the virtual time of the sending process simply by looking at the message timestamp. Moreover, because of spatial synchronization, processes need only exchange timing information with their immediate neighbors. This considerably reduces the number of virtual time update messages created compared to NULL messages, while still providing a (weaker) global drift bound. Modeling a real network would be possible with Chandy and Misra's method, but this would require introducing processes standing for each real link, multiplying the number of necessary processes by the average network graph arity and adding up more NULL messages to the picture.

Jefferson coined the "virtual time" expression as part of a new framework's description, Time Warp [139], allowing processors in a system to be simulated in an even more independent fashion. As opposed to conservative distributed simulation techniques, Time Warp was the first *optimistic* one. Indeed, processes simulated within it only wait for the next incoming message, and handle it as soon as it arrives. However, messages may not actually reach a process in order of their virtual time stamps, since the ordering depends on the relative speed at which processes are simulated. To guarantee a completely faithful simulation, the Time Warp framework requires the possibility of canceling the effects of previously processed messages if a new message with lower virtual time comes in later.

This cancellation is called a *rollback*. It is implemented in Time Warp by keeping for each processor an history of saved states (one before each new message processing) along with their virtual time, which requires some sort of checkpointing. Since processed messages may in turn have generated other indirect messages, a queue of *anti-messages* corresponding to those also has to be maintained so that the simulator can cancel the initial message processing. After a rollback to an early state, the anti-messages stamped with a comparatively higher virtual time are sent. When they arrive at processes' message queues, the system looks if the associated message is still in the queue. If it is, the message has not yet been processed and is simply removed from the queue. If it is not, the receiving process rollbacks to the anti-message's virtual time.

The amount of required history per process raises with the number of exchanged messages. As histories and anti-messages queues grow too large, global resynchronization must happen. At this point, the minimum virtual time of all actors, called the *global virtual time*, is computed and broadcasted, which requires several broadcasts in all the proposed variants [217]. Each state or anti-message that has a comparatively lower time can be discarded. No rollback will ever need them since all new messages will be necessarily stamped with a greater time than the global virtual time.

Compared to Time Warp, our approach differs on two main points. First, synchronization between actors is always done *locally*. No global virtual time is ever computed and broadcasted. Instead, a simulated core knows the current virtual time of its neighbors and only uses this knowledge to determine if it is late or early. We call this *spatial* simulation. Second, we go one step further than optimistic execution, by freeing ourselves

from the need to do any rollbacks and anti-messages generation. We define a local bounded-length virtual time window inside which received messages are allowed to be processed out-of-order with respect to their virtual time of reception. Within that time window, messages can then be processed as soon as they arrive and there is no need for checkpointing because no rollbacks ever need to be performed. This speculative technique considerably increases simulation efficiency and speed, at a reasonable expense of accuracy, as was shown in Section 14.2.

Optimistic time windows applied to the Time Warp framework, as proposed in [237], are similar in spirit than our speculative time windows, but serve a different purpose and are implemented quite differently. They are used to limit optimistic message processing, in the hope to restrain the amount of rollbacks that will have to be performed if too many messages are processed out-of-order. Let us remind that a single message cancellation may require lots of rollbacks, especially if processes have gotten more out of synchrony. Our local time drift limits the number of messages that will be processed out-of-order. In our case, the drift thus controls the degree of accuracy, in addition to the potential parallelism and the simulation performance. What is more, the optimistic time window implementation consists in allowing processing of messages that have a virtual time t smaller than the current global virtual time G plus a fixed window size parameter ω: $t < G + \omega$. As discussed earlier, the computation of G requires a global synchronization, contrary to our distributed spatial approach. The *bounded lag* approach [176] similarly requires several synchronizations and broadcasts. Moreover, it is unefficient when processes may affect each other in a short virtual time frame, i.e., when network links' latency is low.

The interested reader can find additional details on the field in the following survey papers: [98] by Fujimoto and [15] by Bagrodia.

15.2 Single-Core Simulation

There is a large body of work describing how to simulate single-core machines. We hereby do not pretend to be exhaustive, but tried to mention the most influential papers in the field in the past ten years. The main motivations for simulation are to be able to design and test hardware without having to produce real hardware prototypes, which are expensive and hardly modifiable. There have been essentially two directions of research in the simulation field for single-cores: Speeding up the simulation and facilitating simulator implementation and reuse.

15.2.1 Monolithic Simulation

The free and open-source SimpleScalar tool suite [45] has had a considerable success in providing a framework for in-order and superscalar processor simulation. It is an *execution-driven* simulator, i.e., instructions are executed by a realistic architecture model. It is written in C and has been optimized for speed, is quite portable to POSIX operating

systems, easily tunable and relatively extendable with respect to microarchitectural mechanisms, such as cache or branch prediction. The speed it can reach is several hundreds thousands of instructions per second. A large majority of microprocessors studies since then has been performed using it.

SimpleScalar has several limitations. For example, it has an overly simplistic memory model. But the most important limitation is that it is a *monolithic* simulator, where most components are tied up and communicate through shared variables. This makes it hard to change a given component because doing so may impact the code of all other components. Because coding a simulator and making sure it is reliable can take a long time, researchers and engineers strive to reuse code pieces when studying new mechanisms or evolving already established processors. This first lead to the separation of hardware pieces into modules inside the simulator, for cache hierarchy and branch prediction, in the ASIM [86] framework.

15.2.2 Modular Simulation

SystemC [116] is an industry-developed framework promoting the reuse of hardware components' simulators through common interfaces and objects. Schematically, components are separate modules. They declare ports and communicate through them with other modules. Ports are connected to other ports and exchange signals thanks to the underlying channel. Ports present to their module an interface, which the module must use, specifying the possible operations on the associated channel. SystemC is not tied to a particular level of abstraction, but it is somewhat inconvenient to simulate very low-level hardware details (tansistors) with it.

The Liberty Simulation Environment (LSE) [253] introduced a language to describe how modules operate together to form a complete architecture. This description allows to statically discover constant control signal or whether signals going through a single port are actually managed by the same code block. Also, with enough information on inputs/outputs dependencies, optimal static block scheduling can be devised. In other cases, static scheduling is complemented by dynamic scheduling. These optimizations are described in [203]. It is shown that they can more than compensate for the overhead due to the modular implementation.

UNISIM [14] is a framework aiming at building simulators from reusable components including architectural control parts that are traditionally hard to break down. To achieve this, it builds on techniques first proposed by LSE [253] and Microlib [206]. Its design allows the reuse of existing simulators by wrapping them inside UNISIM components. To speed up simulation, UNISIM integrates in its engine support for established techniques such as sampling and TLM, which we discuss below. The UNISIM engine is based on SystemC.

15.2.3 Speeding up the Simulation

Sampling

Simulation sampling is a general approach by which programs are only simulated during enough and/or representative short phases so that the results are deemed representative. It intuitively assumes that the behavior of a program during an interval is directly related to the code being executed during this interval. Basic block execution frequencies, or other architecture-independent metrics, are measured in fixed sized instruction intervals using functional simulation or an instrumented run of the program. These statistics can then serve to find program phases by tracking their changes as intervals get executed. Finally, a representative phase is chosen and simulated, which gives acceptable errors compared to the whole execution for a range of metrics (under 8% for IPC[1]) [233].

Alternatively, metrics can be clustered in a small number of representative ones, indicating that only a few significantly different behaviors actually occur during an execution. Simulation of representative intervals only is shown to produce close results to a full execution's ones, e.g. for cache misses [161] or IPC in SimPoint [234].

Another kind of simulation sampling is statistical sampling, in which enough samples are simulated so that they are deemed statistically representative of the whole execution. The choice of samples is performed randomly or systematically at fixed intervals. SMARTS [259] used systematic sampling, as presented in [67], and performed architectural warm-up before detailed simulation of samples in order to keep the cold start bias low. This latter bias is due to no or insufficient knowledge of the micro-architectural state (caches, pipeline) when the detailed simulation begins, because some of it, e.g., cache and TLB states, may require a very long history to be realistic. Bias can typically be high when *fast-forwarding* (purely functional simulation) is used between instruction ranges where detailed simulation is performed.

Sampling typically yields errors of a few percent, with 10% as the worst-case, compared to a detailed full program execution.

Statistical Simulation

Statistical simulation takes another approach to model program behavior on different architectures. It collects statistics from a program detailed simulation that are later used to generate a synthetic trace exhibiting the same statistical properties. This trace is then fed to a trace-driven simulator that generates the desired metric, usually the IPC. This approach was presented and refined by Nussbaum and Smith [195], Eeckhout and Bosschere [83] and Oskin et al. [199] in the HLS simulator.

Collected statistics typically comprise the distribution of classes of instructions, where each class determines which processor's functional units are used, the distribution of L1 and L2 cache hits/misses for data and instruction, and the distribution of the distance between dependent instructions. These distributions can be split into variants according

[1]Designates the number of Instructions Per Cycle.

to other parameters, such as the basic block size, the proximity to a branch instruction, the particular register dependency type between instructions, ...

Statistical sampling usually exhibits an estimation error of up to 10% for IPC, if the most elaborate mechanisms presented above are used.

Transaction-Level Modeling

Transaction-level modeling (TLM) was formally introduced by the Open SystemC Initiative (OSCI) consortium with the goal to foster interoperability between abstract models that communicate through messages rather than a handshaking protocol. In TLM approaches used for performance evaluation (Timed TLM or TTLM), each event in the system is assigned a timing and the different components have to synchronize before they can continue to make progress. The SystemC TLM 2.0 standard [197] proposes two modeling styles so that component models can provide multiple implementations showing up different accuracy/speed trade-offs.

The *Loosely Timed* style forces both data and timing results of a transaction to be returned as soon as it is initiated. This modeling style allows a high simulation speed and can thus be used for software development and functional verification. However, it can't allow to capture resource contention and its associated timing impact. Additionally, the simulator still issues and processes relatively low-level transactions to interpret local computation or control flow which is largely slower than our mostly direct code execution[2].

In the *Approximately Timed* style, modeling a component requires that other events be simulated or the current time advance enough so that the data and timing results of a transaction can be computed. This style can capture hardware resource contention because system components remain in lock step and is thus adapted to fine-grained hardware modeling and study. On the simulation speed front, each transaction execution causes several costly context switches and events ordering to happen. Moreover, components synchronization happens at each transaction, which strongly limits the achievable simulation speed. This style is thus not suitable to simulate a very large number of components.

The TLM principle can be applied outside of SystemC as well and need not rely on a low-level implementation of handshakes through signals.

Emulation

Emulation is functional simulation, with the goal of executing programs as fast as possible, sacrificing the internal modeling of hardware components. In general, such simulators don't allow to directly assess software and hardware performance. They are geared to validation of large platforms and programs. They often provide virtualization and allow *full-system* simulation. Some prominent examples are Simics [179] and QEMU [23].

[2] The TLM 2.0 whitepaper [198] reports reaching 50 millions of transactions per second. This is to be compared with current processors with frequency ranging from 2 to 3Ghz and an IPC of about 1 in practice.

However, they may be augmented with simple hardware models that allow rough timing and/or the ability to generate traces that can later be fed to *trace-driven* simulators to evaluate performance. Simics, for example, allows to study memory hierarchies by generating traces of memory accesses. Those models, however, can at best provide an order of magnitude estimate of performance.

15.3 Multi-Core and Many-Core Simulation

Multi-core and many-core simulations are difficult because of the number of hardware components involved and their interactions. Simulators able to exploit the several cores found in current machines linearly reduces the simulation time but don't compensate for the ever increasing number of execution units to simulate in many-core architectures, still orders of magnitude larger.

Today's best simulation speed using the techniques presented in Section 15.2 is still slower than native execution by 2 to 3 orders of magnitude for single-core runs. In the ideal case where they could be applied to multi-cores, it would be still practical, although lengthy, to simulate several cores, but, as explained in Chapter 11, many-core architectures would remain out of reach even for small benchmarks.

Unfortunately, the situation is less than ideal, because sampling techniques do not scale yet, and may not scale at all beyond hundreds of cores, as we will see in Section 15.3.1. So, all previous approaches to multi-processor or multi-core simulation rely on discrete-event simulation or other ad-hoc techniques (respectively described in Section 15.3.2 and Section 15.3.4). Very recent frameworks have introduced relaxed synchronization. We review them in Section 15.3.3.

15.3.1 Sampling Techniques May Not Scale

Unfortunately, sampling techniques can't be readily applied to multi-core, because of the influence of scheduling and thread interactions on performance. Sampling was proposed for the SMT case introducing the matrix of observed co-phases [31], i.e., the combination of the different phases the threads may be in concurrently. Phases are determined for each thread individually using SimPoint. Then, threads are simulated jointly. As they enter a new co-phase, the simulator determines if a similar co-phase has already been simulated in detail. If this is the case, the simulator identifies a co-phase class the new co-phase belongs to, and both threads may then be fast-forwarded out of the co-phase using the relative speeds provided by the earlier representatives of this class. Different update policies for the relative speed statistics are possible. The simplest is to store the first representative's behavior only and use it to systematically fast-forward similar co-phases encountered afterwards. Another approach is to continuously update a class' statistics with the numbers obtained by simulating in detail each co-phase of this class out of a given number.

An investigation of how statistical sampling could be applied to multi-threaded simulation is proposed in [154]. This paper also presents an interesting study about how

ideal sampling, i.e., with sample values obtained on a real multi-threaded machine, combined to fast-forwarding, differs from the real observed performance, thus assessing the fast-forwarding assumption alone, and not the effects of detailed simulation or sample length. The conclusion is that the interval between two detailed simulation should not exceed 1 to 10 millions of instructions for a 2-way SMT in order to keep errors under a few percents. This paper also proposes a new functional warm-up technique (Monte-Carlo), but it performs worse than the SMARTS functional warming approach (caches and branch predictors).

These approaches, however, exhibit several drawbacks. First, they have only considered simulation of 2 cores, and, in the first, simulation of independent programs, not taking into account interactions between threads sharing memory regions. Second, albeit the number of really observed co-phases is lower than the product of the number of phases in each program, it is only so by a constant factor. Third, detailed simulation needs to be performed for each co-phase matrix entry.

The suspected exponential growing rate of co-phases with the number of simulated cores is confirmed experimentally in [191], although it is successfully mitigated up to 16 cores in the paper by allowing more samples to be clustered together. Indeed, using a threshold on their Hamming distance, samples that differ only by a small number of threads being in different phases are grouped. This technique, however, doesn't eliminate the exponential nature of the growth in the number of co-phases, which doesn't bode well for its scalability beyond the tested number of cores.

Perelman et al. [205] developed a parallel program phase analysis in which individual threads' instruction intervals are clustered into the same set of phases, instead of a separate set per thread. This approach uses the same number of overall phases (between 5 and 10) as was previously used for a single thread in experiments up to 4 cores. The paper also mentions an application of this clustering to simulation of parallel programs using simulation points. However, the precise experimental setup, and in particular how architectural state is warmed up, is not described. One simulation point is chosen in each thread to represent a given phase. All the simulation points are then fully simulated, presumably jointly with the other threads' corresponding intervals. The results are then appropriately weighted to yield an overall value for execution time (or IPC). It seems that this method requires that the architectural state be saved at each simulation point's start. Moreover, it doesn't take into account the interaction between the threads' concurrent phases when choosing the intervals to simulate. The architectural variability this scheme can support thus appears to be extremely limited.

SimFlex [256] is an extension of statistical sampling to multiprocessors for servers running throughput applications. It benefits from the fact that lots of thread interleavings are exercized under full load during a short period of real execution time (around 30 seconds). Correlation between the number of transactions per second and the number of user instructions per cycle executed is further exploited to reach practical simulation time. However, these techniques are not applicable to general multi-core simulation.

In conclusion, sampling in multi-core simulation exists but doesn't support a large number of cores. The co-phases number increases exponentially with the number of

threads. Without a major breakthrough in the field, sampling will be of no help for many-core architectures.

15.3.2 Conservative Discrete-Events Based Simulators

The Wisconsin Wind Tunnel [213] is a parallel discrete-event simulator for cache-coherent shared-memory machines, designed to run on a specific machine (CM-5, a supercomputer with SPARC processors, back in the 90s). It features a very simple model of instructions (adapted to RISC architectures) with a fixed cost of one cycle, except when a cache request misses. The program's code is instrumented and then directly executed. Instrumentation allows to count the virtual cycles elapsed and to give control back to the simulator when remote memory accesses that the local cache can't service are issued. Only a simple and unrealistic network topology is assumed, where all processors are directly connected to each other processor with a dedicated link having a fixed latency. Except queuing delay, no contention constraints or bandwith limits are enforced.

The discrete-event technique used is the conservative breathing time bucket one [239]. Each host processor simulates a target processor for the duration of a quantum, specified as a count of target cycles. This quantum is taken smaller than the chosen simulated network links' latency. Once all host processors have finished executing a quantum, they have to globally synchronize, ensuring that remote messages are effectively processed. Then, the simulation of the next quantum can begin. This provides lookahead because it is known that no messages sent during a quantum can affect other target processors before the next quantum. The CM-5 features a hardware reduction mechanism, implementing the global synchronization with low latency.

The Wisconsin Wind Tunnel II [190] increases the portability of the WWT approach on two aspects. First, it uses a mostly machine-independent tool for modifying/injecting code to instrument the simulated program. Second, messages are exchanged using SAM, a communication library using active messages when running on clusters of workstations or shared-memory on SMPs. Modelization is the same as in the original WWT, except that contention is modeled at the network interfaces.

Still, active messages are not found on commodity hardware for current popular OS[3], and the disproportionate increase of computing power compared to network latency reduction makes them more necessary than at the time the approach was proposed. What is more, results show that the approach can scale reasonably up to 8 processors (up to 5.4 speedup), but that the dominant performance factor becomes the idle time at global synchronization because of the load imbalance accross processors when they process their quantum.

BigSim [266] is a many-core optimistic discrete-event simulator (see Section 15.1). Our approach shares with it the requirement to annotate instruction blocks to compute the simulation time. BigSim uses a simpler network model that completely neglects contention.

[3]For Linux, the GAMMA project [65] provides networking with low-latency on Ethernet links for only 2 kinds of network chips. It also requires that another card is present for regular IP traffic. Significant improvements in latency may come with the advent of 10 Gb/s Ethernet.

In contrast, we do not model global contention in the network, but we do model contention on individual links.

Part of the efficiency of BigSim seems to come from the chosen class of simulated applications. Indeed, the latter includes linear programs, in which threads receive messages in fixed programmed order, MPI programs following the simple pattern of sequences of computations followed by a global barrier, and programs coded with Structured Dagger [149], a language allowing to specify some kinds of (non-)dependencies, on top of the Charm++ [150] programming environment[4]. By reducing the possible messages and interactions orderings, dependency violations are infrequent and rollbacks can be performed by simple timing adjustements.

The simulator itself is parallelized and scales well to hundreds of processors. Since our approach doesn't perform rollbacks and enforces a relaxed synchronization, the BigSim results bodes well for its scalability if parallelized. BigSim experiments have been conducted on a supercomputer, though, and it remains to be seen if such scalability can be sustained on clusters of workstations. The paper provides no hints on the simulation time needed for thousands of processors.

15.3.3 Relaxed Synchronization

The SystemC TLM 2.0 standard [198] complements the Loosely Timed modeling style (see Section 15.2.3) by introducing *temporal decoupling*, a technique that allows a component to be simulated during a fixed amount of cycles in a row, called a *quantum*, without synchronization with other components, provided that execution stays correct.

For example, a processor may continue its virtual execution as long as it operates on data located in its L1 cache. Some cache line invalidations may be delayed in this case, which slightly changes execution but doesn't affect the result of well-written programs. The aim of this technique is to reduce the number of context switches and synchronization points per executed instruction. However, as soon as the processor needs to communicate with other architecture components, global synchronization still must happen, instead of the local synchronization that we use.

SlackSim [63] is a CMP cycle-accurate simulator derived from SimpleScalar [45] and designed to run Pthread applications. It provides a range of synchronization mechanisms, from global barriers at every cycle to unbound slack where threads are simulated completely independently without any synchronization. Between those two extremes, a WWT-like quantum-based synchronization or a bounded slack scheme, where all threads are allowed to go ahead of the current global time up to a fixed amount of cycles, are selectable. The latter is similar to the Optimistic Time Windows in that the allowed time window is global, although the window in this case corresponds to the allowed imprecision. This can also be seen as an extension of the temporal decoupling practice in SystemC TLM 2.0.

Graphite [22, 182] is an abstract simulator with lax synchronization. It goes further

[4]The Charm++ language and run-time system are detailed in Sections 5.3.2 and 10.4.4.

than SlackSim because processors/cores are allowed to make progress without reference to a global time. Instead, they periodically check how much ahead they are with respect to another randomly chosen process. If their clock is ahead of more than S, a configurable slack parameter, they are put to sleep for a duration computed based on the local clock advance speed observed so far, which represents the supposedly real simulation time necessary for the other process to catch up. This synchronization technique is called *LaxP2P*.

Compared to spatial synchronization, this technique doesn't provide a fixed guarantee about time drift. Also, cores have to communicate with the reference core they randomly choose, which can be any other core, causing communication overhead in the whole network. Provided that local time is checked often against enough referees, LaxP2P is stricter than spatial synchronization, because other cores than the direct neighbors are considered.

A priori, this property may translate into significantly more accurate results. This is not the case in practice since messages between two cores necessarily pass by a chain of direct neighbors. In other words, messages are transmitted along the same topology as is used for spatial synchronization. With an appropriate tuning of the maximum allowed local time drift T, it is not necessary to check for drifts between remote cores, because they can't influence each other in less than the minimum latency for a message to reach one from the other. In practice, this means taking T equal or less than the minimal link latency between two cores. But even for greater values, the occurence of large drift between remote cores is rare enough so as not to change the virtual execution time result significantly, as is highlighted by our experiments with varying T.

Finally, Graphite is largely slower than our simulator. It needs 80 cores to produce speedups that are 10 times smaller, whereas our simulator has not been parallelized yet and takes advantage of one core only.

15.3.4 Other Approaches

A bunch of other techniques have been applied to multi-core and many-core simulation. Some of them are not specific to this particular problem, and can be applied to single-core simulation as well.

Modular simulation (see Section 15.2.2) can be leveraged to speed up simulation of CMPs on a CMP machine [204]. Components simulation is easily parallelized because most of their state is internal and opaque to other components. Communication between components is indeed mostly done thanks to explicit signals. By carefully scheduling the components parallel execution, so that ones that would access the same resources are not executed simultaneously and ones invoking common code are clustered into the same thread, it is possible to avoid lock contention (and even sometimes remove locks) and to better exploit the larger caches of CMPs.

This translates into super-linear speedups when simulating CMPs comprising more than 8 cores. For the best scheduling algorithms explored in the paper, speedups nearly reach the double of host cores when simulating 16 core CMPs (7.7 for 4 host threads).

This simulation technique is however inherently limited by the number of cores of current machines.

Interval analysis provides a mechanistic model of superscalar processors [88], i.e., an explicative model, by contrast with black-box/empirical models. On such balanced processors, the sustained instruction issue rate is equal to the dispatch width except when disruptive miss events, such as instruction cache misses, mispredicted branches or L2 misses, occur. An interval designates a steady execution flow period ended by a miss event and the subsequently incurred penalty. Simple modeling of the different miss event types is proposed and compared to actual measurements on a cycle-accurate simulator.

Thanks to this model, simulation can be performed more efficiently, at a more abstract level [102]. In this approach, there is no need to simulate each pipeline stage. Instead, only some parameters, such as dispatch width, reorder-buffer size and memory latency, are used, coupled to a functional model, to produce accurate performance numbers, with a 6% average error for single-core simulations (spikes to 16%) and a 4.6% average error (spikes to 11%) on the reported execution time for multi-core simulations up to 8 cores. The simulation speedup compared to pure cycle-level simulation is a little less than 10.

The COTson team at HP labs proposed a technique to simulate a thousand of cores with reasonable speed in [186]. Its main contribution is how a trace of events generated thanks to a full-system single-core simulator is fed as multiple streams of instructions representing the cores to a timing simulator. It leverages threading information provided by the guest OS in the first simulator to constitute these streams. Compared to our approach, simulation is slightly more detailed at the level of individual processors and their caches. However, it assumes a perfect memory subsystem, i.e., that all processors are directly connected to the same memory, that they experience the same access latency to it and that bandwidth to memory is unlimited. Moreover, scheduling arbitrations and communications between threads are tied to the way the guest OS performs thread scheduling, leading to dependencies to the host machine's number of cores, to the quantum value and to several other OS implementation details whose influence has not been evaluated. Additionally, the approach is still slower than ours by 1 to 2 orders of magnitudes for 64 to 256 cores, the similar timings obtained for 1024 cores being due to the inefficiencies of our run-time system that automatically moves data at each access. It also exhibits no architectural variability, being limited to pure shared-memory and homogeneous architectures. Finally, the presented results have not been validated against a more accurate implementation of the simulation model, though a comparison with the originally reported scalability for SPLASH-2 benchmarks [258] is performed.

Chapter 16

Conclusion And Future Work

In this Part, we presented an abstract simulation technique that allows to simulate thousands of cores with a considerable speedup (10^2 or more) over existing flexible approaches. It combines spatial synchronization, which is, to our knowledge, the first completely *distributed and local* synchronization approach, with *abstract modeling* and *direct execution* to achieve unprecedented speeds while ensuring realistic simulation. By comparing the results of SiMany with those of a cycle-level simulator up to 64 cores and by verifying that several expected behavior variations for well-known benchmarks actually occur, we showed that the main trends are successfully captured. The analysis of accuracy supports our claim that simulating the microarchitecture's innermost details of processors is less important to many-core than to single-core simulation, in an architecture exploration context.

We also demonstrated how SiMany can be used to quickly explore different kinds of architectures, such as polymorphic cores or clustered networks with shared or distributed memory, and to study the behavior of software on them. An increasing number of researchers are taking the path of heterogeneous architectures to take advantage of cheap silicon. We believe that the results we obtained for the polymorphic and clustered architectures could be improved substantially with specific scheduling policies that would take into account the latency and computing power disparity among cores.

More data should be gathered about forecasts from discrete-event simulators starting from around a hundred of cores, the approximate threshold from which cycle-accurate simulations would take so much time that they cannot practically be conducted. These data could come from other simulator implementations or from actual many-core hardware, e.g., made out of FPGAs. They would help the community check the accuracy of absolute results for a high number of cores, beyond trends.

Finally, the spatial synchronization scheme we presented seems especially suited to parallel simulation because cores can be simulated independently within their locally allowed time window. A preliminary study indicates that most often, at least from networks with 64 cores, enough cores can be simulated during a non-null time window to keep all cores of current multi-core host machines busy. Whether this property can be leveraged to improve again SiMany's efficiency remains to be investigated.

General Conclusion

Contributions

In this thesis, we have started by presenting Capsule, a general parallel programming environment designed to exploit current multi-core processors. It is an enhancement of the proposal by Palatin [201], keeping at its heart the fundamental principles of *task-based programming* and *conditional parallelization* and augmenting them with simple but powerful coarse-grain task synchronization based on the concept of *synchronization groups*. Capsule provides to programmers a small set of simple-to-grasp and expressive platform-independent primitives, effectively easing parallel programming compared to traditional paradigms and their implementations, such as OpenMP [41] or MPI [94], and even more modern ones, such as TBB [133].

We have developed an accompanying portable all-software run-time system that implements the programming interface very efficiently. It can obtain performance results with microbenchmarks and regular programs that are similar to the original version designed in hardware. It is tuned towards embedded systems and programs with abundant regular or irregular parallelism. Experiments on a small to moderate number of cores show that it can obtain linear speedups while significantly reducing the execution time variability of parallel programs compared to a simple parallelization approach. An important application of these results is to enable the efficient parallelization of soft real-time programs on embedded systems. We have performed additional experiments to quantify the influence of the run-time platform on program performance, in particular the influence of the hardware platform and that of task granularity.

In a second part, we have adapted the Capsule environment and run-time system to distributed-memory architectures, which we foresee as the likely future of many-core architectures. First, we have proposed a data structuration and manipulation model with the aim to simplify distributed programming. It allows the run-time system to transparently manage data and their location according to accesses performed by programs, in the context of dynamic task-based environments for which it is generally not possible to know in advance which core will execute a particular task. The programmer only specifies how structures are related and which portions are accessed. It does not have to manage data location explicitly, which would require a knowledge of the particular architecture(s) programs are run on, impairing portability.

Compared to some traditional approaches like distributed-shared memory, it provides a data consistency model adapted to user-defined structures. It also opens up the possibility for the run-time system to exploit links between structures for data placement. A possible application of this property is intelligent data prefetching. Compared to distributed objects, our model is a pure data model which does not provide support for any particular object-oriented paradigm. It is thus free of the associated overhead and can be used also for very small structures. Finally, compared to languages and libraries proposing a set of predefined high-level data structures (like lists, arrays, sets, etc.), it does not force programmers to use the predefined structures to benefit from parallel treatment, which may require cumbersome program transformations.

The second adaptation of Capsule we contributed concerns task management. We

have proposed a class of mechanisms to implement fast probe and divide, work spreading and global load-balancing for distributed architectures. In order to avoid any scalability barrier, all these mechanisms are distributed and local. They permanently balance the available load using the push paradigm, by contrast with work-stealing techniques, over which we showed they have several advantages. Their local control rules are diffusive and gradually lead to a global balance of available tasks over all cores. We have shown in Li et al. [173] that they provide more scalability than the original shared-memory scheme starting from 16 cores.

Finally, in a third part, we have developed a many-core simulator, SiMany, able to sustain program execution on architectures comprising thousands of cores with practical simulation time. It is reasonably realistic and can support substantial architecture variability, allowing to explore a large span of network organizations with a variable number of cores of potentially different computing power. At the time of this writing and to the best of our knowledge, SiMany seems to be the fastest such simulator in existence compared to previously published approaches and the normalized simulation time they report[1].

Speed is essentially achieved through three different techniques. The first is a novel virtual time synchronization technique, called *spatial synchronization*, which ensures that the concurrency a real many-core machine would exhibit is faithfully reproduced. It does so through a relaxed distributed and local scheme that suspends the simulation of a component whose virtual time is too much ahead of its neighbors'. Its maximum local drift parameter acts as a toggle between simulation accuracy and speed. The second technique is the use of a higher level of abstraction than in past simulators. Only simple models for caches and cores are used, following the intuition that, beyond some number of components, simulating accurately their interactions becomes more important than simulating each of them in great detail. The third technique is the architecture and implementation of the simulator. In particular, it can execute a large part of the user code natively.

We have compared the results reported by SiMany with those of a well-known cycle-level/system-level simulator, UNISIM [14]. They show that SiMany reproduces the performance trends exhibited by traditional simulation techniques at least up to 64 cores. This is the first validation of this kind for an abstract discrete-event-based simulator with a relaxed synchronization scheme. We have then exercized the simulator on homogeneous, polymorphic and clustered architectures with up to 1024 cores. The obtained results are compatible with the well-known characteristics of the benchmarks we used.

The results of the simulation of programs parallelized with Capsule on distributed architectures suggest that our approach, which tries to exploit parallelism and to provide data management at a very fine level, is viable to obtain scalable performance on many-core architectures. Performance gains are in large part portable between architectures with different characteristics and increase with a higher number of cores up to several hundreds

[1]The normalized simulation time is the time to simulate a program over a native run on the same machine. Sometimes, speedups over regular cycle-level simulation are reported instead. In this case, we have computed the simulation time assuming that cycle-level simulation is 10^6 times slower than native execution, an optimistic slowdown value. See also Chapter 11.

of cores. Additionally, the Capsule programming model makes parallel programming easier and as such can enable a wider audience of programmers to produce useful parallel code.

Prospects

On the road to achieve our goal to adapt Capsule to many-core architectures and to show its performance potential on them, we have contributed to many different fields. We also undertook a significant programming effort to connect all these pieces together. Inevitably, we had to skim through or even leave out a lot of the interesting opportunities that appeared to us during this time-constrained journey. We will now review the most exciting and promising ones among them.

The Capsule programming model, while being fairly general and expressive, would benefit from supporting simple but powerful primitives for fine-grain synchronization. Programs running on the shared-memory version of Capsule can use traditional synchronization primitives, such as locks and condition variables. However, the latter are not straightforward to use, are difficult to debug and to compose. We briefly mentioned transactional memory, its potential benefits and its current problems in Section 1.2.2. The distributed-memory version guarantees that all accesses to a dereferenced cell are performed atomically. To avoid deadlocks, proper collaboration from the programmer or the compiler is still necessary[2]. Moreover, building complex concurrent schemes may not be straightforward on top of that property.

From these observations, several research directions can be explored. One is to investigate new general schemes or improve emerging ones that provide mutual exclusion on a user-definable set of data. They should provide composable constructs that avoid the current major problems of existing fine-grain synchronization techniques. In this line, improving transactional memory and integrating it into Capsule seems an interesting challenge. Another direction is to simplify implementations by finding a set of constructs allowing to express the most important synchronization patterns in applications. They could be built on top of a limited set of simple primitives. By leveraging the semantical information they provide, it may be possible to optimize sequences of accesses, at the compiler level or in the run-time system. In fact, a proposal has recently started to combine both approaches [159].

An important goal of the Capsule approach is to shield programs from the idiosyncrasies of the architectures they run on. Programmers can (and are encouraged to) specify all tasks that can be performed in parallel, including very small ones. In doing so, they are essentially writing code as if the target architecture was an ideal one that would comprise an infinite number of cores and on which spawning new tasks would always be profitable. With this information, the run-time system adapts the program to the number of hardware cores that are actually present through conditional division.

[2] We here refer to the constraint that only a single cell can be dereferenced at once at any point of a task (see Section 8.2.1). A source code analyzing tool could easily check this property automatically.

However, as Section 4.2.3 showed, this automatic adaptation is sometimes not enough to obtain good performance for programs that create a large portion of very small tasks. Since it does not seem realistic to expect programmers to indicate task size or compilers to be able to infer it, the run-time system should dynamically detect when spawning some tasks is not worthwhile, in addition to the task throttling resulting from conditional parallelization. We have started to propose some possible mechanisms for that purpose in the same Section. As future work, their impact should be evaluated and they should be refined accordingly, with the goal to reduce dependence to task granularity as far as possible.

The data model presented in Chapter 8 has been designed to be able to handle any kind of data relations and accesses, but it is currently biased towards irregular data structures, such as lists, trees or graphs. Random access structures such as arrays or hash tables have to be specified as single cells and their whole content stored into their data section. With the current run-time system implementation, an access to a cell triggers the move of its content to the initiating core. Moving all the data of large arrays or hash tables to fulfill a single element access is obviously extremely inefficient. A possible way to deal with this problem without imposing too much burden on the programmer is to allow such structures to be split into smaller elements that are then managed with the same policies as for small cells. However, it is unclear how this support should be provided and if it can be completely transparent to the programmer. Although the latter property is desirable both to ease programming and enhance portability, some applications may need to use specific policies to run efficiently. The questions of which kind of policies should be supportable and how they could be implemented and integrated with the basic management of the current run-time system are crucial.

A study performed as part of the development of the Munin distributed-shared memory system in the 90s [25] showed that more than 95% of shared accesses conform to a small set of patterns. Another interesting conclusion was that using objects as the proper sharing granularity was not optimal in some situations[3]. The relevance of this study, and in particular the choice of supported patterns, should be assessed for recent applications and multi-core architectures. Our run-time system would very probably benefit from introducing new and more sophisticated policies to handle cell locations, such as selective data replication and invalidate and update schemes. We note that such an approach has partially been developed in the Orca distributed object system. Its aim to support medium-grain full-fledged objects leaves more leeway for sophisticated heuristics, but we are confident that most of those that were actually proposed[4] are simple enough to be adapted to CAPSULE efficiently.

The operation of the schemes for distributed task management we proposed depends on several parameters. Raising the task queue size causes more probes to succeed, which enlarges the number of tasks a program can create and reduces their average granularity. At the same time, doing so allows more tasks to be kept in reserve in queues, allowing the scheme to distribute new tasks more quickly to cores becoming idle. The task queue

[3] More information about Munin can be found in Section 10.3.2.
[4] They are detailed in Section 10.4.3.

difference threshold trades off the load-balancing rate with the bandwidth used. Studying extensively the practical influence of these parameters may have important benefits on performance. Also, some hybrid policies to preserve locality should be investigated.

There are currently only a few simulators fast enough to simulate thousands of cores in a reasonable time frame. To the best of our knowledge, prior to our work, there had been no assessment of the relevance of the results reported by a many-core simulator[5]. In our study, we could not validate the results obtained by SiMany beyond 64 cores because of too long simulation times[6]. Comparison with real hardware machines was not possible, since SiMany was precisely designed to explore possible future architectures. The emergence of many-cores implemented as FPGAs [157] may enable it in the near future, though they have different characteristics than regular silicon dies. In the meantime, comparing the results from other many-core simulators with similar features but using different techniques may provide hints about their accuracy.

Simulation with our new *spatial synchronization* scheme seems particularly well suited to parallelization, since cores can be simulated independently in their allowed time window. More of them are thus likely to be able to make progress at the same time as the simulated architecture grows. Investigating how much performance can be gained by parallelizing SiMany seems appealing, since it may be possible to reduce the simulation time by another order of magnitude or more, allowing to explore more architectures or software designs in the same time frame and/or to simulate larger programs.

Discussion

Will the many-core evolution really take place? Will it be a success? Of course, we do not pretend to answer these questions directly and definitely, but we propose some elements of reflection to feed the discussion.

To the first question, considering the roadmaps of major founders and the current industry situation that was evoked in the Introduction, the answer is likely to be positive. However, the growing trend to offload costly scientific computations to graphic chips may make many-core computing become a reality in other hardware components than the main processors, where it was expected to occur initially. Graphic chips and general-purpose processors have dissimilar structures, with the former having simpler execution units and supporting a higher number of lightweight threads to hide memory latency. The current commercial strategy of founders is not to report the actual number of cores in graphic chips, but rather this number multiplied by the number of SIMD units a core includes, thus presenting SIMD units as if they were individual cores. This feature can give the impression that graphic chips are much ahead of regular processors in the number of cores they include, but in reality the gap is narrow. Moreover, a very recent study [168] has

[5]Except in Monchiero et al. [186], which uses a simulator based on the COTson infrastructure. However, as we argued in Section 15.3.4, this simulation approach cannot be used for architecture exploration nor realistic software evaluation.

[6]And also bugs in the cycle-level simulator used that were exposed because of the high number of components to simulate.

shown that throughput computing performance is on average less than 2.5× higher for an Nvidia GTX 280 over an Intel Core i7 960, so the performance difference is even lower.

Exploiting vectorization units of graphic chips is a distinct problem than utilizing more cores. In particular, candidate program portions must be much more homogeneous to be vectorized properly. When it is possible to do so, performance is generally much better than with threads since a single control flow serves to process large sets of data at once, alleviating the need for additional communication and synchronization. SIMD execution is also a mean to reduce power consumption over multithreading. These are important reasons why Intel and AMD have been enhancing their SIMD instruction set, the latest evolution being the introduction of AVX [90].

More generally, we may be witnessing a convergence of graphic and regular chips. At least can we say that some of the technical advantages of one class are gradually introduced in the other class. Even if tight integration is reached one day, in the meantime both classes will keep their own capabilities and instruction sets. In this thesis, we did not address the problem of distinguishing execution units based on their capabilities. We assumed that they all have the same ones, i.e., that tasks can indifferently be scheduled on any of them. Fortunately, it seems possible to adapt our different proposals to discriminate various types of tasks and send them only to execution units that can run them, without altering the main principles and ideas we presented.

Even considering our contributions and those of a large number of researchers, multi-core and many-core programming remains substantially more difficult than sequential programming. In the past, low-level synchronization primitives and threading paradigms made parallelization of programs a hard task. The situation has considerably progressed and there are now several emerging techniques that ease parallel programming, such as task-based programming, work dispatching, new coarse-grain and fine-grain synchronization primitives, support for speculative execution, etc.

Although we can still foresee plenty of additional improvements in these areas, the intricacy of parallel programming now appears to be shifting rapidly towards the problem of finding ways to parallelize work that seems naturally sequential to programmers. As we experimented when parallelizing benchmarks, it can take a major algorithmic overhaul to uncover independent tasks or ones that can be pipelined. Also, the size of data sets may have to increase to observe worthwhile benefits when throwing more cores at a problem. Finally, we have not studied how commercial applications may benefit from multi-core and many-core programming for the program parts that are not computationally intensive. If customers cannot see a clear benefit from hardware (and software) upgrades, they probably won't be willing to buy new computers. Can many-cores bring benefits for interactive consumer applications, such as office work or games[7]? If they cannot, they may in the end stay confined into the scientific computing, cloud computing and server-based applications areas. Although data centers are increasingly playing a bigger role these days, it is unclear whether they could compensate for a slowdown in consumer hardware demand.

[7]As mentioned in Chapter 1, the position of one experienced game designer towards parallelization and multi-cores is rather contrasted.

Another useful approach to improve performance is to use accelerators, such as DSPs or FPGAs dedicated to particular tasks. A recent proposal [262] has shown that it is possible to synthetize small compound circuits that can accelerate a range of functions. However, it has been applied to scientific benchmarks, and it remains to be seen if it can be useful in consumer applications. In the past years, there has been a growing body of work on automatic parallelization for irregular codes, trying to identify pipeline stages in irregular loops and executing them in separate threads [44, 209]. It is unclear, however, if such techniques can scale to many-cores and cover a sufficiently large number of cases.

The success of many-cores in the end is likely to depend on the maturity of some of the techniques we mentioned and their integration, both in hardware and in software. It will also need important breakthroughs in terms of power consumption. Current projections seem to indicate that improvements in transistor technology will not compensate for the growing number of them that are forecasted to be included on a single chip. In a decade, it may not be possible to power all cores of many-core processors at the same time. Whatever comes out of the mind of scientists and engineers, the next decade is going to be captivating and may decide the fate of most players in the computing industry.

Appendices

Appendix A

Quicksort Example Code

In this Appendix, we present the integrality of the code for the `Quicksort` benchmark. This code comes from the version using the first Capsule programming model and run-time system, i.e., that runs on architectures with global address space. These are described in Part I.

The global variable `treshold` contains the threshold sub-array size under which the sub-arrays are always sorted sequentially (no probe is performed at all). It was used in particular in some experiments of Section 4.2. Other parameters of the program include the array size (global variable `array_size`), a random seed used to generate the elements of the array (global variable `rand_seed`), and the number of times a sort of the random array is performed and whether a purely sequential sort has to be performed (for comparison; global variable `times`). Please look at the `usage` function starting at line 168 for details on how to pass these parameters to the program.

The `main` function starts at line 176. To generate a random array, the program uses the pair `srand`/`rand`, the latter function producing unsigned integers. The result of a call to `rand` is capped to `ELT_MAX`, a compile time constant specifying the maximal value for array elements. Then, the loop that executes each sort variant, Capsule or sequential, is first run empty, i.e., without sorting the array. This part serves to obtain the execution time of the raw loop, in order to substract it from the overall execution time for each sort test to compute an approximation of its execution time. Then, the program proceeds with the sorts. The loop for each sort variant makes a copy of the original array (referenced by the `base_array` pointer) to a work memory area (referenced by `array`) precede the actual sort. It thus makes it possible to sort several times the same random array.

Both the sequential and the Capsule versions perform traditional sequential pivot steps. The corresponding code has thus been isolated in the `qs_pivot_step` function, starting at line 46. The version presented here is slightly faster (around 10%) than a traditional one where `left` and `right` are not changed at the end of the `do` loop. By optimizing the sequential version, we guarantee that the speedup results we report are realistic and computed over excellent sequential versions.

```
/*
 * This file is Copyright (c) 2007-2008 by the INRIA, France.
 * It is released under the GNU GPL v2.
 *
5 * Written by Olivier Certner.
 */

// C library headers
#include <stdio.h>
10 #include <string.h>
#include <stdlib.h>

// CAPSULE API and abstraction layer
#include <capsule_api.h>
15 #include <capsule_abs_itf.h>

// Default parameters for the used array
unsigned const ELT_MAX = 1000000000;

20 unsigned array_size = 100000;
int rand_seed = 27758;
unsigned times = 2;
unsigned threshold = 100;

25
// Encapsulation stuff

typedef struct
{
30   unsigned * left;
  unsigned * right;
}
  qsort_t;

35 qsort_t * alloc_qs (unsigned * left, unsigned * right)
{
  qsort_t * qs = (qsort_t *) malloc (sizeof (* qs));
  qs -> left = left;
  qs -> right = right;
40   return qs;
}


// Pivot step
45
unsigned * qs_pivot_step (unsigned * org_left, unsigned * org_right)
{
  unsigned tmp, pivot;
  unsigned * right = org_right;
50   unsigned * left = org_left;
```

```
    if (org_left >= org_right)
      return NULL;

55  // Pivot is the first element of the sub-array
    pivot = * org_left;

    // Iterative algo to put the elements less or equal
    // to the pivot on the left, the others on the right.
60  do
    {
      // Find an element greater than the pivot,
      // starting from the left.
      while ((* left <= pivot) && (left < right))
65      ++ left;

      // Find an element lower or equal to the pivot,
      // starting from the right.
      while ((* right > pivot) && (left < right))
70      -- right;

      // Exchange left and right.
      tmp = * left;
      * left = * right;
75    * right = tmp;

      // Next ones...
      ++ left;
      -- right;
80  }
    while (left < right);

    // 'right' (+ 1 because of last decrementation in the loop) may
    // not be strictly greater than the pivot.  In this case, it is
85  // the rightmost element lower or equal to the pivot, unless loop
    // exited because of last decrementations.  We thus have to be
    // careful in order to insert the pivot at the right place.
    if (right[1] <= pivot)
      ++ right;
90  else if (* right > pivot)
      -- right;

    // Place the pivot
    * org_left = * right;
95  * right = pivot;

    return right;
  }

100 //-----------------------------------------------------------------------------
```

```
    //-- Sequential version

    void qs_seq (unsigned * left, unsigned * right)
    {
105   unsigned * pivot = qs_pivot_step (left, right);

      if (pivot)
      {
        qs_seq (left, pivot - 1);
110     qs_seq (pivot + 1, right);
      }
    }


115 //------------------------------------------------------------------------
    //-- Capsule version

    void qs_capsule_wrapper (void * arg);

120 void qs_capsule (unsigned * left, unsigned * right)
    {
      // If the array has less than threshold elems,
      // sort it sequentially.
      if (right - left > threshold)
125   {
        // Find pivot
        unsigned * pivot = qs_pivot_step (left, right);

        // Have still something to sort?
130     if (pivot)
        {
          // Probe to execute a new task in parallel
          capsule_ctxt_t * ctxt;
          capsule_probe (& qs_capsule_wrapper, & ctxt);
135
          if (ctxt)
            // Sorting of the left sub-array is done in parallel
            capsule_divide (ctxt, (void *) alloc_qs (left, pivot - 1));
          else
140         // Sorting of the left sub-array is done sequentially
            qs_capsule (left, pivot - 1);

          // Sorting the right sub-array is done in this context
          //
145       // Probing again would be completely useless here (we have
          // to execute this part; so execute it directly, instead of
          // trying to give it to another core).  Keeping the other
          // cores busy will come from the probes from the recursive
          // pivot steps.
150       qs_capsule (pivot + 1, right);
```

```
      }
    }
    else
      qs_seq (left, right);
155 }

  void qs_capsule_wrapper (void * arg)
  {
    qsort_t * qs = (qsort_t *) arg;
160 qs_capsule (qs -> left, qs -> right);
    free (qs);
  }


165
  //--------------------------------------------------------------------------

  void usage (char const * prog_name)
  {
170 fprintf (stderr, "Usage:\n"
      "\t%s [elems_nb iter_nb random_seed threshold do_seq]\n"
      "with do_seq being yes or no.\n",
      prog_name);
  }
175
  int main (int argc, char ** argv)
  {
    unsigned i;
    unsigned * base_array, * array;
180 capsule_rc_t rc;
    capsule_mach_64bits_t ts_begin, ts_end, ts_elapsed;
    float fts_empty_1, fts_empty_2, fts_capsule, fts_seq;
    unsigned uns_h, uns_l;
    unsigned do_seq = 0;
185
    if ((argc != 1) && (argc != 6))
    {
      usage (argv[0]);
      return 1;
190 }

    if (argc == 6)
    {
      char * ec;
195   array_size = (unsigned) strtoul (argv[1], & ec, 0);
      if (argv[1][0] == 0 || * ec != 0)
      {
        fprintf (stderr, "ERROR: 1st arg not parsable.\n");
        return 1;
200   }
```

```
    times = (unsigned) strtoul (argv[2], & ec, 0);
    if (argv[2][0] == 0 || * ec != 0)
    {
205   fprintf (stderr, "ERROR: 2nd arg not parsable.\n");
      return 1;
    }

    rand_seed = (int) strtol (argv[3], & ec, 0);
210 if (argv[3][0] == 0 || * ec != 0)
    {
      fprintf (stderr, "ERROR: 3rd arg not parsable.\n");
      return 1;
    }
215
    threshold = (unsigned) strtoul (argv[4], & ec, 0);
    if (argv[4][0] == 0 || * ec != 0)
    {
      fprintf (stderr, "ERROR: 4th arg not parsable.\n");
220   return 1;
    }

    switch (argv[5][0])
    {
225   case 'y':
      case 'Y':
      case 'o':
      case 'O':
      do_seq = 1;
230   break;
    }
  }

  // Banner
235 printf ("CAPSULE quicksort benchmark.\n\n");
  printf ("Nb of elements: %u.\n", array_size);
  printf ("Tests are repeated %u times.\n", times);
  printf ("Random seed: %i.\n", rand_seed);
  printf ("Threshold: %u.\n", threshold);
240 if (do_seq)
    printf ("Doing sequential sorts for comparison.\n");
  else
    printf ("Not doing sequential sorts.\n");
  printf ("\n");
245
  // CAPSULE runtime initialization
  capsule_sys_init_warmup ();

  // Allocations and array preparation
250 base_array = malloc (array_size * sizeof (* base_array));
```

```
    array = malloc (array_size * sizeof (* array));
    srand (rand_seed);
    for (i = 0; i < array_size; ++ i)
      base_array[i] = rand () % ELT_MAX;
255
    // Loop without sorting
    printf ("First loop without sorting.\n");
    capsule_abs_get_time (& ts_begin);

260 for (i = 0; i < times; ++ i)
      memcpy (array, base_array, array_size * sizeof (* array));

    capsule_abs_get_time (& ts_end);
    capsule_mach_64bits_sub (& rc, & ts_end, & ts_begin, & ts_elapsed);
265
    // Print the elapsed time
    capsule_mach_64bits_get32h (& ts_elapsed, & uns_h);
    capsule_mach_64bits_get32l (& ts_elapsed, & uns_l);
    printf ("Elapsed time (32 bits high, low):  %u, %u.\n", uns_h, uns_l);
270 capsule_mach_64bits_to_float (& ts_elapsed, & fts_empty_1);

    // Testing the CAPSULE regular version
    printf ("\nCAPSULE version.\n");

275 capsule_abs_get_time (& ts_begin);

    for (i = 0; i < times; ++ i)
    {
      memcpy (array, base_array, array_size * sizeof (* array));
280   qs_capsule (array, array + array_size - 1);
      capsule_group_wait ();
    }

    capsule_abs_get_time (& ts_end);
285 capsule_mach_64bits_sub (& rc, & ts_end, & ts_begin, & ts_elapsed);

    // Print the elapsed time
    capsule_mach_64bits_get32h (& ts_elapsed, & uns_h);
    capsule_mach_64bits_get32l (& ts_elapsed, & uns_l);
290 printf ("Elapsed time (32 bits high, low):  %u, %u.\n", uns_h, uns_l);
    capsule_mach_64bits_to_float (& ts_elapsed, & fts_capsule);

    #ifndef CAPSULE_NO_STATS
    // Dump statistics
295 printf ("\nCAPSULE statistics:\n");
    capsule_sys_dump_all_stats (stdout);
    printf ("\n");
    #endif

300 if (do_seq)
```

```
    {
      // Testing the sequential version
      printf ("\nSequential version.\n");

305   capsule_abs_get_time (& ts_begin);

      for (i = 0; i < times; ++ i)
      {
        memcpy (array, base_array, array_size * sizeof (* array));
310     qs_seq (array, array + array_size - 1);
      }

      capsule_abs_get_time (& ts_end);
      capsule_mach_64bits_sub (& rc, & ts_end, & ts_begin, & ts_elapsed);
315
      // Print the elapsed time
      capsule_mach_64bits_get32h (& ts_elapsed, & uns_h);
      capsule_mach_64bits_get32l (& ts_elapsed, & uns_l);
      printf ("Elapsed time (32 bits high, low):  %u, %u.\n", uns_h, uns_l);
320   capsule_mach_64bits_to_float (& ts_elapsed, & fts_seq);
    }

    // Loop without sorting
    printf ("\nSecond loop without sorting.\n");
325 capsule_abs_get_time (& ts_begin);

    for (i = 0; i < times; ++ i)
      memcpy (array, base_array, array_size * sizeof (* array));

330 capsule_abs_get_time (& ts_end);
    capsule_mach_64bits_sub (& rc, & ts_end, & ts_begin, & ts_elapsed);

    // Print the elapsed time
    capsule_mach_64bits_get32h (& ts_elapsed, & uns_h);
335 capsule_mach_64bits_get32l (& ts_elapsed, & uns_l);
    printf ("Elapsed time (32 bits high, low):  %u, %u.\n", uns_h, uns_l);
    capsule_mach_64bits_to_float (& ts_elapsed, & fts_empty_2);

    // Final stats (for perf measurements)
340 float avg_capsule =
      (fts_capsule - (fts_empty_1 + fts_empty_2) / 2) / times;
    printf ("\nAverage execution time, one sort, CAPSULE version:  \t%f.\n",
      avg_capsule);
    if (do_seq)
345 {
      float avg_seq = (fts_seq - (fts_empty_1 + fts_empty_2) / 2) / times;
      printf ("Average execution time, one sort, sequential version:  \t%f.\n",
        avg_seq);
      printf ("Speedup:  \t%f.\n", avg_seq / avg_capsule);
350 }
```

```
    // Free the arrays
    free (array);
    free (base_array);
355
    // Terminate the CAPSULE run-time system
    capsule_sys_destroy ();

    return 0;
360 }
```

Appendix B

Dijkstra Example Code

In this Appendix, we present the integrality of the code for the Dijkstra benchmark[1], in its version that uses the Capsule programming model and run-time system for distributed architectures, which were described in Part II.

In comparison with the code of Quicksort presented in Appendix A, the code for Dijkstra makes use of the additional primitives to handle data structures, which were described in Section 8.1.2, but uses also the functions introduced by SiMany for virtual time accounting. capsule_vtime_advance serves to increment the virtual time counter on the core executing the current thread. capsule_vtime_cur_get gives the value of this counter, for evaluation purposes.

The graph modelization is done in the graph_capsule C module, while the algorithm itself and the main function are in dijkstra_capsule.c.

Since no cache model is directly implemented in the current SiMany version, timing annotations for cache latencies must be introduced by hand, as explained in Section 13.2.1. In dijkstra_capsule.c, they are specified as C expressions using preprocessor constants. The constants PRED_90P_VT, PRED_FAST_VT, PRED_SLOW_VT are used to model branch prediction. The first serves to account for a test for which the prediction outcome is a priori unknown. In this case, a 90% success is assumed. The others account for successful and wrong predictions respectively. L1_VT and L2_VT account for accesses to the L1 and L2 caches. FUNC_VT and ADD_VT account for a function call and an addition on integers. Using preprocessor constants allows to vary the actual values by simple recompilation.

Annotations in graph_capsule do not use preprocessor annotations but rather concrete values. These values have only a small impact on simulation results, since this module is executed only once at benchmark startup to build the graph and once at end of execution to destroy it. They have been minimized in order to avoid including the time of reading a file and building a graph when assessing the performance of our parallel algorithm.

As explained in Section 8.1.2, C was chosen to demonstrate that an implementation of our concepts was feasible. Primitives are provided through library functions. Other

[1] As that of Quicksort, it is released under the GPL v2 license.

choices are possible and even recommended, such as object-oriented languages, which will reduce the verbosity and the low ratio of algorithmic operations over interactions with the run-time system in the following listings.

File graph_capsule.h

```
#ifndef GRAPH_CAPSULE_H
#define GRAPH_CAPSULE_H

/**
 * The graph data structure is as follows:  There are two types of cells,
 * node-cells and edge-cells.  As evident from their names, they are
 * encapsulations of the nodes and edges of a graph inside generic cells.
 * These cells are connected to each other by links.  These two cells are
 * described in greater depth below:
 *
 * 1) node-cell:
 * unsigned name
 * unsigned distance
 * link to the linked-list of edges,
 * link to the parent of the node, and
 * 2) edge-cell:
 * unsigned length
 * link to the node-cell to which it is connected
 * link to the next edge-cell (as it is a part of the linked-list of edges
 * in a node)
 */

// CAPSULE
#include <capsule_api.h>
#include <capsule_ds_api.h>
#include <capsule_vtime_api.h>

// For FILE
#include <stdio.h>


typedef struct node_s
{
  unsigned name; // Used for pretty print
  unsigned distance;
} node_t;

typedef struct edge_s
{
  unsigned length;
} edge_t;
```

```
   capsule_link_t **
   graph_build
45 (FILE * graph_file,
    unsigned * graph_size_ptr);

   void
   graph_destroy
50 (capsule_link_t ** graph,
    unsigned nodes_nb);

   void
   graph_print_distances
55 (capsule_link_t ** graph,
    unsigned start,
    unsigned nodes_nb);

   #endif
```

File graph_capsule.c

```
   #include "graph_capsule.h"

   // For printf
   #include <stdio.h>
 5
   // For malloc
   #include <stdlib.h>

   // Ints of fixed size
10 #include <stdint.h>

   // Assert
   #include <assert.h>
   #undef NDEBUG
15
   #define INFINITY UINT32_MAX

   static void
   set_edge_content
20 (capsule_hdl_t * hdl,
    unsigned length)
   {
     void * void_ptr;
     capsule_rc_t rc = capsule_cell_give_data_access
25     (hdl, 0, sizeof(edge_t), & void_ptr);
     assert (! rc);

     edge_t * edge_ptr = void_ptr;
     edge_ptr -> length = length;
```

```
30
    capsule_vtime_advance (3);

    rc = capsule_cell_revoke_data_access (hdl, 0, sizeof(edge_t));
    assert (! rc);
35 }

  static void
  set_node_content
  (capsule_hdl_t * hdl,
40 unsigned name,
   unsigned distance)
  {
    void * void_ptr;
    capsule_rc_t rc = capsule_cell_give_data_access
45    (hdl, 0, sizeof(node_t), & void_ptr);
    assert (! rc);

    node_t * node_ptr = void_ptr;
    node_ptr -> name = name;
50  node_ptr -> distance = distance;

    capsule_vtime_advance (4);

    rc = capsule_cell_revoke_data_access (hdl, 0, sizeof(node_t));
55  assert (! rc);
  }

  capsule_link_t **
  graph_build
60 (FILE * graph_file,
   unsigned * graph_size_ptr)
  {
    // Return value for DS functions
    capsule_rc_t rc;
65
    // Return value for file functions
    size_t file_rc;

    // Links to the graph nodes (to be returned)
70  capsule_link_t ** graph_link;

    // Get the number of nodes in the data file
    uint32_t nodes_nb_32;

75  file_rc = fread (& nodes_nb_32, sizeof(nodes_nb_32), 1, graph_file);
    assert (file_rc == 1);

    capsule_vtime_advance (100);
```

```
80    unsigned nodes_nb = (unsigned) nodes_nb_32;
      * graph_size_ptr = nodes_nb;

      // Alloc the array of links to the nodes in the graph
      graph_link = (capsule_link_t **)
85      malloc (nodes_nb * sizeof (capsule_link_t *));

      capsule_vtime_advance (10);

      /*
90     * Each node-cell has 2 links:
       * one to the linked-list of edges,
       * one to the parent of the node, and
       *
       * Here, we construct all cells at once.
95     */
      unsigned i, j;
      for (i = 0; i < nodes_nb; ++ i)
      {
        capsule_hdl_t * tmp_hdl;
100
        rc = capsule_cell_create_cell (sizeof(node_t), 2, & tmp_hdl);
        assert (! rc);

        rc = capsule_link_create_from_hdl (tmp_hdl, & graph_link[i]);
105     assert (! rc);

        // Set the name and distance of the node
        set_node_content (tmp_hdl, i, INFINITY);

110     capsule_hdl_release (tmp_hdl);
      }

      // Memory area to contain list of arcs from a given node
      // Format is (see 'gengraph.py'):
115   // - Number of arcs from a node.
      // - A list of couples (dest_node, weight), each one being such an
      // arc.

      uint32_t * per_node_edges = malloc (nodes_nb * sizeof(uint32_t) * 2);
120
      capsule_vtime_advance (10);

      // Loop on each node, to construct its edge linked-list
      for (i = 0; i < nodes_nb; ++ i)
125   {
        capsule_hdl_t * node_hdl;

        rc = capsule_link_deref (graph_link[i], & node_hdl);
        assert (! rc);
```

```
130
        // Read in the edges data
        uint32_t edges_nb;
        file_rc = fread (& edges_nb, sizeof(edges_nb), 1, graph_file);
        assert (file_rc == 1);
135
        file_rc = fread (per_node_edges, sizeof(* per_node_edges),
          edges_nb * 2, graph_file);
        assert (file_rc == edges_nb * 2);

140     capsule_vtime_advance (20);

        for (j = 0; j < edges_nb; ++ j)
        {
          uint32_t edge_weight = per_node_edges[j * 2 + 1];
145
          capsule_hdl_t * edge_hdl;
          rc = capsule_cell_create_cell
            (sizeof(edge_t), 2, & edge_hdl);
          assert (! rc);
150
          // Set the length of the edge
          set_edge_content (edge_hdl, edge_weight);

          // Set the link from the edge-cell to the destination
155       // node of the graph arc.
          uint32_t dest_node_idx = per_node_edges[j * 2];
          rc = capsule_cell_set_link_from_link
            (edge_hdl, 0, graph_link[dest_node_idx]);
          assert (! rc);
160
          // Retrieve the current head of the edge list
          capsule_link_t * head;
          rc = capsule_cell_get_link (node_hdl, 0, & head);
          assert (! rc);
165
          // Link the new edge to the old head
          rc = capsule_cell_set_link_from_link (edge_hdl, 1, head);
          assert (! rc);

170       capsule_link_release (head);

          // Make the new edge the new head of the edge list
          rc = capsule_cell_set_link_from_hdl (node_hdl, 0, edge_hdl);
          assert (! rc);
175
          // Release the handle on this edge-cell
          capsule_hdl_release (edge_hdl);
        } // Loop on all possible edges from one node
```

```
180      capsule_hdl_release (node_hdl);
       } // Loop on all nodes

       // Free buffer
       free (per_node_edges);
185
       capsule_vtime_advance (10);

       // Return the link to the graph
       return graph_link;
190  }

     void
     graph_destroy
     (capsule_link_t ** graph,
195   unsigned nodes_nb)
     {
       unsigned i;

       // Free the array of links to the graph's node-cells
200    for (i = 0; i < nodes_nb; ++ i)
         capsule_link_release (graph[i]);

       free (graph);

205    capsule_vtime_advance (10);
     }

     void graph_print_distances
     (capsule_link_t ** graph,
210   unsigned start,
      unsigned nodes_nb)
     {
       // Used to store return codes
       capsule_rc_t rc;
215
       // Banner
       printf ("Shortest paths starting from nb '%u':\n", start);
       capsule_vtime_advance (20);

220    // Loop on all nodes
       unsigned i;
       for (i = 0; i < nodes_nb; ++ i)
       {
         if (i == start)
225        continue;

         capsule_link_t * cur_node = graph[i];
         capsule_hdl_t * cur_node_hdl;
```

```
230    rc = capsule_link_deref (cur_node, & cur_node_hdl);
       assert (! rc);

       void * void_ptr;
       rc = capsule_cell_give_data_access
235      (cur_node_hdl, 0, sizeof (node_t), & void_ptr);
       assert (! rc);

       node_t * cur_node_content = void_ptr;
       unsigned dist = cur_node_content -> distance;
240    unsigned name = cur_node_content -> name;

       rc = capsule_cell_revoke_data_access (cur_node_hdl, 0, sizeof (node_t));
       assert (! rc);

245    if (dist != INFINITY)
       {
         printf ("Node '%u' is at distance: %u.\n", name, dist);
         printf ("Reverse path:");
       }
250    else
         printf ("Node '%u' is unreachable.\n", name);

       capsule_vtime_advance (20);

255    // First 'from' node
       capsule_link_t * from = NULL;
       rc = capsule_cell_get_link (cur_node_hdl, 1, & from);
       assert (! rc);

260    capsule_hdl_release (cur_node_hdl);

       capsule_hdl_t * cur_from_hdl;

       // Walk the reverse path to the start node
265    while ((rc = capsule_link_deref (from, & cur_from_hdl)) ==
         CAPSULE_S_OK)
       {
         void * void_ptr;
         rc = capsule_cell_give_data_access
270        (cur_from_hdl, 0, sizeof (node_t), & void_ptr);
         assert (! rc);

         node_t * cur_from_content = void_ptr;
         printf (" %u", cur_from_content -> name);
275
         capsule_vtime_advance (10);

         rc = capsule_cell_revoke_data_access
           (cur_from_hdl, 0, sizeof (node_t));
```

```
280        assert (! rc);

           capsule_link_release (from);

           rc = capsule_cell_get_link (cur_from_hdl, 1, & from);
285        assert (! rc);

           capsule_hdl_release (cur_from_hdl);
         }
         assert (CAPSULE_RC_S (rc));
290
         if (dist != INFINITY)
         {
           printf (".\n");
           capsule_vtime_advance (5);
295      }
       }
     }
```

File dijkstra_capsule.c

```
     #include <stdlib.h>
     #include <stdio.h>
     #include <sys/time.h>

   5 // Also includes the CAPSULE APIs
     #include "graph_capsule.h"

     #include <assert.h>

  10 #undef NDEBUG

     //#define DEBUG 1

     // In CAPSULE, a cell has to be built to pass information
  15 // into recursive calls.  The cell will contain as raw data payload
     // the struct arg_t below, containing the distance to the next node
     // to process.  The cell will also hold 2 links, one to the next node
     // and the other to the node we are coming from.
     typedef
  20 struct
     {
       unsigned distance;
     } arg_t;

  25 // Forward declaration since dijkstra and dijkstra_encap are mutually
     // recursive.
     void
```

```
   dijkstra
   (capsule_link_t * node,
30  unsigned distance,
    capsule_link_t * from);

   void
   dijkstra_encap
35 (capsule_link_t * link)
   {
     capsule_hdl_t * hdl;
     capsule_rc_t rc;

40   rc = capsule_link_deref (link, & hdl);
     assert (! rc);

     arg_t * arg;
     void * arg_v_ptr;
45   rc = capsule_cell_give_data_access
       (hdl, 0, sizeof (arg_t), & arg_v_ptr);
     assert (! rc);
     arg = arg_v_ptr;

50   int distance = arg -> distance;
     capsule_vtime_advance (L2_VT);

     rc = capsule_cell_revoke_data_access (hdl, 0, sizeof (arg_t));
     assert (! rc);
55
     capsule_link_t * node, * from;
     rc = capsule_cell_get_link (hdl, 0, & node);
     assert (! rc);
     rc = capsule_cell_get_link (hdl, 1, & from);
60   assert (! rc);
     capsule_hdl_release (hdl);

     dijkstra (node, distance, from);

65   capsule_link_release (node);
     capsule_link_release (from);
   }

   void
70 dijkstra
   (capsule_link_t * node,
    unsigned distance,
    capsule_link_t * from)
   {
75   // Func call overhead
     capsule_vtime_advance (FUNC_VT);
```

```
    capsule_rc_t rc;
    capsule_hdl_t * node_hdl;
80
    rc = capsule_link_deref (node, & node_hdl);
    assert (CAPSULE_RC_S (rc));

    // Is the link pointing to something?
85  if (rc == CAPSULE_S_OK)
    {
      // Test (below) overhead
      capsule_vtime_advance (PRED_FAST_VT);

90    node_t * node_content;
      void * node_v_ptr;
      rc = capsule_cell_give_data_access
        (node_hdl, 0, sizeof (node_t), & node_v_ptr);
      assert (! rc);
95    node_content = node_v_ptr;

      // Test and access
      capsule_vtime_advance (PRED_90P_VT + L2_VT);

100   if (distance >= node_content -> distance)
      {
        rc = capsule_cell_revoke_data_access (node_hdl, 0, sizeof (node_t));
        assert (! rc);

105     capsule_hdl_release (node_hdl);

        return;
      }

110   // Update the distance
      node_content -> distance = distance;
      capsule_vtime_advance (L1_VT);

      rc = capsule_cell_revoke_data_access (node_hdl, 0, sizeof (node_t));
115   assert (! rc);

      // Update the node where we come from
      rc = capsule_cell_set_link_from_link (node_hdl, 1, from);
      assert (! rc);
120
      // Get the linked list of edges
      capsule_link_t * edge;
      rc = capsule_cell_get_link (node_hdl, 0, & edge);
      assert (! rc);
125
      // Done with the origin node
      capsule_hdl_release (node_hdl);
```

```
    // Loop on all edges from the node being processed (pointed to
130 // by 'node').
    capsule_hdl_t * edge_hdl;
    while ((rc = capsule_link_deref (edge, & edge_hdl)) == CAPSULE_S_OK)
    {
      // High proba branch
135   capsule_vtime_advance (PRED_FAST_VT);

      capsule_link_t * next_edge;
      rc = capsule_cell_get_link (edge_hdl, 1, & next_edge);
      assert (! rc);
140
      capsule_link_t * pointed_node;
      rc = capsule_cell_get_link (edge_hdl, 0, & pointed_node);

      edge_t * edge_content;
145   void * edge_v_ptr;
      rc = capsule_cell_give_data_access
        (edge_hdl, 0, sizeof (edge_t), & edge_v_ptr);
      assert (! rc);
      edge_content = edge_v_ptr;
150
      unsigned new_dist = distance + edge_content -> length;
      capsule_vtime_advance (L2_VT + ADD_VT);

      rc = capsule_cell_revoke_data_access (edge_hdl, 0, sizeof (edge_t));
155   assert (! rc);

      capsule_hdl_release (edge_hdl);

      // Now, probe to see if can use another resource to
160   // browse the graph from the pointed node.
      capsule_ctxt_t * ctxt;
      capsule_probe (& dijkstra_encap, & ctxt);

      if (ctxt)
165   {
        // Probe success
        capsule_vtime_advance (PRED_SLOW_VT);

        // Probe succeeded!  Prepare the arg cell.
170     capsule_hdl_t * arg_hdl;
        rc = capsule_cell_create_cell (sizeof (arg_t), 2, & arg_hdl);
        assert (! rc);

        arg_t * arg_content;
175     void * arg_v_ptr;
        rc = capsule_cell_give_data_access
          (arg_hdl, 0, sizeof (arg_t), & arg_v_ptr);
```

```
          assert (! rc);
          arg_content = arg_v_ptr;
180
          arg_content -> distance = new_dist;
          capsule_vtime_advance (L1_VT);

          rc = capsule_cell_revoke_data_access (arg_hdl, 0, sizeof (arg_t));
185       assert (! rc);

          rc = capsule_cell_set_link_from_link (arg_hdl, 0, pointed_node);
          assert (! rc);

190       capsule_link_release (pointed_node);

          rc = capsule_cell_set_link_from_link (arg_hdl, 1, node);
          assert (! rc);

195       capsule_link_t * arg;
          rc = capsule_link_create_from_hdl (arg_hdl, & arg);
          assert (! rc);

          capsule_hdl_release (arg_hdl);
200
          capsule_divide (ctxt, arg);

          capsule_link_release (arg);
        }
205     else
        {
          capsule_vtime_advance (PRED_FAST_VT);

          dijkstra (pointed_node, new_dist, node);
210       capsule_link_release (pointed_node);
        }

        capsule_link_release (edge);

215     edge = next_edge;
      } // Loop on all edges for the current node
      assert (CAPSULE_RC_S (rc));
      // Out of loop penalty
      capsule_vtime_advance (PRED_SLOW_VT);
220
      capsule_link_release (edge);
    } // Is link pointing to something?
    else
      // Test failed
225   capsule_vtime_advance (PRED_SLOW_VT);
  }
```

```
    // Array of pointers to functions that will divide (used for RPC)
    capsule_ctxt_func_ptr_t
230 func_array[] =
    {
      dijkstra_encap
    };

235 int
    main
    (int argc,
     char * argv[])
    {
240   capsule_link_t ** graph;
      FILE * graph_file;
      unsigned root_node;
      unsigned graph_size;

245   // To calculate the execution time
      struct timeval vt1,vt2;
      struct timezone zt1,zt2;

      if ((argc != 2) && (argc != 3))
250   {
        fprintf (stderr, "Usage: %s <graph file> [<root node> (default 0)]\n",
          argv[0]);
        exit (1);
      }
255   if (argc == 3)
        root_node = (unsigned) atoi (argv[2]);
      else
        root_node = 0;

260   capsule_sys_init (func_array, 1);

      // Build graph
      graph_file = fopen (argv[1], "r");
      if (graph_file == NULL)
265   {
        fprintf (stderr, "Could not open file %s\n", argv[1]);
        exit (2);
      }

270   printf ("Building graph from %s...\n", argv[1]);
      graph = graph_build (graph_file, &graph_size);

      if (root_node >= graph_size)
      {
275     fprintf (stderr, "error: root node %d does not exist in graph\n",
          root_node);
        exit (3);
```

```
  }

280  fclose (graph_file);

  printf ("Graph built.\n");

#if DEBUG
285  printf ("Printing graph...\n");
  graph_print_distances (graph, root_node, graph_size);
  printf ("Graph printed.\n");
#endif

290  // Start vtime
  capsule_mach_64bits_t vtime_1;
  capsule_vtime_cur_get (& vtime_1);

  // Start timer
295  gettimeofday (& vt1, & zt1);

  // Launch the computation
  dijkstra (graph[root_node], 0, NULL);

300  // Synchronize
  capsule_group_join ();

  // Stop timer
  gettimeofday (& vt2, & zt2);
305  // Calculate time elapsed
  printf ("Real time: %lf s.\n",
    (double) (vt2.tv_sec - vt1.tv_sec) +
    (double) (vt2.tv_usec - vt1.tv_usec) / 1e6);

310  // End vtime
  capsule_mach_64bits_t vtime_2;
  capsule_rc_t rc;
  rc = capsule_vtime_cur_get (& vtime_2);

315  if (rc == CAPSULE_S_OK)
  {
    // Virtual time is compiled in

    // Compute vtime elapsed
320    capsule_mach_64bits_sub (& rc, & vtime_2, & vtime_1, & vtime_2);
    unsigned vt_low, vt_high;
    capsule_mach_64bits_get32l (& vtime_2, & vt_low);
    capsule_mach_64bits_get32h (& vtime_2, & vt_high);
    printf ("Virtual time elapsed: %u, %u.\n", vt_high, vt_low);
325  }

#if DEBUG
```

```
  printf ("Printing graph...\n");
  graph_print_distances (graph, root_node, graph_size);
330  printf ("Graph printed.\n");

  // Don't destroy the graph to speed simulation time
  graph_destroy (graph, graph_size);
#endif
335
  capsule_sys_destroy ();
  return 0;
}
```

Selected Personal Bibliography

- "A Practical Approach for Reconciling High and Predictable Performance in Non-Regular Parallel Programs", published at DATE'08 (see Certner et al. [54])
- "Scalable Hardware Support for Conditional Parallelization", published at PACT'10 (see Li et al. [173])
- "A Very Fast Simulator For Exploring the Many-Core Future", published at IPDPS'11 (see Certner et al. [55])

Bibliography

[1] A localized dynamic load balancing strategy for highly parallel systems. In *3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 380–383, October . ISBN 0-8186-2053-6. 106

[2] Performance guidelines for AMD Athlon 64 and AMD Opteron ccNUMA multiprocessor systems, June 2006. Rev. 3.0. 135

[3] Sarita V. Adve and Mark D. Hill. Weak ordering - A new definition and some implications. Technical Report CS-TR-1989-902, University of Wisconsin, Madison, December 1989. 123, 124, 125, 126

[4] Sarita V. Adve and Mark D. Hill. Implementing sequential consistency in cache-based systems. In Benjamin W. Wah, editor, *ICPP (1)*, pages 47–50. Pennsylvania State University Press, 1990. ISBN 0-271-00728-1. 125

[5] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 2–14, New York, NY, USA, 1990. ACM. ISBN 0-89791-366-3. doi: http://doi.acm.org/10.1145/325164.325100. 126, 128

[6] K. Agrawal, C.E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2010*, pages 1–12, 19-23 2010. doi: 10.1109/IPDPS.2010.5470403. 67

[7] Mustaque Ahamad, James E. Burns, Phillip W. Hutto, and Gil Neiger. Causal memory. In Sam Toueg, Paul G. Spirakis, and Lefteris M. Kirousis, editors, *WDAG*, volume 579 of *Lecture Notes in Computer Science*, pages 9–30. Springer, 1991. ISBN 3-540-55236-7. 128

[8] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 251–260, New York, NY, USA, 1993. ACM. ISBN 0-89791-599-2. doi: http://doi.acm.org/10.1145/165231.165264. 128

[9] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.485843. 142

[10] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, and Nathan Thomas Nancy M. Amato. Stapl: A standard template adaptive parallel c++ library. In *In Int. Wkshp on Adv. Compiler Technology for High Perf. and Embedded Processors*, page 10, 2001. 64

[11] Robert W. Brodersen Anantha P. Chandrakasan, Samuel Sheng. Low power CMOS digital design. *Journal of Solid-State Circuits*, 27(4):473–484, April 1992. 3

[12] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html. 46, 191

[13] ATI. Ati radeon hd5870. URL: http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-overview.aspx. 4, 77

[14] David August, Jonathan Chang, Sylvain Girbal, Daniel Gracia-Perez, Gilles Mouchard, David A. Penry, Olivier Temam, and Neil Vachharajani. UNISIM: An open simulation environment and library for complex architecture design and collaborative development. *IEEE Comput. Archit. Lett.*, 6(2):45–48, 2007. ISSN 1556-6056. doi: http://dx.doi.org/10.1109/L-CA.2007.12. 161, 190, 209, 224

[15] Rajive L. Bagrodia. Perils and pitfalls of parallel discrete-event simulation. In *WSC '96: Proceedings of the 28th Winter simulation conference*, pages 136–143, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-7803-3383-7. doi: http://doi.acm.org/10.1145/256562.256592. 208

[16] Henri E. Bal. Orca: a language for distributed programming. *SIGPLAN Not.*, 25 (5):17–24, 1990. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/382080.382082. 150

[17] Henri E. Bal and M. Frans Kaashoek. Object distribution in orca using compile-time and run-time techniques. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 162–177, New York, NY, USA, 1993. ACM. ISBN 0-89791-587-9. doi: http://doi.acm.org/10.1145/165854.165884. 152

[18] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng.*, 18(3): 190–205, 1992. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.126768. 150, 153

[19] Henri E. Bal, M. Frans Kaashoek, Andrew S. Tanenbaum, and Jack Jansen. Replication techniques for speeding up parallel applications on distributed systems. *Concurrency: Pract. Exper.*, 4(5):337–355, 1992. ISSN 1040-3108. doi: http://dx.doi.org/10.1002/cpe.4330040502. 150, 151, 152

[20] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Rühl, and M. Frans Kaashoek. Orca: a portable user-level shared object system. Technical Report IR-408, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, July 1996. Technical Report IR-408, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam. 153

[21] BBN Laboratories. Butterfly parallel processor overview. Technical Report 6148, BBN Laboratories, Cambridge, MA, March 1986. 136

[22] Nathan Beckmann, Jonathan Eastep, III Charles Gruenwald, George Kurian, Harshad Kasture, Jason E. Miller, Christopher Celio, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores, November 09 2009. URL http://hdl.handle.net/1721.1/49809. 215

[23] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. 175, 211

[24] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–176, New York, NY, USA, 1990. ACM. ISBN 0-89791-350-7. doi: http://doi.acm.org/10.1145/99163.99182. 139

[25] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 125–134, New York, NY, USA, 1990. ACM. ISBN 0-89791-366-3. doi: http://doi.acm.org/10.1145/325164.325124. 139, 226

[26] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, Pittsburgh, PA (USA), September 1991. 129, 130, 144

[27] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Presto: a system for object-oriented parallel programming. *Softw. Pract. Exper.*, 18(8):713–732, 1988. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.4380180802. 149

[28] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The midway distributed shared memory system. In *Proceedings of Compcon '93*, pages 528–537, San Francisco, February 1993. IEEE. 144

[29] Raoul Bhoedjang, Tim Ruhl, Rutger Hofman, Koen Langendoen, Henri Bal, and Frans Kaashoek. Panda: a portable platform to support parallel programming languages. In *Sedms'93: USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems*, pages 11–11, Berkeley, CA, USA, 1993. USENIX Association. 153

[30] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, New York, NY, USA, October 2008. ACM. ISBN 978-1-60558-282-5. doi: http://doi.acm.org/10.1145/1454115.1454128. 46

[31] Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *ISPASS*, pages 45–56, 2004. 160, 212

[32] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 78–86, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. doi: http://doi.acm.org/10.1145/28697.28706. 148

[33] Barney Blaise. Posix threads programming. https://computing.llnl.gov/tutorials/pthreads. 16

[34] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995. URL `citeseer.ist.psu.edu/blumofe95cilk.html`. 43, 66, 160, 168

[35] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 362–371, New York, NY, USA, 1993. ACM. ISBN 0-89791-591-7. doi: http://doi.acm.org/10.1145/167088.167196. 67

[36] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *SFCS '94: Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Washington, DC, USA, 1994. IEEE Computer Society. ISBN 0-8186-6580-7. doi: http://dx.doi.org/10.1109/SFCS.1994.365680. 60, 66

[37] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/324133.324234. 60, 66

[38] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 132–141, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7255-2. 120

[39] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 297–308, New York, NY, USA, 1996. ACM. ISBN 0-89791-809-6. doi: http://doi.acm.org/10.1145/237502.237574. 119

[40] OpenMP Architecture Review Board. Openmp fortran application programming interface, version 1.0, October 1997. 22

[41] OpenMP Architecture Review Board. Openmp application programming interface, version 3.0, May 2008. 22, 223

[42] Hans-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *Proc. HotPar '09 (1st USENIX Workshop on Hot Topics in Parallelism)*, Berkeley, CA, USA, March 2009. Usenix Association. HP Labs. 12

[43] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix market: A web resource for test matrix collections. In *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137. Chapman & Hall, 1997. 192

[44] Matthew Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. Revisiting the sequential programming model for multi-core. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3047-8. doi: http://dx.doi.org/10.1109/MICRO.2007.35. 4, 229

[45] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/268806.268810. 159, 208, 215

[46] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194, New York, NY, USA, 1981. ACM. ISBN 0-89791-060-5. doi: http://doi.acm.org/10.1145/800223.806778. 106

[47] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. Stapl: standard template adaptive parallel library. In *SYSTOR '10: Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, pages 1–10, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-908-4. doi: http://doi.acm.org/10.1145/1815695.1815713. 65

[48] William W. Carlson and Jesse M. Draper. Distributed data access in ac. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 39–47, New York, NY, USA, 1995. ACM. ISBN 0-89791-701-6. doi: http://doi.acm.org/10.1145/209936.209942. 118

[49] John B. Carter. Design of the munin distributed shared memory system. *J. Parallel Distrib. Comput.*, 29(2):219–227, 1995. 139

[50] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 152–164, New York, NY, USA, 1991. ACM. ISBN 0-89791-447-3. doi: http://doi.acm.org/10.1145/121132.121159. 139

[51] John Bruce Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency*. Ph.D. thesis, Department of Computer Science, Rice University, Houston, TX, USA, September 1993. 139, 141

[52] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008. ISSN 1542-7730. doi: http://doi.acm.org/10.1145/1454456.1454466. 12

[53] John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael O'Boyle, Grigori Fursin, and Olivier Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2006)*, pages 24–34, Seoul, Korea, October 2006. ACM Press. 49

[54] Olivier Certner, Zheng Li, Pierre Palatin, Olivier Temam, Frederic Arzel, and Nathalie Drach. A practical approach for reconciling high and predictable performance in non-regular parallel programs. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, pages 740–745, New York, NY, USA, 2008. ACM. ISBN 978-3-9810801-3-1. doi: http://doi.acm.org/10.1145/1403375.1403555. 29, 41, 261

[55] Olivier Certner, Zheng Li, Arun Raman, and Olivier Temam. A very fast simulator for exploring the many-core future. In *Proceedings of the twenty-fifth International Parallel and Distributed Processing Symposium (IPDPS)*, May 2011. To appear. 161, 261

[56] Luis Ceze, Karin Strauss, George Almasi, Patrick J. Bohrer, Jose R. Brunheroto, Calin Cascaval, Jose G. Castanos, Derek Lieber, Xavier Martorell, José E. Moreira, Alda Sanomiya, and Eugen Schenfeld. Full circle: Simulating linux clusters on linux clusters. In *In Proceedings of the Fourth LCI International Conference on Linux Clusters: The HPC Revolution 2003*, 2003. 160

[57] Rohit Chandra. *The COOL parallel programming language: design, implementation, and performance*. PhD thesis, Stanford, CA, USA, 1995. 65

[58] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Softw. Eng.*, 5(5):440–452, 1979. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1979.230182. 205, 206

[59] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/358598.358613. 206

[60] K. M Chandy and Jayadev Misra. Termination detection of diffusing computations in communicating sequential processes. Technical report, Austin, TX, USA, 1980. 206

[61] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, 1983. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/357360.357365. 206

[62] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The amber system: parallel programming on a network of multiprocessors. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 147–158, New York, NY, USA, 1989. ACM. ISBN 0-89791-338-8. doi: http://doi.acm.org/10.1145/74850.74865. 149

[63] Jianwei Chen, Murali Annavaram, and Michel Dubois. Slacksim: a platform for parallel simulations of cmps on cmps. *SIGARCH Comput. Archit. News*, 37(2): 20–29, 2009. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1577129.1577134. 160, 215

[64] Francis Y. Chin, John Lam, and I-Ngo Chen. Efficient parallel algorithms for some graph problems. *Commun. ACM*, 25(9):659–665, 1982. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/358628.358650. 191

[65] Giovanni Chiola and Giuseppe Ciaccio. Efficient parallel processing on low-cost clusters with gamma active ports. *Parallel Computing*, 26(2-3):333–354, 2000. 214

[66] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 249–261, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3047-8. doi: http://dx.doi.org/10.1109/MICRO.2007.16. 159

[67] Thomas M. Conte, Mary Ann Hirsch, and Kishore N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, ICCD '96, pages 468–477, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7554-3. URL `http://portal.acm.org/citation.cfm?id=645464.653497`. 210

[68] Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of intel threading building blocks. In David Christie, Alan Lee, Onur Mutlu, and Benjamin G. Zorn, editors, *IISWC*, pages 57–66. IEEE, 2008. ISBN 978-1-4244-2778-9. 108

[69] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of dijkstra's shortest path algorithm. In *MFCS '98: Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, pages 722–731, London, UK, 1998. Springer-Verlag. ISBN 3-540-64827-5. 192

[70] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7(2):279–301, 1989. 106, 111

[71] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput.*, 36(5):547–553, 1987. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.1987.1676939. 181

[72] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0. doi: http://doi.acm.org/10.1145/1168857.1168900. 12

[73] Vincent Danjean and Raymond Namyst. Controling Kernel Scheduling from User Space: an Approach to Enhancing Applications' Reactivity to I/O Events. In *Proceedings of the 2003 International Conference on High Performance Computing (HiPC '03)*, Hyderabad, India, December 2003. URL http://www.springerlink.com/link.asp?id=r99f1x65v4gw07pp. 179

[74] Johan de Gelas. The quest for more processing power, part two: "multi-core and multi-threaded gaming", 3 2005. URL http://www.anandtech.com/show/1645. 10, 16

[75] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, 1970. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356571.356573. 136

[76] Edsger Wybe Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965. ISSN 0001-0782. 11

[77] Edsger Wybe Dijkstra and Carel S. Scholten. Termination detection for diffusing computations. *Inf. Process. Lett.*, 11(1):1–4, 1980. 206

[78] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 155–165, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: http://doi.acm.org/10.1145/1542476.1542494. 12

[79] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 434–442, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press. ISBN 0-8186-0719-X. doi: http://doi.acm.org/10.1145/17407.17406. 122, 125, 127, 133

[80] Christian Dufour, Jean Belanger, Simon Abourida, and Vincent Lapointe. Fpga-based real-time simulation of finite-element analysis permanent magnet synchronous machine drives. In *Proceedings of the 38th Annual IEEE Power Electronics Specialists Conference, PESC'07*, June 2007. 42

[81] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *SC '08: Proceedings of the 2008 ACM/IEEE conference*

on Supercomputing, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9. 67

[82] Sandhya Dwarkadas, Peter Keleher, Alan L. Cox, and Willy Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 144–155, New York, NY, USA, 1993. ACM. ISBN 0-8186-3810-9. doi: http://doi.acm.org/10.1145/165123.165150. 142, 143, 144

[83] Lieven Eeckhout and Koenraad De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, page 25, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1363-8. 210

[84] Lieven Eeckhout, Sebastien Nussbaum, James E. Smith, and Koen De Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, 2003. ISSN 0272-1732. doi: http://dx.doi.org/10.1109/MM.2003.1240210. 160

[85] Lieven Eeckhout, Hans Vandierendonck, and Koenraad De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2003. 50

[86] Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan Binkert, Roger Espasa, and Toni Juan. Asim: A performance model framework. *Computer*, 35:68–76, 2002. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/2.982918. 209

[87] Ralf S. Engelschall. Portable multithreading: the signal stack trick for user-space thread creation. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association. 38, 179

[88] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 27(2), 2009. 217

[89] W. Fenton, B. Ramkumar, V.A. Saletore, A.B. Sinha, and L.V. Kale. Supporting machine independent programming on diverse parallel architectures. In *Proceedings of the International Conference on Parallel Processing*, pages 193–201, St. Charles, IL, August 1991. 70

[90] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency, October 2008. 228

[91] B. Fleisch and G. Popek. Mirage: a coherent distributed shared memory design. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 211–223, New York, NY, USA, 1989. ACM. ISBN 0-89791-338-8. doi: http://doi.acm.org/10.1145/74850.74871. 141

[92] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/367766.368168. 182

[93] Michael J. Flynn and Patrick Hung. Microprocessor design issues: Thoughts on the road ahead. *IEEE Micro*, 25(3):16–31, 2005. ISSN 0272-1732. doi: http://dx.doi.org/10.1109/MM.2005.56. 3, 4

[94] The MPI Forum. MPI: A message passing interface. In Bob Borchers, editor, *Proceedings of the Supercomputing '93 Conference*, pages 878–885, Portland, OR, November 1993. IEEE Computer Society Press. ISBN 0-8186-4340-4. 14, 171, 176, 186, 223

[95] *Using the GNU Compiler Collection (GCC) version 4.4*. The Free Software Foundation, 2008. URL http://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/. 39

[96] Matteo Frigo. The weakest reasonable memory model. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, 1998. URL ftp://theory.lcs.mit.edu/pub/cilk/frigo-ms-thesis.ps.gz. 119

[97] Matteo Frigo. A fast fourier transform compiler. *SIGPLAN Not.*, 34(5):169–180, 1999. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/301631.301661. 65

[98] R. M. Fujimoto. Parallel discrete event simulation. In *WSC '89: Proceedings of the 21st Winter simulation conference*, pages 19–28, New York, NY, USA, 1989. ACM. ISBN 0-911801-58-8. doi: http://doi.acm.org/10.1145/76738.76741. 208

[99] G. R. Gao and V. Sarkar. Location consistency: Stepping beyond the barriers of memory coherence and serializability. Technical Report ACAPS Technical Memo 78, School of Computer Science, McGill University, December 1993. Revised Sep 1994. 132, 133

[100] G. R. Gao and V. Sarkar. Location consistency: Stepping beyond the memory coherence barrier. In *Proceedings of the 24th International Conference on Parallel Processing*, volume II, Software, pages II:73–76, Boca Raton, FL, August 1995. CRC Press. 132

[101] Guang R. Gao and Vivek Sarkar. Location consistency-a new memory model and cache consistency protocol. *IEEE Trans. Comput.*, 49(8):798–813, 2000. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/12.868026. 132, 133

[102] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of the 16th International Symposium on High-Performance Computing Architecture*, 01 2010. 160, 161, 164, 217

[103] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26, New York, NY, USA, 1990. ACM. ISBN 0-89791-366-3. doi: http://doi.acm.org/10.1145/325164.325102. 128, 129

[104] Tony Givargis and Frank Vahid. Platune: a tuning framework for system-on-a-chip platforms. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(11): 1317–1327, 2002. 161

[105] GNU. GNU Portable Threads. URL: http://www.gnu.org/software/pth/. 179

[106] Brice Goglin. Design and Implementation of Open-MX: High-Performance Message Passing over generic Ethernet hardware. In *Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2008*, Miami États-Unis d'Amérique, 2008. doi: 10.1109/{IPDPS}.2008.4536140. URL dx.doi.org/10.1109/{IPDPS}.2008.4536140http://hal.inria.fr/inria-00210704/en/. 178

[107] Brice Goglin and Nathalie Furmento. Finding a Tradeoff between Host Interrupt Load and MPI Latency over Ethernet. In IEEE, editor, *Cluster 2009*, New Orleans États-Unis d'Amérique, 2009. URL http://hal.inria.fr/inria-00397328/en/. 178

[108] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: implementing a fast parallel call. *J. Parallel Distrib. Comput.*, 37(1):5–20, 1996. ISSN 0743-7315. doi: http://dx.doi.org/10.1006/jpdc.1996.0104. 67

[109] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989. URL ftp://ftp.cs.wisc.edu/tech-reports/reports/91/tr1006.ps.Z. Also available as University of Wisconsin-Madison, Computer Sciences Department technical report TR #1006. 127, 128

[110] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 124–131, New York, NY, USA, 1983. ACM. ISBN 0-89791-101-6. doi: http://doi.acm.org/10.1145/800046.801647. 125

[111] J. A. Gosden. Explicit parallel processing description and control in programs for multi- and uni-processor computers. In *AFIPS '66 (Fall): Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 651–660, New York, NY, USA, 1966. ACM. doi: http://doi.acm.org/10.1145/1464291.1464361. 20

[112] W. S. Gosset. The probable error of a mean. *Biometrika*, 6(1):1–25, 1908. URL http://www.york.ac.uk/depts/maths/histstat/student.pdf. Published under the pseudonym of "Student". 49

[113] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969. 67

[114] R. L. Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *AFIPS '72 (Spring): Proceedings of the May 16-18, 1972, spring joint computer conference*, pages 205–217, New York, NY, USA, 1972. ACM. doi: http://doi.acm.org/10.1145/1478873.1478901. 67

[115] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulations of the barnes-hut method for n-body simulations. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 439–448, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-6605-6. 192

[116] T Grotker, S Liao, G Martin, and S Swan. System design with systemc, 2002. 209

[117] Erik Hagersten. *Towards Scalable Cache-Only Memory Architectures*. PhD thesis, Royal Institute of Technology, Stockholm / Swedish Institute of Computer Science, 1992. 81

[118] Yijie Han and Robert A. Wagner. An efficient and fast parallel-connected component algorithm. *Journal of the ACM*, 37(3):626–642, 1990. URL citeseer.ist.psu.edu/han90efficient.html. 191

[119] Tim Harris, Adrián Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, May/June 2007. ISSN 0272-1732. doi: http://dx.doi.org/10.1109/MM.2007.63. 12

[120] Allan Hartstein, Viji Srinivasan, Thomas R. Puzak, and Philip G. Emma. On the nature of cache miss behavior: Is it $\sqrt{2}$? *Journal of Instruction-Level parallelism (JILP)*, 10, June 2008. 78

[121] Abdelsalam Heddaya and Himanshu Sinha. Coherence, non-coherence and local consistency in distributed shared memory for parallel computing. Technical Report BU-CS-92-004, Computer Science Department, Boston University, Boston, MA, May 1992. 134

[122] John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau, et al. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, fourth edition, 2006. ISBN 0-12-370490-1 (paperback), 0-08-047502-7 (e-book), 0-12-373590-4. URL http://www.loc.gov/catdir/toc/ecip0618/2006024358.html&http://www.loc.gov/catdir/enhancements/fy0665/2006024358-d.html. 77

[123] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 13–26, New York, NY, USA, 1987. ACM. ISBN 0-89791-215-2. doi: http://doi.acm.org/10.1145/41625.41627. 122

[124] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM. ISBN 0-8186-3810-9. doi: http://doi.acm.org/10.1145/165123.165164. 11, 28, 43

[125] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/78969.78972. 122

[126] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/355620.361161. 65, 69

[127] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, and Victor Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 1-59593-061-2. doi: http://dx.doi.org/10.1109/SC.2005.53. 9, 82

[128] Phillip W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *Proc. 10th International Conference on Distributed Computing Systems*, pages 302–309, Paris, France, May-June 1990. IEEE Computer Society Press. 128

[129] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press, pub-IEEE:adr, 1995. 14, 23, 37

[130] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. *Theory of Computing Systems*, 31(4): 451–473, 1998. URL `http://springerlink.metapress.com/openurl.asp?genre=article&issn=1432-4350&volume=31&issue=4&spage=451`. formerly Mathematical Systems Theory. 131

[131] Liviu Iftode, Cezary Dubnicki, Edward W. Felten, and Kai Li. Improving release-consistent shared virtual memory using automatic update. In *HPCA '96: Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, page 14, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7237-4. 144

[132] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: a bridge between release consistency and entry consistency. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 277–287, New York, NY, USA, 1996. ACM. ISBN 0-89791-809-6. doi: http://doi.acm.org/10.1145/237502.237567. 131, 132, 145

[133] Intel. Intel threading building blocks. URL: http://www.threadingbuildingblocks.org/, . 68, 108, 160, 168, 223

[134] Intel. Intel tera-scale computing research program. URL: http://techresearch.intel.com/articles/Tera-Scale/1421.htm, . 4, 159

[135] Intel. Intel xeon processor 7500 series: a catalyst for mission-critical transformation, May 2010. Version 1.0. 77

[136] International Technology Roadmap for Semiconductors. System drivers, 2009. 77

[137] ISO/IEC. Programming languages — C, 1999. 78, 84, 98

[138] Aamer Jaleel, Matthew Mattina, and Bruce Jacob. Last level cache (llc) performance of data mining workloads on a cmp – a case study of parallel bioinformatics workloads. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 88–98, February 2006. doi: 10.1109/HPCA.2006.1598115. 46

[139] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/3916.3988. 207

[140] Christopher F. Joerg. *The Cilk system for parallel multithreaded computing*. PhD thesis, Cambridge, MA, USA, 1995. 66, 119, 120

[141] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-performance all-software distributed shared memory. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 213–226, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. doi: http://doi.acm.org/10.1145/224056.224073. 155, 156

[142] Kirk Lauritz Johnson. *High-performance all-software distributed shared memory*. PhD thesis, Cambridge, MA, USA, 1996. 155

[143] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, 1988. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/35037.42182. 148

[144] M. Frans Kaashoek and Andrew S. Tanenbaum. Group communication in the amoeba distributed operating system. In *ICDCS*, pages 222–230. IEEE Computer Society, 1991. 153

[145] M. Frans Kaashoek, Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. FLIP: An internetwork protocol for supporting distributed systems. *ACM Trans. Comput. Syst.*, 11(1):73–106, 1993. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/151250.151253. 153

[146] L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM parallel programming language and system: Part i – description of language features. Technical Report PPL-95-02, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1995. 70

[147] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM parallel programming language and system: Part ii – the runtime system. Technical Report PPL-95-03, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1995. 70

[148] Laxmikant V. Kalé. Comparing the performance of two dynamic load distribution methods. In *ICPP (1)*, pages 8–12, 1988. 154

[149] Laxmikant V. Kale and Milind A. Bhandarkar. Structured dagger: A coordination language for message-driven programming. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing*, pages 646–653, London, UK, 1996. Springer-Verlag. ISBN 3-540-61626-8. 215

[150] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++ : A Portable Concurrent Object-Oriented System Based on C++. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, New York, NY, USA, September 1993. ACM. ISBN 0-89791-587-9. doi: http://doi.acm.org/10.1145/165854.165874. 43, 70, 215

[151] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 13–21, New York, NY, USA, 1992. ACM. ISBN 0-89791-509-7. doi: http://doi.acm.org/10.1145/139669.139676. 142

[152] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: distributed shared memory on standard workstations and operating systems. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association. 142, 143

[153] Peter John Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Department of Computer Science, Rice University, Houston, TX, USA, January 1995. 142, 143

[154] Joshua L. Kihm and Daniel A. Connors. Statistical simulation of multithreaded architectures. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 67–74, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2458-3. doi: http://dx.doi.org/10.1109/MASCOT.2005.69. 160, 212

[155] Charles Koelbel. An overview of high performance fortran. *SIGPLAN Fortran Forum*, 11(4):9–16, 1992. ISSN 1061-7264. doi: http://doi.acm.org/10.1145/140734.140736. 63, 117

[156] Tim Kogel, Malte Doerper, Andreas Wieferink, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Serge Goossens. A modular simulation framework for architectural exploration of on-chip interconnection networks. In *CODES+ISSS '03: Proceedings*

of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, pages 7–12, New York, NY, USA, 2003. ACM. ISBN 1-58113-742-7. doi: http://doi.acm.org/10.1145/944645.944648. 161

[157] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre-Yves Droz. RAMP blue: A message-passing manycore system in FPGAs. In Koen Bertels, Walid A. Najjar, Arjan J. van Genderen, and Stamatis Vassiliadis, editors, *International Conference on Field Programmable Logic and Applications (FPL)*, pages 54–61. IEEE, 2007. ISBN 1-4244-1060-6. URL http://dx.doi.org/10.1109/FPL.2007.4380625. 227

[158] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 262–273, New York, NY, USA, 1993. ACM. ISBN 0-8186-4340-4. doi: http://doi.acm.org/10.1145/169627.169724. 63, 117, 118

[159] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: http://doi.acm.org/10.1145/1250734.1250759. 10, 225

[160] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. doi: http://doi.acm.org/10.1145/1504176.1504181. 16

[161] Thierry Lafage and André Seznec. Choosing representative slices of program execution for microarchitecture simulations: a preliminary application to the data stream. pages 145–163, 2001. 210

[162] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.1979.1675439. 120

[163] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/361082.361093. 11

[164] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359545.359563. 205

[165] Douglas Lea. libg++, the GNU C++ library. In USENIX Association, editor, *USENIX proceedings: C++ Conference*, pages 243–256, Denver, CO, October 1988. USENIX Association. 84

[166] Thomas J. LeBlanc, Michael L. Scott, and Christopher M. Brown. Large-scale parallel programming: experience with bbn butterfly parallel processor. In *PPEALS '88: Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 161–172, New York, NY, USA, 1988. ACM. ISBN 0-89791-276-4. doi: http://doi.acm.org/10.1145/62115.62131. 136

[167] Peizong Lee and Zvi Meir Kedem. Automatic data and computation decomposition on distributed memory parallel computers. *ACM Trans. Program. Lang. Syst.*, 24 (1):1–50, 2002. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/509705.509706. 79

[168] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. doi: http://doi.acm.org/10.1145/1815961.1816021. URL http://doi.acm.org/10.1145/1815961.1816021. 227

[169] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 148–159, New York, NY, USA, 1990. ACM. ISBN 0-89791-366-3. doi: http://doi.acm.org/10.1145/325164.325132. 81

[170] Man-Lap Li, Ruchira Sasanka, Sarita V. Adve, Yen kuang Chen, and Eric Debes. The alpbench benchmark suite for complex multimedia applications. In *IEEE International Symposium on Workload Characterization*, pages 34–45, 2005. 46

[171] Yong Li, Ahmed Abousamra, Rami Melhem, and Alex K. Jones. Compiler-assisted data distribution for chip multiprocessors. In *Proceedings of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 501–512, September 2010. ISBN 978-1-4503-0178-7. 79

[172] Zheng Li. *Architectural Support for the CAPSULE Parallelization Environment.* PhD thesis, École doctorale d'informatique de l'université de Paris-Sud XI, December 2010. 104

[173] Zheng Li, Olivier Certner, José Duato, and Olivier Temam. Scalable hardware support for conditional parallelization. In *Proceedings of the nineteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 157–168, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi: http://doi.acm.org/10.1145/1854273.1854297. 104, 115, 224, 261

[174] Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Trans. Softw. Eng.*, 13(1):32–38, 1987. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1987.232563. 106, 107, 108, 155

[175] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Princeton, NJ, September 1988. 127

[176] B. Lubachevsky, A. Shwartz, and A. Weiss. Rollback sometimes works...if filtered. In *WSC '89: Proceedings of the 21st Winter simulation conference*, pages 630–639, New York, NY, USA, 1989. ACM. ISBN 0-911801-58-8. doi: http://doi.acm.org/10.1145/76738.76819. 208

[177] Reinhard Lüling, Burkhard Monien, and Friedhelm Ramme. Load balancing in large networks: a comparative study. In *SPDP*, pages 686–689. IEEE Computer Society, 1991. ISBN 0-8186-2310-1. 106

[178] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0475-2. 42

[179] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.982916. 211

[180] Doug Matzke. Will physical scalability sabotage performance gains? *Computer*, 30 (9):37–39, 1997. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.612245. 3

[181] Sally A. McKee. Reflections on the memory wall. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, page 162, New York, NY, USA, 2004. ACM. ISBN 1-58113-741-9. doi: http://doi.acm.org/10.1145/977091.977115. 77

[182] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the 16th International Symposium on High-Performance Computing Architecture*, 01 2010. 160, 161, 175, 194, 215

[183] J. C. Mogul and G. Minshall. Rethinking the TCP Nagle Algorithm. *SIGCOMM Comput. Commun. Rev.*, 31(1):6–20, 2001. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/382176.382177. 178

[184] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):264–280, 1991. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/71.86103. 67

[185] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. *Parallel Architectures and Compilation Techniques, International Conference on*, 0: 261–270, 2009. ISSN 1089-795X. doi: http://doi.ieeecomputersociety.org/10.1109/PACT.2009.22. 156, 177

[186] Matteo Monchiero, Jung Ho Ahn, Ayose Falcon, Daniel Ortega, and Paolo Faraboschi. How to simulate 1000 cores. Technical Report HPL-2008-190, Hewlett Packard Laboratories, November 9 2008. URL http://www.hpl.hp.com/techreports/2008/HPL-2008-190.pdf. 160, 217, 227

[187] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965. 3

[188] Gordon E. Moore. Progress in digital integrated electronics. *International Electron Devices Meeting*, 21:11–13, 1975. 3

[189] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: log-based transactional memory. In *HPCA*, pages 254–265. IEEE Computer Society, 2006. 12

[190] Shubhendu Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Huss-lederman, Mark D. Hill, James R. Larus, and David A. Wood. Wisconsin Wind Tunnel II: A fast and portable parallel architecture simulator. In *In Workshop on Performance Analysis and Its Impact on Design (PAID)*, 1997. 214

[191] Jeffrey Namkung, Dohyung Kim, Rajesh Gupta, Igor Kozintsev, Jean-Yves Bouget, and Carole Dulong. Phase guided sampling for efficient parallel application simulation. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 187–192, New York, NY, USA, 2006. ACM. ISBN 1-59593-370-0. doi: http://doi.acm.org/10.1145/1176254.1176301. 160, 194, 213

[192] Ramanathan Narayanan, Berkin Özıs. Ikyılmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. Minebench: A benchmark suite for data mining workloads. In *IEEE International Symposium on Workload Characterization*, pages 182–188, 2006. 46

[193] Angeles Navarro, Emilio Zapata, and David Padua. Compiler techniques for the distribution of data and computation. *IEEE Trans. Parallel Distrib. Syst.*, 14(6): 545–562, 2003. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/TPDS.2003.1206503. 79

[194] Lionel M. Ni, Chong-Wei Xu, and Thomas B. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Trans. Softw. Eng.*, 11(10):1153–1161, 1985. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1985.231863. 106

[195] Sébastien Nussbaum and James E. Smith. Modeling superscalar processors via statistical simulation. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1363-8. 160, 210

[196] Nvidia. Fermi, the next generation CUDA architecture. URL: http://www.nvidia.fr/object/fermi_architecture_fr.html. 4, 77

[197] Open SystemC Initiative (OSCI). *TLM 2.0 Language Reference Manual*, . 211

[198] Open SystemC Initiative (OSCI). TLM 2.0 Whitepaper, . 211, 215

[199] Mark Oskin, Frederic T. Chong, and Matthew K. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *ISCA*, pages 71–82, 2000. 160, 210

[200] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009. ISBN 978-3-642-03358-2. URL http://dx.doi.org/10.1007/978-3-642-03359-9. 134

[201] Pierre Palatin. *Exploitation pratique et efficace du parallélisme sur processeurs multi-cœurs*. PhD thesis, Université Paris XI Orsay, September 2008. 5, 13, 25, 29, 33, 34, 35, 223

[202] Pierre Palatin, Yves Lhuillier, and Olivier Temam. CAPSULE: Hardware-assisted parallel execution of component-based programs. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 247–258, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9. doi: http://dx.doi.org/10.1109/MICRO.2006.13. 5, 13, 29, 48, 60, 191

[203] David A. Penry and David I. August. Optimizations for a simulator construction system supporting reusable components. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 926–931, New York, NY, USA, 2003. ACM. ISBN 1-58113-688-9. doi: http://doi.acm.org/10.1145/775832.776065. 209

[204] David A. Penry, Daniel Fay, David Hodgdon, Ryan Wells, Graham Schelle, David I. August, and Dan Connors. Exploiting parallelism and structure to accelerate the simulation of chip multi-processors. In *HPCA*, pages 29–40, 2006. 160, 216

[205] Erez Perelman, Marzia Polito, Jean yves Bouguet, John Sampson, Brad Calder, and Carole Dulong. Detecting phases in parallel applications on shared memory architectures. In *In International Parallel and Distributed Processing Symposium*, pages 25–29, 2006. 160, 194, 213

[206] Daniel Gracia Perez, Gilles Mouchard, and Olivier Temam. MicroLib: A case for the quantitative comparison of micro-architecture mechanisms. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–54, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2126-6. doi: http://dx.doi.org/10.1109/MICRO.2004.25. 209

[207] Dac Pham, Shigehiro Asano, Mark Bolliger, Michael N. Day, H. Peter Hofstee, Charles R. Johns, James A. Kahle, Atsushi Kameyama, John M. Keaty, Yoshio Masubuchi, Mack W. Riley, David Shippy, Daniel Stasiak, Masakazu Suzuoki, M. Wang, James D. Warnock, Steve Weitzel, Dieter F. Wendel, Takeshi Yamazaki,

and K. Yazawa. The design and implementation of a first-generation CELL processor. In *IEEE International Solid-State Circuits Conference*, pages 184–185,592, February 2005. 99

[208] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1484–1489, New York, NY, USA, 2007. ACM Press. ISBN 978-3-9810801-2-4. 42

[209] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In Mary Lou Soffa and Evelyn Duesterwald, editors, *CGO*, pages 114–123. ACM, 2008. ISBN 978-1-59593-978-4. 229

[210] Keith Harold Randall. *Cilk: Efficient multithreaded computing.* PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1998. 66, 119, 120

[211] Lawrence Rauchwerger, Francisco Arzu, and Koji Ouchi. Standard templates adaptive parallel library (stapl). In *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 402–409, London, UK, 1998. Springer-Verlag. ISBN 3-540-65172-1. 64

[212] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and typhoon: user-level shared memory. *SIGARCH Comput. Archit. News*, 22(2):325–336, 1994. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/192007.192062. 146

[213] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. *SIGMETRICS Perform. Eval. Rev.*, 21(1):48–60, 1993. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/166962.166979. 214

[214] Mack W. Riley, James D. Warnock, and Dieter F. Wendel. Cell broadband engine processor: Design and implementation. *IBM Journal of Research and Development*, 51(5):545–558, 2007. 99

[215] Philip E. Ross. Why cpu frequency stalled. *IEEE Spectrum*, 45(4):72, April 2008. 3

[216] Bernard Roy. Transitivité et connexité. *C. R. Acad. Sci. Paris*, 249:216–218, 1959. 182

[217] Behrokh Samadi. *Distributed simulation, algorithms and performance analysis (load balancing, distributed processing).* PhD thesis, University of California, Los Angeles, 1985. 207

[218] Farideh A. Samadzadeh and G. E. Hedrick. Near-optimal multiprocessor scheduling. In *CSC '92: Proceedings of the 1992 ACM annual conference on Communications*, pages 477–484, New York, NY, USA, 1992. ACM. ISBN 0-89791-472-4. doi: http://doi.acm.org/10.1145/131214.131275. 67

[219] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-cc multiprocessor machine code. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 379–391, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: http://doi.acm.org/10.1145/1480881.1480929. 128

[220] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989. ISBN 0262691302. 67

[221] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple coma. In *HPCA '95: Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, page 276, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-6445-2. 81

[222] Ali Sayinta, Gorkem Canverdi, Marc Pauwels, Amer Alshawa, and Wim Dehaene. A mixed abstraction level co-simulation case study using SystemC for system on chip verification. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 20095, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1870-2-2. 161

[223] D. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on smp clusters. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 125, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8323-6. 147

[224] Daniel J. Scales and Kourosh Gharachorloo. Design and performance of the shasta distributed shared memory protocol. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 245–252, New York, NY, USA, 1997. ACM. ISBN 0-89791-902-5. doi: http://doi.acm.org/10.1145/263580.263643. 146, 147

[225] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 174–185, New York, NY, USA, 1996. ACM. ISBN 0-89791-767-7. doi: http://doi.acm.org/10.1145/237090.237179. 146, 147

[226] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *ISCA '87: Proceedings of the 14th annual international symposium on Computer architecture*, pages 234–243, New York, NY, USA, 1987. ACM. ISBN 0-8186-0776-9. doi: http://doi.acm.org/10.1145/30350.30377. 124, 125

[227] C. E. Scheurich. *Access ordering and coherence in shared memory multiprocessors*. PhD thesis, Los Angeles, CA, USA, 1989. 123

[228] Jörn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 35–44, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-136-4. 42

[229] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. *SIGOPS Oper. Syst. Rev.*, 28(5):297–306, 1994. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/381792.195575. 146

[230] Robert Schöne, Wolfgang E. Nagel, and Stefan Pflüger. Analyzing cache bandwidth on the Intel Core 2 architecture. In Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 365–372. IOS Press, 2007. ISBN 978-1-58603-796-3. 156, 177

[231] Loren Schwiebert. Deadlock-free oblivious wormhole routing with cyclic dependencies. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 149–158, New York, NY, USA, 1997. ACM. ISBN 0-89791-890-8. doi: http://doi.acm.org/10.1145/258492.258507. 182

[232] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997. 12

[233] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1363-8. 210

[234] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, New York, NY, USA, 2002. ACM. ISBN 1-58113-574-2. doi: http://doi.acm.org/10.1145/605397.605403. 159, 210

[235] W. Shu and L. V. Kale. A dynamic scheduling strategy for the chare-kernel system. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, 1989.*, pages 389 –398, 12-17 1989. doi: 10.1145/76263.76306. 155

[236] Jaswinder P Singh, Wolf Weber, and Anoop Gupta. Splash: Stanford parallel applications for shared-memory*. Technical report, Stanford, CA, USA, 1992. 46

[237] Lisa M. Sokol, Jon B. Weissman, and Paula A. Mutchler. MTW: An empirical performance study. In *WSC '91: Proceedings of the 23rd Winter simulation conference*, pages 557–563, Washington, DC, USA, 1991. IEEE Computer Society. ISBN 0-7803-0181-1. 208

[238] SPARC International, Inc. *The SPARC Architecture Manual—Version 8*. Prentice-Hall, pub-PH:adr, 1992. ISBN 0-13-825001-4. 134

[239] Jeff Steinman. SPEEDES: A multiple-synchronization environment for parallel discrete-event simulation. *The International Journal for Computer Simulation*, 2(3): 241–286, 1992. 214

[240] Alexander Stepanov and Meng Lee. The standard template library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994. 64

[241] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005. 4

[242] R. J. Swan, S. H. Fuller, and D. P. Siewiorek. The structure and architecture of cm*: A modular, multi-microprocessor. In *Carnegie-Mellon University Computer Science Department Research Review*. Pittsburgh, PA, December 1976. 136

[243] Richard J. Swan, Andy Bechtolsheim, Kwok-Woon Lai, and John K. Ousterhout. The implementation of the cm* multi-microprocessor. In *AFIPS '77: Proceedings of the June 13-16, 1977, national computer conference*, pages 645–655, New York, NY, USA, 1977. ACM. doi: http://doi.acm.org/10.1145/1499402.1499516. 136

[244] C. K. Tang. Cache system design in the tightly coupled multiprocessor system. In *AFIPS '76: Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 749–753, New York, NY, USA, 1976. ACM. doi: http://doi.acm.org/10.1145/1499799.1499901. 125

[245] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/62.2160. 138

[246] Chandramohan A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 110–119, New York, NY, USA, 1994. ACM. ISBN 0-89791-660-3. doi: http://doi.acm.org/10.1145/195473.195515. 146

[247] Samuel Thibault. A flexible thread scheduler for hierarchical multiprocessor machines. In *Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, Cambridge, USA, June 2005. URL `http://hal.inria.fr/inria-00000138`. 179

[248] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 49–84. Springer

Berlin / Heidelberg, 2002. URL http://dx.doi.org/10.1007/3-540-45937-5_14. 10.1007/3-540-45937-5_14. 42

[249] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in stapl. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 277–288, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. doi: http://doi.acm.org/10.1145/1065944.1065981. 65

[250] Tilera. Tilera website. URL: http://www.tilera.com/. 4, 159

[251] Alan Turing. On computable numbers, with an application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936. URL doi:10.1112/plms/s2-42.1.230. 3

[252] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3. doi: http://doi.acm.org/10.1145/1693453.1693479. 69

[253] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, Sharad Malik, and David I. August. The liberty simulation environment: A deliberate approach to high-level system modeling. *ACM Trans. Comput. Syst.*, 24(3):211–249, 2006. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/1151690.1151691. 209

[254] Scott Vetter, Giuliano Anselmi, Bruno Blanchard, Younghoon Cho, Christopher Hales, and Marcos Quezada. Ibm power 770 and 780 technical overview and introduction. Technical Report REDP-4639-00, IBM, March 2010. 77

[255] David L. Weaver and Tom Germond. *The SPARC Architecture Manual—Version 9*. Prentice-Hall PTR, pub-PHPTR:adr, 1994. ISBN 0-13-099227-5. 134

[256] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006. URL http://dblp.uni-trier.de/db/journals/micro/micro26.html#WenischWFAFH06. 159, 213

[257] Marc Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel Distrib. Syst.*, 4(9):979–993, 1993. 106, 107

[258] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM. ISBN 0-89791-698-0. doi: http://doi.acm.org/10.1145/223982.223990. 46, 217

[259] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *in Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97, 2003. 159, 210

[260] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. Spl: a language and compiler for dsp algorithms. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 298–308, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: http://doi.acm.org/10.1145/378795.378860. 4, 65

[261] Di Xu, Chenggang Wu, and Pen- Chung Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *Proceedings of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 237–247, September 2010. ISBN 978-1-4503-0178-7. 78

[262] Sami Yehia, Sylvain Girbal, Hugues Berry, and Olivier Temam. Reconciling specialization and flexibility through compound circuits. In *HPCA*, pages 277–288. IEEE Computer Society, 2009. 229

[263] Joshua J. Yi, Resit Sendag, David J. Lilja, and Douglas M. Hawkins. Speed versus accuracy trade-offs in microarchitectural simulations. *IEEE Transactions on Computers*, 56(11):1549–1563, 2007. ISSN 0018-9340. doi: http://doi.ieeecomputersociety.org/10.1109/TC.2007.70744. 205

[264] Hao Yu, Dongmin Zhang, and Lawrence Rauchwerger. An adaptive algorithm selection framework. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 278–289, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2229-7. doi: http://dx.doi.org/10.1109/PACT.2004.6. 65

[265] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software write detection for a distributed shared memory. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 8, Berkeley, CA, USA, 1994. USENIX Association. 144, 145, 146

[266] G. Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kale. Bigsim: a parallel simulator for performance prediction of extremely large parallel machines. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 78–, April 2004. doi: 10.1109/IPDPS.2004.1303013. 214

[267] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 75–88, New York, NY, USA, 1996. ACM. ISBN 1-880446-82-0. doi: http://doi.acm.org/10.1145/238721.238763. 145

Index

Index

cell, 85
  data section, 86
  header, 96
  links section, 87
  sections, 86
conditional parallelization, 17
control message, 165, 176, 184
  shutdown, 184

divide, *see* primitive

group, 24
  inheritance, 25
  new, 25
  wait, 25

handle, 85
historization, 183

lazy update, 95
link, 85

memory model, 82

ping-pong effect, 110
primitive
  divide, 17
  probe, 17
probe, *see* primitive
proxy
  cell, 93
  chain, 95
  link, 94
  virtual time, *see* virtual time

radius effect, 110
recursive work declaration, 20

spatial synchronization, 161, 165
synchronization group, *see* group

task queue, 104
  global end, 115
  local end, 104, 115

virtual time, 163
  propagation, 164
  proxy, 165
  shadow, 168

This document was prepared with TEX Live 2010, using the
XEmacs 21.5.28 editor on systems running FreeBSD 7-STABLE
(which includes a bunch of GNU utilities). Figures were done
with OpenOffice, Ti*k*Z and gnuplot.

Résumé

L'accroissement régulier de la fréquence des micro-processeurs et des importants gains de puissance qui en avaient résulté ont pris fin en 2005. Les autres principales techniques matérielles d'amélioration de performance pour les programmes séquentiels (exécution dans le désordre, antémémoires, prédiction de branchement, etc.) se sont largement essouflées. Conduisant à une consommation de puissance toujours plus élevée, elles posent de délicats problèmes économiques et techniques (refroidissement des puces). Pour ces raisons, les fabricants de micro-processeurs ont choisi d'exploiter le nombre croissant de transistors disponibles en plaçant plusieurs cœurs de processeurs sur une même puce.

Par cette thèse, nous avons pour objectif et ambition de préparer l'arrivée probable, dans les prochaines années, de processeurs multi-cœur à grand nombre de cœurs (une centaine ou plus). À cette fin, nos recherches se sont orientées dans trois grandes directions. Premièrement, nous améliorons l'environnement de parallélisation Capsule, dont le principe est la parallélisation conditionnelle, en lui adjoignant des primitives de synchronization de tâches robustes et en améliorant sa portabilité. Nous étudions ses performances et montrons ses gains par rapport aux approches usuelles, aussi bien en terme de rapidité que de stabilité du temps d'exécution. Deuxièmement, la compétition entre de nombreux cœurs pour accéder à des mémoires aux performances bien plus faibles va vraisemblablement conduire à répartir la mémoire au sein des futures architectures multi-cœur. Nous adaptons donc Capsule à cette évolution en présentant un modèle de données simple et général qui permet au système de déplacer automatiquement les données en fonction des accès effectués par les programmes. De nouveaux algorithmes répartis et locaux sont également proposés pour décider de la création effective des tâches et de leur répartition. Troisièmement, nous développons un nouveau simulateur d'évènements discrets, SiMany, pouvant prendre en charge des centaines à des milliers de cœurs. Nous montrons qu'il reproduit fidèlement les variations de temps d'exécution de programmes observées sur un simulateur de niveau cycle jusqu'à 64 cœurs. SiMany est au moins 100 fois plus rapide que les meilleurs simulateurs flexibles actuels. Il permet l'exploration d'un champ plus large d'architectures ainsi que l'étude des grandes lignes du comportement des logiciels sur celles-ci, ce qui en fait un outil majeur pour le choix et l'organisation des futures architectures multi-cœur et des solutions logicielles qui les exploiteront.

Abstract

Since 2005, chip manufacturers have stopped raising processor frequencies, which had been the primary mean to increase processing power since the end of the 90s. Other hardware techniques to improve sequential execution time (out-of-order processing, caches, branch prediction, etc.) have also shown diminishing returns, while raising the power envelope. For these reasons, commonly referred to as the frequency and power walls, manufacturers have turned to multiple processor cores to exploit the growing number of available transistors on a die.

In this thesis, we anticipate the probable evolution of multi-core processors into *many-core* ones, comprising hundreds of cores and more, by focusing on three important directions. First, we improve the Capsule programming environment, based on *conditional parallelization*, by adding robust coarse-grain synchronization primitives and by enhancing its portability. We study its performance and show its benefits over common parallelization approaches, both in terms of speedups and execution time stability. Second, because of increased contention and the memory wall, many-core architectures are likely to become more distributed. We thus adapt Capsule to distributed-memory architectures by proposing a simple but general data structure model that allows the associated run-time system to automatically handle data location based on program accesses. New distributed and local schemes to implement conditional parallelization and work dispatching are also proposed. Third, we develop a new discrete-event-based simulator, *SiMany*, able to sustain hundreds to thousands of cores with practical execution time. We show that it successfully captures the main program execution trends by comparing its results to those of a cycle-accurate simulator up to 64 cores. SiMany is more than a hundred times faster than the current best flexible approaches. It pushes further the limits of high-level architectural design-space exploration and software trend prediction, making it a key tool to design future many-core architectures and to assess software scalability on them.