Un modèle de transition logico-matérielle pour la simplification de la programmation parallèle

Chong Li

UNIVERSITÉ
PARIS-EST

ÉCOLE DOCTORALE
Mathématiques et STIC

THÈSE DE DOCTORAT

Informatique

UN MODÈLE DE TRANSITION LOGICO-MATÉRIELLE POUR LA SIMPLIFICATION DE LA PROGRAMMATION PARALLÈLE

A SOFTWARE-HARDWARE BRIDGING MODEL FOR SIMPLIFYING PARALLEL PROGRAMMING

Présentée par:

LI Chong

Soutenue le 03 juillet 2013

Rapporteurs:	KUCHEN Herbert	Professeur	Universität Münster
	CHAILLOUX Emmanuel	Professeur	Univ. P-et-M-Curie
Examinateurs:	MILLER Quentin	Lecturer (MdC)	Somerville C. Oxford
	LOULERGUE Frédéric	Professeur	Univ. d'Orléans
	GAVA Frédéric	Maître de Conf. - HdR	Univ. Paris-Est
	GREBET Jean-Guillaume	Resp. de Recherche	S.A.S EXQIM
Dir. de thèse:	HAINS Gaétan	Professeur	Univ. Paris-Est

REMERCIEMENTS

J'aimerais, en premier lieu, remercier vivement M. Herbert KUCHEN, Professeur à Universität de Münster, d'Allemagne, et Emmanuel CHAILLOUX, Professeur à Université Pierre-et-Marie-Curie (Paris VI), pour l'honneur qu'ils m'ont fait en acceptant d'être les rapporteurs de ma thèse.

Mes remerciements s'adressent également à M. Quentin MILLER, Lecturer à Somerville College de University of Oxford, d'Angleterre, M. Frédéric LOULERGUE, Professeur à Université d'Orléans, M. Frédéric GAVA, Maître de Conférences - HDR à Université Paris Paris-Est Créteil (Paris XII), et M. Jean-Guillaume GREBET, Directeur Général Délégué et Responsable de Recherche de EXQIM S.A.S., du Luxembourg, pour l'honneur qu'ils m'ont fait en acceptant de bien vouloir participer à ce jury de soutenance.

J'aimerais exprimer toute ma profonde gratitude à M. Gaétan Joseph Daniel Robert HAINS, Professeur à Université Paris-Est, qui a assuré l'encadrement de cette thèse avec beaucoup de patience et d'enthousiasme. Je tiens tout particulièrement à lui exprimer ma plus profonde gratitude pour son soutien de tous les jours qui m'a permis de mener à terme ces travaux. Son expérience et ses conseils ont été décisifs pour le déroulement de ce travail. Je le remercie pour son extrême gentillesse et sa disponibilité de tous les instants.

J'adresse également des vifs remerciements à M. Jacques LUCAS, Président-Directeur Général de EXQIM S.A.S., et au Dr. Jean-Guillaume GREBET, pour m'avoir choisi pour réaliser cette thèse industrielle et co-financé sous la forme d'une Convention Industrielle de Formation par la Recherche (CIFRE) avec le Ministère de l'Enseignement Supérieur et de la Recherche. Un grand merci pour m'avoir donné l'accès du supercalculateur SGI Altix de EXQIM S.A.S. dont j'ai mené les expérimentations.

Je tiens également à remercier M. Murray COLE, Reader à University of Edinburgh à l'Écosse, M. Youry KHMELEVSKY, Adjunct Professor à University of British Columbia au Canada, M. Mostafa BAMHA, Maître de Conférences à Université d'Orléans, M. Muath ALRAMMAL, Assistant Professor

à Al Khawarizmi International College à l'Abou Dabi, et de nombreux collègues de mon laboratoire LACL, de mon école doctorale MSTIC, de mon université PRES Paris-Est, et du labotatoire LIFO de l'université d'Orléans pour les nombreux conseils scientifiques qu'ils m'ont prodigués tout au long de cette thèse.

J'adresse mes sincères remerciements à M. Bernard LAPEYRE, Directeur du Département des Études doctorales d'Université Paris-Est, Mme Claude TU, Ajointe au directeur du Département des Études doctorales, Mme Sylvie CACH, Responsable administrative de l'école docotrale MSTIC, Mme Flore TSILA, Secrétaire du labotatoire LACL, et al. pour leur soutiens administratifs. Un grand merci à l'école doctorale MSTIC, pour m'avoir financé mes formations doctorales organisées par ENPC SIM et par CEA-INRIA-EDF, et pour m'avoir autorisé de rédiger cette thèse en anglais, ce qui m'a facilité la tâche et surtout épargné du travail.

Je souhaite aussi remercier M. Aurélien GONNAY, Ingénieur architecte logiciel, M. Mohamad AL HAJJ HASSAN, Docteur en algorithmes parallèlese et spécialiste de base de données distributée, M. Moez HAMMAMI, Ingénieur statisticien économiste, M. Gwénolé LE MENN, Normalien gestionnaire de risques financiers, M. Olivier PHILIPPE, Normalien gérant de portefeuilles, et tous les collègues de EXQIM S.A.S. pour leur soutien et les nombreux conseils divers depuis la fondation de EXQIM S.A.S..

J'adresse un grand merci à toute ma famille qui a toujours été présente lorsque j'en ai eu besoin, en particulier, à ma mère et mon père. Je suis là grâce à vos sacrifices, votre soutien et vos encouragements pendant toutes ces années.

Enfin et surtout, il y a de nombreuses personnes à qui je souhaite adresser un grand merci. Cependant, il est impossible de tous citer ici. Je vous tous remercie vivement, les amis!

Paris, le 8 juin 2013.

Cette thèse a été effectuée auprès de :

Laboratoire d'Algorithmique, Complexité et Logique
Département d'Informatique
Faculté des Sciences et Technologies
Université Paris-Est Créteil
61, avenue du Général de Gaulle
94010 Créteil, France
www.lacl.fr

et au sein de :

Exclusive Quantitative Investment Management
24, rue de Caumartin
75009 Paris, France
www.exqim.com

RÉSUMÉ LONG EN FRANÇAIS

La programmation parallèle et les algorithmes data-parallèles sont depuis plusieurs décennies les principales techniques qui sous-tendent l'informatique haute performance. Comme toutes les propriétés non-fonctionnelles du logiciel, la conversion des ressources informatiques dans des performances évolutives et prévisibles implique un équilibre délicat entre abstraction et automatisation avec une précision sémantique. Au cours de la dernière décennie, de plus en plus de professions ont besoin d'une puissance de calcul très élevée. Cependant, comme dans les secteurs financiers de trading algorithmique, les équipes sont structurées en binôme, composées d'un analyste quantitatif, spécialisé dans les algorithmes financiers, et un développeur informatique chargé de l'implémentation et des problématiques de haute performance. De plus, la résolution d'un algorithme complexe requiert parfois l'expertise d'un spécialiste en logico-matériel haute performance, notamment pour paralléliser l'algorithme sur un super-calculateur donné. Par ailleurs, la migration des programmes existants vers une nouvelle configuration matérielle ou le développement de nouveaux algorithmes à finalité spécifique dans un environnement parallèle n'est jamais un travail facile, ni pour les développeurs de logiciel, ni pour les spécialistes du domaine.

Dans cette thèse, nous décrivons le travail qui vise à simplifier le développement de programmes parallèles, en améliorant également la portabilité du code de programmes parallèles et la précision de la prédiction de performance d'algorithmes parallèles pour des environnements hétérogènes. Avec ces objectifs à l'esprit, nous avons proposé un modèle de transition nommé SGL pour la modélisation des architectures parallèles hétérogènes et des algorithmes parallèles, et une mise en œ uvre de squelettes parallèles basés sur le modèle SGL pour le calcul haute performance. SGL simplifie la programmation parallèle à la fois pour les machines parallèles classiques et pour les nouvelles machines hiérarchiques. Il généralise les primitives de la programmation BSML. SGL pourra plus tard en utilisant des techniques de Model-Driven pour la génération de code automatique à partir d'une fiche technique sans codage complexe, par exemple pour le traitement de Big-Data sur un sys-

tème hétérogène massivement parallèle. Le modèle de coût de SGL améliore la clarté de l'analyse de performance des algorithmes, permet d'évaluer la performance d'une machine et la qualité d'un algorithme.

Le chapitre 1 est l'introduction de la thèse. Dans le chapitre 2, nous suivons l'histoire de supercalculateurs pour examiner les différentes architectures de supercalculateurs afin d'avoir une vision globale des architectures de machines parallèles sur lesquels un programme parallèle peut être exécuté. L'évolution du matériel informatique parallèle montre que la vue à plat d'une machine parallèle comme un ensemble de communication de machines séquentielles reste un modèle utile et pratique, mais est de plus en plus incomplète. En outre, on observe que les multiprocesseurs hétérogènes présentent des opportunités uniques pour améliorer le débit du système et réduire la consommation du processeur. L'engouement par le green-computing met encore plus de pression sur l'utilisation optimale des architectures qui ne sont pas seulement hautement évolutives mais hiérarchiques et non-homogènes. Tous ces changements rendent la programmation parallèle plus difficile qu'avant. Un modèle de transition logico-matérielle[1] réaliste est souhaitable pour la manipulation de ces machines hiérarchiques hétérogènes.

Nous avons ensuite traversé dans le chapitre 3 l'état de l'art des modèles de programmation parallèle. Nous avons constaté que la programmation concurrente multi-threadée est bien pour démarrer, mais elle ne peut être appliquée qu'à une architecture à mémoire partagée; l'approche du passage de messages gère l'architecture à mémoire distribuée, mais la gestion de la communication n'est jamais une tâche facile; les modèles d'acteurs fournissent des schémas de communication, mais leur application est trop difficile à optimiser sans donner une transition algorithme-machine; le modèle de transition BSP [Val90] relie le logiciel et le matériel, fournit une vue séquentielle de programme parallèle grâce aux super-étapes, simplifie la conception et l'analyse d'algorithmes grâce à sa barrière, mais les ordinateurs parallèles se développent aujourd'hui de plus en plus dans une architecture hiérarchique contrairement à la structure plate proposée par BSP; MapReduce simplifie le traitement de données à grand échelle sur une grappe d'ordinateurs distribués en masquant la communication, mais sa capacité à gérer des algorithmes complexes tout en gardant de bonnes performances peut être remise en question. Toutes ces observa-

1 Nous traduisons ici le mot anglais "bridging model" en français "modèle de transition logico-matérielle".

tions nous conduisent à proposer un modèle de transition logico-matérielle hiérarchique pour simplifier la programmation parallèle.

Nous avons donc introduit dans le chapitre 4 notre modèle de programmation et d'exécution parallèle sous la forme d'un langage impératif et simple (Scatter-Gather Language ou SGL) [LH12a, LH11b]. Des études existantes ont permis d'identifier l'originalité et l'utilité de certains aspects de SGL. SGL est muni d'une sémantique opérationnelle claire et d'un modèle de coût précis. Le modèle de coût de SGL améliore la clarté de l'analyse de la performance des algorithmes; il permit d'analyser à la fois la performance d'une machine et la qualité d'un algorithme. Nous estimons que l'ordinateur SGL peut couvrir la plupart des ordinateurs parallèles modernes. Le coût de la synchronisation de l'algorithme de SGL pour un ordinateur massivement parallèle peut être considérablement réduit par sa structure hiérarchique. Le coût de la communication de SGL entre les différents niveaux est plus réaliste que la structure plane.

SGL a été utilisé dans le chapitre 5 avec son modèle de programmation pour programmer des opérations parallèles de base et des squelettes parallèles plus complexes, tels que la réduction parallèle, le préfixe parallèle, le tri parallèle par échantillonnage et un algorithme de papillon: homomorphisme distribuable [LGH12]. Dans tous les cas, l'évolutivité et la précision du modèle de coûts ont été mesurées. Il a été constaté qu'un sous-ensemble des modèles de communication parallèle n'est pas naturellement couvert par des opérations centralisées de SGL. Ceci est visible dans une implémentation récursive du tri parallèle par échantillonnage.

Une analyse plus approfondie nous a conduit à proposer dans le chapitre 6 deux éléments d'une solution générale à ce dilemme. Le premier est le théorème GPS: une équivalence sémantique entre les sous-programmes **gather** ; `P` ; **scatter** (où `P` est un programme séquentiel local dans le *maître*) et les opérations horizontales comme **put** de BSML (BSP-CAML). Ce résultat sert de base des optimisations de future compilateur par lequel une gamme de programmes SGL propres mais inefficaces peuvent être compilés en programmes de plus bas niveau mais plus efficaces à l'aide de la communication horizontale. La deuxième solution proposée est une forme simplifiée de **put** que nous avons conçu et expérimenté dans BSML. Avec celle-ci, l'algorithme BSP de tri parallèle d'échantillonnage de Tiskin-McColl [Tis99] a été programmé sans devoir coder une matrice de communication générale: seulement la relation

de communication 1-à-n pour le côté d'émetteur et n-à-1 pour le côté de récepteur.

La définition et la validation de SGL détaillées dans cette thèse ne sont qu'une première étape vers l'application sûre et efficace pour la programmation parallèle de haut niveau. La conclusion ainsi que nos perspectives de travail sont présentées dans le chapitre 7. De nombreuses perspectives sont envisageables par les travaux futurs. La compilation de code SGL sera étudiée en définissant un langage complet qui comprendra la syntaxe et la sémantique de SGL données ici. Une autre direction qui sera explorée est l'utilisation de SGL comme langage intermédiaire pour des langages plus abstraits, par example EXQIL [HGL+], le langage dédié au trading algorithmique développé chez EXQIM S.A.S.. Une approche encore plus haut niveau dont nous avons lancé l'étude est la génération automatique de code par laquelle les programmes parallèles sont générés directement à partir de descriptions UML spécifiques au domaine [KHL12].

CONTENTS

1 INTRODUCTION 1

2 SUPERCOMPUTER ARCHITECTURES 7

2.1 Early Parallel Systems . . . 8

2.1.1 Superscalar Processing . . . 9

2.1.2 Vector Processing . . . 10

2.2 Massive Processing . . . 12

2.2.1 Clustering . . . 12

2.2.2 Distributed Computing . . . 18

2.3 Hybrid Computing . . . 20

2.3.1 Multiprocessing . . . 20

2.3.2 Hybrid Clusters . . . 23

3 PARALLEL PROGRAMMING MODELS 29

3.1 Low-Level Parallel Models . . . 30

3.1.1 Shared-Memory Communication . . . 30

3.1.2 Message-Oriented Models . . . 35

3.1.3 SWAR Programming Models . . . 38

3.2 High-Level Parallel Models . . . 41

3.2.1 Concurrent Computation Models . . . 41

3.2.2 Bridging Models . . . 47

3.2.3 Other Parallel Models . . . 57

4 A NEW SIMPLE BRIDGING MODEL 65

4.1 Motivation . . . 66

4.2 The SGL Model . . . 70

4.2.1 The Abstract Machine . . . 71

4.2.2 Execution Model . . . 72

4.2.3 Cost Model . . . 74

4.3 Case study: Modelling Parallel Computers . . . 77

4.3.1 Modelling Multi-core Computers . . . 77

4.3.2 Modelling Hierarchical Clusters . . . 80

4.3.3 Modelling Heterogeneous Computers . . . 82

5 SGL PROGRAMMING 85

5.1 Programming Model of SGL . . . 86

5.1.1 Language Syntax . . . 86

5.1.2 Environments . . . 88
5.1.3 Operational Semantics . . . 88
5.2 Parallel Skeletons Implementation . . . 94
5.2.1 Implementing Basic Data-Parallel Skeletons . . . 95
5.2.2 Implementing Distributable Homomorphism . . . 104
5.2.3 Speedup and Efficiency . . . 110
6 GATHER-SCATTER FOR SIMPLIFIED COMMUNICATIONS 113
6.1 The GPS Theorem . . . 113
6.1.1 Gather-Scatter Communication . . . 114
6.1.2 The GPS Theorem . . . 115
6.2 Simplifying BSML's Put . . . 116
6.2.1 Dilemma in BSML: Proj vs. Put . . . 116
6.2.2 The GPS Function . . . 119
6.2.3 Experimentation in BSML . . . 120
7 CONCLUSION 127
7.1 Conclusion . . . 127
7.2 Future work . . . 129
A APPENDIX 131
A.1 Communication Throughput . . . 131

BIBLIOGRAPHY 133

1

INTRODUCTION

Parallel programming and data-parallel algorithms have been the main techniques supporting high-performance computing for many decades. Like all non-functional properties of software, the conversion of computing resources into scalable and predictable performance involves a delicate balance of abstraction and automation with semantic precision. During the last decade, more and more professions require a very high computing power. However, as in the algorithmic trading financial companies, people tend to work in pairs composed of a quantitative analyst, specialized in financial algorithms, and a software engineer in charge of coping with implementation and high-performance related work. Furthermore, solving a complex algorithm sometimes requires a high-performance software-hardware specialist for parallelizing the algorithm on a specified supercomputer. Moreover, migrating existing programs to a new hardware configuration or developing new specific-purpose algorithms on a parallel environment is never an easy work, neither for software developers nor for domain specialists.

From a programming point of view, paper [Adv09] gives a perspective on the collective work spanning for approximately 30 years. It shows how difficult it is to formalize the seemingly simple and fundamental property of "what value a read should return in a multi-threaded program". Safe languages must be given semantics that computer science graduates and developers can understand with a reasonable effort. The author of this survey believes that we need to rethink higher-level disciplines that make it much easier to write parallel programs and that can be enforced by the languages and systems.

As the above remark highlights, multi-threaded *semantics* is far too complex for realistic software development. Yet parallel execution is synonymous with multiple processes or multi-threading, without which there can be no parallel speed-up. So how should programmers avoid the complexity of multi-

threaded programming and yet expect scalable performance? Part of the answer comes from the observation that the vast majority of parallel algorithms are deterministic. Along this line of reasoning, researchers like M. Cole [Col89] and H. Kuchen [SK93, BK96] have developed the paradigm of *algorithmic skeletons* for deterministic and deadlock-free parallel programming. Skeletons are akin to design patterns for parallel execution. A large body of programming research literature supports the view that most if not all parallel application software should be based on families of algorithmic skeletons.

A deterministic and high-level parallel programming interface is indeed a major improvement over explicit message passing. But the diminished expressive power of skeletons is not only an advantage. Unlike sequential equivalents, skeletons are not libraries in the classical sense because their host language (e. g., C) is necessarily less expressive than the language in which they are written (e. g., C+MPI). This is due to the lack of a base language that is not just Turing-complete but complete for parallel algorithms, a notion that has not even been well defined yet. As a result there is no fixed notion of a set of skeleton *primitives* but instead the message-passing primitives used to implement them.

Meanwhile, a major conceptual step was taken by L. Valiant [Val90] who introduced his Bulk-Synchronous Parallel (BSP) model. Inspired by the complexity theory of PRAM model of parallel computers, Valiant proposed that parallel algorithms can be designed and measured by taking into account not only the classical balance between time and parallel space (hence the number of processors) but also communication and synchronization. The BSP performance model is both realistic and tractable so that researchers like McColl et al. [MW98] were able to define BSP versions of all important PRAM algorithms, implement them and verify their portable and scalable performances as predicted by the model. BSP is thus a *bridging model* relating parallel algorithms to hardware architectures.

From the mid-1990's, it became clear that BSP is the model of choice for implementing algorithmic skeletons: its view of the parallel system included explicit processes and added a small set of network performance parameters to allow predictable performance. G. Hains et al. designed BS-lambda [LHF00] as a minimal model of computation with BSP operations. BS-lambda became the basis for Bulk-Synchronous ML (BSML) [Lou00] a variant of CAML under development by Loulergue et al. since 2000. BSML has a simplified program-

ming interface of only four operations: `mkpar` to construct parallel vectors indexed by processors, `proj` to map them back to lists/arrays, `apply` to generate asynchronous parallel computation and `put` to generate communication and global synchronization from a two-dimensional processor-processor pairing. As a result, parallel performance mathematically follows from program semantics and the BSP parameters of the host architecture.

While BSML was evolving and practical experience with BSP algorithms was accumulating, one of its basic assumptions about parallel hardware was changing. The flat view of a parallel machine as a set of communicating sequential machines remains true but is more and more incomplete. Recent supercomputers like Blue Gene/L [ABB+03], Blue Gene/P [sD08], and Blue Gene/Q [HOF+12] feature multi-processors on one card, multi-core processors on one chip, multiple-rack clusters etc. The Cell/B.E. [KDH+05, JB07], Cell-based RoadRunner [BDH+08] and GPU's feature a CPU with Master-Worker architecture. Moreover, [KTJR05] observes that heterogeneous chip multiprocessors present unique opportunities for improving system throughput and reducing processor consumption. The trend towards green-computing[1] puts even more pressure on the optimal use of architectures that are not only highly scalable but hierarchical and non-homogeneous.

In this thesis, we describe work that attempts to improve the simplicity of parallel program development, the portability of parallel program code, and the precision of parallel algorithm performance prediction for heterogeneous environments. With these goals in mind we proposed a bridging model named SGL (formerly Scatter-Gather Language) for modeling heterogeneous parallel architectures and parallel algorithms, and an implementation of parallel skeletons based on SGL model for high-performance computing. SGL simplifies the parallel programming either on the classical parallel machines or on the novel hierarchical machines. It generalizes the BSML programming primitives. SGL can be used later with model-driven techniques for automatic code generation from specification sheet without any complex coding, for example processing Big Data on heterogeneous massively parallel systems. The SGL cost model improves the clarity of algorithms performance analysis. At the same time, it allows benchmarking machine performance and algorithm scalability.

1 "Green computing is the study and practice of designing, manufacturing, using, and disposing of computers, servers, and associated subsystems efficiently and effectively with minimal or no impact on the environment." – San Murugesan [Muro8]

The rest of this dissertation is organized as follow:

In chapter 2, we follow the history of supercomputing to review the different architectures of supercomputer in order to have some general notions of parallel machine architectures on which a parallel program may be executed. We start by the first supercomputer CDC 6600, then the Cray family which was optimized for vector processing. After that, we present several contemporary representative supercomputing clusters such as one of world's fastest supercomputers IBM Blue Gene/L, standard x86-based Linux-OS disk-less SGI Altix ICE, inexpensive everyone-can-build Beowulf cluster and Volunteer Grids, etc. At the end of this chapter, we review several multi-core architectures and the up-to-date hybrid clusters such as Cell-accelerated first PetaFLOPS in the world IBM RoadRunner, and GPGPU-based 2010 world fastest Tianhe-1A supercomputer.

In chapter 3, we review the state of the art of parallel programming models in order to understand the portability of code, the scalability of algorithms, practicality of machine modelling, simplicity of programming, and feasibility of optimisation analysis between different models. This chapter is organised in two sections: the first one presents low-level parallel models for shared-memory multi-threaded programming, shared-nothing message-passing programming, and SWAR programming; the second one presents high-level portable models such as process calculi, bridging models, and MapReduce etc. We emphasize, in this chapter, the BSP model which enforces a strict separation of communication and computation, removes non-determinism and guarantees the absence of deadlocks, gives an accurate model of performance prediction, provides much simplicity and efficiency for parallel programming.

In chapter 4, we introduce our Scatter-Gather parallel-programming and parallel execution model in the form of a simple imperative Scatter-Gather Language (SGL). Its design is based on past experience with Bulk Synchronous Parallel (BSP) programming and BSP language design. SGL's novel features are motivated by the last decade move towards multi-level and heterogeneous parallel architectures involving multi-core processors, graphics accelerators and hierarchical routing networks in the largest multiprocessing systems. The design of SGL is coherent with L. Valiant's multi-BSP while offering a programming interface that is even simpler than the primitives of BSML. We also attempted to analyse, in this chapter, how to link the SGL abstract machine to

several typical parallel computers – from small multi-core microcomputer to large supercomputer clusters and grids.

We propose in chapter 5 the programming model of SGL and several skeletons implementation to attempt to motivate and support the view that BSP's advantages for parallel software can be enhanced by the recursive hierarchical and heterogeneous machine structure of SGL, while simplifying the programming interface even further by replacing point-to-point messages with logically centralised communications. Our initial experiments with language definition, programming and performance measurement show that SGL combines clean semantics, simplified programming of BSP-like algorithms and dependable performance measurement. However, SGL does not express "horizontal" communication patterns, so it overly favours simplicity at the expense of expressive power.

Chapter 6 is our first attempt to demonstrate that this defect only affects a minority of algorithms and can be compensated by automated compilation/interpretation. We thus introduce, in this chapter, the GPS theorem which can be implemented later in a compiler to optimize the SGL's "horizontal" all-to-all communication. We then propose a simplified version of BSML's *put* based on GPS and implement the parallel sample-sort algorithm with it. The comparison of BSML's *put* and SGL's *GPS* shows that *GPS* has a better code readability and lower execution time.

Finally, we conclude our work in chapter 7 and present several perspectives as future work.

2

SUPERCOMPUTER ARCHITECTURES

2.1 Early Parallel Systems . . . 8
2.1.1 Superscalar Processing . . . 9
2.1.1.1 CDC 6600 . . . 9
2.1.2 Vector Processing . . . 10
2.1.2.1 CDC STAR-100 . . . 10
2.1.2.2 Cray-1 . . . 11
2.1.2.3 Successors of Cray-1 . . . 11
2.2 Massive Processing . . . 12
2.2.1 Clustering . . . 12
2.2.1.1 Hitachi SR2201 . . . 13
2.2.1.2 Blue Gene/L . . . 13
2.2.1.3 Altix ICE . . . 15
2.2.1.4 Beowulf . . . 17
2.2.2 Distributed Computing . . . 18
2.2.2.1 Grid Computing . . . 18
2.2.2.2 Volunteer Computing . . . 19
2.3 Hybrid Computing . . . 20
2.3.1 Multiprocessing . . . 20
2.3.1.1 Multi-core Processors . . . 21
2.3.1.2 Coprocessors / Accelerators . . . 22
2.3.2 Hybrid Clusters . . . 23
2.3.2.1 RoadRunner . . . 24
2.3.2.2 Tianhe-1A . . . 26

A computer hardware is useless without computer program; a computer program cannot be executed without computer hardware. Programs are developed on a well-defined computer architecture for performing a specified task. Most of contemporary sequential computers were designed based on the Von Neumann architecture. However, we have no such standard for parallel computers. There are a wide variety of parallel computer architectures. Designing

a parallel algorithm is already much harder than designing a sequential one. How to develop a portable parallel program is still a big challenge today.

Before talking about the parallel codes, in this chapter, we follow the history of supercomputing to review the different architectures of supercomputer in order to have some general notions of parallel machine architectures on which a parallel program may be executed. We start by the first supercomputer CDC 6600, then the Cray family optimized for vector processing in Section 2.1. After that, several contemporary representative supercomputing clusters such as one of world's fastest supercomputers IBM Blue Gene/L, standard X86-based Linux-OS disk-less SGI Altix ICE, inexpensive everyone-can-build Beowulf cluster and Volunteer Grids, etc. are presented in Section 2.2. Finally, we review in Section 2.3 several multi-core architectures and the up-to-date hybrid clusters such as Cell-accelerated first Peta-FLOPS in the world IBM RoadRunner, and GPGPU-based 2010 world fastest Tianhe-1A supercomputer.

We can see in this chapter that with the passage of time, the parallel computer architectures evolve from vector processor to multi processor; from shared-memory to distributed-memory; from flat parallel to nested parallel; from symmetric unit to hybrid unit; from homogeneous to heterogeneous more and more. All these evolutions lead us to propose a hierarchical bridging model (Chapter 4) for simplifying parallel programming.

2.1 EARLY PARALLEL SYSTEMS

The history of high performance computing goes back at least to the 1960s, when the first supercomputer was created for high-energy nuclear physics research. We review in this section CDC 6600 and Cray computers, the well-known first supercomputers in the world, to see how computers became parallel where they were only sequential. The main techniques used by supercomputer designers at that time are superscalar processing and vector processing with special hardware to execute multiple instructions or process multiple data during a clock cycle. All programs at that time were developed in a low-level language, none was portable.

2.1.1 *Superscalar Processing*

A scalar processor manipulates at a time one or two data with one instruction. The technique allowing multiple instructions to be worked on at the same time is known today as **superscalar**. Supercomputer designers invented in the beginning such technique to obtain parallelism. Superscalar CPU design emphasizes improving the instruction dispatcher accuracy, and allowing it to keep the multiple functional units in use at all times.

2.1.1.1 *CDC 6600*

The first supercomputer CDC 6600 [Tho63, Tho70] was built by *Control Data Corporation* (CDC) and delivered in 1964 to the *Lawrence Radiation Laboratory*. In contrast to a typical machine in the era that used a single CPU to drive the entire system, and CPUs generally ran slower than the main memory they were attached to at the time. The 6600 CPUs, took another approach, handled only arithmetic and logic, in order to reduce the size of CPU to obtain a higher clock speed. The instructions for memory access, input/output (I/O), and other "housekeeping" tasks were implemented separately to allow the central processor (CP), peripheral processors (PPs) and I/O to operate in parallel.

The major elements of the Control Data 6600 is shown in Figure 1. The 6600 CP had 8 general purpose 60-bit registers, 8 18-bit address registers, and 8 18-bit scratchpad registers; but it had no instructions for input and output. The CP included 10 parallel functional units, allowing multiple instructions to be worked on at the same time. The "housekeeping" tasks were accomplished through 10 PPs. One of the PPs was in overall control of the machine, including control of the program running on the main CPU, while the others were dedicated to various I/O tasks. Each PP included its own memory of 4096 12-bit words. This memory served for both for I/O buffering and program storage. The 6600 thus gained speed by "farming out" work to PPs freeing the CP to process actual data.

Software developers should master machine-specific assembly language [Gri74] and handle the whole instruction set of this complex supercomputer. The algorithms should be re-designed according to the hardware architecture, none was re-usable. Furthermore, debugging was even more difficult.

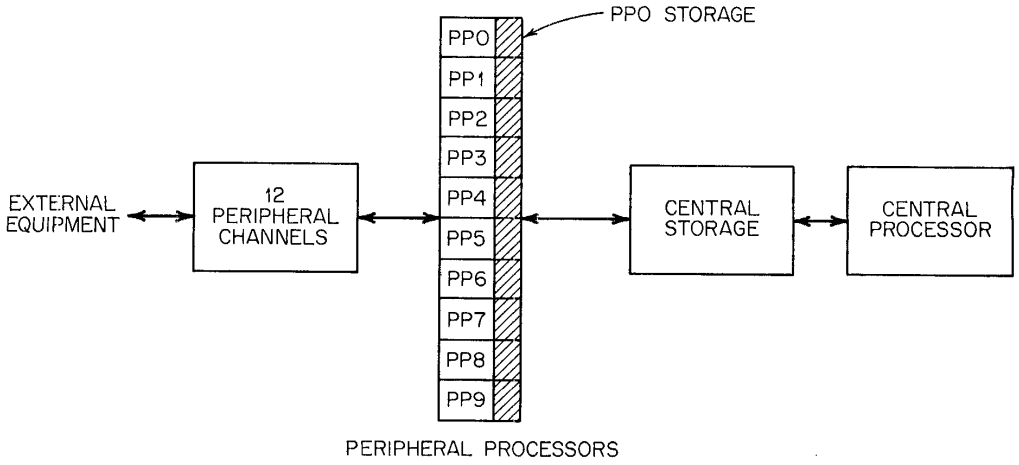

Figure 1: Major elements of Control Data 6600 [Tho70]

2.1.2 *Vector Processing*

Vector processing allows processing instruction set containing instructions that operate on one-dimensional arrays of data called vectors. This technique removes the overhead of superscalar. Vector processors can greatly improve performance on certain workloads, notably numerical simulation and similar tasks. The most successful vector supercomputers were the Cray computers whose designers had learned from CDC STAR-100 failure experience.

2.1.2.1 *CDC STAR-100*

The Control Data Corporation STAR-100 [Pur74] was one of the first supercomputers using vector processing technique to improve computing performance. The name STAR is constructed of the words ***ST****rings* and ***AR****rays*. In contrast to a scalar processor, whose instructions operate on single data items, the STAR-100 CPU implemented an instruction set containing instructions that operate on one-dimensional arrays of data called *vectors* to set up additional hardware that fed in data from the main memory as quickly as possible [Kog81]. However, very few programs can be effectively vectorized into a series of single instructions. Any time that the program had to run scalar instructions, the overall performance of the machine dropped dramatically.

2.1.2.2 *Cray-1*

Former CDC 6600's designer and Cray-1's architect, Seymour Cray, learned from the failure experience of STAR-100, founded *Cray Reseaarch, Inc.* and delivered the 80 MHz Cray-1 supercomputer [Inc77] in 1976 for *Los Alamos National Laboratory*. Cray-1 is one of the best known and most successful supercomputers in history. It used the **chaining** technique in which scalar and vector registers generate interim results which can be used immediately, without additional memory references which reduce computational speed [HJS99].

The Cray-1 had 12 pipelined functional units. An add unit and a multiply unit performed the 24-bit address arithmetic. The scalar portion of the system consisted of an add unit, a logical unit, a population count, a leading zero count unit and a shift unit. The vector portion consisted of add, logical and shift units. The floating point functional units were shared between the scalar and vector portions, and these consisted of add, multiply and reciprocal approximation units. The CRAY-1 machine was the first Cray design to use Integrated Circuits (ICs). The high-performance ICs generated considerable heat. The Cray-1 computer needed a liquid Freon refrigeration system as cooling system.

Cray released in 1978 the first standard software package for the Cray-1. Software engineers of Cray-1 could hence have Cray Operating System (COS), Cray Assembly Language (CAL) and Cray FORTRAN (CFT) – the first automatically vectorizing FORTRAN compiler. The development became easier than before but a good comprehension of the physical architecture was still needed, and the algorithms should be designed for this specific machine.

2.1.2.3 *Successors of Cray-1*

After Cray-1, the Cray X-MP supercomputer was delivered by Cray Research, Inc. in 1982. It was a 105 MHz shared-memory parallel vector processor with better chaining support and multiple memory pipelines. All three floating point pipelines on the XMP could operate simultaneously [THS03].

The Cray-2 supercomputer delivered by Cray Research, Inc. in 1985 was the fastest machine in the world with 1.9 GFLOPS peak performance when it was released. It attempted to be designed with more functional units to give the

system higher parallelism, tighter packaging to decrease signal delays, and faster components to allow a higher clock speed. Cray-2 had four vector processors built with tight-packed Emitter-Coupled Logic (ECL). For resolving the heat problem that had not been met before, it was totally immersed in a tank of Fluorinert, which bubbled as it operated [Mur97].

Cray Research, Inc. developed subsequently the Cray Y-MP, Cray-3, Cray C90, Cray T90, Cray T3D and Cray T3E models and others. Along with the release of Cray-3 supercomputer, C compilers, Network File System (NFS) and TCP/IP stack etc. were now supported [Cra91]. Programmers could use high-level programming languages and standard communication protocols that facilitate much the development. However, the algorithm design depended still on the physical machine.

2.2 MASSIVE PROCESSING

The early parallel systems had fewer than 10 processors. The computer clustering approach allows connection of a number of readily available computing nodes in order to accumulate processors' computational power. Supercomputers with thousands of processors began to appear in the 1990s. We review in this section several representative centralized computer clusters such as one of earliest distributed memory parallel supercomputers Hitachi SR2201, one of the world's fastest parallel supercomputers Blue Gene/L, standard X86-based Linux-OS disk-less SGI Altix ICE, and inexpensive everyone-can-build Beowulf cluster; then distributed systems such as Grids and volunteer computing systems etc.

2.2.1 *Clustering*

To reduce the communication cost, today's supercomputer components are almost all built as close as possible to one another. A massive parallel distributed-memory centralized cluster provides considerable scalability, performance, and resiliency. We review here four most representative supercomputer clusters: Hitachi SR2201, IBM Blue Gene/L, SGI Altix ICE, and NASA Beowulf.

2.2.1.1 *Hitachi SR2201*

HITACHI SR2201[FYA+97], introduced by *Hitachi* in 1996, was one of earliest distributed memory parallel supercomputers. The cache miss penalty in SR2201 was solved by 150 MHz HARP-1E processor' Pseudo Vector Processing (PVP): data was loaded by pre-fetching to a special register bank, bypassing the cache. The SR2201 could have up to 2048 processing elements (PEs), connected via a high-speed three dimensional crossbar network (Figure 2), which was able to transfer data at 300 MB/s over each link. All these gave HITACHI SR2201 a peak performance of 600 GFLOPS.

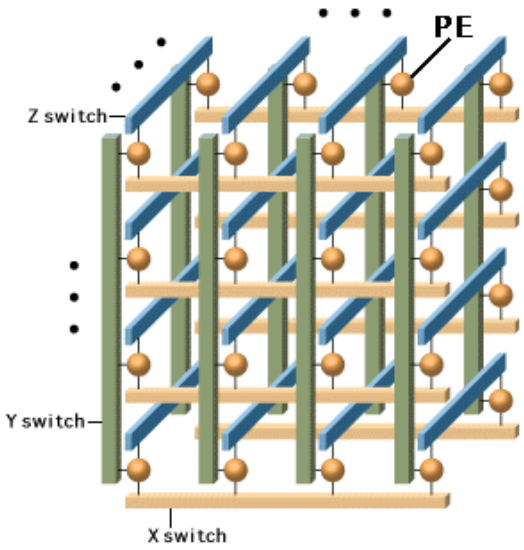

Figure 2: HITACHI SR2201 three-dimensional crossbar switch network [Hit97]

Many high-level programming languages and parallel computing interfaces, such as FORTRAN77, FORTRAN90, C, C++, Parallel FORTRAN, Parallel Virtual Machine (PVM) and MPI (c. f., Section 3.1.2.1) etc. were supported by SR2201 [Hit97]. Express™ parallel development support tool facilitated programming and performance tuning. Nevertheless, the network topology of SR2201 played an important role for the performance.

2.2.1.2 *Blue Gene/L*

Blue Gene is an IBM project aimed at designing supercomputers that can reach operating speeds in the PFLOPS range, with low power consumption. Blue Gene/L [AAA+01, ABB+03] was the first generation of supercomputers

issued from the project. The Blue Gene/L system installed at LLNL[1] that achieved 596 TFLOPS peak performance was organised as follows (Figure 3):

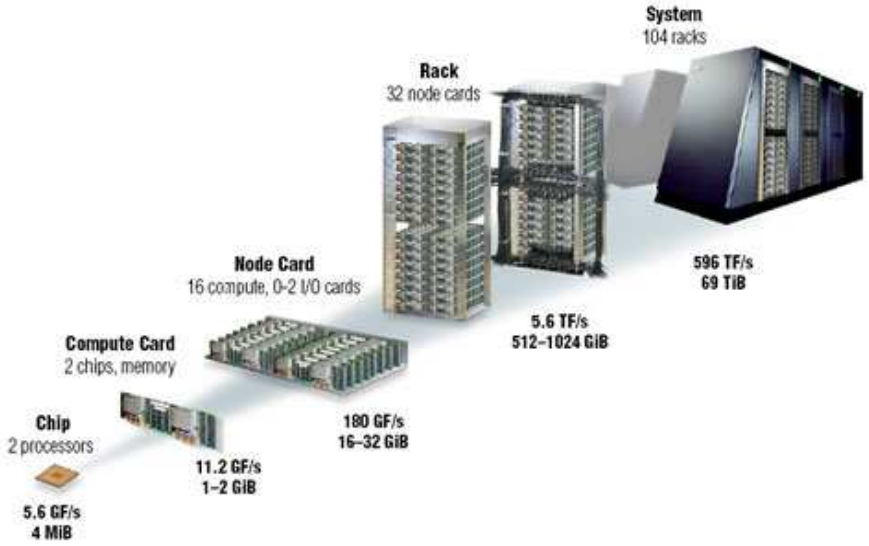

Figure 3: Hierarchy of Blue Gene processing units [LLN11]

Each Blue Gene/L Compute or I/O node was a single Application-Specific Integrated Circuit (ASIC) with associated Dynamic Random-Access Memory (DRAM) chips. The ASIC integrated two 700 MHz PowerPC 440 embedded processors, each with a double-pipeline-double-precision Floating Point Unit (FPU), a cache sub-system with built-in DRAM controller and the logic to support multiple communication sub-systems.

Blue Gene/L Compute nodes were packaged two per compute card, with 16 compute cards plus up to 2 I/O nodes per node board. There were 32 node boards per rack. By the integration of all essential sub-systems on a single chip, and the use of low-power logic, a Blue Gene/L standard 19-inch rack could package up to 1024 compute nodes plus additional I/O nodes, within reasonable limits of electrical power supply and air cooling.

Each Blue Gene/L node was attached to three parallel communication networks [LLN11] (Figure 4): a three-dimensional (3D) toroidal network for peer-to-peer communication between compute nodes, a collective network for collective communication (e. g., broadcasts or reduce), and a global interrupt network for fast barriers. The I/O nodes provided communication to storage and external hosts via an Ethernet network. The I/O nodes handled filesystem operations on behalf of the compute nodes. Finally, a separate and private

1 Lawrence Livermore National Laboratory.

Ethernet network provided access to any node for configuration, booting and diagnostics.

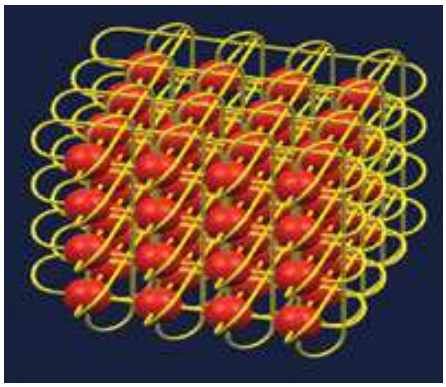

Figure 4: BlueGene/L three-dimensional torus network [LLN11]

Blue Gene/L compute nodes used lightweight operating systems for minimum system overhead. However, only a subset of POSIX[2] calls was supported, and only one process could run at a time on nodes in coprocessor mode. Developers could use MPI to manage the inter-node communication. Apart from the high-level programming languages such as C, C++ and Fortran, programmers could also use some scripting languages such as Ruby or Python on the compute nodes.

IBM has developed subsequently new Blue Gene generations named Blue Gene/P [sD08, NHBY09] and Blue Gene/Q [HOF+12].

2.2.1.3 *Altix ICE*

The Altix ICE supercomputer delivered by Silicon Graphics (SGI) also used a Hierarchical Management Framework (HMF) for scalability and resiliency. But unlike IBM Blue Gene/L, SGI Altix ICE is an Intel Xeon-based system featuring diskless compute blades using standard SUSE Linux Enterprise Server (SLES) or Red Hat Enterprise Linux (RHEL) distributions. One of the largest SGI Altix ICE 8200EX systems NASA's *Pleiades* with 111 104 cores for 1315.3 TFLOPS as peak performance at the *Ames Research Center* was the 7th faster supercomputer in the world in 2011 according to Top500[3].

2 Portable Operating System Interface, a family of standards specified by the IEEE.

3 Top500 project ranks and details the 500 most powerful computer systems in the world. http://www.top500.org/

A typical Altix ICE system (Figure 5) is built with one or more Altix ICE racks. Each rack can control up to 4 *Individual Rack Units* (IRUs) with a Leader node that hold a single system image for all compute blades of the rack. An Altix ICE 8200EX IRU contains 16 compute blades interconnected by four 4x DDR InfiniBand (IB) switch blades. A compute blade is configured with 2 multi-core *Intel* Xeon processors.

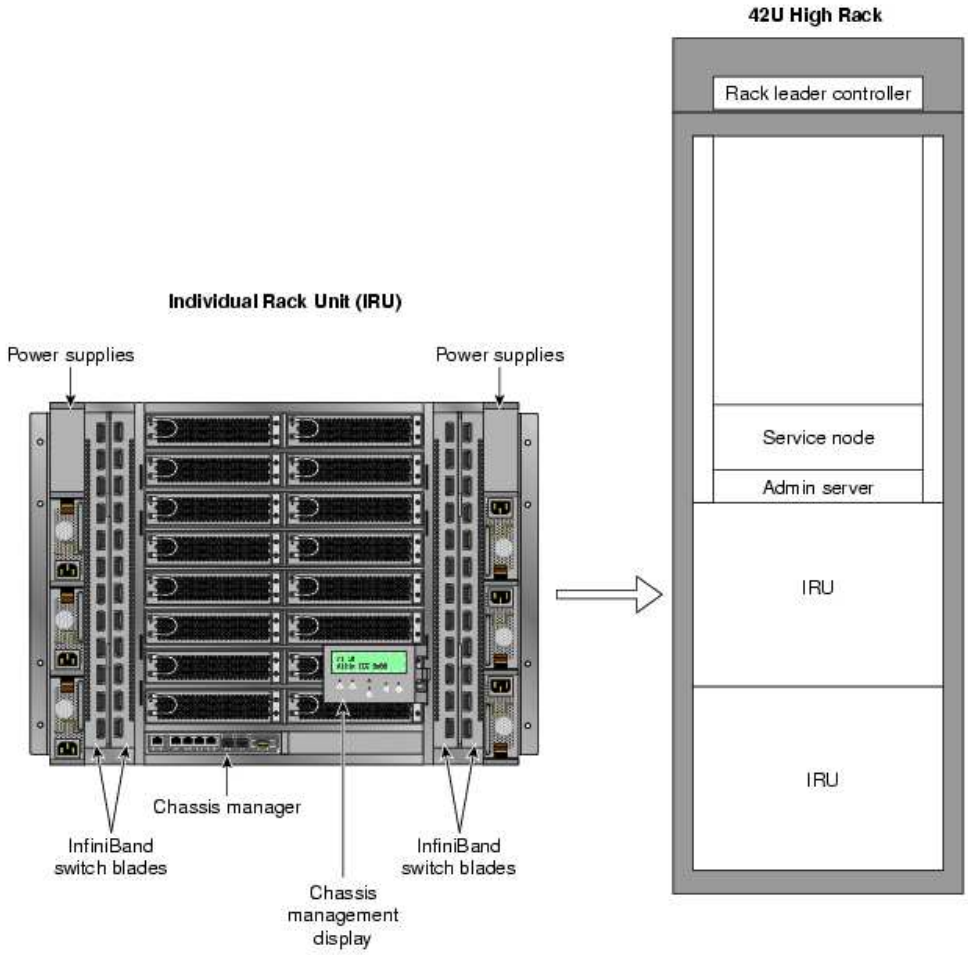

Figure 5: SGI Altix ICE 8200 IRU and Rack Components Example [Lib10]

Standard microprocessor architecture (X86-64), standard communication interfaces (IP-over-InfiniBand protocol, Message Passing Toolkit), standard operation system (SLES or RHEL), all these features of Altix ICE reduce the programming difficulty though the network topology affects always algorithms performance.

The Altix ICE 8200EX supercomputer used by *Exclusive Quantitative Investment Management (EXQIM) S.A.S.* has 2 IRUs in a 42U-high rack, 1 Service

node managing a InfiniteStorage system shared by the 2 IRUs, and 1 Admin node administrating the entire system (Figure 6).

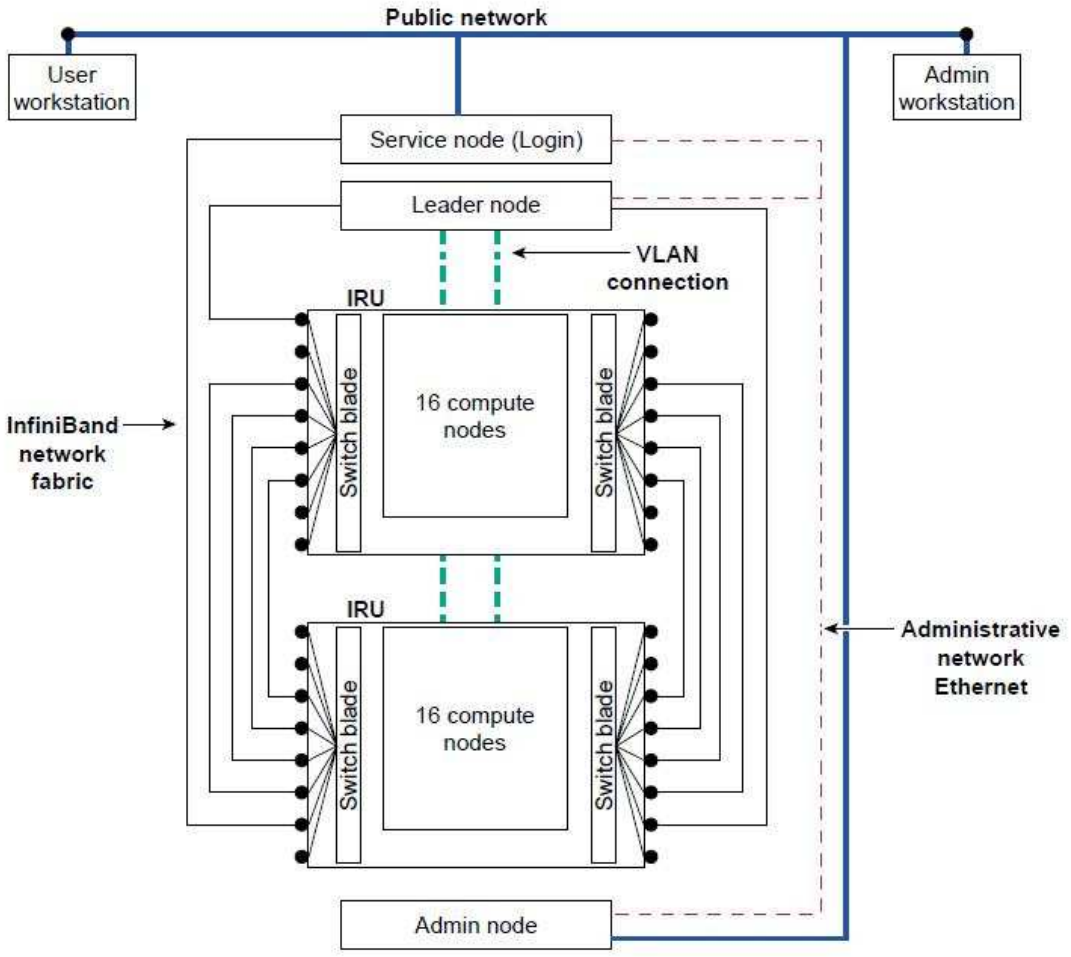

Figure 6: Overview of Altix ICE 8200EX Supercomputer at EXQIM

Each of EXQIM's Altix ICE compute blade is configured with two Intel Xeon E5440 Quad-core 2.83 GHz processors and 32 GByte DDR2 memory running SLES Linux distribution. All compute blades interconnect in a **fat-tree** topology. More details about the processors can be found in Section 2.3.1.1.

2.2.1.4 *Beowulf*

Beowulf [Ste01, Sit99] is a multi-computer architecture which can be used for parallel computations. The first Beowulf cluster was built in 1994 by Thomas Sterling and Donald Becker at NASA.

A Beowulf cluster is a kind of high-performance massively parallel computer built primarily out of commodity hardware components, running a free-software operating system such as Linux or FreeBSD, interconnected by a private high-speed network. It consists of a cluster of PCs or workstations dedicated to running high-performance computing tasks. The nodes in the cluster don't sit on people's desks; they are dedicated to running cluster jobs. It is usually connected to the outside world through only a single node.

The performance prediction on Beowulf for algorithm design and software development is very important because the hardware is neither optimized nor easily configurable.

2.2.2 *Distributed Computing*

Scientists sometimes integrate geographically distributed computer resources to obtain more computational power. We review briefly here Grid computing and Volunteer Computing to understand how computer resources may be federated into one system to create a "virtual supercomputer".

2.2.2.1 *Grid Computing*

A computational grid [FK99, FKT01] is a hardware and software infrastructure coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.

Grids are a form of distributed computing whereby a "super virtual computer" is composed of many networked loosely coupled computers acting together to perform large tasks. A grid is more loosely coupled, heterogeneous, and geographically dispersed compared to a cluster supercomputer.

There are many projects for building grid infrastructures such as: the Austrian Grid[4], D-Grid[5], BEgrid[6], EGEE[7], Grid'5000[8] and EGI[9] etc. The French one is Grid'5000[BCC+06]. It currently connects resources distributed on 9 sites in France. Each local site platform is formed of at least one cluster. The total number of processors of all the sites is 2218 (7896 cores) in 1171 nodes [Gri12]. The sites are connected by the RENATER Education and Research Network. All clusters are connected to Renater with at least 1Gb/s link. The processors of each cluster are connected via Myrinet or Infiniband.

Since grid hardware is shared, reserving right resources for a specific algorithm is essential. A bridging model (c. f., Section 3.2.2) which provides machine-algorithm cost model can greatly help us in resolving this issue.

2.2.2.2 *Volunteer Computing*

Volunteer Computing (sometimes called *quasi-opportunistic supercomputing*) is a computational paradigm for supercomputing on a large number of geographically disperse computers [KCD+08]. The computational power is donated by different computer owners to achieve high performance computing. Thus, the system is fully heterogeneous, and the hosts can arrive or leave randomly. Like Beowulf we had seen in Section 2.2.1.4, the nodes of a volunteer computing system are mainly commodity computers, often much more heterogeneous than Beowulf. Some were built with multi-core CPUs and/or GPGPU [KDH11]. The architectures of multi-core CPUs and GPGPU will be reviewed in Section 2.3.1.

The first volunteer computing project was GIMPS[10]. After that there are many volunteer computing projects such as: SETI@home[11], Folding@home[12],

4 The Austrian Grid. http://www.austriangrid.at/
5 D-Grid: The German Grid Initiative. http://www.d-grid.de/
6 BEgrid: The Belgian Grid for Research. http://www.begrid.be/
7 EGEE: Enabling Grids for E-sciencE. http://www.eu-egee.org/
8 Grid'5000. https://www.grid5000.fr/
9 EGI: The European Grid Initiative. http://web.eu-egi.eu/
10 The Great Internet Mersenne Prime Search. http://www.mersenne.org/
11 Search for ExtraTerrestrial Intelligence at Home. http://setiathome.berkeley.edu/
12 Folding@home disease research project. http://folding.stanford.edu/

MilkyWay@Home[13], Einstein@Home[14], and BOINC[15]. Between June 2007 and June 2011, Folding@home exceeded the performance of Top500's fastest supercomputer [Gro12, Ter07].

How to maintain the correctness of parallel programs and keep a good performance in this dynamic environment is still a big challenge.

2.3 HYBRID COMPUTING

Increasing the number of processors is not the only solution to obtain higher computational power. Adding processors requires more space or smaller components. Bigger space implies longer distance, the performance will be decreased by the overhead inter-processor communication cost, and smaller components produce a lot of heat.

For acquiring high performance with limited energy consummation and heat generated by the semiconductors, different special computational units are employed, such as: digital signal processor (DSP), graphics processing unit (GPU), application-specific integrated circuit (ASIC), field-programmable gate array (FPGA), or coprocessor etc. Hybrid computing refers to the combination of these special computational units with standard units in order to perform high-performance computing.

In this section, we firstly present multi-core processors and coprocessors. After that we review two notable hybrid supercomputers: RoadRunner and Tianhe-1A.

2.3.1 *Multiprocessing*

Multiprocessing is the use of two or more CPUs within a single computer system. CPUs can be implemented in a single processor or in a coprocessor.

13 MilkyWay@home astrophysics project. http://milkyway.cs.rpi.edu/
14 Einstein@Home project. http://einstein.phys.uwm.edu/
15 Berkeley Open Infrastructure for Network Computing. http://boinc.berkeley.edu/

2.3.1.1 *Multi-core Processors*

The number of central processing units (CPU) can be increased not only by replicating the processor, but also can be implemented inside a processor. A multi-core processor is an integrated circuit (IC) with two or more independent actual CPU (cores). Cores may or may not share caches according to the architecture, and the inter-core communication may be implemented either through message passing or shared memory. The single core processors reached the physical limits of possible complexity and speed, silicon industries such as Intel and AMD propose thus dual-core, quad-core, hexa-core, octo-core, deca-core, even 48-core X86-architecture processors.

For example, the processors in *EXQIM*'s Altix ICE supercomputer's compute blades (c. f., Section 2.2.1.3) are quad-core Harpertown-based Xeon. As we can see in Figure 7, they use unified L2 caches. Two cores share one L2 cache with 6 MByte. The core pairs of one compute blade that share an L2 cache are 0+2, 1+3, 4+6, and 5+7. The eight cores from two processors of one compute blade share 32 GByte DDR2 memory, and they communicate with each other via a 1333 MHz Front-Side Bus (FSB).

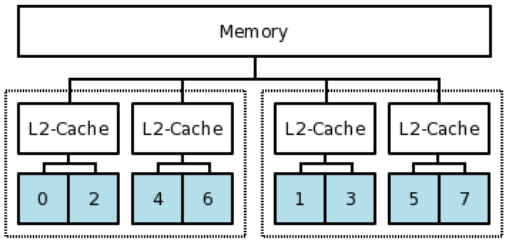

Figure 7: Intel Xeon Harpertown CPUs in a single ICE 8200EX [Nor11]

The newest *Intel* processors (e. g., Nehalem-based Xeon) use a point-to-point technology QuickPath Interconnect (QPI) instead of FSB to increase the inter-core communication performance.

Multi-core processors can also be heterogeneous. STI (Sony, Toshiba, and IBM) alliance's Cell processor [KDH+05] (Figure 8), for example, is a multi-core chip composed of one *Power Processor Element* (PPE), and multiple *Synergistic Processing Elements* (SPE). The PPE and SPEs are linked together by an internal high speed bus dubbed *Element Interconnect Bus* (EIB) [CRDI07, JB07].

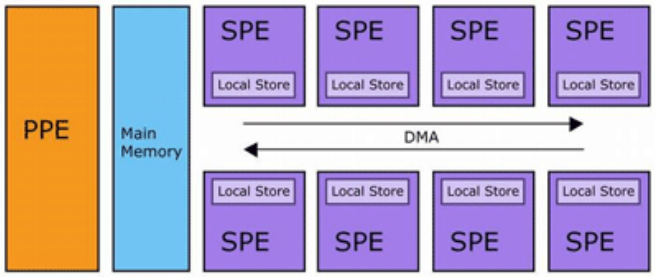

Figure 8: STI Cell B/E processor abstract overview [SPI]

Some low-level programming models are presented in Section 3.1.1 for shared-memory multi-core architectures.

2.3.1.2 *Coprocessors / Accelerators*

A coprocessor is a computer processor used to supplement the functions of the primary processor. As in a STI Cell B/E architecture (c. f., Section 2.3.1.1), the SPEs can be considered as the coprocessors and the PPE as the main processor.

A coprocessor may also be an accelerator such as: graphics processing unit (GPU), crypto accelerator, digital signal processor, etc. The most commonly used coprocessors for high-performance computing nowadays are general-purpose GPU (GPGPU). Instead of optimising cores' speed and data access latency, GPGPUs use hundreds or even thousands of cores to increase computational throughput. Figure 9 shows a 512-core Fermi architecture.

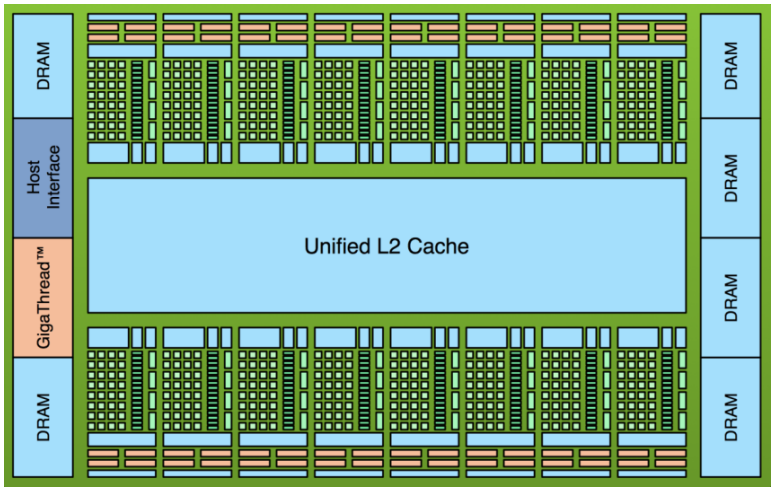

Figure 9: Nvidia Fermi Architecture [Co.09]

Fermi [Gla09] is the current generation CUDA (formerly Compute Unified Device Architecture) architecture introduced by *Nvidia Corporation*. Each core has its own L1 cache, all 512 cores share a unified L2 cache. Fermi has up to 1 TByte GPU memory [Co.09]. As shown in Figure 10, copying data from main memory to GPU memory is demanded before processing parallel computation; and copying result from GPU memory to main memory is also necessary after the computation [NBGS08].

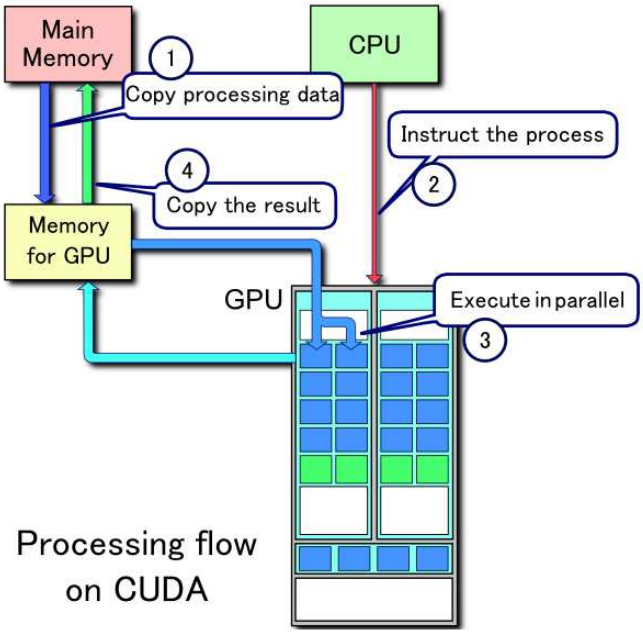

Figure 10: CUDA processing flow [Wik08a]

AMD-ATi and other silicon manufacturers propose a wide variety of GPGPU implementations for high-performance computing. A specific language such as OpenCL or OpenACC is needed for programming on GPGPU.

Intel announced, in 2012, its many-core coprocessor Xeon Phi. Even though its architecture is X86-based, software developers still need specific language and compiler such as OpenMP, Intel Threading Building Blocks (TBB), or Intel Cilk Plus (c. f., Section 3.2.3.2) to create threads in a program [Int12].

2.3.2 *Hybrid Clusters*

Hybrid computing Clusters combine the special computational units (e. g., GPGPU, coprocessor, etc.) with standard units (e. g., CPU) in order to perform

high-performance computing. We review here 2 notable hybrid supercomputers: RoadRunner and Tianhe-1A.

2.3.2.1 *RoadRunner*

RoadRunner [BDH+08] built by *IBM* at the *Los Alamos National Laboratory* was the world's first Top500 LINPACK[16] sustained 1.0 PFLOPS system. It was also the first hybrid supercomputer in the world.

Like *IBM* Blue Gene (c. f., Section 2.2.1.2) and *SGI* Altix ICE (c. f., Section 2.2.1.3), RoadRunner cluster architecture is also tiered (Figure 11):

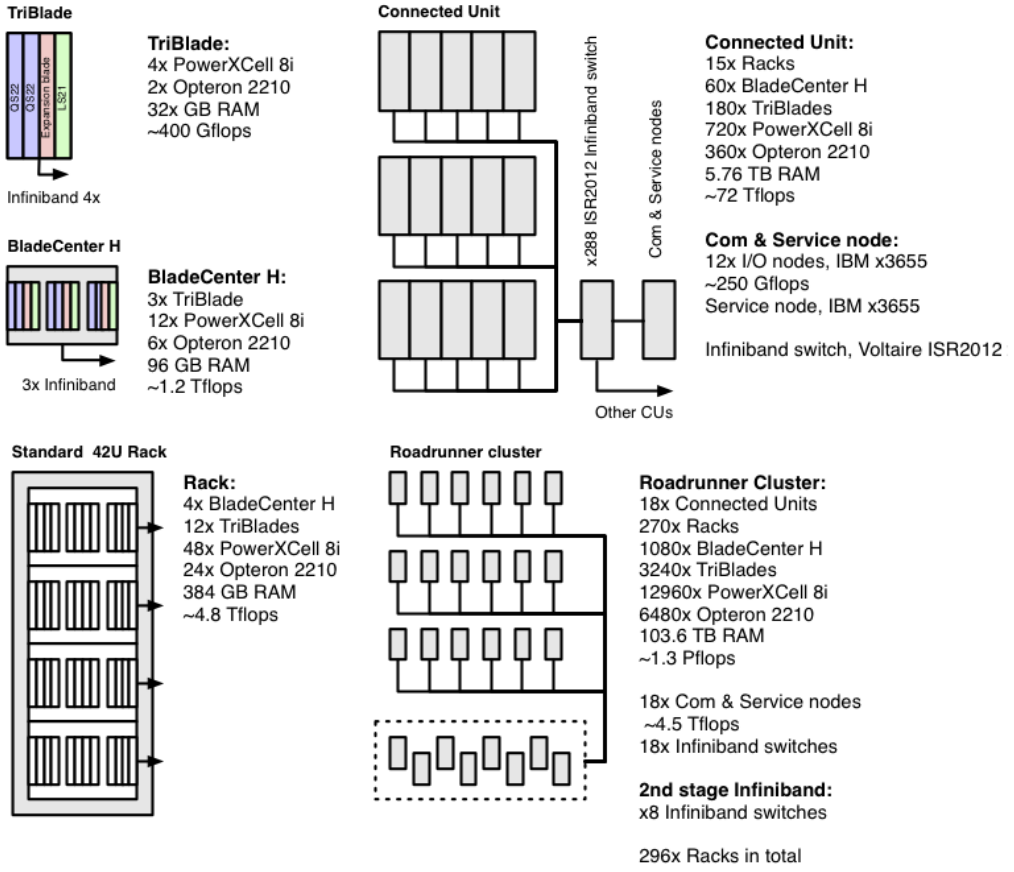

Figure 11: Overview of Roadrunner tiered composition [Wiko8b]

- The **cluster** is made up of 18 connected units (CUs), which are connected via 8 additional (second-stage) Infiniband ISR2012 switches. Each CU

16 A numerical linear algebra software library. http://www.netlib.org/linpack/

is connected through 12 uplinks for each second-stage switch, which makes a total of 96 uplink connections.

- A **CU** is 60 BladeCenter H full of TriBlades, that is 180 TriBlades. All TriBlades are connected to a 288-port Voltaire ISR2012 Infiniband switch. Each CU also has access to the Panasas file system through 12 System x3755 servers.

- A **TriBlade** (Figure 12) consists of one LS21 Opteron blade, an expansion blade, and two QS22 Cell blades. The LS21 has two 1.8 GHz dual-core Opterons with 16 GByte memory. Each QS22 has two PowerXCell 8i CPUs, running at 3.2 GHz and 8 GB memory. The expansion blade connects the two QS22 via four PCIe x8 links to the LS21, two links for each QS22. It also provides outside connectivity via an Infiniband 4x DDR adapter. Three TriBlades fit into one BladeCenter H chassis.

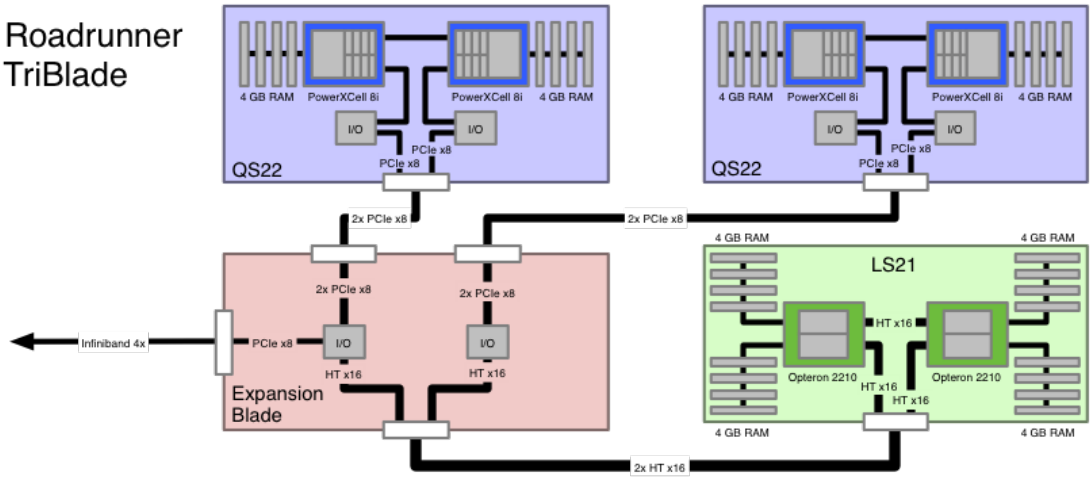

Figure 12: RoadRunner TriBlade [Wiko8b]

- The Opteron 2210 **processor** on LS21 created by *AMD* is a dual-core processor using for feeding the Cells with useful data and passing data between computing nodes. The *IBM* PowerXCell 8i featuring 1 PPE and 8 SPEs plays a coprocessor role here in charge of the heaviest computation. More information about the processors can be found in Section (2.3.1).

Either algorithm design or program development on RoadRunner requires a good understanding of different processor architectures and network topology for this extremely heterogeneous hierarchical system.

2.3.2.2 *Tianhe-1A*

The Tianhe-1A (天河一号) supercomputer developed by the *Chinese National University of Defense Technology* (NUDT) located at the *National Supercomputing Center in Tianjin* was the fastest computer in the world from October 2010 to June 2011 and is one of the few Petascale supercomputers in the world [Top].

Tianhe-1A [YLL+11] (Figure 13) is a hybrid supercomputer equipped with 14 336 Xeon X5670 processors and 7 168 Nvidia Tesla M2050 GPGPUs: The system is composed of 112 computer racks, 12 storage racks, 6 communications racks, and 8 I/O racks. Each computer rack is composed of 4 frames, with each frame containing 8 blades, plus a 16-port switching board. Each blade is composed of 2 computer nodes, with each computer node containing 2 Intel Xeon X5670 6-core processors and 1 Nvidia M2050 GPU processor.

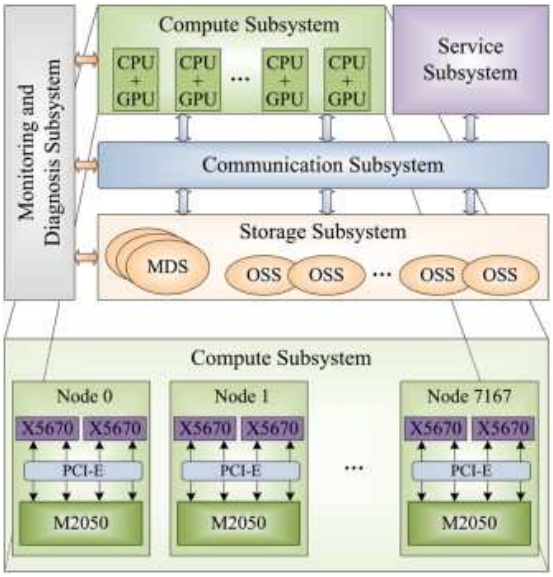

Figure 13: Tianhe-1 architecture [YLL+11]

Parallel computer hardware continues its evolution. The flat view of a parallel machine as a set of communicating sequential machines remains a useful and practical model but is more and more incomplete. Moreover, we can observe that heterogeneous chip multiprocessors present unique opportunities for improving system throughput and reducing processor consumption. The trend towards green-computing puts even more pressure on the optimal use of architectures that are not only highly scalable but hierarchical and non-homogeneous. All these changes make parallel programming harder than be-

fore. Thus, a novel programming and execution model is desirable for handling these heterogeneous hierarchical machines.

In the following chapters, we will introduce our SGL model which is based on scatter-gather primitives, together with a parallel execution primitive pardo. It is an attempt to continue the trend for simplification in the set of primitives of BSP-like programming and yet provides a better match with modern architectures that are both heterogeneous and hierarchically structured. For example, the Tianhe-1A's architure is neither flat nor completely homogeneous but can be cleanly represented as a tree of processing units with 6 or even 7 levels: system – racks – frames – blades – nodes – processors – cores. We will use our hierarchical SGL model in Section 4.3 of Chapter 4 to model several previous presented machines.

3

PARALLEL PROGRAMMING MODELS

3.1 Low-Level Parallel Models 30
3.1.1 Shared-Memory Communication 30
3.1.1.1 Multi-threaded programming 31
3.1.1.2 Directive Programming 32
3.1.2 Message-Oriented Models 35
3.1.2.1 Message Passing Interface 35
3.1.2.2 Message Queuing Protocols 36
3.1.3 SWAR Programming Models 38
3.2 High-Level Parallel Models 41
3.2.1 Concurrent Computation Models 41
3.2.1.1 Process Calculi 41
3.2.1.2 Actor Model 44
3.2.2 Bridging Models 47
3.2.2.1 Parallel Random Access Machine 47
3.2.2.2 Bulk-Synchronous Parallel 48
3.2.2.3 Extensions of the BSP Model 54
3.2.3 Other Parallel Models 57
3.2.3.1 The LogP Model 57
3.2.3.2 Divide and Conquer – Cilk 58
3.2.3.3 Algorithmic skeletons & MapReduce 60

We review in this chapter the state of the art of parallel programming models in order to understand the portability of code, the scalability of algorithms, practicality of machine modelling, simplicity of programming, and feasibility of optimisation analysis between different models.

The chapter is organised in two sections: the first one presents low-level parallel models for shared-memory multi-threaded programming, shared-nothing message-passing programming, and SWAR programming; the second one presents high-level portable models such as process calculi, bridging models, and MapReduce etc.

We emphasize, in this chapter, the BSP model (Section 3.2.2.2) which enforces a strict separation of communication and computation, removes nondeterminism and guarantees the absence of deadlocks, gives an accurate model of performance prediction, provides much simplicity and efficiency for parallel programming.

3.1 LOW-LEVEL PARALLEL MODELS

In this section, we review different parallel programming techniques for either shared-memory architecture or distributed-memory architecture. Shared-memory architecture can simplify program design; it is easy to use thanks to its implicit communications. Distributed-memory architecture is more realistic for large supercomputer design. But the explicit communication is not easy to handle. However, these techniques depend greatly on machine architecture, that is why we call them low-level parallel models here.

3.1.1 *Shared-Memory Communication*

Shared memory is a memory that may be simultaneously accessed by multiple processes with an intent to provide communication among them or avoid redundant copies. A shared memory system (Figure 14) is relatively easy to program since all processors share a single view of data and the communication between processors can be as fast as memory accesses to a same location.

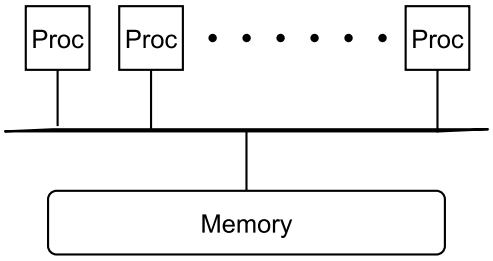

Figure 14: Shared-Memory Architecture

The issue with shared memory systems is that CPU-to-memory connection could be a bottleneck. Thus, the number of processors is limited. Shared-

memory hardware architecture may use different techniques such as: Uniform Memory Access (UMA), Non-Uniform Memory Access (NUMA), Cache-Only Memory Architecture (COMA) etc. A shared-memory hardware architecture is not mandatory in the logical point of view, direct access of memory can be simulated by the middleware if needed.

3.1.1.1 *Multi-threaded programming*

Process' resources may be shared by threads efficiently. Multi-threading paradigm provides developers with a useful abstraction of concurrent execution. Developers do not need to worry about the communication which is implicit. But this kind of program can be executed only on a single OS system.

To avoid race conditions [Ung95] in a shared-memory environment, different locking techniques to coordinate between threads were proposed, such as mutexes, semaphores, monitors, etc.

The semaphore concept was invented in 1965 by Edsger W. Dijkstra [Dij65, HDH02]. A semaphore is a variable or abstract data type that provides an abstraction for controlling access by multiple processes to a common resource in a parallel programming environment. A Counting semaphore S is equipped with two operations: V and P. They work as follow:

1. In the beginning, S is initialized with a non-negative integer (the number of available resources).
2. When P is called, S is decremented (the resource is distributed to the process). If S is negative, the process is blocked (no available resource).
3. When V is called, S will be incremented (the resource is gave back by the process).

Edsger W. Dijkstra [Dij65] identified and solved the mutual exclusion problem in the same year: a mutex is essentially the same thing as a binary semaphore. C. A. R. Hoare [Hoa74] invented the monitor concept in 1974: a monitor is an object or module intended to be used safely by more than one thread.

POSIX[1] provides a standardized application programming interface (API) for using shared memory on UNIX-like platform. The POSIX Shared Memory

1 Portable Operating System Interface, a family of standards specified by the IEEE.

API contains the following functions for managing memory allocation and interprocess communication: *shm_open*, *shmat*, *shmctl*, *shmdt* and *shmget* etc. POSIX defines also an API named Pthreads for creating and manipulating threads. Some measures performed in Section 4.3 are based on it.

Recently, Thompson et al. found that cache misses at the CPU-level and locks requiring kernel arbitration are both extremely costly [TFM+11]. They thus introduced a lock-free concept named *Disruptor* using a ring buffer to avoid these issues in order to obtain very low-latency and high-throughput programs. Based on this concept, a high performance inter-thread messaging library was developed by LMAX[2] on Java platform. However, LMAX disruptor is still limited on shared memory architecture.

3.1.1.2 *Directive Programming*

Preprocess directives handled by the compiler may be used to describe some programming language constructs. Directive programming offers the possibility of automatic parallelization. A program with parallel directives can also be compiled correctly in sequential.

In C, for example, the *#pragma* directive is used to instruct the compiler to use pragmatic or implementation-dependent features. Two notable users of such directive for parallel programming are OpenMP and OpenACC.

OpenMP [3] [CJvdPK07] is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. It is a multi-threading implementation. The threads run concurrently, with the runtime environment allocating threads to different processors.

The core elements of OpenMP are:

- **Thread creation**: *parallel* directive.
- **Work sharing**: *do/parallel do* and *section* directives.
- **Data-environment management**: *shared* and *private* clauses.
- **Thread synchronization**: *critical*, *atomic* and *barrier* directives.
- **Runtime routines and environment variables**: *omp_set_num_threads()*, *omp_get_thread_num()*, *OMP_NUM_THREADS* and *OMP_SCHEDULE*.

2 LMAX Exchange. http://www.lmax.com/

3 The OpenMP API specification for parallel programming. http://openmp.org/

Compared to POSIX API, OpenMP API may greatly simplify parallel programming using automatic parallelism techniques. The code in Listing 1 simply uses a *parallel for* directive to initialize the value of a large array in parallel using each thread to do part of the work. This code can be compiled correctly in parallel and in sequential (if the compiler does not support the parallel directives). The number of parallel threads is defined before the compilation either implicitly (same as the number of processors) or explicitly.

```
// Automatically parallel large array initialization

#include <omp.h>

int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];

    #pragma omp parallel for
    for (i = 0; i < N; ++i)
        a[i] = 2 * i;

    return 0;
}
```

Listing 1: OpenMP code in C for automatically parallelizing array initialization

Parallelization using OpenMP can be programmed not only automatically but also explicitly, the latter makes data locality clearer for developers. However, this causes a loss of code portability and increase programming complexity. Listing 2 shows how programming the same job in Listing 1 is complicated using explicit parallelization approach and how clear the data locality is.

The experimentation that we performed in Section 5.2 used OpenMP for the shared-memory node-level part.

```c
// Explicitly parallel large array initialization

#include <omp.h>

int main(int argc, char *argv[]) {
    int rank,size;
    const int N = 100000;
    int i, a[N], block;

    size = omp_get_num_threads();
    block = N / size;

    // Each thread has its own variables rank and i
    #pragma omp parallel private(rank, i)
    {
        rank = omp_get_thread_num();

        // For the threads where rank = 0 .. size-2
        if (rank != (size - 1)) {
        for (i = rank * block; i < (rank + 1) * block; ++i)
            a[i] = 2 * i;
        }
        // For the last thread, rank = size-1
        else {
            for (i = rank * block; i < N; ++i)
                a[i] = 2 * i;
        }
        #pragma omp barrier
    }
    return 0;
}
```

Listing 2: OpenMP code in C for explicitly parallelizing array initialization

OpenACC[4] is a programming standard for parallel computing proposed in 2012 by *Cray*, *CAPS*, *Nvidia* and *PGI*. Like in OpenMP, C, C++, and Fortran source code can be annotated to identify the areas that should be accelerated using *progma* compiler directives and additional functions. But different from OpenMP, OpenACC's objective is to simplify parallel programming of heterogeneous CPU/GPU systems. Thus, its code can be compiled not only on the CPU, but also on the GPU.

4 OpenACC Home. http://www.openacc.org/

3.1.2 *Message-Oriented Models*

Shared-memory models simplify the parallel programming with their implicit communication. However, the communication cost always exists in a parallel program and can never be neglected. A communication-explicit model allows developers to better understand parallel program performance, opens the possibility to optimize communication cost.

The message-oriented approach focuses on the communication. It is often used on distributed-memory architectures (Figure 15). It can also be applied on shared-memory architectures to gain program portability.

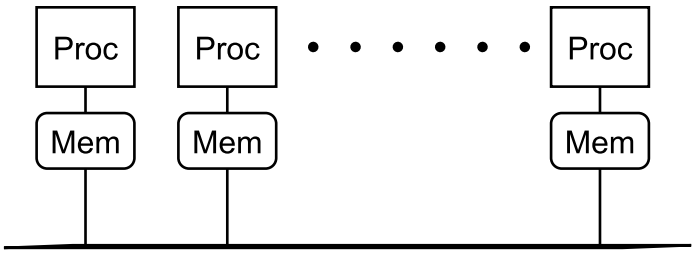

Figure 15: Distributed-Memory Architecture

Message-oriented middleware allows application modules to be distributed over heterogeneous platforms and reduces the complexity of developing applications that span multiple operating systems and network protocols.

3.1.2.1 *Message Passing Interface*

Message passing is the paradigm of communication. The messages are sent from a sender to one or more recipients. Depending on implementation, communication may be synchronous or asynchronous, messages may be passed one-to-one (unicast), one-to-many (broadcast), many-to-one (gather), or many-to-many (All-to-All). Many high-level data parallel models (c. f., Section 3.2) are based on message passing.

Message Passing Interface (MPI) is a language-independent communications protocol used to program parallel computers. The goals of MPI are high

performance, scalability, and portability. It remains today the dominant model used in high-performance computing [DD95, SKP06].

An independent identifier is attributed to each process by MPI communicator. MPI supports both point-to-point and collective communication using functions such as: *MPI_Send*, *MPI_Receive*, *MPI_Bcast*, *MPI_Reduce* and *MPI_Alltoall* etc.

There are many MPI implementations. MPICH is one of the most popular implementations of MPI. It is used as the foundation for the vast majority of MPI implementations, including *IBM* MPI (used for example by Blue Gene which was viewed in Section 2.2.1.2), *Intel* MPI, *Cray* MPI, and many others. *SGI* has its own implementation named Message Passing Toolkit (MPT) used by SGI supercomputer such as Altix ICE (presented in Section 2.2.1.3).

The scalability and portability of MPI programs can be improved by applying SPMD (Single Program, Multiple Data) technique. Most of today's parallel programs are developed on MPI. The experimentation that we performed in Section 5.2 uses MPI too.

3.1.2.2 *Message Queuing Protocols*

Message Queuing Protocols provide patterns to simplify the communication. It can also be employed as a middleware for implementing actor models (c. f., Section 3.2.1.2). Application layer protocols are the tendency of message-oriented middleware.

AMQP [5] [O'H07] originated by JPMorgan Chase & Co., co-developed by Cisco Systems, IONA Technologies, iMatix, Red Hat, and TWIST is an open standard application layer protocol for passing business messages between applications. It is defined to provide flexible routing, including common messaging paradigms like point-to-point, fanout, publish/subscribe, and request-response.

ZeroMQ [6] [Hin13] is a high-performance asynchronous messaging library introduced by iMatix after leaving AMQP workgroup for providing a signif-

5 The Advanced Message Queuing Protocol. http://www.amqp.org/
6 ∅MQ. http://www.zeromq.org/

icantly simpler and faster communication. It acts like a concurrency framework. ZeroMQ has four basic patterns:

- **Request-reply** (Figure 16): The REQ-REP socket pair is in lockstep. The client issues *zmq_send()* and then *zmq_recv()* in a loop. This is a remote procedure call and task distribution pattern.

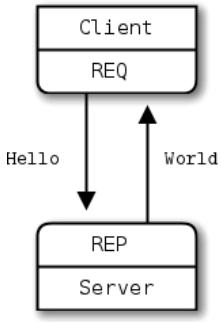

Figure 16: ZeroMQ Request-Reply Pattern [Hin13]

- **Publish-subscribe** (Figure 17): The PUB-SUB socket pair is asynchronous. The client calls *zmq_recv()* in a loop. Similarly, the service calls *zmq_send()* as often as needed. This is a data distribution pattern.

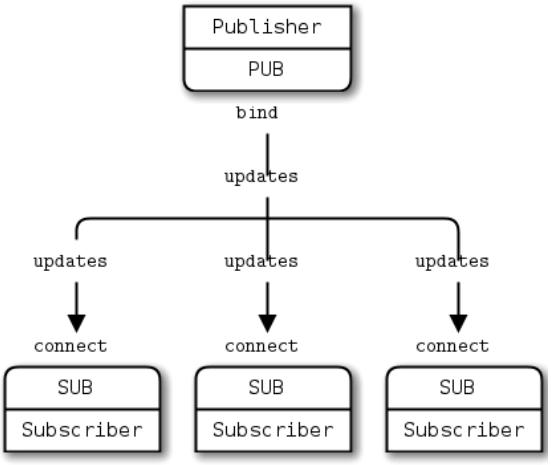

Figure 17: ZeroMQ Publish-Subscribe Pattern [Hin13]

- **Parallel pipeline** (Figure 18): This is a "ventilator" that produces tasks that can be done in parallel. The workers connect upstream to the ventilator and downstream to the sink. The sink that collects results back from the worker processes. This is a parallel task distribution and collection pattern.

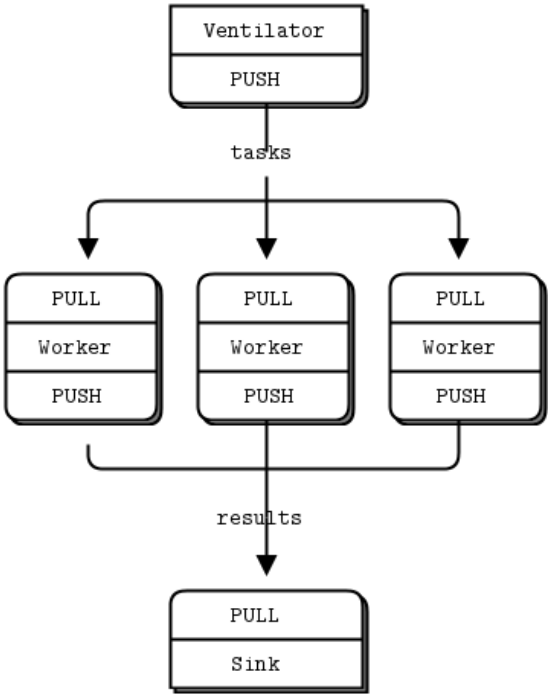

Figure 18: ZeroMQ Parallel Pipeline Pattern [Hin13]

- **Exclusive pair**: which connects two sockets exclusively. This is a pattern for connecting two threads in a process, not to be confused with "normal" pairs of sockets.

Message queuing models provide different communication patterns to reduce the complexity of parallel programming, but the communication latency is difficult to manage because of the implicit routing. Twitter's *Storm* platform (c. f., Section 3.2.1.2) and some streaming modules of EXQIM's algorithmic trading system are based on ZMQ protocol.

3.1.3 *SWAR Programming Models*

SWAR is an acronym for **S**IMD **W**ithin **A** **R**egister. SIMD, in turn, stands for **S**ingle **I**nstruction, **M**ultiple **D**ata. On x86 processors, the well-known SWAR architectures contain AMD's 3DNow!, Intel's MMX, SSE (Streaming SIMD Extensions) and its successors, and AVX (Advanced Vector Extensions), etc.

SWAR processing has been applied in a wide range of computation-intensive fields such as image processing [PPdQN+01], cryptographic pairings [GGP09], raster processing [PG04], computational fluid dynamics [HML+03] and communications [Spr01] etc.

SWAR programs are architecture-depended. It is not easy to develop SWAR programs but they provide considerable acceleration thanks to the hardware implementation.

Several SWAR programming models were mentioned in Randall J. Fisher's thesis [Fis03]:

- **Scalar models**. The code written in a scalar language is analyzed by a so-called "vectorizing" compiler to find parallel-able operations and functions. These are then translated into vector- or array-based parallel code for the target architecture. The programming model is thus a scalar one.

- **Array models**. Multi-dimensional array models are the most commonly used non-scalar models in parallel processing. The array object can be operated on as a single aggregate object rather than as a set of scalar elements via looping or parallelizing constructs.

- **Vector models**. Unlike array models, vector models are single-dimensional and non-scalar. True vector models are more consistent with the operation of SWAR architecture than scalar or array models.

- **Sequential model**. The NESL language [BHC+93] describes parallel data as recursive sequences. It allows complex, irregular, and nested data structures to be described and operated on.

The code in Listing 3 shows a C++ function that adds two 4-element vectors using inline assembly calling SSE instructions.

SWAR is always a good low-cost option for accelerating certain expensive operations. A new instruction named FMA (fused multiple-add) will be supported by both Intel and AMD's X86 microprocessors in 2013.

```
// A 16byte = 128bit vector struct
struct Vector4
{
   float x, y, z, w;
};

// Add two constant vectors and return the resulting vector
Vector4 SSE_Add ( const Vector4 &Op_A, const Vector4 &Op_B )
{
   Vector4 Ret_Vector;

   __asm
   {
      MOV EAX Op_A              // Load pointers into CPU regs
      MOV EBX, Op_B

      MOVUPS XMM0, [EAX]        // Move unaligned vectors to SSE regs
      MOVUPS XMM1, [EBX]

      ADDPS XMM0, XMM1          // Add vector elements
      MOVUPS [Ret_Vector], XMM0 // Save the return vector
   }

   return Ret_Vector;
}
```

Listing 3: SSE: Adding Vectors

In order to reduce the complexity of SWAR code development, the LLVM compiler[7] supports since 2012 auto-vectorization [LLV12]. Vector operations in LLVM can be automatically translated by the compiler to benefit the SWAR architecture.

Low-level parallel programming models allow developers to make even finer performance tuning. However, all presented low-level models are architecture-depended, high-level models are desirable for developing more complex and portable algorithms.

7 Formerly Low-Level Virtual Machine. http://llvm.org/

3.2 HIGH-LEVEL PARALLEL MODELS

We present, in this section, some notable portable high-level parallel programming models for which algorithms could be developed/analysed more precisely while keeping their portability. First of all, we review three concurrent computation models: CSP, CCS, and Actor Model which provide algebraic laws or simulate the physical world that allow process descriptions to be manipulated and analysed. After that, we review PRAM, BSP and some extensions of BSP as bridging models that provide a conceptual bridge between the physical implementation of the machine and the abstraction available to a programmer of that machine. The sequential view of a parallel program featured by bridging models can greatly help developers to analyse and/or create parallel algorithms. Finally, we finish the section by reviewing LogP, Cilk, and MapReduce.

3.2.1 *Concurrent Computation Models*

In parallel programming, each process is concurrent to others. Several models inspired by algebra (e. g., process calculus) or physical world interactions (e. g., the actor model) were used for modelling these concurrent processes.

3.2.1.1 *Process Calculi*

A *process calculus* is intended for modelling concurrent system. It provides a tool for the high-level description of interactions, communications, and synchronizations between a collection of independent agents or processes. Process descriptions can be manipulated and analysed using process calculus algebraic laws. Formal reasoning about equivalences between processes can be carried out with process calculus. The most notable process calculi are Communicating Sequential Processes (CSP) and Calculus of Communicating Systems (CCS) with its evolution as the π-calculus [MPW92, Mil93].

CSP was first described by C. A. R. Hoare [Hoa78] in 1978 as concurrent programming language, and has evolved substantially to refine the theory of process algebraic form [AJS04]. CSP has been practically applied in industry

as a tool for specifying and verifying the concurrent aspects of a variety of different systems [Bar95, HC02].

CSP allows the description of systems in terms of component processes that operate independently and interact with each other solely through message-passing communication. The syntax of CSP defines the "legal" ways in which processes and events may be combined. Let e be an event, and X be a set of events, then the basic syntax of CSP can is shown in Table 1.

Proc	::=	STOP	
	|	SKIP	
	|	$e \rightarrow$ Proc	(prefixing)
	|	Proc $\Box$ Proc	(external choice)
	|	Proc $\sqcap$ Proc	(non-deterministic choice)
	|	Proc $\|\|\|$ Proc	(interleaving)
	|	Proc $\|[\{X\}]\|$ Proc	(interface parallel)
	|	Proc $\setminus$ X	(hiding)
	|	Proc; Proc	(sequential composition)
	|	if b then Proc else Proc	(boolean conditional)
	|	Proc $\triangleright$ Proc	(timeout)
	|	Proc $\triangle$ Proc	(interrupt)

Table 1: Syntax of CSP

The following example shows how to apply the CSP syntax to represent a chocolate vending machine interacting with a person wishing to buy some chocolate. The vending machine may receive a payment (event "coin") and offer a chocolate (event "choc"). A machine which demands payment before delivering a chocolate can be written as:

$$VendingMachine = coin \rightarrow choc \rightarrow \text{STOP}$$

A person may pay coins as well as by debit/credit card can be modelled as:

$$Person = (coin \rightarrow \text{STOP}) \Box (card \rightarrow \text{STOP})$$

These 2 processes can be in parallel to interact with each other:

$$VendingMachine\ |[\{coin,\ card\}]|\ Person \equiv coin \rightarrow choc \rightarrow STOP$$

When synchronization is only required on "coin", we have:

$$VendingMachine\ |[\{coin\}]|\ Person \equiv (coin \rightarrow choc \rightarrow STOP)\Box(card \rightarrow STOP)$$

The "coin" and "card" events can be hidden:

$$((coin \rightarrow choc \rightarrow STOP)\Box(card \rightarrow STOP))\backslash\{coin,\ card\}$$

We thus get a **non-deterministic** process:

$$(choc \rightarrow STOP) \sqcap STOP$$

A single-program-multiple-data (SPMD) parallel program E can be described in CSP as:

$$[|E|]_{SPMD} = [|E@0 \parallel \ldots \parallel E@(p-1)|]_{CSP}$$

where $[|\ |]_{SPMD}$ is the defined meaning of program E, $[|\ |]_{CSP}$ is the CSP transition semantics, and $E@i = E[pid \leftarrow i]$.

However, the `pid` (Processor ID, i. e., processor number) variable is bound **outside** the source program in most cases [Gav05] which is not a standard use of lexical scoping for identifiers [Wik]. This is what led to the design of the `mkpar` primitive in BSML(BSP-CAML c. f., Section 3.2.2.2): it takes as input a function from `pid` processor indexes to "local" values and creates a parallel value. This has three advantages over previous uses of `pid` (like MPI [GLS99]):

1. the binding of variable PID is explicit in parallel programs;
2. the variable's name is arbitrary, not necessarily `pid`; and
3. the binding scope is local to the definition of `mkpar` argument.

CCS was introduced by Robin Milner [Mil82] around 1980 as a process calculus. Its actions model indivisible communications between exactly two participants. CCS is useful for evaluating the qualitative correctness of properties of a system such as **deadlock** or **livelock** [KB07].

The formal language of CCS includes primitives for describing parallel composition, choice between actions and scope restriction. The set of CCS processes is defined by the BNF grammar in Table 2.

P	::=	$\emptyset$	(empty process)
	|	$a.P_1$	(action)
	|	A	(process identifier)
	|	$P_1 + P_2$	(choice)
	|	$P_1 \| P_2$	(parallel composition)
	|	$P_1[b/a]$	(renaming)
	|	$P_1 \backslash a$	(restriction)

Table 2: Syntax of CCS

The development of CSP was influenced by CCS and vice versa. There is a fair number of cross-references, acknowledgements, and reciprocal citations.

CSP and CCS can help higher level parallel model implementation analysis and validation. For example, Simpson et al. [SHD99] have applied CSP for analysing BSPlib's transport layer protocol for BSP programming and have shown that this protocol is fault-tolerant and free from the potential for deadlock and livelock. Merlin and Hains [MH05, MH07] integrated CCS semantics with BSP model, proposed a BSP process algebra and generalized its cost model for concurrent and data-parallel meta-computing.

3.2.1.2 *Actor Model*

The actor model, inspired by physical world interactions, is a mathematical model of concurrent computation that treats "actors" as the universal primitives. The actor model was introduced in 1973 by Carl Hewitt [HBS73].

The actor model adopts the philosophy that *everything is an actor*. An actor is a computational entity that, in response to a message it receives, can concurrently:

- send a finite number of messages to other actors;
- create a finite number of new actors;
- designate the behaviour to be used for the next message it receives.

Decoupling the sender from communications sent is a fundamental advance of the actor model enabling asynchronous communication and control structures as patterns of high-level messages passing [Hew77].

Many expressive actor programming platforms have been proposed in recent years. The most notable are: *Apache*'s S4[8], *Twitter*'s Storm[9] and *Typesafe*'s Akka[10] etc.

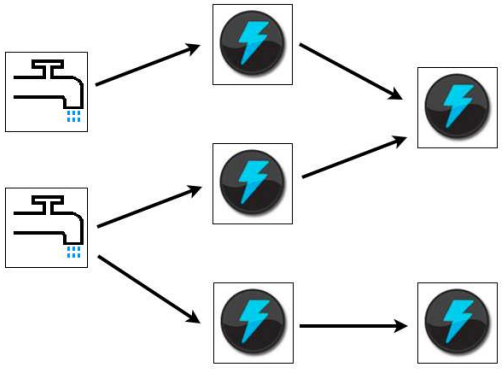

Figure 19: A Storm topology (spouts are represented by water-taps, and bolts are represented by lightnings) [Sto12]

Storm [Sto12] performs real-time computations based on "topologies" (Figure 19). A topology is a graph of computation. Each Storm node in a topology contains processing logic, and links between nodes indicate how data should be passed around between nodes. Nodes are used to transform a stream, which is an unbounded sequence of tuples, into a new stream in a distributed and reliable way. A node can be either Spout or Bolt. A spout is a source of streams in a topology. A bolt consumes any number of input streams, does some processing, and possibly emits new streams. Bolts can do anything from run functions, filter tuples, do streaming aggregations, do

8 http://incubator.apache.org/s4/
9 http://storm-project.net/
10 http://akka.io/

streaming joins, talk to databases, and more. Links between nodes in a topology indicate how tuples should be passed around.

Spouts and bolts execute in parallel as many tasks across the cluster (Figure 20). Stream grouping defines how to send tuples between sets of tasks. There are seven built-in stream groupings in Storm:

- **Shuffle grouping**: Tuples are randomly distributed across the bolt's tasks in a way such that each bolt is guaranteed to get an equal number of tuples.
- **Fields grouping**: The stream is partitioned by the fields specified in the grouping.
- **All grouping**: The stream is replicated across all the bolt's tasks.
- **Global grouping**: The entire stream goes to a single one of the bolt's tasks.
- **None grouping**: This grouping specifies that you do not care how the stream is grouped.
- **Direct grouping**: This is a special kind of grouping. A stream grouped this way means that the producer of the tuple decides which task of the consumer will receive this tuple.
- **Local or shuffle grouping**: If the target bolt has one or more tasks in the same worker process, tuples will be shuffled to just those in-process tasks. Otherwise, this acts like a normal shuffle grouping.

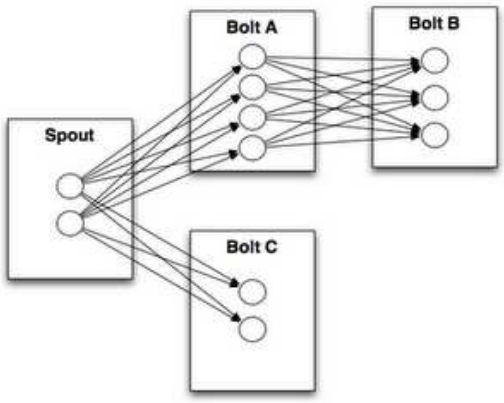

Figure 20: A task-level view of Storm topology [Mar12]

Strom is related to dataflow models which have a long history. It is fine-tuned to handle unbounded data streams such as stock market data. But for lack of space we will not describe them further here.

3.2.2 *Bridging Models*

The models reviewed in Section 3.2.1 greatly simplify parallel programming. However, the performance of supercomputer cannot only be measured in FLOPS [FMM+13], thus a bridging model for accurate performance prediction is demanded for designing large-scale systems or algorithms [BDH+09]. The term *bridging model*[11] was introduced in 1990 by Leslie Valiant [Val90]. A bridging model is an abstract model of a computer which provides a conceptual bridge between the physical implementation of the machine and the abstraction available to a programmer of that machine. It provides a common level of understanding between hardware and software engineers. Thus, one can develop portable and predictable algorithms on it.

Bridging models provide also a sequential view of a parallel program with **supersteps**, this feature could greatly help developers to analyse and/or create parallel algorithms. The sequential view *SEQ of PAR* [12] was analysed by L. Bougé et al. [Bou96] in their study of data-parallel semantics.

3.2.2.1 *Parallel Random Access Machine*

Forturne et al. [FW78] have introduced the Parallel Random Access Machine (PRAM) model for parallel computing. The PRAM model consists of p processors (Figure 21). Each one has its own private local memory, and they all share a global memory. In PRAM model, processors execute the computation instructions synchronously. At each step of a PRAM algorithm, some processors are active and execute : read, write or compute instructions. The other processors are inactive. In a read step, each active processor reads one global memory location into its local memory. In a compute step, each active processor executes a single operation and writes the result into its local memory. In a write step, each active processor writes one local memory location into the global memory.

It was necessary to define some memory access restrictions to resolve read and write conflicts to the same shared memory location. Depending on the restrictions on memory access, we have 4 different PRAM models:

11 which we translate in this thesis in French "modèle de transition logico-matérielle"
12 equivalent to *PAR of SEQ* execution

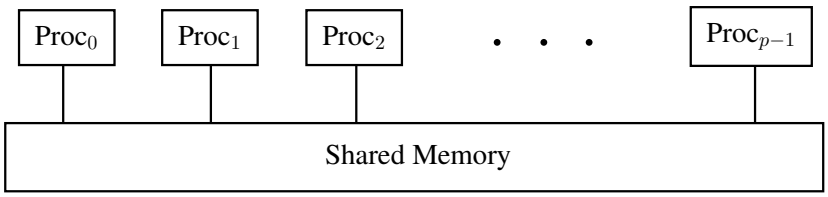

Figure 21: A PRAM computer

- **Exclusive Read, Exclusive Write (EREW) PRAM**: At each time step, one and only one processor can read or write the same shared memory location.

- **Concurrent Read, Exclusive Write (CREW) PRAM**: At each time step, simultaneous reads of the same memory location are allowed, but only one processor can write to a shared memory location.

- **Concurrent Read, Concurrent Write (CRCW) PRAM**: At each time step, both simultaneous reads and writes of the same memory location are allowed. In CRCW PRAM model, we also need to specify what happens when several processors write to the same memory locations.

- **Queue Read, Queue Write (QRQW) PRAM** [GMR94] : At each time step, each memory location can be read or written by any number of processors. Concurrent read or write to a location are serviced one-at-a-time. The access time to read or write a location is proportional to the number of concurrent readers or writers to the same location.

The PRAM model is used by parallel-algorithm designers to model parallel algorithmic performance like other bridging models. Synchronisation and communication are neglected in this model. Algorithm cost is thus estimated only using two parameters: O(time) and O(time $\times$ processor_number). The time complexity of a PRAM algorithm depends on the number of instructions needed to be executed; and the space complexity depends on the number of memory cells needed to be allocated.

3.2.2.2 *Bulk-Synchronous Parallel*

The Bulk-Synchronous Parallel (BSP) model is a programming model introduced by Leslie Valiant [Val90]. It offers a high degree of abstraction like

PRAM models (c. f., Section 3.2.2.1) and yet allows portable and predictable performance on a wide variety of multi-processor architectures [SHM97]. The major difference between BSP and PRAM is that the local computations of BSP are asynchronous, and the cost of inter-processor communications in BSP is not neglected.

A BSP computer (Figure 22) contains:

- a homogeneous set of uniform processor-memory pairs;
- a communication network allowing inter-processor delivery of messages;
- and a global synchronization unit which executes collective requests for a synchronization barrier.

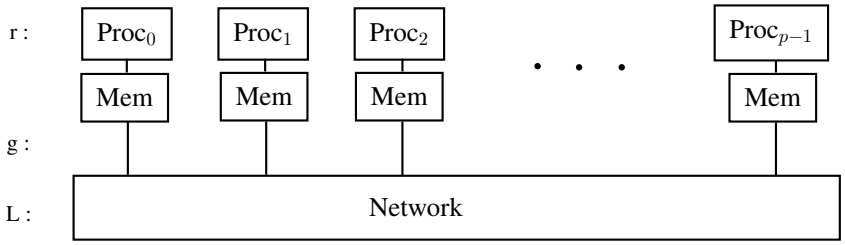

Figure 22: A BSP computer

A wide range of actual architectures can be seen as BSP computers. For example, shared-memory machines could be used in a way such as each processor only accesses a sub-part of the shared memory and communications could be performed using a dedicated part of the shared memory. Moreover, the synchronization unit is very rarely a hardware but rather a software event [HS98]. Supercomputers and clusters of PCs can be modelled as BSP computers.

A BSP program is executed as a sequence of **supersteps** (Figure 23), each one divided into (at most) three successive and logically disjoint phases:

1. In the first phase, each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes.
2. In the second phase, the network delivers the requested data transfers.
3. And in the third phase, a global synchronization barrier occurs, making the transferred data available for the next superstep.

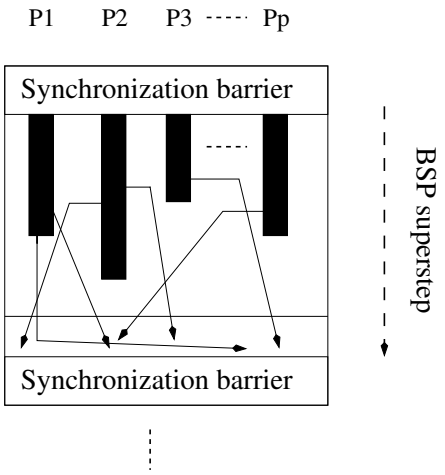

Figure 23: A BSP superstep

The performance of the BSP machine is characterised by 4 parameters:

- the local processing speed r;
- the number of processor-memory pairs p;
- the time L required for a global synchronization (barrier);
- and the time g for collectively delivering a 1-relation communication phase where every processor receives/sends at most one word.

The network can deliver an h**-relation** (every processor receives/sends at most h words) in time $g \times h$. To accurately estimate the execution time of a BSP program these 4 parameters could be easily benchmarked [Bis04].

The execution time (cost) of a superstep s is the sum of the maximum local processing time, the data delivery and the global synchronisation times. It is expressed by the following formula:

$$Cost(s) = \max_{0 \leqslant i \leqslant p}\{w_i^s\} + \max_{0 \leqslant i \leqslant p}\{h_i^s \times g\} + L$$

where w_i^s is the local processing time on processor i during superstep s, and $h_i^s = \max\{h_{i+}^s,\ h_{i-}^s\}$ where h_{i+}^s (resp. h_{i-}^s) is the number of words transmitted (resp. received) by processor i during superstep s.

The total cost of a BSP program composed of S supersteps is $\sum_S Cost(s)$. It is, therefore, the sum of three terms:

$$W + H \times g + S \times L$$

where $W = \sum_S \max_i\{w_i^s\}$ and $H = \sum_S \max_i\{h_i^s\}$.

In general, W, H and S are functions of p and of the size of data n, or of more complex parameters such as data skew. To minimize execution time, the BSP algorithm design must jointly minimize the number S of supersteps and the total volume H (resp. W). In addition, for each superstep s, the volume h_i^s (resp. w_i^s) must be balanced on all processors.

The BSP model has been implemented in different languages, as libraries or platforms. The most notable are:

- The BSP programming library – BSPlib[13] [HMS+98] maintained at *University of Oxford Parallel Applications Centre*. BSPlib can be used with C, C++, or Fortran. It supports **SPMD** parallelism based on efficient one-sided communications. The core library (excluding collective communications) consists of just 20 primitives such as: *bsp_nprocs*, *bsp_pid*, *bsp_sync*, *bsp_put*, *bsp_get*, *bsp_set*, *bsp_send*, etc.

- The Paderborn University BSP library (PUB)[14] [BJOR99, BJvOR03] is a C-Library to support development of parallel algorithm based on the BSP model. Like BSPlib, the PUB-Library offers buffered asynchronous message-passing between the nodes organized in supersteps.

- MulticoreBSP[15] [YBRM13] developed at *Katholieke Universiteit Leuven* is designed for shared-memory multi-core architecture. Its interface is directly derived from the BSPlib. Compared to BSPlib, MulticoreBSP for C adds two new high-performance primitives and updates the interface of existing primitives. The library depends on only two established standards: POSIX threads (PThreads) and the POSIX realtime extension for maximum portability.

- BSML[16] [LGB05] developed at *Université d'Orléans* and *Université Paris XII* (now called *Université Paris-Est Créteil* or UPEC) is a library for OCaml[17] implementing partially the Bulk Synchronous Parallel ML language [Gav05]. There is in BSML an abstract polymorphic type α par which represents the type of p-wide parallel vectors of values of type α, one per process. It is very different from usual SPMD programming

13 Oxford BSPlib. http://www.bsp-worldwide.org/implmnts/oxtool/
14 PUB-Library. http://www2.cs.uni-paderborn.de/~pub/
15 MulticoreBSP. http://www.multicorebsp.com/
16 Bulk Synchronous Parallel ML. http://traclifo.univ-orleans.fr/BSML/
17 Objective Caml. http://caml.inria.fr/

where messages and processes are explicit, and programs may be non-deterministic or may contain deadlocks. In fact a large subset of BSML parallel programs are purely functional. The newest version (0.5) of core BSML library is based on the following primitives:

mkpar : $(int \to \alpha) \to \alpha\ par$

proj : $\alpha\ par \to (int \to \alpha)$

apply : $(\alpha \to \beta)\ par \to \alpha\ par \to \beta\ par$

put : $(int \to \alpha)\ par \to (int \to \alpha)\ par$

The semantics of BSML primitives is described by the use of parallel values. Parallel value <x_0, x_1,..., x_{p-1}> represents a set of local values of a given type, such that x_i is stored on processor i and p is the number of processors.

In BSML, **mkpar** is the parallel constructor:
mkpar f computes the value <f 0, f 1,..., f (p-1)>.

proj is the parallel destructor:
proj <x_0, x_1,..., x_{p-1}> computes a function f such that (f i) = x_i.

apply is the asynchronous parallel transformer:
apply <f_0, f_1,..., f_{p-1}> <x_0, x_1,..., x_{p-1}> computes
<f_0x_0, f_1x_1,..., $f_{p-1}x_{p-1}$>.

Finally, **put** is the synchronous (communicating) parallel transformer:
put <g_0, g_1,..., g_{p-1}> computes a parallel vector of functions that contain the transported messages that were specified by the g_i. The input local functions are used to specify the outgoing messages thus: g_i j is the value that processor i wishes to send to processor j. The result of applying **put** is a parallel vector of functions dual to the g_i: they specify which value was received from a given distant processor.

The execution of **mkpar** is purely local and so is the execution of the **apply** primitive. The execution of **proj** uses an all-to-all communication and the execution of **put** is a general BSP communication (any processor-processor relation can be implemented with it). Experience with BSML for more than a decade has shown that **proj** is much easier to use than **put** , that **proj** can be used to program a large subset of all parallel functions, but that algorithms such as sample-sort cannot be im-

plemented without **put** . Chapter 6 will propose elements of a solution to this dilemma based on the correspondence **mkpar** = **scatter** , and **proj** = **gather** for a flat BSP machine.

- Google Pregel [MAB+10] inspired by the BSP model is a platform for large-scale graph processing. It provides a fault-tolerant framework for the execution of graph algorithms in parallel over many machines. Programs are expressed in Pregel as a sequence of iterations (superstep). In each iteration, a vertex can, independently of other vertices, receive messages sent to it in the previous iteration, send messages to other vertices, modify its own and its outgoing edges states, and mutate the graph's topology. "Thinking like a vertex" is the essence of programming in Pregel.

- Apache Hama[18] is a pure BSP computing open-source framework on top of Hadoop Distributed File System (HDFS) for massive scientific computations such as matrix, graph and network algorithms. Hama architecture is similar to Hadoop[19] architecture, except in the portion of communication and synchronization mechanisms. It consists of three major components:
  - **BSPMaster**, used for maintaining groom server status, supersteps and other counters in a cluster, and job progress information; scheduling jobs and assigning tasks to groom servers; distributing execution classes and configuration across groom servers; providing users with the cluster control interface (web and console based).
  - **GroomServer**, is a process that launches BSP tasks assigned by BSPMaster.
  - **ZooKeeper**[20], used to manage the efficient barrier synchronization of the BSPPeers.

Many BSP algorithms have been developed and are widely applied [Tis99, Bis95]. The BSP model has been used with success in a wide variety of problems such as scientific computing [Ger93, BM94, DM02, HB99], parallel datastructure [LGG97, LG02], genetic algorithms and programming [DK96b, DK96a], neural network [RS98], etc.

18 Apache Hama. http://hama.apache.org/
19 Apache Hadoop. http://hadoop.apache.org/
20 Apache ZooKeeper. http://zookeeper.apache.org/

The BSP model enforces a strict separation of communication and computation: during a superstep, no communication between the processors is allowed, only at the synchronisation barrier they are able to exchange information. This execution policy has two main advantages: first, it guarantees the absence of deadlocks and allows its implementation remove non-determinism; second, it allows for an accurate model of performance prediction based on the throughput and latency of the interconnection network, and on the speed of processors. This performance prediction model can even be used online to dynamically make decisions, for instance choose whether to communicate in order to re-balance data or to continue an unbalanced computation.

The BSP model greatly facilitates debugging. The computations going on during a superstep are completely independent and thus can be debugged independently. Moreover, it is easy to measure during the execution of a BSP program, the time spent to communicate and to synchronise by just adding chronometers before and after the primitive of synchronisation.

3.2.2.3 *Extensions of the BSP Model*

BSP has been extended by many authors to address concerns about BSP's unsuitability for modelling specific architectures or computational paradigms.

The E-BSP [BFMR96] extends the basic BSP model to deal with unbalanced communication patterns. i. e., patterns in which the amount of data sent or received by each node is different. The cost function supplied by E-BSP is a non-linear function that strongly depends on the network topology. The model essentially differentiates between communication patterns that are insensitive to the bisection bandwidth and those that are not.

The Decomposable-BSP model (D-BSP) [TK96] extends the BSP model by introducing the possibility of sub-machine synchronizations. A D-BSP computer is basically a BSP computer where the synchronization device allows subgroup of processors to synchronize independently. The D-BSP remembers the HPRAM [HR92] and CLUMPS [CT94] models in which the costs are expressed in terms of BSP supersteps. In this framework network locality can be exploited assuming that sub-machines parameters are a decreasing function of the diameter of the subset of processors involved in communication and synchronization.

The EM-BSP model [DDH97] includes secondary local memories. In this model, each processor has, in addition to its main memory, an external memory formed of a set of disks. The model is restricted to the case where all processors have the same number of disks because it is mostly the case in practice. Each disk drive consists of a sequence of tracks. The tracks can be accessed by direct random access using their unique track number. Each processor can use all of its disk drives concurrently.

The H-BSP model [CL01] adds a hierarchical concept to the BSP model. An H-BSP program consists of a number of BSP groups which are dynamically created at run time and executed in a hierarchical fashion. H-BSP provides a group-based programming paradigm and supports *Divide & Conquer* algorithms efficiently.

The Bulk-Synchronous Parallel Random Access Machine (BSPRAM) [Tis98] reconciles shared-memory style programming with BSP's efficient exploitation of data locality (Figure 24). In this model, BSP communication network is replaced by a global shared main memory. Different from BSP, a BSPRAM superstep consists of an *input phase*, a *local computation phase*, and an *output phase* (Figure 25).

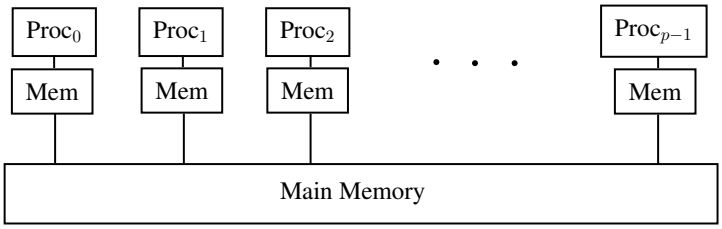

Figure 24: A BSPRAM computer

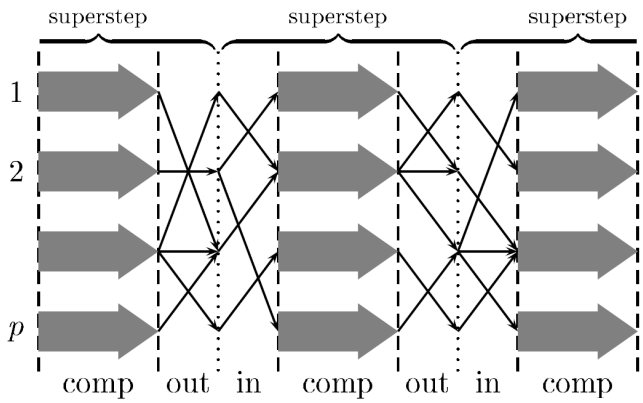

Figure 25: A BSPRAM computation [Tis98]

The original goal of Multi-BSP model [Val11] is capturing the most basic resource parameters of multi-core architectures. It is a multi-level model that has explicit parameters for processor numbers, memory/cache sizes, communication costs, and synchronization costs. The lowest level corresponds to shared memory or the PRAM, acknowledging the relevance of that model for whatever limitations on memory and processor numbers it may be efficient to emulate it.

The Multi-BSP model extends BSP in two ways. First, it is a hierarchical model, with an arbitrary number of levels. It recognizes the physical realities of multiple memory and cache levels both within single chips as well as in multi-chip architectures. The aim is to model all levels of an architecture together, even possibly for whole datacenters. Second, at each level, Multi-BSP incorporates memory size as a further parameter. After all, it is the physical limitation on the amount of memory that can be accessed in a fixed interval of time from the physical location of a processor that creates the need for multiple levels.

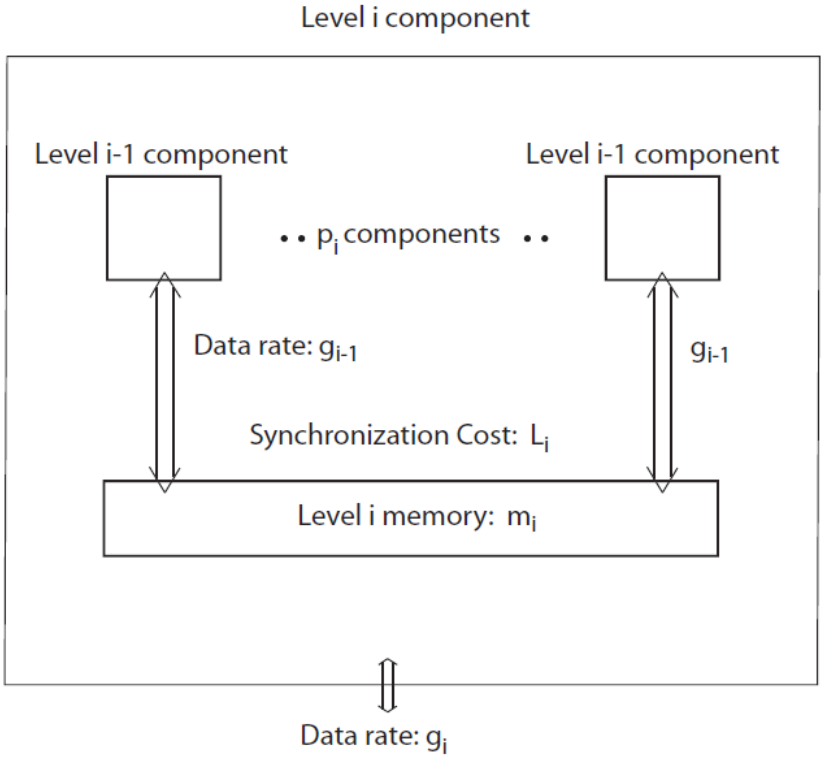

Figure 26: Schematic diagram of a Multi-BSP level i component [Val11]

An instance of a Multi-BSP is a tree structure of nested components where the lowest level or leaf components are processors and each other level con-

tains some storage capacity. The model does not distinguish memory from cache as such, but does assume certain properties of it.

The performance of a Multi-BSP computer can be described with parameter d, the depth or number of levels, and $4d$ further parameters (p_1, g_1, L_1, m_1) $(p_2, g_2, L_2, m_2)(p_3, g_3, L_3, m_3) \ldots (p_d, g_d, L_d, m_d)$. At the i^{th} level there are a number of *components* specified by the parameters (p_i, g_i, L_i, m_i)) each component containing a number of $i-1^{st}$ level components as illustrated in Figure 26.

3.2.3 *Other Parallel Models*

A large number of parallel computation models has been developed. We review here the three most representative ones: LogP, which is equivalent to BSP from an asymptotic point of view, but has a more sophisticated communication design; Cilk, which takes care of the load balancing, synchronization and communication by its runtime system, may handle the divide-and-conquer algorithms that the flat nature of BSP is not easily reconciled with; and Algorithmic skeletons, which have a higher level than the previous ones, for deterministic and deadlock-free parallel programming.

3.2.3.1 *The LogP Model*

LogP [CKP+93] is a distributed memory multiprocessor model where processors communicate by point-to-point messages. The model parameters are:

- **L**: an upper bound on the *latency* incurred in communicating a message containing a word (or small number of words) from its source module to its target module.
- **o**: the processor time *overhead* required to transmit or receive a message, during which the processor cannot perform other operations.
- **g**: the *gap*, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. $\frac{1}{g}$ corresponds to the available per-processor communication bandwidth.
- **P**: the number of processor/memory couples.

The term L, o and g parameters are measured as multiples of the processor cycle. In the LogP model, processors work asynchronously and at most $\lceil \frac{L}{g} \rceil$ messages can be in transit, on the network, at any time. In the LogP model, sending a small message (a datum) from one processor to another requires a time of $L + 2 \times o$. Sending a long message formed of k bytes, by point-to-point messages, requires sending $\lceil \frac{k}{w} \rceil$ in $2 \times o + (\lceil \frac{k}{w} \rceil - 1) \times \max\{g, o\} + L$ cycles, where w is the underlying message size of the machine.

The LogP model deals only with short messages. So, an extension of this model, named LogGP was described in [AISS95] to model small and long messages communication. LogGP extends LogP model where a G parameter is added. This parameter captures the bandwidth obtained for long messages and $\frac{1}{G}$ represents the available per processor communication bandwidth for long messages. Thus, sending a k byte message, in the LogGP model, requires $2 \times o + (k - 1) \times G + L$. Other LogP extensions, such as [LZE97], were also presented in the literature for the same goal.

BSP can be efficiently simulated by LogP and vice-versa. However, Bilardi et al. [BHP+96] claim that BSP is somewhat preferable to LogP due to its greater simplicity and portability. However, it was shown in [BHP+96, ELZ98], that from an asymptotic point of view, the two models are equivalent.

3.2.3.2 *Divide and Conquer – Cilk*

Divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until they become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. Divide and conquer algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.

Cilk[21] [BJK+95] is a C-based runtime system for algorithmic multithreaded programming developed at MIT. The philosophy behind Cilk is that a programmer should concentrate on structuring her or his program to expose

21 The Cilk Project. http://supertech.csail.mit.edu/cilk/

parallelism and exploit locality, leaving Cilk's runtime system with the responsibility of scheduling the computation to run efficiently on a given platform. The Cilk runtime system takes care of details such as load balancing, synchronization, and communication protocols. Cilk is algorithmic in that the runtime system guarantees efficient and predictable performance. With these features, the Cilk system is very suitable for developing divide and conquer algorithms.

The Cilk scheduler uses a policy called "work-stealing" to divide procedure execution efficiently among multiple processors. Each processor has a stack for storing frames whose execution has been suspended; the stacks are more like deques, in that suspended states can be removed from either end. A processor can only remove states from its own stack from the same end that it puts them on; any processor which is not currently working (having finished its own work, or not yet having been assigned any) will pick another processor at random, through the scheduler, and try to "steal" work from the opposite end of their stack — suspended states, which the stealing processor can then begin to execute. The states which get stolen are the states that the processor stolen from would get around to executing last.

```
cilk int fib (int n)
{
   if (n < 2) return n;
   else
   {
      int x, y;

      x = spawn fib (n-1);
      y = spawn fib (n-2);

      sync;

      return (x+y);
   }
}
```

Listing 4: A recursive implementation of the Fibonacci function in Cilk

Divide-and-conquer algorithms can be easily parallelized in Cilk. Listing 4 shows a recursive implementation of the Fibonacci function in Cilk. The *spawn*

keyword indicates that the called `fib` function can safely operate in parallel with other executing code. And the *sync* keyword indicates that execution of the current procedure cannot proceed until all previously spawned procedures have completed and returned their results to the parent frame.

3.2.3.3 *Algorithmic skeletons & MapReduce*

Algorithmic skeletons[22] are a high-level parallel programming model for parallel and distributed computing. M. Cole [Col89] and H. Kuchen have developed the paradigm of algorithmic skeletons for deterministic and deadlock-free parallel programming. Algorithmic skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Skeletons are akin to design patterns for parallel execution.

Table 3 defines the functional semantics of a set of data-parallel skeletons [Col04a, Alt07]. It can also be seen as a naive sequential implementation using lists. The skeletons work as follow:

$$
\begin{aligned}
\textbf{repl}\ x\ n &= [x, \ldots, x] \\
\textbf{map}\ f\ [x_1, \ldots, x_n] &= [(f\ x_1), \ldots, (f\ x_n)] \\
\textbf{mapidx}\ g\ [x_1, \ldots, x_n] &= [(g\ 1\ x_1), \ldots, (g\ n\ x_n)] \\
\textbf{zip}\ \oplus\ [x_1, \ldots, x_n]\ [y_1, \ldots, y_n] &= [x_1 \oplus y_1, \ldots, x_n \oplus y_n] \\
\textbf{reduce}\ \oplus\ [x_1, \ldots, x_n] &= x_1 \oplus \cdots \oplus x_n \\
\textbf{scan}\ \oplus\ [x_1, \ldots, x_n] &= [x_1, (x_1 \oplus x_2), \cdots, ((x_1 \oplus x_2) \cdots \oplus x_n)]
\end{aligned}
$$

Table 3: Simple data-parallel skeletons

- Skeleton **repl** creates a new list containing n times element x. Here we speak of lists for the specification but parallel implementations would use more efficient data-structures as arrays (e. g., in BSML) or a stream (e. g., in a client/server or grid environment) since the size of the lists remain constant.

- The **map**, **mapidx** and **zip** skeletons are equivalent to the classical *Single-Program-Multiple-Data* (SPMD) style of parallel programming, where a

22 Skeletal Parallelism homepage. http://homepages.inf.ed.ac.uk/mic/Skeletons/

single program f is applied on different data in parallel. Parallel execution is obtained by assigning a share of the input list to each available processor.

- **reduce** is an elementary data-parallel skeleton: the function **reduce** $\oplus$ e l computes the "sum" of all elements in a list l, using the associative binary operator $\oplus$ and its identity e. Reduction has traditionally been very popular in parallel programming and is provided as the collective operation `MPI_Reduce` in the MPI standard. Note that the binary operator $\oplus$ may itself be time-consuming. To parallelize the **reduce** skeleton, the input list is divided into sub-lists that are assigned to each processor. The processors compute the $\oplus$-reductions of their elements locally in parallel, and the local results are then combined either on a single processor or using a tree-like pattern of computation and communication, making use of associativity in the binary operator.

- The **scan** skeleton is similar to **reduce** (and is provided as the collective operation `MPI_Scan`), but rather than the single "sum" produced by **reduce**, **scan** computes the partial (prefix) sums for all list elements. Parallel implementation is done as for **reduce**.

Google's MapReduce [DG08] is a simplified version of algorithmic skeletons (only two of them: *map* and *reduce*) for processing large data sets. MapReduce is typically used to do distributed computing on clusters of computers. MapReduce provides regular programmers the ability to produce parallel distributed programs much more easily, by requiring them to write only simple *Map* and *Reduce* functions which focus on the logic of the specific problem at hand, while the *MapReduce System* automatically takes care of marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, providing for redundancy and failures, and overall management of the whole process.

Dean and Ghemawat stated that they have inspired their MapReduce model by Lisp and other functional languages. Users must implement two functions *Map* and *Reduce* having the following signatures:

$$\begin{aligned} &\textbf{map}: && (k_1, v_1) \longrightarrow \mathrm{list}(k_2, v_2), \\ &\textbf{reduce}: && (k_2, \mathrm{list}(v_2)) \longrightarrow \mathrm{list}(v_3). \end{aligned}$$

In **Map**, the master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem and passes the answer back to its master node. The *map* function must be written with two input variables, a key k_1 and an associated value v_1. Its output is a list of intermediate key/value pairs (k_2, v_2). This list is partitioned by the Map-Reduce library depending on the values of k_2, where all pairs having the same value of k2 belong to the same group.

In **Reduce**, the master node then collects the answers of all the sub-problems and combines them in some way to form the output — the answer to the problem it was originally trying to solve. The *reduce* function must be written with two input parameters: an intermediate key k_2 and a list of intermediate values $\mathtt{list}(v_2)$ associated with k_2. It applies the user-defined merge logic on $\mathtt{list}(v_2)$ and outputs a list of values $\mathtt{list}(v_3)$.

A MapReduce computation can be refined as 5 phases:

1. **Prepare the Map input**: the MapReduce system designates Map processors, assigns the K_1 input key value each processor would work on, and provides that processor with all the input data associated with that key value.
2. **Run the user-provided Map code**: Map is run exactly once for each K_1 key value, generating output organized by key values K2.
3. **"Shuffle" the Map output to the Reduce processors**: the MapReduce system designates Reduce processors, assigns the K_2 key value each processor would work on, and provides that processor with all the Map-generated data associated with that key value.
4. **Run the user-provided Reduce code**: Reduce is run exactly once for each K_2 key value produced by the Map step.
5. **Produce the final output**: the MapReduce system collects all the Reduce output, and sorts it by K_2 to produce the final outcome.

MapReduce has been written in many programming languages. Apache Hadoop[23] is a popular open-source implementation.

Many parallel programming models have been proposed today — for example, the multi-threaded concurrent programming is easy to use, but it can be

23 Apache Hadoop. http://hadoop.apache.org/

applied only on shared-memory architectures; the message-passage approach handles the distributed-memory architectures, but the management of communication is not an easy job; actor models provide patterns for the communication, but its application is too hard to optimise without any algorithm-machine bridging; the BSP bridging model links software and hardware, offers a sequential view of a parallel program with supersteps, simplifies algorithm design and analyse with the barrier, but more and more nowadays parallel computers are not developed in BSP-proposed flat structure but in a hierarchical architecture; MapReduce simplifies large data set processing on distributed cluster with implicit communication, but how it handles a complex algorithm with a good performance, is still a question. A new simple and realistic parallel programming model should be proposed. The next chapters introduce our programming and execution models for SGL, and motivate its suitability as a successor for existing BSP programming libraries/languages.

4

A NEW SIMPLE BRIDGING MODEL

4.1 Motivation 66

4.2 The SGL Model 70

4.2.1 The Abstract Machine 71

4.2.2 Execution Model 72

4.2.3 Cost Model 74

4.3 Case study: Modelling Parallel Computers 77

4.3.1 Modelling Multi-core Computers 77

4.3.2 Modelling Hierarchical Clusters 80

4.3.3 Modelling Heterogeneous Computers 82

We introduce, in this chapter, our Scatter-Gather parallel-programming and parallel execution model in the form of a simple imperative Scatter-Gather Language (SGL) [LH12a, LH11b]. Its design is based on past experience with Bulk Synchronous Parallel (BSP) programming and BSP language design [Val90, HMS+98, BJOR99, YBRM13, LGB05] (c. f., Section 3.2.2.2). SGL's novel features are motivated by the last decade move towards multi-level and heterogeneous parallel architectures (c. f., Section 2.2.1 and Section 2.3.2) involving multi-core processors, graphics accelerators and hierarchical routing networks in the largest multiprocessing systems. The design of SGL is coherent with L. Valiant's multi-BSP [Val11] (c. f., Section 3.2.2.3) while offering a programming interface that is even simpler than the primitives of bulk-synchronous parallel ML (BSML) [LGB05]. SGL appears to cover a large subset of all BSP algorithms [Tis99] while avoiding complex message-passing programming.

Like all BSP-inspired systems [HMS+98, BJOR99, YBRM13, LGB05, MAB+10], it supports predictable, portable, and scalable performance. Moreover, SGL's explicit data distribution allows automatic or programmable load-balancing.

4.1 MOTIVATION

Parallel programming and data-parallel algorithms have been the main techniques supporting high-performance computing for many decades. Like all non-functional properties of software, the conversion of computing resources into scalable and predictable performance involves a delicate balance of abstraction and automation with semantic precision.

From a programming point of view, paper [Adv09] gives a perspective on the collective work spanning for approximately 30 years. It shows how difficult it is to formalize the seemingly simple and fundamental property of "what value a read should return in a multi-threaded program". Safe languages must be given semantics that computer science graduates and developers can understand with a reasonable effort. The author of this survey believes that we need to rethink higher-level disciplines that make it much easier to write parallel programs and that can be enforced by our languages and systems. As we move toward more disciplined programming models, there is also a new opportunity for a hardware/software co-designed approach that rethinks the hardware/software interface and the hardware implementations of all concurrency mechanisms.

As the above remark highlights, multi-threaded *semantics* is far too complex for realistic software development. Yet parallel execution is synonymous with multiple processes or multi-threading, without which there can be no parallel speedup. So how should programmers avoid the complexity of multi-threaded programming and yet expect scalable performance? Part of the answer comes from the observation that the vast majority of parallel algorithms are deterministic. Along this line of reasoning, researchers such as M. Cole [Col89] and H. Kuchen [SK93, BK96] have developed the paradigm of *algorithmic skeletons* for deterministic and deadlock-free parallel programming. Skeletons are akin to design patterns for parallel execution. A large body of programming research literature supports the view that most if not all parallel application software should be based on families of algorithmic skeletons.

A deterministic and high-level parallel programming interface is indeed a major improvement over explicit message passing(c. f., Section 3.1.2). But the diminished expressive power of skeletons is not only an advantage. Unlike sequential equivalents, skeletons are not libraries in the classical sense because their host language (e. g., C) is necessarily less expressive than the language in which they are written (e. g., C+MPI). This is due to the lack of a base language that is not just Turing-complete but complete for parallel algorithms, a notion that has not even been well defined yet. As a result there is no fixed notion of a set of skeleton *primitives* but instead the message-passing primitives used to implement them. That feature of skeleton languages is similar to that of a language with automatic memory management: if the only concern is time complexity then a simpler automatic-memory language is sufficient to implement it; but if space complexity is to be explicit, then it is necessary to use an explicit-memory allocation language to implement the original one.

These remarks gave rise to our notion of *explicit processes*: without an explicit notion of the number of physical parallel processes, parallel speedup is not part of programming semantics. A language that is expected to express parallel algorithms must express not only a function from computation events to physical units (this function may not be injective), but also the inverse. In [HF93] G. Hains and C. Foisy introduced this notion of explicit processes through a deterministic parallel dialect of ML called DPML: the program's semantics is parametrized on the number of physical processes and their local indexes (processor ID's as they are often called). We conclude that DPML can serve as implementation language for skeletons, yet it remains a deterministic language.

Meanwhile, a major conceptual step was taken by L. Valiant [Val90] who introduced his Bulk-Synchronous Parallel (BSP) model (c. f., Section 3.2.2.2). Inspired by the complexity theory of PRAM model (c. f., Section 3.2.2.1) of parallel computers, Valiant proposed that parallel algorithms can be designed and measured by taking into account not only the classical balance between time and parallel space (hence the number of processors) but also communication and synchronization. A BSP computation is a sequence of so-called supersteps. Each superstep combines asynchronous local computation with point-to-point communications that are coordinated by a global synchronization to ensure coherence and deadlock-freedom. The resulting performance model is both realistic and tractable so that researchers such as McColl et al. [MW98] were able to define BSP versions of all important PRAM algorithms,

implement them and verify their portable and scalable performances as predicted by the model. BSP is thus a *bridging model* relating parallel algorithms to hardware architectures.

From the mid-1990's, it became clear that BSP is the model of choice for implementing algorithmic skeletons: its view of the parallel system included explicit processes and added a small set of network performance parameters to allow predictable performance. Our earlier language design DPML was still too flexible to be restrained to the class of BSP executions. G. Hains et al. then designed BS-lambda [LHF00] as a minimal model of computation with BSP operations. BS-lambda became the basis for Bulk-Synchronous ML (BSML) [Lou00] a variant of CAML under development by F. Loulergue et al. since 2000. BSML improves on DPML by offering a purely functional semantics, and much simplified programming interface of only four operations: `mkpar` to construct parallel vectors indexed by processors, `proj` to map them back to lists/arrays, `apply` to generate asynchronous parallel computation and `put` to generate communication and global synchronization from a two-dimensional processor-processor pairing. As a result, parallel performance mathematically follows from program semantics and the BSP parameters of the host architecture.

While BSML was evolving and practical experience with BSP algorithms was accumulating, one of its basic assumptions about parallel hardware was changing. The flat view of a parallel machine as a set of communicating sequential machines remains true but is more and more incomplete. Recent supercomputers (c. f., Section 2.2.1) like Blue Gene/L [ABB+03], Blue Gene/P [sD08], and Blue Gene/Q [HOF+12] feature multi-processors on one card, multi-core processors on one chip, multiple-rack clusters etc. The Cell/B.E. [KDH+05, JB07], Cell-based RoadRunner [BDH+08] and GPU's feature a CPU with Master-Worker architecture (c. f., Section 2.3). Moreover, [KTJR05] observes that heterogeneous chip multiprocessors present unique opportunities for improving system throughput and reducing processor consumption. The trend towards green-computing puts even more pressure on the optimal use of architectures that are not only highly scalable but hierarchical and non-homogeneous.

Towards the middle of the 2000's decade it was obvious that models such as BSP should be adapted or generalized to the new variety of architectures. Yet, programming simplicity and performance portability should be retained

as much as possible. With these goals in mind, Valiant introduced Multi-BSP [Val11] (c. f., Section 3.2.2.3) a multi-level variant of the BSP model and showed how to design scalable and predictable algorithms for it. The main new feature of Multi-BSP is its hierarchical nature with nested levels that correspond to physical architectures' natural layers. In that sense it preserves the notion of explicit processes and, as we will show in this section with our SGL design, allows to solve three pending problems with BSP programming:

1. The flat nature of BSP is not easily reconciled with divide-and-conquer parallelism [Hai98], yet many parallel algorithms (e. g., Strassen matrix multiplication, quad-tree methods etc.) are highly artificial to program any other way than recursively.

2. BSP was designed natively to scale with the number of processors, and it was assumed the clock speeds of CPU would continue to improve. However, there are barriers to further significant improvements in operating frequency due to voltage leakage across internal chip components and heat dissipation limits [SMD+10]. The parallelism is thus used to scale up nowadays computing power using multi-threaded cores, multi-core CPUs, Cell processor, GP-GPU, etc. We do not know how to design algorithms for nested level systems with BSP model. The hierarchical architectures share communication resources inside every level but not between different levels. The BSP cost model is not suitable for these kinds of architectures.

3. After teaching BSML programming at Universitié Paris-Est Créteil, we observed that many postgraduate[1] students can master the constructor **mkpar**, destructor **proj**, and asynchronous parallel **apply** while having much trouble with the general communication primitive **put**.

Based on all the above developments and the last three observations, we will now define the SGL model which realizes a programming model for Multi-BSP, generalizes it to heterogeneous systems and yet simplifies the BSML primitives to a set of three. SGL assumes a tree-structured machine and uses only the following parallel primitives:

scatter to send data from master to workers,

pardo to request asynchronous computations from the workers, and

1 French Master 1, fourth year higher education after Baccalauréat.

gather to collect data back to the master.

We show how to embed this model in an imperative language, how to define its operational semantics, its performance model, how to use it to code useful algorithms (reduce, parallel scan and parallel sort) and provide some initial measurements to validate its performance model.

4.2 THE SGL MODEL

The PRAM model [FW78] of parallel computation begins with a set of p sequential Von-Neumann machines (*processors*). It then connects them with a shared memory and an implicit global controller (as in a SIMD architecture). The BSP model [Val90] relaxes the global control and direct memory access hypotheses. It also begins with a set of p sequential machines but assumes asynchronous execution except for global synchronisation barriers and point-to-point communications that complete between two successive barriers. It is common to assume that the PRAM/BSP processors are identical, but it is relatively easy to adapt algorithms to a set of heterogeneous processors with varied processing speeds. In both cases the set of processors has no structure other than its numbering from 0 to $p-1$. This absence of structure on the set of processors can be traced back to the failure of a trend of popular research in the 1980s: algorithms for specific interconnect topologies such as hypercube algorithms, systolic algorithms, rectangular grid algorithms, etc. We trace this failure to the excessive variety of algorithms and architectures without a model to *bridge* the portability gap between algorithms and parallel machines.

The SGL model we propose does introduce a degree of topology on the set of processors. But this topology is purely hierarchical and is not intended to become an explicit factor of algorithm diversity. Just as PRAM/BSP algorithms use the p parameter to adapt to systems of all sizes, our SGL algorithms use the machine parameters to adapt to all possible systems.

4.2.1 *The Abstract Machine*

The abstract machine or an SGL Computer (SGLC) is defined as the combination of three attributes (Figure 27):

1. A set of **workers**, which are sequential processors composed of a computation element ("computing core") and local memory unit. The *workers* provide the primary computing power.

2. A set of **masters**, which are also sequential processors composed of a coordination element ("coordinating core") and central memory unit whose data is accessible by its own children via the coordinating core, i. e., the coordinating core can *scatter* (resp. *gather*) message to (resp. from) its children. The *masters* limit the communication cost.

3. A **tree** structure that the root is a *master* and the *master*'s children may be either *masters* themselves or leaf-*workers*. The number of children is unbounded so that the BSP/PRAM concept of a flat p-vector of processors is easily simulated in SGL. The *tree* structure offers a vertical scalability to SGLC that the BSP concept can scale only horizontally the number of processors.

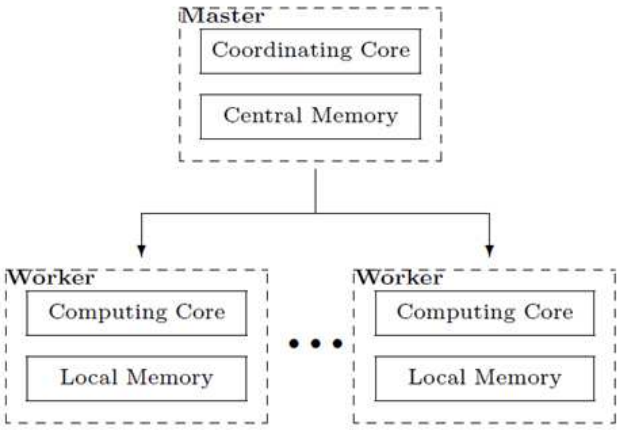

Figure 27: A 2-level SGL computer

Different forms of a SGLC are possible:

- A sequential machine can be modelled as only one worker without master.

- A flat parallel machine, or a BSP computer, can be modelled as a 1-level SGL computer (master + workers).
- A hierarchical machine of any shape can be modelled as a multi-level SGL computer (Figure 28).

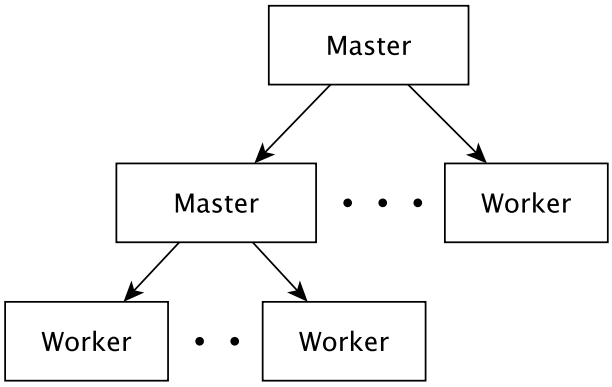

Figure 28: A multi-level SGL computer

The logical structure of an SGL computer satisfies the following constraints:

- A system shall have one and only one *root-master* (a rooted tree).
- A *master* coordinates its *children* through communication.
- A *worker* shall be controlled by one and only one *master* (*masters* can be replicated by underlying libraries for fault-tolerance).
- Communication is always between a *master* and its *children*.

Alexandre Tiskin has shown in his BSPRAM paper [Tis98] that communication through a higher-level memory (*master* node in our case) can be simulated by a horizontal inter-node direct communication as in the BSP model.

4.2.2 *Execution Model*

An SGL *program execution* is a sequence of *supersteps* (Figure 29). The initial computing data and final result can be either distributed in workers or centralised in the *root-master*. Each *superstep* is composed of four phases:

1. A **scatter communication** phase initiated by the *master*. The *master* scatters data to its *children* if the data is not yet distributed, then it engages its *children* to start the second phase.

2. An **asynchronous computation** phase performed by the *children*. The *children* execute the task initialized by their *master* in the first phase. A *child* can be either a *master* or a *worker*. The *workers* accomplish its *master*'s task; and the *child-master* can start a sequence of *supersteps* with its own children nested in this *child* computation phase so-called *sub-supersteps*.

3. A **gather communication** phase centred on the *master*. The *master* synchronizes its *children*'s task of the second phase, then gathers the computation results from its *children* if necessary.

4. A **local computation** phase on the *master*. The *master* post-processes the gathered data of the third phase and terminates the *superstep*.

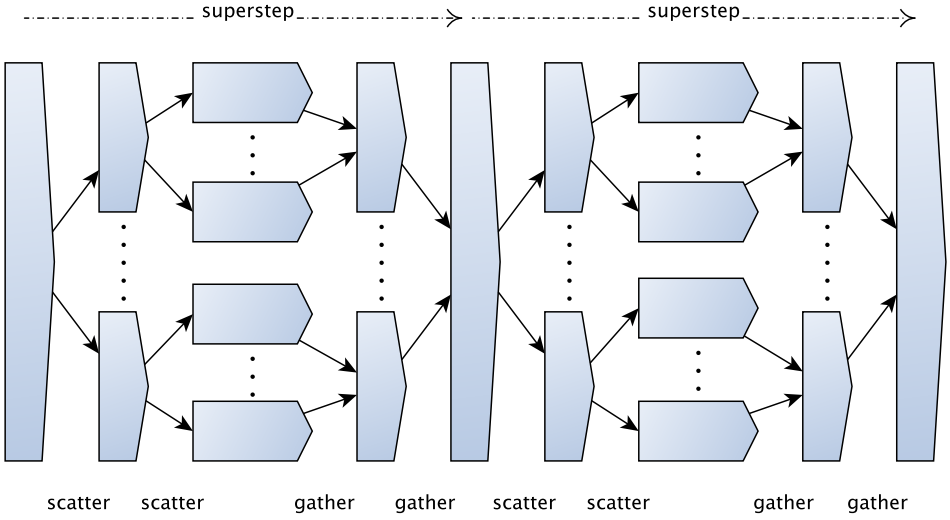

Figure 29: A 2-level SGL computation

With these four phases, we can usually write an SGL program recursively:

The choice of using (or not) children in the recursive program depends on performance parameters that combine (known) parameters of: communication costs, synchronization costs, computation costs, and load balancing.

```
if node is a master and its children are to be used then
    split input data into blocks;
    scatter data blocks;
    compute on individual data blocks in children (parallel);
    gather children's results;
    compute children results in master (if necessary);
else
    compute on local data on node;
end if
```

4.2.3 *Cost Model*

Like all such models, SGL's main goal of expressing parallel algorithms requires a precise notion of the execution time for a program. We give here the mathematical form of this *cost model* with the understanding that a full definition should be based on the operational semantics and will be defined in a further document. The cost equations below are nevertheless sufficient for informal algorithm design/understanding and comparison with other parallel models.

The cost of an SGL algorithm (i. e., of its execution on given input data) is the sum of the costs of its supersteps. That follows from the understanding that supersteps execute sequentially. The cost of an individual superstep if split into two independent terms: computation cost and communication cost.

$$\begin{aligned} \mathrm{Cost}_{\mathrm{total}} &= \sum \mathrm{Cost}_{\mathrm{supstep}} \\ \mathrm{Cost}_{\mathrm{supstep}} &= \mathrm{Comp}_{\mathrm{total}} + \mathrm{Comm}_{\mathrm{total}} \end{aligned}$$

Computation cost is the sum of local computation times in the children (parallel, hence combined with max) and of the local computation in the master. Addition of both terms realizes the hypothesis that the master's local work may not overlap with the children's work. Communication cost is split into the time for performing the scatter and the gather operation.

$$\begin{aligned}
Cost_{superstep} &= Comm_{scat} + \max_{i=children}\{Comp_i\} + Comm_{gath} + Comp_{master} \\
\\
Comm_{scatter} &= Words_{scat} \times Bandwidth_{scat} + Synchronization \\
Comm_{gather} &= Words_{gath} \times Bandwidth_{gath} + Synchronization \\
Comp_{master/child} &= Operations \times \frac{1}{Speed_{node}}
\end{aligned}$$

Finally, communication costs are estimated by a linear term similar to BSP's $g \times h + L$, based on machine parameters for (the inverse of) bandwidth and synchronization between the master and its children. As always, local computation cost is the number of instructions executed divided by processing speed. The machine-dependent parameters can be made as abstract or concrete as desired. In other words theoretical SGL algorithms can be investigated with respect to their asymptotic complexity classes, while portable concrete SGL algorithms can be analysed for more precise cost formulae using bytecode-like instruction counts and normalized communication parameters, and finally SGL programs can be compiled and measured for actual performance on a given architecture.

We remind the reader that, like the machine architecture, all the above cost formulae are intended to be recursive. The local computation cost on a child node can itself be an SGL algorithm cost if the machine structure is such.

In contrast to multi-BSP [Val11], concrete cost estimations and measurements use the following machine and algorithm parameters. Each level of a hierarchy can have different parameter values.

Concrete cost estimates and measurements use the following parameters:

- Machine parameters:
  - **p** : the number of *children* processors that a *master* has. We use P for the total number of leaf-*workers* of a machine.
  - **c** : computation speed of processors, c_0 denotes the time interval for performing a unit of *work* on the *master* and $c_i(i=1..p)$ denotes the time interval for performing a unit of *work* on *children* processor. Parameter c without an index refers to a local quantity.

  - **g** : the gap, $g_{\downarrow}$ defined as the minimum time interval for transmitting one word from *master* to its *children*, and $g_{\uparrow}$ for *children* to its *master*. We use a single g in case of symmetric communication cost.

  - **l** : the latency to perform a *gather* communication synchronization. i. e., the time to execute a 1-bit gather. Since a *scatter* communication should be initialized by the *master*, synchronisation is not necessary for *scatter* phase. We use L for the total cost of all-level synchronisations of one global superstep.

- Algorithm parameters:

  - **w** : work or number of local operations performed by processors, w_0 denotes the *master*'s work and $w_i(i=1..p)$ denote the work of a *child* i. A parameter w without an index refers to a local quantity.

  - **k** : number of words to transmit, $k_{\downarrow}$ denotes number of words that *master* scatter to its *children*, and $k_{\uparrow}$ denotes number of words that *master* gather from its *children*.

In general, the cost formulae are:

$$\begin{aligned} Cost_{Master} &= \overbrace{\max_{i=1..p}\{Cost_{child_i}\} + w_0 \times c_0}^{\text{computation}} + \overbrace{k_{\downarrow} \times g_{\downarrow} + k_{\uparrow} \times g_{\uparrow} + l}^{\text{communication}} \\ Cost_{Worker} &= w_i \times c_i \end{aligned}$$

which clearly covers the possibility of a heterogeneous architecture.

Typically, but not necessarily, we have symmetric communication costs:

$$Cost_{supstep} = w \times c + \underbrace{\left[\max_{i=1..p}\{Cost_{chd_i}\} + (k_{\downarrow} + k_{\uparrow}) \times g + l\right]}_{=0, \text{ when it is a worker (leaf).}}$$

The SGL cost model is experimented with the SGL programming model and its semantics in Chapter 5.

4.3 CASE STUDY: MODELLING PARALLEL COMPUTERS

As advocated by L. Valiant [Val11], any bridging model should be faithful to physical realities in terms of numerical parameters. Unlike BSP's flat architecture, SGL is hierarchical. It means, for modelling a physical machine, many combinations are possible. We thus attempt to analyse in this section how to link the SGL abstract machine to several typical parallel computers – from small multi-core microcomputer to big supercomputer cluster and grid.

4.3.1 *Modelling Multi-core Computers*

A multi-core computer is nowadays the most commonly used computer in all the fields. Even a laptop or a smart phone has a multi-core CPU today. A multi-core computer is typically a shared-memory architecture machine.

First of all, we try to model the simplest one, a mono-processor multi-core parallel computer. As we presented in Section 2.3.1.1, the Xeon Harpertown CPU used by Altix ICE 8200EX has four cores, each core has its own L1 cache, but each two cores share one L2 cache, and all four cores share the main memory. A parallel computer (Figure 30) using one Harpertown-based Xeon quad-core CPU can be modelled either as an one-level SGL computer with which we ignore the shared L2 cache, or a two-level SGL computer which groups the shared-cache pair cores in level 1.

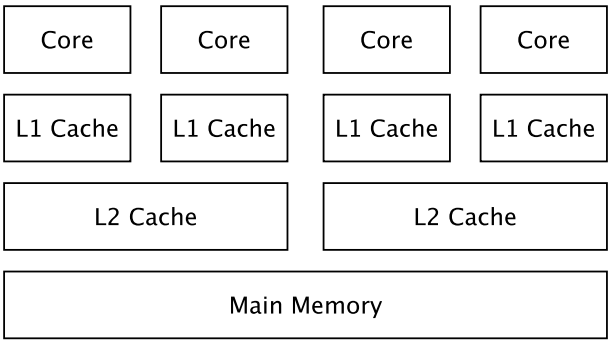

Figure 30: A multi-core computer with one Harpertown-based Xeon CPU

One of the advantages of one-level SGL computer is that it is equivalent to the BSPRAM model (c. f., Section 3.2.2.3), and the BSP model can be simulated

by BSPRAM efficiently [Tis98]. Many BSP algorithms such as ones presented in [Tis99] can be implemented directly on one-level SGL computer. However, the unified L2 Cache creates a memory affinity problem. The performance decreases significantly because of the cache sharing.

A two-level SGL computer takes consideration of memory sharing[2]. Meanwhile, it distinguishes g_1 for fetching data from L2 cache, and g_2 for fetching data from main memory. This separation may improve the communication cost prediction precision for algorithm design when the data exchange happens between the shared cache cores. Furthermore, the synchronisation cost l can be reduced from $O(2^{p_1+p_2})$ to $O(2^{p_1} + 2^{p_2})$ [3]. Thus, it is possible to design a more optimised algorithm on the two-level SGL computer than on the one-level one.

We then model the real Altix ICE 8200EX compute node, a multi-processor multi-core parallel computer. A 8200EX compute node contains 2 Harpertown-based Xeon quad-core CPUs that share the main memory (Figure 31).

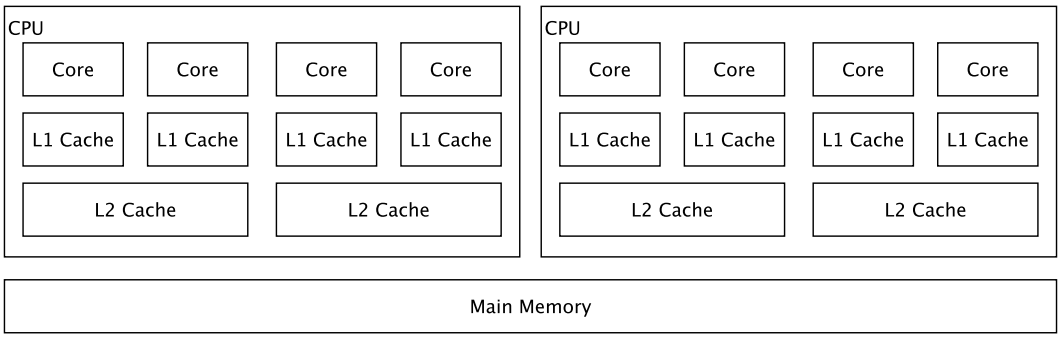

Figure 31: A 8200EX compute node with two Harpertown-based Xeon CPUs

In this case, shall we create a three-level SGL computer for the additional layer (CPU), or should keep the abstract machine in two-level? If the physical machine is modelled in two-level, which physical layer shall be merged? Let's compare these propositions:

1. **two-level SGL computer merged L1 cache and L1 cache layers**: It means the physical machine is modelled according its hardware coupling – the machine contains 2 CPUs, each CPU contains 4 cores. The synchronisation cost l is $O(2^{p_1} + 2^{p_2})$, where $P_1 = 4$ and $P_2 = 2$. However, the 4 cores

2 We do not distinguish memory and cache here.
3 Harpertown architecture has not hardware support for the synchronisation.

of one CPU do not share the same L2 cache, the communication cost g_1 is thus the cost of fetching data from main memory. Since g_1 is already the total cost from lowest level to highest level, g_2 shall be 0.

2. **two-level SGL computer merged L2 cache and main memory layers**: The physical machine is modelled according the physical memory sharing structure. The synchronisation cost l is $O(2^{p_1} + 2^{p_2})$, where $P_1 = 2$ and $P_2 = 4$. The communication costs g_1 and g_2 can be modelled the same way as we did for the one-Harpertown machine. We obtain thus the same architecture as the one-Harpertown machine but why more processors.

3. **three-level SGL computer**: This proposition takes account all layers. The synchronisation cost l becomes $O(2^{p_1} + 2^{p_2} + 2^{p_3})$, where $P_1 = 2$, $P_2 = 2$ and $P_3 = 2$; the communication costs g_1 and g_2 can be modelled as same as we did for the one-Harpertown machine, and g_3 shall be 0 because it does not increase the communication cost here. Thus, the processors are modelled in a binary-tree structure.

Proposition 1 is not practical, because it does not make known the real structure of the physical connection. Proposition 2 corresponds to the real physical connection of the machine. And Proposition 3 adds the CPU level which does not change the cost analysis; but thanks to the 8200EX architecture, it provides a binary-tree structure which can simplify algorithm design.

Programming with cache requires some special techniques. We measured here the Altix ICE 8200EX compute node only as one-level SGL computer. OpenMP's barrier is used for measuring the synchronisation cost l, and the C language's function *memcpy* [4] for the communication cost g [5]. Table 4 shows that the synchronisation cost l increases significantly when the number of cores increases. This confirms our claim that the cost of barrier for a large system may be reduced by a hierarchical architecture.

Many modern CPUs such as Nehalem-EP architecture which has unified L3 Cache shared by all four cores have this issue.

4 Here instead of transferring directly the pointers to data, we use memcpy() for replacing data in different memory regions to avoid concurrent access between CPU cores.

5 $g_\downarrow$ and $g_\uparrow$ are symmetric here.

# of cores	l (µs)	g (ns/32b)
2	12.08	0.59
4	25.64	0.59
6	37.80	0.59
8	52.00	0.59

Table 4: 8200EX compute node core-level machine parameters

4.3.2 *Modelling Hierarchical Clusters*

A typical supercomputer is composed of many multi-core compute components connected by a network in a distributed-memory architecture. Such a system could contain many hierarchical levels: Compute Card – Node Card – Rack – System, like IBM Blue Gene/L (c. f., Section 2.2.1.2); or Compute Node – IRU – Rack – System, like SGI Altix ICE (c. f., Section 2.2.1.3). The network is sometimes heterogeneous among different levels.

The EXQIM's Altix ICE system is configured on a fat-tree topology (Figure 32): all 16 Compute Nodes in the same IRU interconnect via the 4X DDR InfiniBand switches; all 2 IRUs in the same Rack interconnect via a "fatter" network[6]; and we have only one Rack in this system. Thus, we need only one more SGL level for modelling this network after 8200EX Compute Nodes. The distributed-memory nature can be simulated to the shared-memory architecture [Tis98].

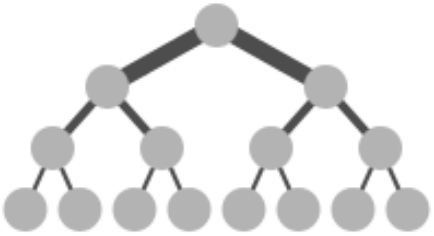

Figure 32: A binary fat-tree topology

6 Here all 32 Compute Nodes interconnect between them

When we used the collective functions *MPI_Barrier*, *MPI_Scatterv* and *MPI_Gatherv* of SGI's Message Passing Toolkit (MPT) to measure l, $g_{\downarrow}$ and $g_{\uparrow}$ for this level (Table 5) [7], we found that increasing the number of nodes aggravates not only the synchronisation cost but also the communication cost (Figure 33) in SGI MPT implementation. This approves again that the communication costs for a large system may be reduced by a hierarchical architecture; and the distinction of $g_{\downarrow}$ and $g_{\uparrow}$ may be useful in some cases.

# of procs	nodes × cores	l (µs)	$g_{\downarrow}$ (ns/32b)	$g_{\uparrow}$ (ns/32b)
2	= 2 × 1	1.48	1.38	2.15
4	= 4 × 1	2.85	1.69	2.00
8	= 8 × 1	4.37	1.89	2.05
16	= 16 × 1	5.96	2.04	2.09
32	*= 16 × 2*	*7.62*	*2.14*	*2.09*
64	*= 16 × 4*	*7.93*	*2.63*	*2.11*
96	*= 16 × 6*	*8.81*	*2.88*	*2.13*
128	*= 16 × 8*	*9.89*	*3.01*	*2.77*

Table 5: Altix ICE node-level machine parameters

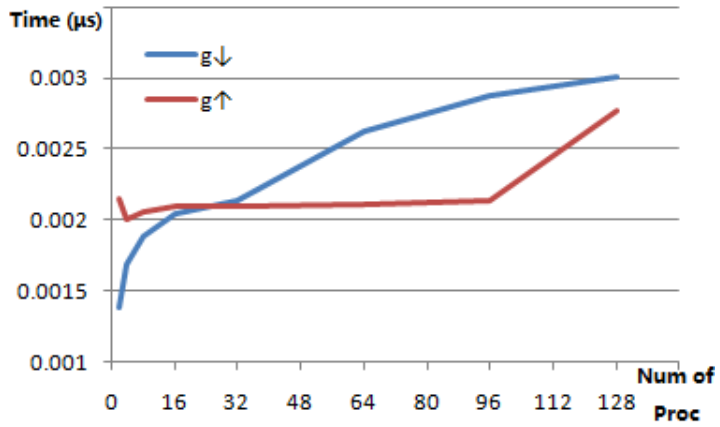

Figure 33: Measurement of $g_{\downarrow}$ and $g_{\uparrow}$ on Altix ICE IRU

A geographically distributed computational grid is often a set of supercomputers. Grid5000 (c. f., Section 2.2.2.1), for example, used originally 1Gbit/s the Ethernet Over MPLS (EoMPLS) solution for the inter-site connection; and

7 The last 4 lines in the table are only referential information.

most of the sites use InfiniBand or Myrinet technologies for inter-node connection (from 8 Gbit/s to faster than 100 Gbit/s). These characteristics require again a SGL level for modelling. More generally, a grid can be roughly modelled as a three-level SGL machine: the shared-memory multi-core compute nodes as level 1; the distributed-memory LAN-interconnected computer clusters as level 2; the distributed-memory WAN-interconnected grid system as level 3.

4.3.3 *Modelling Heterogeneous Computers*

As we presented in Section 2.3.2, more and more parallel computers have not only a hierarchical architecture, but also at the same time a heterogeneous architecture. GPGPU is the most used component for heterogeneous computers.

Nvidia's Fermi (c. f., Section 2.3.1.2) is the world's first complete GPU computing architecture [Gla09]. It contains 16 Streaming Multiprocessors (SMs) sharing the GPU memory; each SM is composed of 32 CUDA cores sharing uniform cache; each CUDA core can process multiple threads in parallel. A parallel computer equipped with 2 Fermi GPUs (Figure 34) can be modelled as a three-level SGL machine[8]:

- **level 1**: 32 CUDA cores communicated through the uniform cache;
- **level 2**: 16 SM processors communicated through the GPU memory;
- **level 3**: 2 GPUs communicated through the main memory.

IBM RoadRunner (c. f., Section 2.3.2.1) is one of the most sophistic heterogeneous supercomputers: the cluster is made up of 18 Connected Units (CU); each CU is composed of 15 Racks; each Rack contains 4 BladeCenters; each BladeCenter has 3 TriBlades; each TriBlade consists of 1 LS21 blade and 2 QS22 blades; each LS21 blade manages 2 dual-core Opteron CPUs; and each QS22 blade controls 2 8-SPE Cell processors. The CUs, Racks, BladeCenters, TriBlades are interconnected through InfiniBand (IB) switches; the LS21 and QS22 blades of the same TriBlade are interconnected via PCIe or HyperTransport links; each multi-core Opteron CPU or Cell processor has its own memory shared between the cores.

8 The CPU is used as controller of GPUs here.

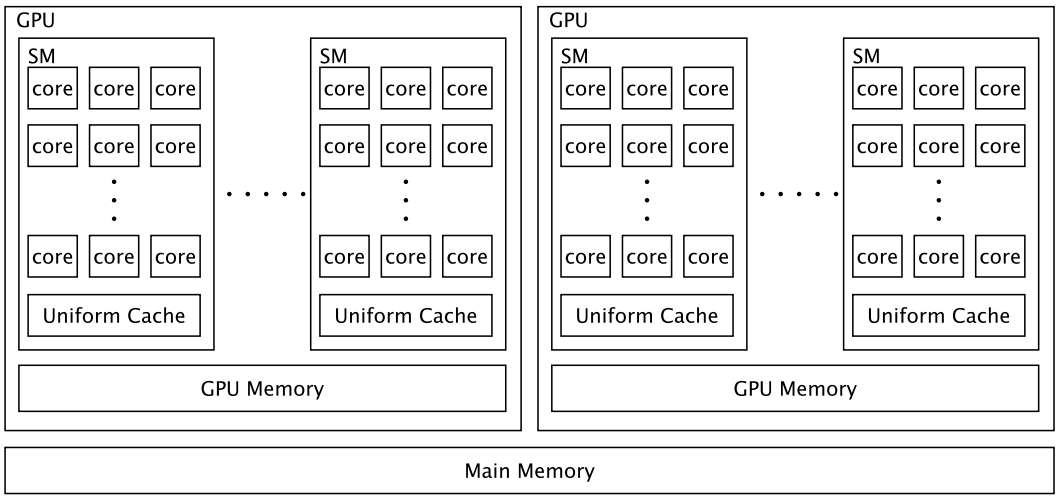

Figure 34: A parallel computer equipped with 2 Fermi GPUs

We propose here a 4-level SGL machine for modelling the RoadRunner:

- **Level 1**: is the 8-SPE Cell processor in shared-memory architecture. 8 SPEs of each Cell processor are interconnected via EIB. All Cell processors can communicate other components through its PCIe link.
- **Level 2**: is the 4-Cell TriBlade in distributed-memory architecture interconnected via PCIe links. One core of Opteron CPU of the same TriBlade is assigned to each CELL processor as the main controller, that to say 2 dual-core Opteron CPUs are assigned to 2 dual-CELL QS22 for each Triblade. Each TriBlade is equipped with an expansion blade to send/receive to/from others via IB.
- **Level 3**: is the 180-TriBlade CU in distributed-memory architecture interconnected via 288-port Voltaire IB switches. Each CU is equipped with a Panasas file system[9].
- **Level 4**: is the 18-CU cluster in distributed-memory architecture interconnected through the second-stage IB switches.

We believe that the SGL computer can cover most of the modern parallel computers. The synchronisation cost of SGL algorithms for a massively parallel computer can be greatly reduced by its hierarchical structure. The communication cost of SGL between different levels is more realistic than the

9 We do not distinguish memory and file system here.

flat structure. The SGL cost model is experimented in next chapter with SGL programming model and its operational semantics.

5

SGL PROGRAMMING

5.1 Programming Model of SGL 86
5.1.1 Language Syntax 86
5.1.2 Environments 88
5.1.3 Operational Semantics 88
5.2 Parallel Skeletons Implementation 94
5.2.1 Implementing Basic Data-Parallel Skeletons 95
5.2.1.1 Parallel Reduce 95
5.2.1.2 Parallel Scan 97
5.2.1.3 Parallel Sort 100
5.2.2 Implementing Distributable Homomorphism 104
5.2.2.1 Program Examples using DH 104
5.2.2.2 Implementation of DH programs 107
5.2.3 Speedup and Efficiency 110

We propose in this chapter the programming model of SGL and several skeleton implementations [LH12a, LGH12] to attempt to motivate and support the view that BSP's advantages for parallel software can be enhanced by the recursive hierarchical and heterogeneous machine structure of SGL, while simplifying the programming interface even further by replacing point-to-point messages with logically centralised communications. Our initial experiments with language definition, programming and performance measurement show that SGL combines clean semantics, simplified programming of BSP-like algorithms and dependable performance measurement.

This chapter is organized as follow: in Section 5.1, we present the language syntax, environment, and operational semantics; in Section 5.2, we experiment the SGL language with *reduce*, *scan*, *sort* and *DH* skeletons.

5.1 PROGRAMMING MODEL OF SGL

Bougé advocated in his paper [Bou96], that an abstract computing model needs an execution model, but also a programming model. Neither BSP [Val90] nor Mutli-BSP [Val11] propose a language implementation, but only the execution models.

We thus enrich Winskel's basic imperative language IMP [Win93] to yield our deterministic parallel programming language – SGL.

5.1.1 *Language Syntax*

Values are integers, booleans and arrays (vectors) built from them. Vectors of vectors are necessary for building blocks of work to be scattered among workers.

- n ranges over numbers **Nat**
- $< n_1, n_2, \ldots n_\ell >$ ranges over arrays of number **Vec**
  Here ℓ denotes length of array and $\ell \in \mathbb{N}at$
- $< v_1, v_2, \ldots v_\ell >$ ranges over arrays of array **VecVec**

The scatter operation takes a vector of vectors in the master and distributes it to workers/children. The gather operation inverts this process.

Imperative variables are abstractions of memory positions and are called *locations*. They are many-sorted like the language's values.

- X ranges over scalar locations **NatLoc**, i. e., names of memory elements to store numbers. Here $X_{i=pid}$ denotes *master/children* locations; X without index denotes *master* location.
- $\overrightarrow{V}$ ranges over vectorial locations **VecLoc**, i. e., names of memory elements to store arrays. Here $\overrightarrow{V}_{i=pid}$ denotes *master/children* locations; $\overrightarrow{V}$ without index denotes *master* location.

- $\widetilde{W}$ ranges over vectorial vectorial locations **VVecLoc**, i. e., names of memory elements to store arrays of arrays.

The basic operations are defined as follows:

- $\odot$ denotes binary operations such as $+, -, \times, /$.
- $[\,]$ denotes element access of a vector.
- $:=$ denotes data to be transferred between a *master* and its *child* (i. e., an array of arrays).

Expressions are relatively standard with the convenience of scalar-to-vector (sequential) operations.

- a ranges over scalar arithmetic expressions.
  Aexp $::= n \mid X \mid a \odot a \mid \overrightarrow{V}[a]$
- b ranges over scalar boolean expressions.
  Bexp $::=$ **true** $\mid$ **false** $\mid a = a \mid a \leqslant a \mid \neg b \mid b \wedge b \mid b \vee b$
- v ranges over vectorial expressions
  Vexp $::= < a_1, a_2, \ldots a_\ell > \mid \overrightarrow{V} \mid v \odot a \mid v \odot v \mid \widetilde{W}[a]$
- w ranges over vectorial vectorial expressions.
  VVexp $::= < v_1, v_2, ..v_\ell > \mid \widetilde{W}$

The language's commands include classical sequential constructs with SGL's three primitives: **scatter**, **pardo** and **gather**. Their exact meanings of parallel statements are defined in the semantic rules below in Section 5.1.3.

- c ranges over primitive commands.
  Com $::=$
  skip $\mid X := a \mid \overrightarrow{V} := v \mid \widetilde{W} := w \mid c\ ;\ c$
  $\mid$ **if** b **then** c **else** $c \mid$ **for** X **from** a **to** a **do** c
  $\mid$ **scatter** w **to** $\overrightarrow{V} \mid$ **scatter** v **to** X
  $\mid$ **gather** $\overrightarrow{V}$ **to** $\widetilde{W} \mid$ **gather** X **to** $\overrightarrow{V}$
  $\mid$ **pardo** $c \mid$ **if master then** c **else** c

- Auxiliary commands.
  Aux ::= **numChd** | **len** $\overrightarrow{V}$ | **len** $\widetilde{W}$

5.1.2 *Environments*

States (or environments) are maps from imperative variables (locations) to values of the corresponding sort. Like values they are many-sorted and we use the following notations and definitions for them.

The functions in States Σ are defined as follow:

- $\sigma : \mathrm{NatLoc} \to \mathrm{Nat}$, thus $\sigma(X) \in \mathrm{Nat}$
- $\sigma : \mathrm{VecLoc} \to \mathrm{Vec}$, thus $\sigma(\overrightarrow{V}) \in \mathrm{Vec}$
- $\sigma : \mathrm{VVecLoc} \to \mathrm{VecVec}$, thus $\sigma(\widetilde{W}) \in \mathrm{VecVec}$

Here $\mathrm{Pos} \in \mathrm{Nat}$ is what we call the (relative) *position*:
$\mathrm{Pos} = 0$ denotes *master* position (same as above), and $\mathrm{Pos} = i \in \{1..p\}$ denotes position in i^{th} child. It is the recursive analog of BSP's (or MPI's) `pids`.

- $\sigma : \mathrm{NatLoc} \to \mathrm{Pos} \to \mathrm{Nat}$, thus $\sigma(X_{pos}) = \sigma_{pos}(X) \in \mathrm{Nat}$
- $\sigma : \mathrm{VecLoc} \to \mathrm{Pos} \to \mathrm{Vec}$, thus $\sigma(\overrightarrow{V}_{pos}) = \sigma_{pos}(\overrightarrow{V}) \in \mathrm{Vec}$

5.1.3 *Operational Semantics*

The semantics of vector expressions is standard and deserves no special explanations.

- **Construction**:

$$\frac{\forall_{i=1..\ell} \langle a_i,\ \sigma\rangle \to n_i}{\langle < a_1, a_2, \ldots a_\ell >,\ \sigma\rangle \to < n_1, n_2, \ldots n_\ell >}$$

  A vector is constructed by enumerating all its elements: the elements'

expressions are evaluated one by one in a local environment, then the evaluated values are assembled into a vector.

- **Variable access**:

$$\langle \overrightarrow{V},\ \sigma\rangle \rightarrow \sigma(\overrightarrow{V}),\ \textit{a value of the form} < n_1, n_2, \ldots n_\ell >$$

The value of a variable is accessible from the local environment according to its location.

- **Vector variable access**:

$$\frac{\langle \widetilde{W},\ \sigma\rangle \rightarrow \sigma(\widetilde{W}),\ \textit{a value of the form} < v_1, v_2, \ldots v_\ell > \quad \langle a,\ \sigma\rangle \rightarrow n}{\langle \widetilde{W}[a],\ \sigma\rangle \rightarrow v_n}$$

To have the value of a element from a vector variable, we need to get the location of the vector from the local environment, and the indexation of the element evaluated from an expression in the same environment; then the value of the element is accessible according to the location and the indexation.

- **Vector-scalar operation**:

$$\frac{\begin{array}{ll}\langle v,\ \sigma\rangle \rightarrow < n'_1, n'_2, \ldots n'_\ell > & \\ \langle a,\ \sigma\rangle \rightarrow n & \forall_{i=1\ldots\ell}\langle n'_i \odot n,\ \sigma\rangle \rightarrow n_i\end{array}}{\langle v \odot a,\ \sigma\rangle \rightarrow < n_1, n_2, \ldots n_\ell >}$$

A binary vector-scalar operation takes two operands: a vector and a scalar. The result has the same length as the vector operand, and the value of each result vector element is evaluated from operating the value of vector operand's element in the same indexation as the result vector element and the scalar operand's value.

- **Vector-vector operation**:

$$\frac{\begin{array}{ll}\langle v_1,\ \sigma\rangle \rightarrow < n'_1, n'_2, \ldots n'_\ell > & \\ \langle v_2,\ \sigma\rangle \rightarrow < n''_1, n''_2, \ldots n''_\ell > & \forall_{i=1\ldots\ell}\langle n'_i \odot n''_i,\ \sigma\rangle \rightarrow n_i\end{array}}{\langle v_1 \odot v_2,\ \sigma\rangle \rightarrow < n_1, n_2 \ldots n_\ell >}$$

(Note: length of v_1 and length of v_2 shall be equal.)

A binary vector-vector operation takes two vector operands. These vector operands shall have the same length and the result has the same length too. The value of each result vector element is evaluated from operating the values of vector operands' elements in the same indexation as the result vector element.

Similarly for the rules of the other expressions.

Before defining the primitive commands, we define here some auxiliary commands. The first one is relative to the SGL machine structure.

- $\langle \textbf{numChd},\ \sigma \rangle \to n$
  Return number of children that the processor has.

- $\langle \textbf{len}\ \overrightarrow{V},\ \sigma \rangle \to n$
  Return the length of $\overrightarrow{V}$.

- $\langle \textbf{len}\ \widetilde{W},\ \sigma \rangle \to n$
  Return the length of $\widetilde{W}$.

The primitive commands are defined as follow:

- **Skip**

$$\langle \textbf{skip},\ \sigma \rangle \to \sigma$$

  A null statement which does nothing.

- **Assignments**

$$\frac{\langle a,\ \sigma \rangle \to n}{\langle X := a,\ \sigma \rangle \to \sigma[n/X]}$$

  The expression a is evaluated in the environment σ, then the value is stored in the location X.

$$\frac{\langle v,\ \sigma\rangle \rightarrow < n_1, n_2, \ldots n_\ell >}{\langle \overrightarrow{V} := v,\ \sigma\rangle \rightarrow \sigma[< n_1, n_2, \ldots n_\ell > / \overrightarrow{V}]}$$

The expression v is evaluated in the environment σ, then the value is stored in the location V.

$$\frac{\langle w,\ \sigma\rangle \rightarrow < v_1, v_2, \ldots v_\ell >}{\langle \widetilde{W} := w,\ \sigma\rangle \rightarrow \sigma[< v_1, v_2, \ldots v_\ell > / \widetilde{W}]}$$

Same as before, the expression w is evaluated in the environment σ, then the value is stored in the location W.

- **Sequencing**

$$\frac{\langle c_1,\ \sigma\rangle \rightarrow \sigma'' \quad \langle c_2,\ \sigma''\rangle \rightarrow \sigma'}{\langle c_1; c_2,\ \sigma\rangle \rightarrow \sigma'}$$

The command c_1 is evaluated in the initial environment σ that gives us a new environment σ'. After that, the command c_2 is then evaluated in this new environment σ'.

- **Conditionals**

$$\frac{\langle b,\ \sigma\rangle \rightarrow \textbf{true} \quad \langle c_1,\ \sigma\rangle \rightarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2,\ \sigma\rangle \rightarrow \sigma'}$$

$$\frac{\langle b,\ \sigma\rangle \rightarrow \textbf{false} \quad \langle c_2,\ \sigma\rangle \rightarrow \sigma''}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2,\ \sigma\rangle \rightarrow \sigma''}$$

The condition b is evaluated in the initial environment σ. If its value is `true`, then the command c_1 will be evaluated in the environment σ that give us a new environment σ'; otherwise, the command c_2 will be evaluated in the initial environment σ that give us a new environment σ''.

- **For-Loops**

$$\frac{\langle X := a_1,\ \sigma\rangle \to \sigma'' \quad \left\langle \begin{array}{l} \textbf{if } X \leqslant a_2 \\ \textbf{then } c;\ X := X + 1;\ \textbf{for } X \textbf{ from } X \textbf{ to } a_2 \textbf{ do } c \\ \textbf{else skip} \end{array} ,\ \sigma'' \right\rangle \to \sigma'}{\langle \textbf{for } X \textbf{ from } a_1 \textbf{ to } a_2 \textbf{ do } c,\ \sigma\rangle \to \sigma'}$$

The scalar expressions a_1 and a_2 are evaluated in the initial environment σ then the value of a_1 is stored in the scalar location X that gives us a new environment σ''. After that, the value in the location X and the value of a_2 are compared: if the value in X is bigger than the value of a_2, a null statement **skip** will be executed; otherwise, the command c will be evaluated in the environment σ'', then the value in the location X will be increased by 1, and a new for-loop statement (**for** X **from** X **to** a_2 **do** c) will be at the end evaluated.

- **Scatters**

$$\frac{\langle v,\ \sigma\rangle \to < n_1, n_2, \ldots n_p > \quad \forall_{i=1..numChd} \langle X_i := n_i,\ \sigma\rangle \to \sigma'_i}{\langle \textbf{scatter } v \textbf{ to } X,\ \sigma\rangle \to \sigma'}$$

The vector expression v is evaluated in the *master*'s local environment which is a part of the initial global environment σ to get the scalar value of each element of v; and the length of v shall be the same as the number of *children* processors that the *master* has. After that, the scalar value of each element of v is sent to a *child*'s environment that the position of the child is the same as the indexation of the sending element of v. We thus have a new global environment σ' composed of all new local environments of the master and its children. The communication cost of this statement is (p words $\times$ $g_\downarrow$).

$$\frac{\langle w,\ \sigma\rangle \to < v_1, v_2, \ldots v_p > \quad \forall_{i=1..numChd} \langle \overrightarrow{V}_i := v_i,\ \sigma\rangle \to \sigma'_i}{\langle \textbf{scatter } w \textbf{ to } \overrightarrow{V},\ \sigma\rangle \to \sigma'}$$

Same as the statement (**scatter** v **to** X) but for scattering vectorial vectorial expression. The communication cost of this statement is ($\sum$ number of words of v $\times$ $g_\downarrow$).

- **Gathers**

$$\frac{\langle \overrightarrow{V} :=< X_1, X_2, \ldots X_{\mathrm{numChd}} >,\ \sigma\rangle \rightarrow \sigma'}{\langle \textbf{gather } X \textbf{ to } \overrightarrow{V},\ \sigma\rangle \rightarrow \sigma'}$$

The values in location X on the *children* are sent to the *master*, then they are stored as a vector in the location $\overrightarrow{V}$ on the *master*. A synchronisation is presented here to ensure the reception of all values. The communication cost of this statement is (p words $\times\ g_{\uparrow} + l$).

$$\frac{\langle \widetilde{W} :=< \overrightarrow{V}_1, \overrightarrow{V}_2, \ldots \overrightarrow{V}_{\mathrm{numChd}} >,\ \sigma\rangle \rightarrow \sigma'}{\langle \textbf{gather } \overrightarrow{V} \textbf{ to } \widetilde{W},\ \sigma\rangle \rightarrow \sigma'}$$

Same as the statement (**gather** X **to** $\overrightarrow{V}$) but for gathering vectorial vectorial expression. The communication cost of this statement is ($\sum$ number of words of $v \times g_{\uparrow} + l$).

- **Parallel**

$$\frac{\forall_{i=1..\mathrm{numChd}} \langle c,\ \sigma_i\rangle \rightarrow \sigma'_i}{\langle \textbf{pardo } c,\ \sigma\rangle \rightarrow \sigma'}$$

Each *child* of the *master* evaluates independently the command c in its own local environment σ_i where the position $i = 1 \ldots p$ and p is the number of children. This statement costs $\max_{i=1..p}\{\mathrm{Cost}(c)_i\}$.

$$\frac{\langle \mathrm{numChd} = 0,\ \sigma\rangle \rightarrow \textbf{true} \quad \langle c_2, \sigma\rangle \rightarrow \sigma'}{\langle \textbf{if master then } c_1 \textbf{ else } c_2,\ \sigma\rangle \rightarrow \sigma'}$$

$$\frac{\langle \mathrm{numChd} = 0,\ \sigma\rangle \rightarrow \textbf{false} \quad \langle c_1, \sigma\rangle \rightarrow \sigma'}{\langle \textbf{if master then } c_1 \textbf{ else } c_2,\ \sigma\rangle \rightarrow \sigma'}$$

If the local environment is on a *master*, i. e., the number of children is zero in the local environment, the command c_1 will be evaluated; otherwise, the command c_2 will be evaluated.

The operational semantics of SGL can be used to design and validate its implementations. In Section 5.2 we tested standard skeletons programs in SGL

but not a full language. The semantics can also verify future compiler optimisations like the $\mathcal{G};\mathcal{P};\mathcal{S}$ (gather; process; scatter or GPS-) theorem presented in Section 6.1.1.

5.2 PARALLEL SKELETONS IMPLEMENTATION

There exist two kinds of algorithmic skeletons [KC02]: tasks and data-parallel ones. The former can capture parallelism that originates from executing several tasks, *i.e.* different function calls, in parallel. They mainly describe various patterns for organizing parallelism, including pipelining, farming, client-server, *etc.* The latter parallelize computation on a data structure by partitioning it among processors and performing computation simultaneously on different parts of it.

A well-know disadvantage of skeleton languages is that the only admitted parallelism is usually that of skeletons, while many parallel applications are not easily expressible as instances of known skeletons. Skeleton languages must be constructed as to allow the integration of skeletal and ad-hoc parallelism in a well defined way [Col04b]. In this light, having skeletons in SGL would combine the expressive power of collective communication patterns with the clarity of the skeleton approach.

The SGL operational semantics have been defined in Section 5.1.3. In this section, we consider the implementation of well-known data-parallel skeletons (c. f., Section 3.2.3.3) because they are simpler to use than task-parallel ones for coarse-grained models[1] and also because they encode many scientific computation problems and scale naturally. Even if SGL's implementation is certainly less efficient compared to a dedicated skeleton language (using MPI send/receive [FSCL06]), the programmer can compose skeletons when it is natural for him and use a SGL programming style when new patterns are needed.

The experimentation in this section was conducted on EXQIM's Altix ICE supercomputer (c. f., 2.2.1.3). This physical machine can be represented by a 2-level SGL computer (Figure 35). The SGL cost model machine parameters can be found in Section 4.3.1 and Section 4.3.2.

1 An efficient BSP implementation for those has nevertheless been shown in [GG11]

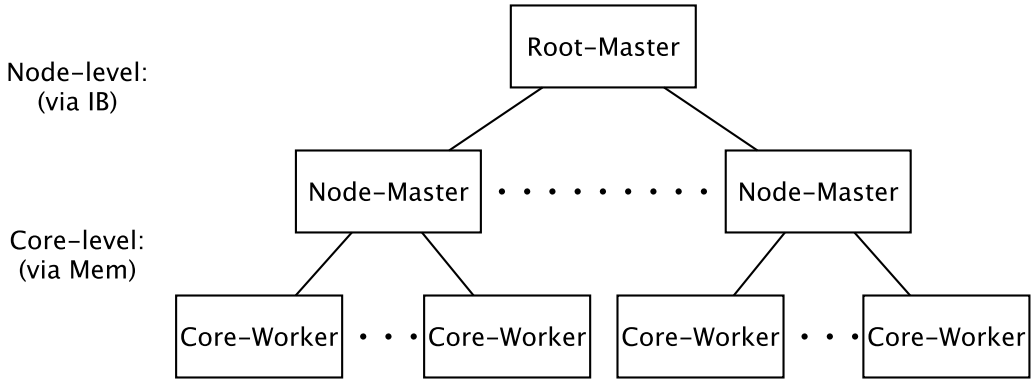

Figure 35: Altix ICE cluster modelled as a 2-level SGL computer

We tested, in this section, SGL variants of some of the most important basic parallel algorithms: parallel reduction (with the *product* operation, Section 5.2.1.1), parallel prefix reductions also called scan (with the *sum* operation, Section 5.2.1.2), a sorting algorithm (parallel sorting by regular sampling, Section 5.2.1.3) and a butterfly algorithm (distributable homomorphism, Section 5.2.2). For each one we wrote an SGL algorithm, implemented it by hand using the MPI/OpenMP operations mentioned in Section 4.3.1 and Section 4.3.2, and then compared the model's predicted vs observed run time for increasing data sizes. Finally, we computed speedup and parallel efficiency values (Section 5.2.3).

5.2.1 *Implementing Basic Data-Parallel Skeletons*

First of all, we tried to implement the 3 basic data-parallel skeletons: parallel reduce, parallel scan and parallel sort. We did not test parallel broadcast here because it can be applied naturally by *scatter*.

5.2.1.1 *Parallel Reduce*

Reduce is an elementary data-parallel skeleton. It computes the "sum" of all elements in a "list", with an associative binary operator. Reduction has traditionally been very popular in parallel programming. To parallelize the reduce skeleton, the input data is divided into subsets that are assigned to each pro-

cessor. The processors compute the reduce operations of their data locally in parallel, and the local results are then combined either on a single processor or using a tree-like pattern of computation and communication, making use of associativity in the binary operator.

In Algorithm 1, each *worker* computes the product of its local scalar numbers [Line 8]. After that, the master fetches the computed product from its *children* [Line 5] and calculates the final product [Line 6]. Comments on the right of the algorithm are cost estimations for the corresponding lines.

Algorithm 1 SGL implementation of parallel reduce

Reduce(IN $\overrightarrow{src}$, OUT res)
begin

```
1: if master then
2:     pardo
3:         Reduce(src→, res);                          ⟸ Reduce_child_i
4:     end pardo
5:     gather res to lst→;                             ⟸ p × g↑ + l
6:     Product(lst→, res) ;                            ⟸ O(p)
7: else
8:     Product(src→, res);                             ⟸ O(n)
9: end if
```

end

Line 3 is a recursive call to the algorithm and line 8, the no-children case, is a local sequential loop. Product operates on $\overrightarrow{src}$ whose elements are the res outputs on each child.

The cost of the supersteps is (subscripts $\langle\ \rangle$ refer to the pseudo-code line):

$$\begin{aligned} Cost_{Master} = &\ \max_{i=1..p}(Reduce_{Child_i\ \langle 3\rangle}) + O(p)_{\ \langle 6\rangle} \times c \\ &+ p_{\ \langle 5\rangle} \times g_{\uparrow} + l \\ Cost_{Worker} = &\ O(n_{worker})_{\ \langle 8\rangle} \times c \end{aligned}$$

The measurements (Figure 36) show a nearly perfect match of measured performance with predicted performance (obtained from the cost formula with machine parameters measured independently of this algorithm).

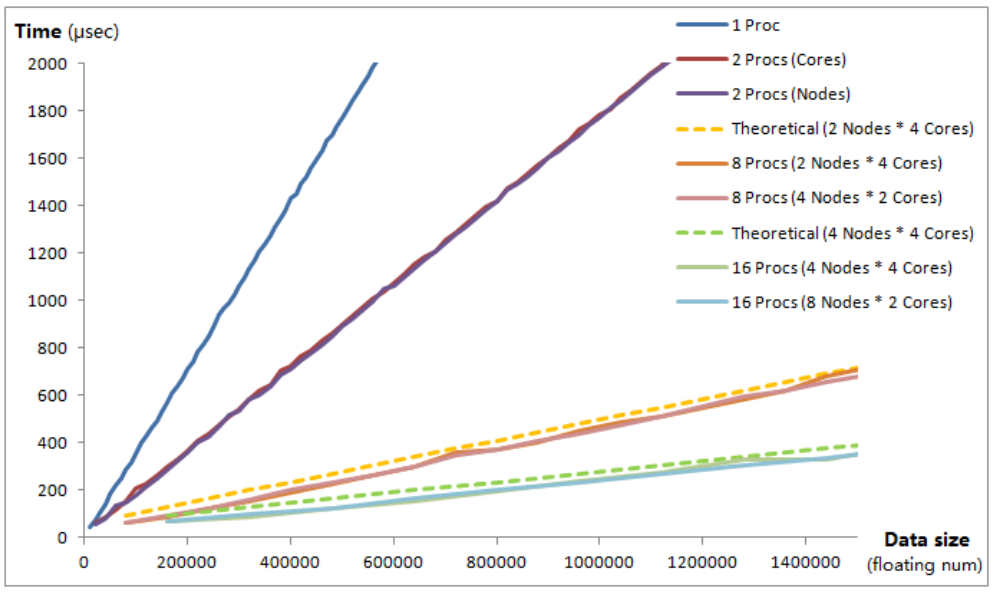

Figure 36: Predicted and actual execution times of SGL reduce

5.2.1.2 *Parallel Scan*

The *scan* skeleton is similar to *reduce*, but rather than the single "sum" produced by reduce, scan computes the partial (prefix) "sums" for all "list" elements. Parallel implementation is done as for reduce.

We perform the parallel scan algorithm in two steps (Algorithm 2):

1. Each *worker* performs a sequential scan [Line 10 of Step 1]. Then, the *master* fetches the last computed element from each *child* [Line 6 of Step 1]. After that, it performs a sequential scan of the fetched data [Line 8 of Step 1].

2. *Master* scatters the results among its *children* [Line 2 of Step 2]. Then, each *child* computes the sum of the received value and its computed data [Line 4 of Step 2]. Finally, the *children* obtain the final result.

Algorithm 2 SGL implementation of parallel scan

Step1(IN $\overrightarrow{src}$, OUT $\overrightarrow{mid}$)

begin

1: **if master then**
2: **pardo**
3: Step1($\overrightarrow{src}$, $\overrightarrow{mid}$); $\Longleftarrow Step1_{child_i}$
4: $x := \overrightarrow{mid}$[**len** $\overrightarrow{mid}$]; $\Longleftarrow O(1)_{child_i}$
5: **end pardo**
6: **gather** x **to** $\overrightarrow{mid}$; $\Longleftarrow p \times g_{\uparrow} + l$
7: ShiftRight($\overrightarrow{mid}$); $\Longleftarrow O(p)$
8: LocalScan ($\overrightarrow{mid}$, $\overrightarrow{mid}$); $\Longleftarrow O(p)$
9: **else**
10: LocalScan ($\overrightarrow{src}$, $\overrightarrow{mid}$); $\Longleftarrow O(n)$
11: **end if**

end

Step2(IN $\overrightarrow{mid}$, OUT $\overrightarrow{res}$)

begin

1: **if master then**
2: **scatter** $\overrightarrow{mid}$ **to** x; $\Longleftarrow p \times g_{\downarrow}$
3: **pardo**
4: $\overrightarrow{res} := \overrightarrow{mid} + x$; $\Longleftarrow O(n_{child_i})$
5: Step2($\overrightarrow{res}$, $\overrightarrow{res}$); $\Longleftarrow Step2_{child_i}$
6: **end pardo**
7: **else**
8: **skip**;
9: **end if**

end

The pseudo-code for parallel scan is given in Algorithm 2 and the cost of the supersteps is:

$$\begin{aligned} Cost_{Master} &= \max_{i=1..p}(Step1_{Child_i} + O(1) \times c_i) \\ &+ \max_{i=1..p}(Step2_{Child_i} + O(n_{Child_i}) \times c_i) \\ &+ 2 \times O(p) \times c \\ &+ p \times g_{\uparrow} + p \times g_{\downarrow} + l \\ Cost_{Worker} &= O(n_{worker}) \times c \end{aligned}$$

Again the pseudo-code and measurements (Figure 37) support our claims that SGL is simple to use and that its performance model is reliable. Several observations can be made from this graph: the performance prediction is correct to within a few percent, the discrepancy between prediction and observation increases with communication cost or with the number of processing units. Moreover, configurations with more cores per processor exhibit a larger variability due to the local shared memory.

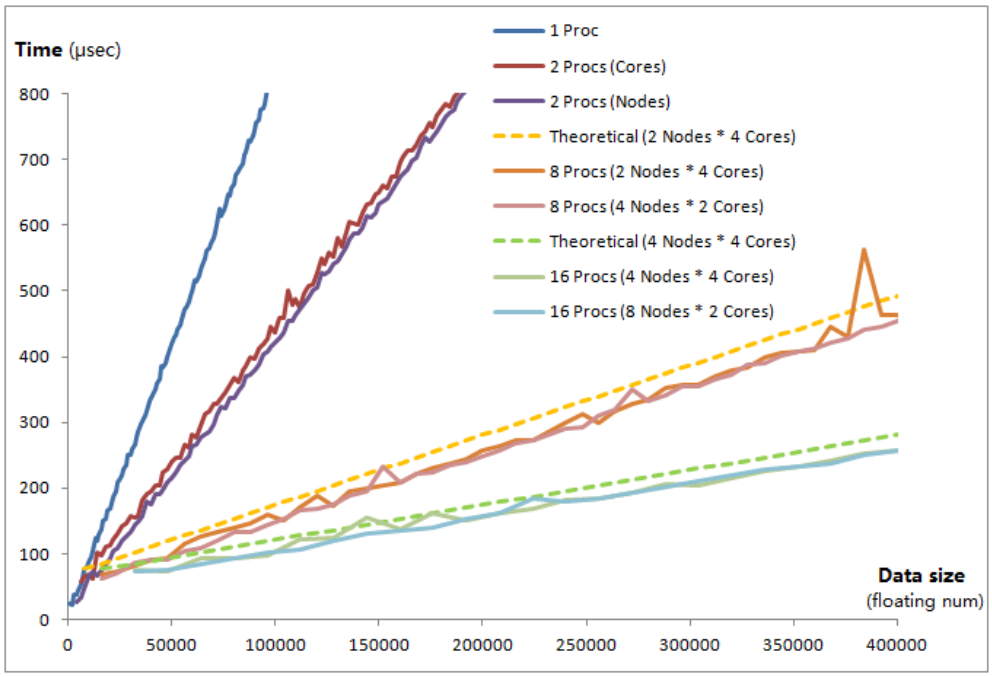

Figure 37: Predicted and actual execution times of SGL scan

We now experiment with a BSP sorting algorithm in SGL.

5.2.1.3 *Parallel Sort*

Our parallel sorting algorithm is based on Parallel Sorting by Regular Sampling(PSRS) [SS92] a *partition-based* algorithm. A. Tiskin proposed an implementation of this algorithm in BSP [Tis99]. We could also use the algorithm of [HJB98] to improve the performance.

We implement this algorithm in 5 steps (Algorithms 3 and 4):

1. Each *worker* performs a local sort [Line 8 of Step 1] and selects p (number of workers) samples [Line 9 of Step 1] which are gathered onto *root-master* [Line 5 of Step 1].

2. *Root-master* performs a local sort of the p^2 gathered samples [Line 1 of Step 2]. Then, it picks p-1 almost-equally spaced pivots from the sorted samples [Line 2 of Step 2].

3. *Root-master* broadcasts these pivots to all *workers* [Line 5 of Step 3]. After that, each *worker* produces p partitions of its local data using the p-1 pivots [Line 10 of Step 3]. Partition i holds values that should now be moved to *worker* i.

4. *Master* gathers the partitions which are not already in place [Line 6 of Step 4].

5. *Master* scatters the gathered partitions to its *children* according to partitions' index [Line 2 of Step 5]. Then, each *worker* performs a local merge of the received partitions [Line 11 of Step 5].

[Suj96] showed that the computational cost of this algorithm is $2\frac{n}{p}(\log n - \log p + \frac{p^3}{n}\log p)$, and the communication cost is $g\frac{1}{p}(p^2(p-1)+n)+4l$ in BSP. Thus, the total cost of this algorithm implemented in SGL is:

$$2\frac{n}{p}(\log n - \log p + \frac{p^3}{n}\log p) \times c$$

$$+\frac{1}{p}(p^2(p-1)+n) \times G + 2 \times L$$

Algorithm 3 SGL implementation of Parallel Sorting by Regular Sampling (1)

Stepo

1: initialize pid := 1 in each *worker*
2: use parallel scan to computer the pid for each *worker*
3: set lowerPid, upperPid and maxPid in *masters*

Step1(IN $\overrightarrow{arr}$, OUT $\overrightarrow{sam}$)
begin
1: **if master then**
2: **pardo**
3: Step1($\overrightarrow{arr}$, $\overrightarrow{sam}$);
4: **end pardo**
5: **gather** $\overrightarrow{sam}$ **to** $\widetilde{tmp}$;
6: $\overrightarrow{sam}$:= Concatenate($\widetilde{tmp}$);
7: **else**
8: QuickSort($\overrightarrow{arr}$);
9: SelectSamples($\overrightarrow{arr}$, $\overrightarrow{sam}$);
10: **end if**
end

Step2(IN $\overrightarrow{sam}$, OUT $\overrightarrow{pvt}$)
begin
1: QuickSort($\overrightarrow{sam}$)
2: PickPrivots($\overrightarrow{sam}$, $\overrightarrow{pvt}$);
end

Step3(IN $\overrightarrow{arr}$, IN $\overrightarrow{pvt}$, OUT $\widetilde{blk}$)
begin
1: **if master then**
2: **for** i **from** 1 **to** numChd **do**
3: $\widetilde{tmp}$[i] := $\overrightarrow{pvt}$;
4: **end for**
5: **scatter** $\widetilde{tmp}$ **to** $\overrightarrow{pvt}$;
6: **pardo**
7: Step3($\overrightarrow{arr}$, $\overrightarrow{pvt}$, $\widetilde{blk}$);
8: **end pardo**
9: **else**
10: BuildPartitions($\overrightarrow{arr}$, $\overrightarrow{pvt}$, $\widetilde{blk}$);
11: **end if**
end

Algorithm 4 SGL implementation of Parallel Sorting by Regular Sampling (2)

Step4(IN $\widetilde{\texttt{blk}}$, OUT $\widetilde{\texttt{stay}}$, OUT $\widetilde{\texttt{move}}$)

begin

1: **if master then**
2: **pardo**
3: Step4($\widetilde{\texttt{blk}}$, $\widetilde{\texttt{stay}}$, $\widetilde{\texttt{move}}$);
4: **end pardo**
5: **for** i **from** 1 **to** maxPid **do**
6: **gather** $\widetilde{\texttt{move}}$[i] **to** $\widetilde{\texttt{tmp}}$;
7: **if** i $<$ lowerPid or i $>$ upperPid **then**
8: $\widetilde{\texttt{move}}$[i]) := Concatenate($\widetilde{\texttt{tmp}}$);
9: **else**
10: $\widetilde{\texttt{stay}}$[i] := Concatenate($\widetilde{\texttt{tmp}}$);
11: **end if**
12: **end for**
13: **else**
14: **for** i **from** 1 **to** maxPid **do**
15: **if** i $=$ pid **then**
16: $\widetilde{\texttt{stay}}$[pid] := $\widetilde{\texttt{blk}}$[pid];
17: **else**
18: $\widetilde{\texttt{move}}$[i] := $\widetilde{\texttt{blk}}$[i];
19: **end if**
20: **end for**
21: **end if**

end

Step5(IN $\widetilde{\texttt{stay}}$, IN $\widetilde{\texttt{move}}$, OUT $\overrightarrow{\texttt{arr}}$)

begin

1: **if master then**
2: **scatter** Bundle($\widetilde{\texttt{move}}$) **to** $\overrightarrow{\texttt{arr}}$;
3: **pardo**
4: $\widetilde{\texttt{tmp}}$:= Unbundle($\overrightarrow{\texttt{arr}}$);
5: **for** i **from** lowerPid **to** upperPid **do**
6: $\widetilde{\texttt{move}}$[i] := Concatenate($\widetilde{\texttt{tmp}}$[i], $\widetilde{\texttt{stay}}$[i]);
7: **end for**
8: Step5($\widetilde{\texttt{stay}}$, $\widetilde{\texttt{move}}$, $\overrightarrow{\texttt{arr}}$);
9: **end pardo**
10: **else**
11: $\overrightarrow{\texttt{arr}}$:= MergeSort($\widetilde{\texttt{move}}$[pid]);
12: **end if**

end

where G is the sum of all g from each level and L is the sum of all l from each level.

Our performance prediction is based on $SeqSort(n) = O(n \log n)$ for the sequential sorting algorithm and $O(n/p)$ for the data to be transferred from one node to another node after local sort. Our implementation includes a node-to-node horizontal communication optimization which we have begun to investigate formally (see Chapter 6) and will be added to a SGL compiler.

The measurements shown in Figure 38 lead to the same observations and explanations as above for parallel scan. Higher local computation costs lead to a larger prediction error, in particular our theoretical estimate is *lower* than observation.

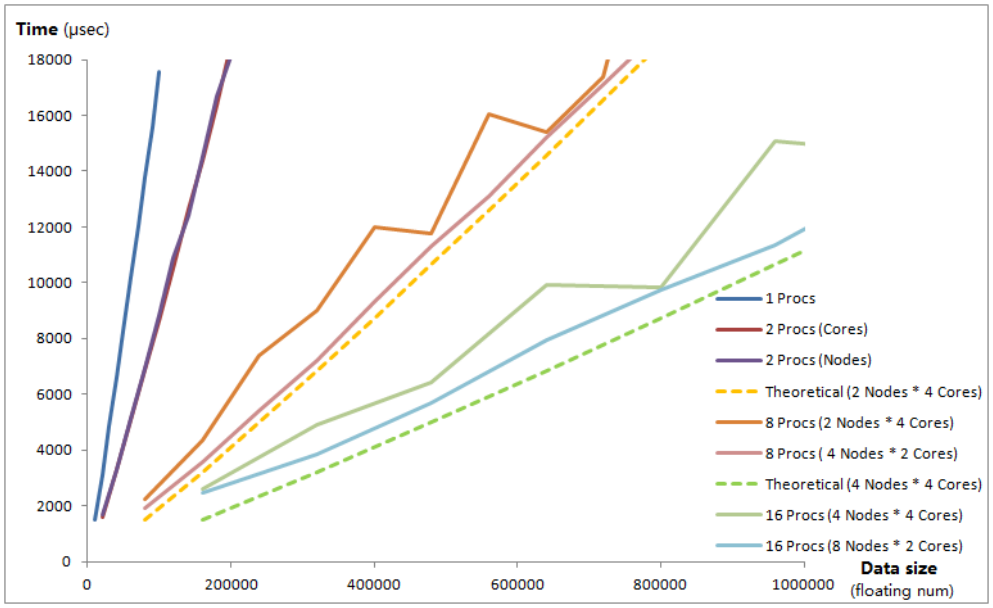

Figure 38: Predicted and actual execution times of SGL sort

The pseudo-code shows that our primitive algorithms are easily implemented in SGL. Future work will investigate SGL programming for more general algorithms.

The three tests illustrate SGL's relative programming simplicity on a relevant set of algorithms, convincing proof of the cost model's quality and initial evidence that SGL programming has no intrinsic cost overhead.

5.2.2 *Implementing Distributable Homomorphism*

Reduce and *scan* are basic skeletons, we study here a more complex data-parallel skeleton: the *Distributable Homomorphism* (DH) [Gor96], which may be used to express a special class of divide-and-conquer algorithms. **dh** $\oplus \otimes l$ transforms a list $l = [x_1, \cdots, x_n]$ of size $n = 2^m$ into a result list $r = [y_1, \cdots, y_n]$ of the same size, whose elements are recursively computed as follows:

$$y_i = \begin{cases} u_i \oplus v_i & \text{if } i \leqslant \frac{n}{2} \\ u_{i-\frac{n}{2}} \otimes v_{i-\frac{n}{2}} & \text{otherwise} \end{cases}$$

where $u = \textbf{dh}\ \oplus \otimes [x_1, \ldots, x_{\frac{n}{2}}]$, *i.e.* **dh** applied to the left half of the input list l and $v = \textbf{dh}\ \oplus \otimes [x_{\frac{n}{2}+1}, \ldots, x_n]$, *i.e.* **dh** applied to the right half of l. The **dh** skeleton provides the well-known butterfly pattern of computation which can be used to implement many computations.

5.2.2.1 *Program Examples using DH*

In this section, we give some examples of classical parallel numerical computations that can be performed using the skeletons presented above.

As first application example, we consider the solution of the *Tridiagonal System Solver* (TDS) of equations [Alt07]: $A.x = b$ where A is a $n \times n$ sparse matrix representing coefficients, x a vector of unknowns and b a right-hand-side vector. The only values of matrix A different from 0 are on the main diagonal, as well as directly above and below it – we call them the upper- and lower diagonal, respectively.

We represent the TDS as a list of rows, each inner row consisting of four values (a_1, a_2, a_3, a_4): the value a_1 that is part of the lower diagonal of matrix A, the value a_2 at the main diagonal, the value a_3 at the upper diagonal, and the value a_4 that is part of the right-hand-side vector b. The first and last row of A only contain two values, but in order to obtain a consistent representation we set $a_1 = 0$ for the first and $a_3 = 0$ for the last row. This corresponds to adding a column of zeros at the left and right of matrix A.

We now use the **dh** skeleton to parallelize the problem as a divide-and-conquer parallel algorithm. Since in the conquer phase, two subsystems can

be combined using the first and last row of the systems, our implementation works on triples (a, f, l) of rows, containing for each initial input row the actual row value a, and the first f and the last l row of the subsystem the row is part of. Using this list representation, the algorithm can be expressed as follows:

$$(\mathrm{TDS}\, m) = \mathbf{map}\, \pi_1\, (\mathbf{dh}\ \oplus\ \otimes\, (\mathbf{map}\, triple\, m))$$

where

$$\begin{pmatrix} a_1 \\ f_1 \\ l_1 \end{pmatrix} \oplus \begin{pmatrix} a_2 \\ f_2 \\ l_2 \\ t_2 \end{pmatrix} = \begin{pmatrix} a_1 \diamond (l_1 \star f_2) \\ f_1 \diamond (l_1 \star f_2) \\ (l_1 \circ f_2) \bullet l_2 \end{pmatrix}$$

$$\begin{pmatrix} a_1 \\ f_1 \\ l_1 \end{pmatrix} \otimes \begin{pmatrix} a_2 \\ f_2 \\ l_2 \\ t_2 \end{pmatrix} = \begin{pmatrix} (l_1 \circ f_2) \bullet a_2 \\ (f_1 \diamond (l_1 \star f_2) \\ (l_1 \circ f_2) \bullet l_2 \end{pmatrix}$$

and where

$$\begin{aligned}
a \star b &= (a_1, a_3 - (\frac{a_2}{b_1}) \times b_2, b_3 \times (-\frac{a_2}{b_1}), a_4 - (\frac{a_2}{b_1}) \times b_4) \\
a \diamond b &= (a_1 - (\frac{a_3}{b_2}) \times b_1, a_2, (-\frac{a_3}{b_2}) \times b_3, a_4 - (\frac{a_3}{b_2}) \times b_4) \\
a \circ b &= (a_1, a_2 - (b_1 \times \frac{a_3}{b_2}), (-b_3 \times a_3/b_2), a_4 - (\frac{a_3}{b_2}) \times b_4) \\
a \bullet b &= (a_1, -(\frac{a_2}{b_1}) \times b_2, a_3 - (b_3 \times \frac{a_2}{b_1}), a_4 - \frac{b_4 \times a_2}{b_1})
\end{aligned}$$

and where

$$a = (a_1, a_2, a_3, a_4)$$

$$b = (b_1, b_2, b_3, b_4)$$

which are row operations in a Gaussian elimination.

The **dh** method works as follows. In the divide phase, the matrix is subdivided into single rows. The conquer phase starts by combining neighbouring rows, applying first $\star$ (resp. $\circ$) and then $\diamond$ (resp. $\bullet$), which results in systems of two equations each, with non-zero elements in the first column, the main diagonal and the last column. The sub-matrices are then combined into matrices of four rows with the same structure, *i.e.*, where all non-zero elements are either on the diagonal, or in the first or last column. This process continues until, finally, the entire system of equations has this form. Note that the first

and last column of the matrix remain zero throughout the process, thus the the solution for the initial system of equations.

Combining two subsystems is achieved using a special row, obtained from the last row l of the first system and the first row f of the second one, using operator $\star$ (resp. $\circ$). This row is applied using operator $\diamond$ (resp. $\bullet$) to each row of the first system. Similarly, the rows of the second subsystem are adjusted using another special row obtained from the first and last rows.

The *Fast Fourier Transform* (FFT) of a list $x = [x_0, \ldots, x_{n-1}]$ of length $n = 2^m$ yields a list whose ith element is defined as:

$$(\mathrm{FFT}\, x)_i = \sum_{k=0}^{n-1} x_k \omega_n^{ki}$$

where ω_n denotes the nth complex root of unity $e^{2\pi\sqrt{-1}/n}$.

The FFT can be expressed in a divide-and-conquer form:

$$(\mathrm{FFT}\, x)_i = \begin{cases} (\mathrm{FFT}\, u)_i \oplus_{i,n} (\mathrm{FFT}\, v)_i & \text{if } i < \dfrac{n}{2} \\ (\mathrm{FFT}\, u)_j \otimes_{j,n} (\mathrm{FFT}\, v)_j & \text{otherwise} \end{cases}$$

where $u = [x_0, x_2, \ldots, x_{n-2}]$, $v = [x_1, x_3, \ldots, x_{n-1}]$, $j = i - \frac{n}{2}$, and $a \oplus_{i,n} b = a + \omega_n^i b$ and $a \otimes_{j,n} b = a - \omega_n^j b$. This formulation is close to the **dh** skeleton except $\oplus_{i,n}$ and $\otimes_{j,n}$ being parametrized with i and n.

These operators repeatedly compute the roots of unity. Instead of computing them for every call, they can be computed once *a priori* and stored in a list $\Omega = [\omega_n^1, \ldots, \omega_n^{\frac{n}{2}}]$ accessible by both operators. For this, we first use a **scan**. FFT can thus be expressed as follow:

$$\begin{aligned} (\mathrm{FFT}\, l) \;=\; & \mathbf{let}\, \Omega = \mathbf{scan}\, +\, 1\, (\mathbf{repl}\, (\omega\, n)\, \frac{n}{2}) \\ & \mathbf{in}\ \mathbf{map}\, \pi_1\, (\mathbf{dh}\, \oplus\, \otimes\, (\mathbf{mapidx}\, \mathrm{triple}\, l)) \end{aligned}$$

where

$$\begin{pmatrix} x_1 \\ i_1 \\ n_1 \end{pmatrix} \oplus \begin{pmatrix} x_2 \\ i_2 \\ n_2 \\ t_2 \end{pmatrix} = \begin{pmatrix} x_1 \oplus_{n_1}^{i_1} x_2 \\ i_1 \\ 2n_1 \end{pmatrix}$$

and $\otimes$ is defined similarly.

The first element of each triple contains the input value, the second one its position and the last one the current list length. In each **dh** step, these operators are applied element-wise to two lists of length $n_1 = n_2$, resulting in a list of length $2n_1$. $x_1 \oplus^{i_1}_{n_1} x_2$ (resp. for $\otimes$) is defined as $x_1 + x_2 \times (\mathrm{th}\,(n \times \frac{i_1}{n_2})\,\Omega)$ where $\mathrm{th}\,n\,[x_1, \ldots, x_n, \ldots] = x_n$.

5.2.2.2 *Implementation of DH programs*

We have implemented the **dh** skeleton using SGL which maps its model of execution since they are both recursive ones. We have then applied it to TDS and FFT.

All data is distributed over p *workers* and the algorithm performs recursively as follows (Algorithm 5): first of all, each *worker* performs a sequential **dh** with its own local data [Lines 22-31]; then, the *master* gathers the computed data [Line 6], permutes them according to the position [Lines 7-11], and scatters the permuted data to the *workers* [Line 12]; after that, each *worker* performs either $\oplus$ operation or $\otimes$ operation according to its position [Lines 13-19]. After $\log(p)$ times above achievements, we obtain the final result.

In the pseudo code, line 3 is a recursive call to the algorithms, lines 14 - 18 are executed in parallel, and lines 22 - 31, the no-children case, represent a local sequential loop. The cost of the superstep is below:

$$\begin{aligned} \mathrm{Cost}_{Master} &= \max_{i=1..p}(\mathrm{DH}_{\mathrm{Child}_i}) + \\ &\quad \log(p) \times (2^n \times (g_\uparrow + g_\downarrow) + l \\ &\quad + 2^n \times \max\{c_\oplus, c_\otimes\}) \end{aligned}$$

$$\mathrm{Cost}_{Worker} = 2^n \times n \times \frac{c_\oplus + c_\otimes}{2}$$

Figures 39 and 40 show the measurement of the execution time of TDS and FFT implemented with our DH algorithm.

Algorithm 5 SGL implementation of distributable homomorphism (**DH**)

DH(IN $\oplus$, IN $\otimes$, INOUT data)

```
1:  if master then
2:      pardo
3:          DH(⊕, ⊗, data);
4:      end pardo
5:      for n from 1 to log2(numChd) do
6:          gather data to tmp;
7:          for i from 1 to (len(numChd)) do
8:              if ((i-1) % exp2(n))/2 = 0 then
9:                  Swap(tmp[i], tmp[i+exp2(n)])
10:             end if
11:         end for
12:         scatter tmp to list;
13:         pardo
14:             if ((PID-1) % exp2(n))/2 = 0 then
15:                 data := data ⊕ list;
16:             else
17:                 date := data ⊗ list;
18:             end if
19:         end pardo
20:     end for
21: else
22:     for n from 1 to log2(len(data)) do
23:         for i from 1 to (len(data)) do
24:             if ((i-1) % exp2(n))/2 = 0 then
25:                 tmp := data[i] ⊕ data[i+exp2(n)];
26:             else
27:                 tmp := data[i-exp2(n)] ⊗ data[i];
28:             end if
29:         end for
30:         data := tmp;
31:     end for
32: end if
```

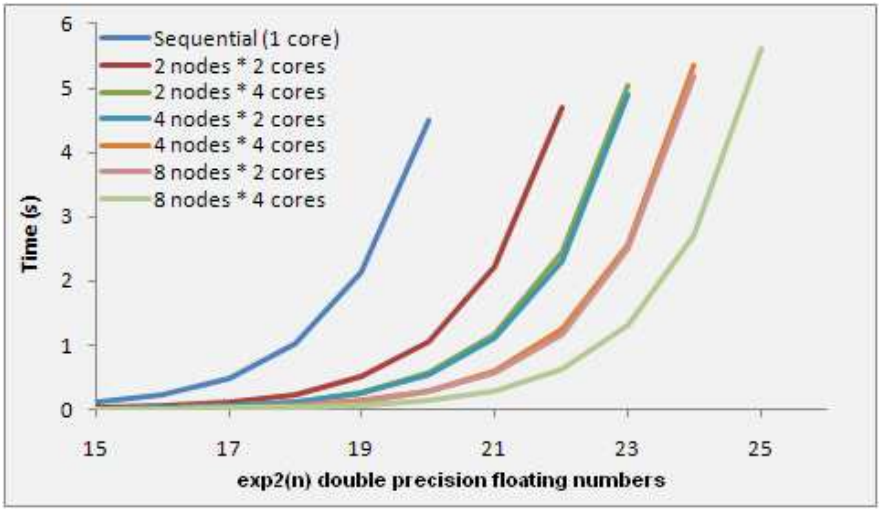

Figure 39: Execution times of TDS using SGL DH

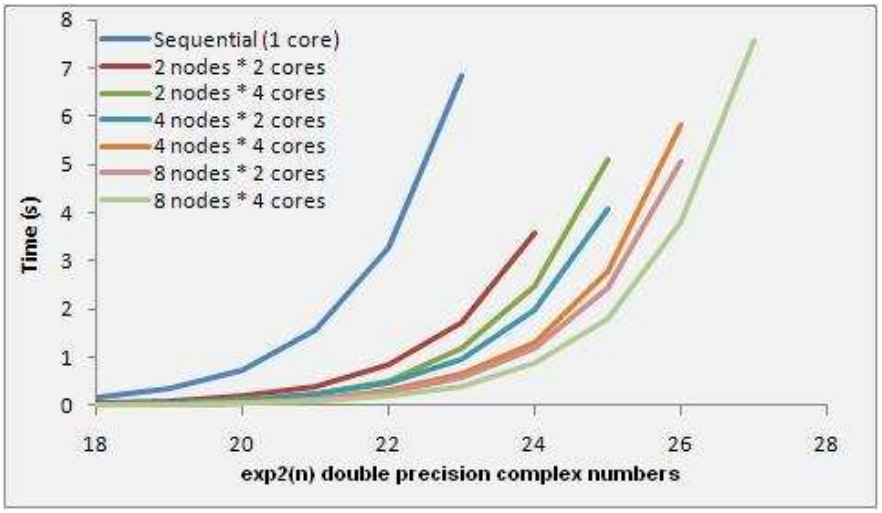

Figure 40: Execution times of FFT using SGL DH

dh imposes many "horizontal" communications that would be programmed in SGL as sequences of the form $\mathcal{G};\mathcal{P};\mathcal{S}$ as in Section 6.1.1. An SGL compiler or even interpreter will be able to optimize them[2] into a flat horizontal communication-synchronization phase, and this is how we have implemented it in our experiments. Also, it is easy to see, using our semantics, that the SGL's implementation of **dh** is a correct one thanks to the simple programming model.

2 The G;P;S pattern recognition could be guaranteed by a specific language construct or documented constraints on the user code.

5.2.3 *Speedup and Efficiency*

We firstly tested speedup and efficiency using the *scan* algorithm of Section 5.2.1.2. The formulas we used are:

$$\text{Speedup} = \frac{\text{ExecutionTime}_1}{\text{ExecutionTime}_p}$$

$$\text{Efficiency} = \frac{\text{Speedup}_p}{p}$$

where p is the number of used nodes (resp. cores).

First of all, we fixed the size of input data at 2 560 000 floating numbers, the number of cores at 4 for each node and varied the number of nodes from 1 to 16. We obtained the node-level speedup and efficiency in Table 6.

Num of proc	4	8	16	32	64
Node scale-out	1	2	4	8	16
Speedup	1	1.999	3.974	7.560	13.815
Efficiency	1	0.999	0.994	0.945	0.863

Table 6: Node-level speedup and efficiency of SGL scan

After that, we fixed the size of input data at 9 600 000 floating numbers, fixed the number of nodes at 16 and varied the number of core of each node from 1 to 4. We obtained the core-level speedup and efficiency in Table 7.

Num of proc	16	32	48	64
Core scale-out	1	2	3	4
Speedup	1	1.965	2.884	3.852
Efficiency	1	0.983	0.961	0.963

Table 7: Core-level speedup and efficiency of SGL scan

Our measurements show that core-level scale-out is more efficient than node-level scale-out. The theoretical explanation is that the cost of node-level communication is more expensive than core-level communication. Figures 41 and 42 illustrate the two tables from the point of view of acceleration.

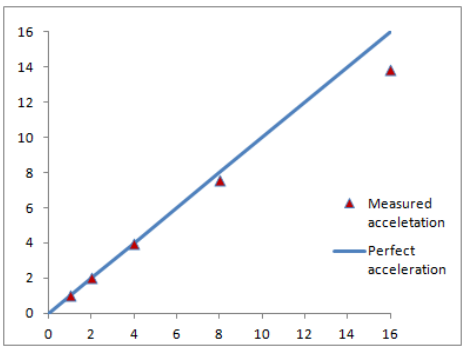

Figure 41: Node-level speedup of SGL scan (used 4 cores per node)

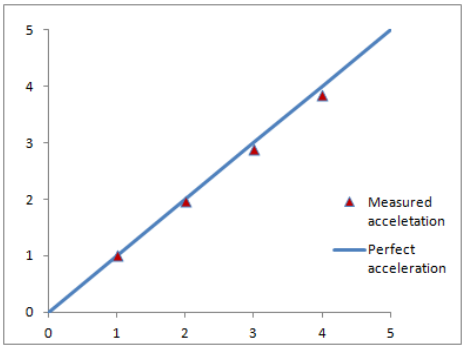

Figure 42: Core-level speedup of SGL scan (used 16 nodes)

We have used the same technique to measure the efficiency of TDS (HD skeleton, Section 5.2.2). Figure 43 shows the speedup according to the measures from our TDS algorithm. The measurement shows that when the number of core increases, the overhead increases too. It confirms our claims that the communication cost is very expensive and we need to find solutions for reducing it.

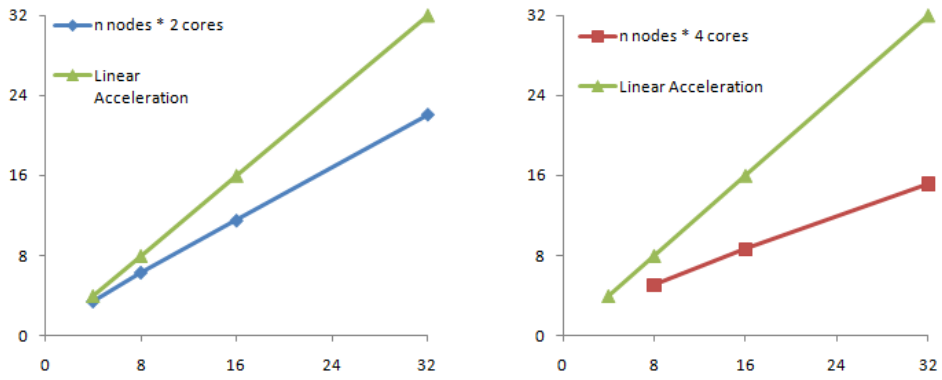

Figure 43: Speedup of SGL HD (TDS)

This chapter is not about finding the best algorithms. The algorithms used in previous sections may be not the optimized ones; they were used for experimenting our SGL cost model and programming model. This dissertation has attempted to motivate and support the idea that BSP's advantages for parallel software can be enhanced by the recursive hierarchical and heterogeneous machine structure of SGL, while simplifying the programming interface even further by replacing point-to-point messages with logically centralised communications.

6

GATHER-SCATTER FOR SIMPLIFIED COMMUNICATIONS

6.1 The GPS Theorem 113

6.1.1 Gather-Scatter Communication 114

6.1.2 The GPS Theorem 115

6.2 Simplifying BSML's Put 116

6.2.1 Dilemma in BSML: Proj vs. Put 116

6.2.2 The GPS Function 119

6.2.3 Experimentation in BSML 120

With the limitation presented in the previous chapter that SGL does not express "horizontal" communication patterns in mind, we attempt in this chapter to demonstrate that this defect only affects a minority of algorithms and can be compensated by automated compilation/interpretation. We then introduce the GPS theorem (Section 6.1) which can be implemented later in a compiler to optimize the SGL's "horizontal" all-to-all communication. We also propose a simplified version of BSML's *put* based on GPS and implement the parallel sample-sort algorithm with it (Section 6.2). The comparison of BSML's *put* and SGL's *GPS* shows that *GPS* has a better code readability and lower execution time.

6.1 THE GPS THEOREM

In this section we firstly review the *gather-scatter* communication. After that, we propose the GPS theorem, the basis for future language-based solutions

providing a compromise between simple centralized communications and more general horizontal communications that are more difficult to program and error-prone.

6.1.1 *Gather-Scatter Communication*

While experimenting with SGL programming it was found that many basic algorithms can be efficiently and cleanly programmed with the model, while satisfying a simple performance model. But the case of the sample-sort algorithm poses a fundamental problem for SGL. The tree topology architecture suggests a limited bandwidth around the master. Either this is experimentally true, in which case sample-sort cannot be generally efficient (for which SGL's simplicity could not be blamed) or the topology abstracts the actual communication bandwidth assuming that all links in the architecture have the same "width". This may hold if:

(a) indeed the network is a fat-tree; or
(b) the architecture's level below the master allows for horizontal communications that SGL does not express.

Case (a) is already covered by our SGL cost model but it was not observed in our experiments: the *master*'s bandwidth can be overwhelmed when an algorithm such as sorting moves a large subset of all data across the master. Case (b) is not only likely, but is the norm for flat architectures.

The pending issue with SGL is thus: can we program general communication patterns such as those involved in parallel sample-sort. The `if master` conditional selects an instruction branch depending on the local node's number of children: zero or more. Also, SGL's communications do not provide direct "horizontal" communications because of their hierarchical *logical* structure. But if the hardware supports them (see Appendix A.1), inter-worker communications can be extracted from SGL semantics either statically or dynamically.

6.1.2 *The GPS Theorem*

In this section we propose solutions to SGL's algorithmic incompleteness: a compromise between simple centralised communications and more general horizontal communications that are more difficult to program and error-prone. We start from the notion that we may program communications that are logically centralised but physically "horizontal". To support this claim we first present a theoretical result showing how successive supersteps can be partially compressed into a single horizontal communication-synchronization phase.

A first type of solution to SGL's lack of horizontal communications can come in the form of compiler optimisations. Sub-programs that concentrate data (gather), then locally process it on the master, then redistribute it (scatter) can be compiled to horizontal communications without the need for concentration. This possibility was not implemented so far for lack of a complete SGL language with syntax, compiler and code generation, a rather long sub-project to develop. But we can prove that this kind of optimisation is possible in the form of a theorem based on our operational semantics.

Let $\mathcal{G} \equiv$ **gather** $\overrightarrow{V}$ **to** $\widetilde{W}$ and $\mathcal{S} \equiv$ **scatter** $\widetilde{W}$ **to** $\overrightarrow{V}$ (c. f., Section 5.1.3 for their semantics) in a system with one master and p workers. Assume also that values for $\overrightarrow{V}$ are all vectors of length p. As a result values for $\widetilde{W}$ are equivalent to $p \times p$ matrices of scalars. SGL code for reorganizing such a parallel matrix of values is a sequence $\mathcal{G};\mathcal{P};\mathcal{S}$ where $\mathcal{P}$ is a sequential program in the master that realizes a permutation of the matrix.

GPS-theorem
Let $\mathcal{G};\mathcal{P};\mathcal{S}$ be as above, $\mathcal{P}$ a sequential program whose non-local variables are $\widetilde{W}$, $\overrightarrow{V}$, and π a permutation of $\{1,\ldots,p\}$ such that (π): $\forall i,j.\ \sigma''(\widetilde{W})_{i,j} = \sigma'(\widetilde{W}_{(\pi(i,j))})$ whenever $\langle \mathcal{P},\ \sigma' \rangle \to \sigma''$. Then $\sigma'''(\overrightarrow{V})_{(i,j)} = \sigma(\overrightarrow{V})_{\pi(i,j)}$ whenever $\langle (\mathcal{G};\mathcal{P};\mathcal{S}),\ \sigma \rangle \to \sigma'''$.

Proof. The sub-programs must be evaluated through steps: (g) $\langle \mathcal{G},\ \sigma \rangle \to \sigma'$; (p) $\langle \mathcal{P},\ \sigma' \rangle \to \sigma''$ and (s) $\langle \mathcal{S},\ \sigma'' \rangle \to \sigma'''$. Recall that environments σ are maps from identifiers and machine positions (master, child 1, child 2, child of child i ...) to values. The former is written as indices. The semantics translates

step (g) into (g'): $\sigma' = \sigma[\widetilde{W}/\sigma(\overrightarrow{V})_i \mid i = 1,\ldots,p]$ and step (s) into (s'): $\sigma''' = \sigma''[\overrightarrow{V}/\sigma''(\widetilde{W})_i \mid i = 1,\ldots,p]$. We thus have:

$$\begin{aligned}
\sigma'''(\overrightarrow{V})_{(i,j)} = (\sigma'''(\overrightarrow{V})_i)_j &= (\sigma''(\widetilde{W})_i)_j \quad (s')\\
&= \sigma''(\widetilde{W})_{(i,j)}\\
&= \sigma'(\widetilde{W})_{\pi(i,j)} \quad (\pi)\\
&= \sigma(\overrightarrow{V})_{\pi(i,j)} \quad (g'). \square
\end{aligned}$$

If the proposition's hypotheses are satisfied then permutation π can be applied locally to the subset of matrix data available on one worker node, and then given as local argument to a collective communication operation, thus combining two vertical communications into a single horizontal one. The local interpretation of sub-program $\mathcal{P}$ into π is beyond the scope of this chapter and would require a more complex language than what is covered by our current semantics for SGL.

6.2 SIMPLIFYING BSML'S PUT

We attempted, in this section, to propose a GPS function implemented in BSML for simplifying BSML's **put** primitive. We then experimented this new function in BSML with a parallel simple-sort algorithm and compared it with a standard BSML implementation.

6.2.1 *Dilemma in BSML: Proj vs. Put*

In BSML (c. f., Section 3.2.2.2), **proj** is the parallel destructor: **proj** <x_0, x_1,..., x_{p-1}> computes a function f such that (f i) = x_i. And **put** is the synchronous (communicating) parallel transformer: **put** <g_0, g_1,..., g_{p-1}> computes a parallel vector of functions that contain the transported messages that were specified by the g_i. The input local functions are used to specify the outgoing messages thus: g_i j is the value that processor i wishes to send to processor j. The result of applying **put** is a parallel vector of functions dual to the g_i: they specify which value was received from a given distant processor.

The execution of **proj** uses an all-to-all communication and the execution of **put** is a general BSP communication (any processor-processor relation can be implemented with it). Experience with BSML for more than a decade has shown that **proj** is much easier to use than **put**, that **proj** can be used to program a large subset of all parallel functions, but that algorithms such as sample-sort cannot be implemented without **put**.

Let us take here the student assignments of M1[1] course *Algorithmes Parallèles et Distribués* at Université Paris-Est Créteil (UPEC, ex-Paris 12) as examples. In April 2012, we asked the students to recode all functions of Caml's *List* library[2] using `'a parlist = 'a list par` instead of `'a list` for parallel programming; and in April 2013 the next-term students to recode all functions of Caml's *Array* library[3] using `'a parray = 'a array par` instead of `'a array`. Table 8 and Table 9 show the signatures of all required functions.

Recoded function	**Signature**
par_length:	'a parlist -> int
par_hd:	'a parlist -> 'a
par_tl:	'a parlist -> 'a parlist
par_nth:	'a parlist -> int -> 'a
par_rev:	'a parlist -> 'a parlist
par_flatten:	'a list parlist -> 'a list
par_map:	('a -> 'b) -> 'a parlist -> 'a parlist
par_fold_left:	('a -> 'a -> 'a) -> 'a -> 'a parlist -> 'a
par_for_all:	('a -> bool) -> 'a parlist -> bool
par_exists:	('a -> bool) -> 'a parlist -> bool
par_mem:	'a -> 'a parlist -> bool
par_find:	('a -> bool) -> 'a parlist -> 'a
par_filter:	('a -> bool) -> 'a parlist -> 'a parlist
par_partition:	('a -> bool) -> 'a parlist -> 'a parlist * 'a parlist

Table 8: Signatures of recoded functions of Caml's List library in parallel

1 French first-year Master, the fourth year of European higher education.
2 OCaml reference manual - List library. http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html
3 OCaml reference manual - Array library. http://caml.inria.fr/pub/docs/manual-ocaml/libref/Array.html

In Table 8 for Caml's List library, the functions `par_tl`, `par_flatten`, `par_map` and `par_filter` do not require any inter-processor communication. The functions `par_hd` and `par_nth` require only a one-to-one communication, they can be implemented easily and efficiently with **proj** . The functions `par_length`, `par_fold_left`, `par_for_all`, `par_exists`, `par_mem` and `par_find` require an all-to-one communication with p data, an implementation using **proj** can be efficient enough too. However, the functions `par_rev` and `par_partition` require an all-to-all communication, an efficient implementation can only be done by using **put** . There are only 2 functions out of 14 that need **put** for implementing an efficient all-to-all communication.

Recoded function	**Signature**
parray_length:	'a parray -> int
parray_get:	'a parray -> int -> 'a
parray_set:	'a parray -> int -> 'a -> unit
parray_init:	int -> (int -> 'a) -> 'a parray
parray_map:	('a -> 'b) -> 'a parray -> 'b parray
parray_fold_left:	('a -> 'a -> 'a) -> 'a -> 'a parray -> 'a
parray_append:	'a parray -> 'a parray -> 'a parray
parray_sub:	'a parray -> int -> int -> 'a parray

Table 9: Signatures of recoded functions of Caml's Array library in parallel

In Table 9, the functions `parray_init` and `parray_map` do not require any inter-processor communication; the functions `parray_get` and `parray_set` require only a one-to-one communication; the functions `parray_length` and `parray_fold_left` require an all-to-one communication with p data; and the functions `parray_append` and `parray_sub` require an all-to-all communication. In the case of Caml's Array library, there are thus only 2 functions out of 8 that need **put** for implementing an efficient all-to-all communication.

These exercises confirm again our claims that **proj** can be used to program a large subset of all parallel functions, but **put** is indispensable for implementing efficiently the all-to-all communication. We thus propose in the next section the `GPS` function for parallel programming to simplify BSML's general communication `put` function.

6.2.2 *The GPS Function*

A second type of solution to SGL's lack of horizontal communication is developed in this section within the framework of BSML i. e., flat BSP programming within OCAML programs. It is based on the following simulation of SGL by BSML for flat BSP machines.

There is a natural correspondence between a two-level SGL machine and a BSML program:

- the *master* corresponds to all non-*par* types in the BSML program i. e., all values that are replicated on each processor;
- the *workers* correspond to all local elements of *par* types in the BSML program;
- **scatter** corresponds to **mkpar** (*seq-to-par* primitive);
- **gather** corresponds to **proj** (*par-to-seq* primitive) and
- **pardo** corresponds to **apply** .

An algorithm such as parallel sample-sort cannot be programmed with pure SGL primitives i. e., with only **mkpar**, **apply**, **proj** and without the general communication primitive **put**. Our proposed solution is a simplified form of **put** that leads to the same parallel performance but resembles a G;P;S program as in the GPS theorem presented in Section 6.1.2.

The general BSML communication primitive is **put** and it has the following type:

```
put: (int -> 'a) par -> (int -> 'a) par
```

with the input parallel vector containing a *destination-value* map at each processor, and the output parallel vector containing a *sender-value* map at each processor.

The *simplified* version that we propose is called `sgl_gps` and has the following type:

```
(int -> 'a -> 'b list) -> (int -> 'b list -> 'c) -> 'a par -> 'c par
```

The first argument relates to data input for the communication phase. It specifies how each processor (rank an integer) splits a value into a list[4] of p

4 We use `list` here for prototyping. `Array` should be allowed too in a real implementation.

values, one for every destination. The second argument relates to data reception after communication. It specifies how each processor (rank an integer) aggregates a list of p received values into a local value. The third argument is the input parallel vector and the function produces its output parallel vector by applying a single communication-synchronisation superstep, like BSML's **put**.

The new function is considered to be simpler to be used than **put** because: (a) it separates meta-data (first two arguments) from the actual data to be communicated (third argument); and (b) the meta-data is sequential.

6.2.3 *Experimentation in BSML*

The following (Listing 5) is a BSML implementation for the new communication function:

```
(* ====================== *)
(* === BSML libraries === *)
(* ====================== *)
open Bsml
open Base
open Tools

(* ============================ *)
(* === Auxiliary functions === *)
(* ============================ *)
let rec (from_to: int -> int -> int list) = fun debut fin ->
  if debut > fin then [] else debut::(from_to (debut+1) fin);;

let (procs_list: int list) = from_to 0 (bsp_p-1) ;;

(* ========================== *)
(* === SGL g-p-s function === *)
(* ========================== *)
let (sgl_gps: (int -> 'a -> 'b list) -> (int -> 'b list -> 'c) -> 'a par -> 'c
     par) = fun split assemble indata ->
   let splitted = apply (mkpar split) indata in
   let exchange = put (parfun List.nth splitted) in
   let permuted = parfun (fun l -> List.map l procs_list) exchange in
   apply (mkpar assemble) permuted
;;
```

Listing 5: Simplified BSML horizontal communication

Here **parfun** is an abbreviation for `fun f x -> apply (replicate f) x` provided by BSML's module *Bsmlbase*, where **replicate** is an abbreviation for `fun f -> mkpar (fun i -> f)` from the same module. This function is similar to List.map for parallel vectors:

$$\textbf{parfun}\ f < x_0, \ldots, x_{(p-1)} > = < f\ x_0, \ldots, f\ x_{(p-1)} >$$

And **bsp_p** is a machine parameter accessor of BSML giving the number p of processors in the parallel machine.

We have implemented Tiskin-McColl parallel sample-sort [Tis99] with this new communication primitive (Listing 6). There are two communications in the main function. The first one is when `glosampl` is computed with `gather_list` which is a modified **proj** (all-to-all) communication [Line 65]. This one is used to constitute meta-data. The second communication (for transport) is when return value uses `sgl_gps` [Line 69].

```
1  (* === BSML libraries === *)
   open Bsml
3  open Stdlib
   open Base
5  open Comm
   open Tools
7
   open Sgl_gps
9
   (* === Auxiliary functions === *)
11 let (filter_nth: (int -> bool) -> 'a list -> 'a list ) = fun f l ->
     let rec aux i = function
13     | x::r -> if f i then x::(aux (i+1) r) else aux (i+1) r
       | [] -> []
15   in aux 0 l
   ;;
17
   let (extract_n: int -> int -> 'a list -> 'a list) = fun n len l ->
19   filter_nth (fun i -> (n * i - 1) mod len >= len - n) l
   ;;
21
   let rec (split_lt: 'a list -> 'a list -> 'a -> 'a list * 'a list) =
23   fun acc l p -> match l with
     | x::r -> if x < p then (split_lt (x::acc) r p) else ( ( List.rev acc ), l)
25   | [] -> ((List.rev acc), [])
   ;;
27
```

```
let rec (slice_p: 'a list -> 'a list -> 'a list list) = fun l pivots ->
  match pivots with
  | p::r -> let l1,l2 = split_lt [] l p in
    l1::(slice_p l2 r)
  | [] -> [l] ;;

let rec (merge: 'a list -> 'a list -> 'a list -> 'a list) = fun acc l1 l2 ->
  match l1,l2 with
  | x1::r1,x2::r2 -> if x1 < x2 then merge (x1::acc) r1 l2
    else merge (x2::acc) l1 r2
  | [],l | l,[] -> List.rev_append acc l ;;

let rec (splitn: int -> 'a list -> 'a list * 'a list) = fun n (x::r) ->
  if n=0 then [],x::r
  else let l1,l2 = splitn(n-1) r in (x::l1,l2) ;;

let rec (p_merge: int -> 'a list list -> 'a list) = fun p -> function
  | [] -> []
  | l::[] -> l
  | ll -> let ll1,ll2 = splitn (p/2) ll in
    merge [] (p_merge (p/2) ll1) (p_merge (p-p/2) ll2) ;;

let (proj_list: 'a par -> 'a list) = fun v -> List.map (proj v) procs_list ;;

let rec (concat: 'a list -> 'a list list -> 'a list) = fun acc ll ->
  match ll with
  | l::lr -> concat (List.rev_append l acc) lr
  | [] -> List.rev acc ;;

let rec (gather_list: 'a list par -> 'a list) =
  fun parlist -> concat [] (proj_list parlist) ;;

(* === Parallel Sorting by Regular Sampling === *)
let (regular_sample_sort: int par -> 'a list par -> 'a list par) =
  fun lvlengths lv ->
  let locsort = parfun (List.sort compare) lv in
  let regsampl = apply (parfun (fun l len -> extract_n bsp_p len l) locsort)
      lvlengths in
  let glosampl = List.sort compare (gather_list regsampl) in
  let pivots = extract_n bsp_p (bsp_p * (bsp_p - 1)) glosampl in
  let split = fun pid send_raw -> slice_p send_raw pivots in
  let assemble = fun pid recv_raw ->  p_merge bsp_p recv_raw in
  sgl_gps split assemble locsort
;;
```

Listing 6: Implementing Tiskin-McColl parallel sample-sort with **GPS** function

Our **GPS** implementation was experimented on an 8-core 2.67 GHz Xeon-based workstation with BSML 0.5 and MPICH 3.0.1 running on OpenSuse 11.4 (Celadon). We fixed the total size of input data at 200 000 integer numbers, and varied the number of processors from 1 to 8. The execution time was measured by OCaml's function `Unix.gettimeofdate`.

The formula of speedup we used is same as in Section 5.2.3 :

$$\mathrm{Speedup} = \frac{\mathrm{ExeTime}_{\mathrm{Seq}}}{\mathrm{ExeTime}_{\mathrm{Par}}}$$

where $\mathrm{ExeTime}_{\mathrm{Seq}} = 182.62ms$ is based on OCaml's sequential function `List.sort` processing 200 000 random integer numbers generated by the function `Random.int`.

Since the bound of sequential sorting algorithm is $O(n \log n)$, the formula for the efficiency we used is slightly different from the standard one which is simply $speedup \times \frac{1}{p}$. Here is

$$\mathrm{Efficiency} = \mathrm{Speedup} \times \frac{n \log n}{200000 \log 200000}$$

where n is the size of input data per processor. This formula is coherent with Shi's method presented in the paper [SS92] introducing Parallel Sorting by Regular Sampling (PSRS).

We obtained therefore the results in Table 10.

Num of proc	1	2	4	8
time (ms)	215.7	106.7	45.7	20.4
Speedup	0.676	1.367	3.192	7.154
Efficiency	0.847	0.807	0.886	0.929

Table 10: Speedup and efficiency of parallel sample-sort in GPS implementation

The efficiency of $p = 1$ is smaller than 1, because we used here the parallel program for the 1-processor test, it thus computed the meta-data. Furthermore, BSML does not optimize the communication that a processor send data to itself. This created a communication cost that decreased the efficiency of $p = 1$.

We have also tested on the same machine the same parallel sample-sort algorithm implemented in standard BSML (with **put** , shown in Listing 7),

which can be found in L. Gesbert's PhD thesis [Ges09].

```
(* === Parallel Sorting by Regular Sampling === *)
let (psrs : int par -> 'a list par -> 'a list par) = fun lvlengths lv ->
  let locsort = parfun (List.sort compare) lv in
  let regsampl = apply (parfun (fun l len -> extract_n bsp_p len l) locsort)
      lvlengths in
  let glosampl = List.sort compare (gather_list regsampl) in
  let pivots = extract_n bsp_p (bsp_p * (bsp_p - 1)) glosampl in
  let comm = parfun (fun l -> slice_p l pivots) locsort in
  let recv = put (parfun List.nth comm) in
    parfun(fun ll -> p_merge bsp_p (List.map ll procs_list)) recv
;;
```

Listing 7: Gesbert's implementation of parallel sample-sort [Ges09]

The main difference between these two implementations can be found in lines 67-69 of Listing 6 and lines 7-9 of Listing 7. The `sgl_gps` function provides a sequential view for coding. Developers need only to focus on how to split the local data and how to assemble the received data on a local node. The all-to-all communication and synchronisation are performed implicitly by `sgl_gps`. The cost of communication may be optimised during the compilation (if the compiler supports natively the `sgl_gps` function). Contrariwise, the `put` function focusses on the communication. A *destination-value* map for each processor before `put` and a *sender-value* map for each processor after `put` must be designed by the developer. Moreover, developers must work on a parallel environment.

We use the same method as before for measuring the execution time, speedup and efficiency of this **put** implementation. The results are shown in Table 11.

Num of proc	1	2	4	8
time (ms)	218.0	105.4	44.6	20.7
Speedup	0.669	1.383	3.273	7.029
Efficiency	0.838	0.817	0.908	0.913

Table 11: Speedup and efficiency of parallel sample-sort using BSML **put**

Figure 44 shows the speedup of the **GPS** implementation and the **put** implementation of parallel sample-sort. Figure 45 compares the real cost (execution time) of these two implementations.

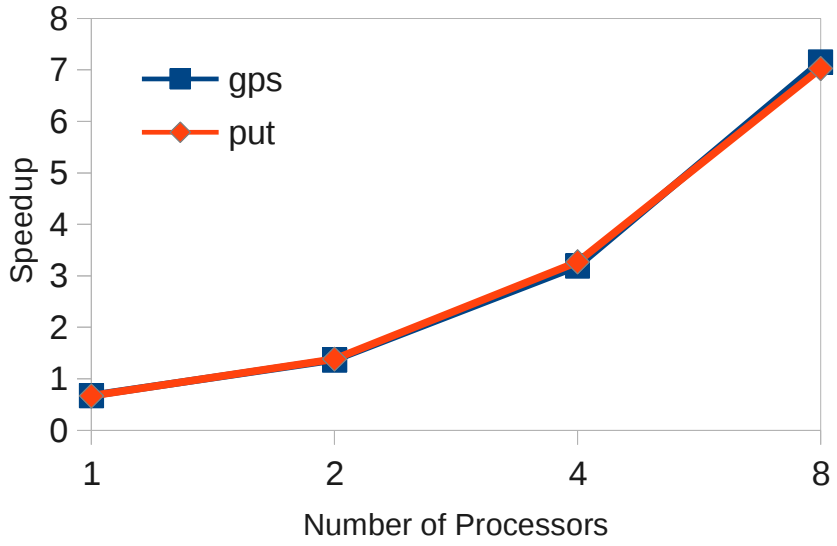

Figure 44: Speedup of **GPS** and **put** parallel sample-sort implementations

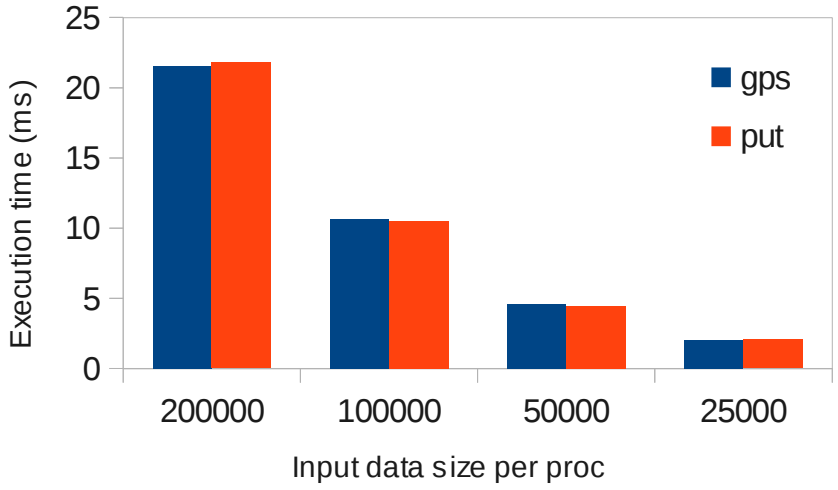

Figure 45: Execution time of **GPS** and **put** parallel sample-sort implementations

Our measurement shows that the **GPS** implementation is as fast as the **put** implementation. Furthermore, the **GPS** implementation keeps the same speedup as the standard BSML one, because the sorting algorithms are the same for both implementations. These results confirm our claim that **GPS** can simplify programming while keeping a good performance.

We conclude this programming experiment by observing that **put** is used in only a minority of parallel functions, that it can be replaced by a new function that is logically simpler and that this new function is close to SGL semantics. With `sgl_gps` we can program horizontal communications to optimise performance in an extended version of SGL. Future work will experiment multi-level and heterogeneous versions of this extension to SGL.

7

CONCLUSION

7.1 Conclusion 127

7.2 Future work 129

7.1 CONCLUSION

We have described in this dissertation attempts to improve the simplicity of parallel program development, while ensuring portability and the accuracy of parallel algorithm performance prediction for heterogeneous and hierarchical environments.

The term *bridging model* introduced by L. Valiant in 1990 is a model of a computer which provides a conceptual bridge between the physical implementation of the machine and the abstraction available to a programmer of that machine. It provides a common level of understanding between hardware and software engineers. Thus, one can develop portable and predictable algorithms on it.

With this in mind, we reviewed in chapter 2 a wide variety of different parallel computer architectures – from commodity hardware to specified heterogeneous hardware, from multi-core computers to peta-FLOPS supercomputers, from centralized parallel computers to geographically distributed grids – in order to have some general notions about the modern parallel computers' design, for preparing a realistic general abstract machine to model these different architectures. Parallel computer hardware evolution shows that the flat view of a parallel machine as a set of communicating sequential machines remains a useful and practical model but is more and more incomplete. Moreover, we can observe that heterogeneous chip multiprocessors present unique opportunities for improving system throughput and reducing processor con-

sumption. The trend towards green-computing puts even more pressure on the optimal use of architectures that are not only highly scalable but hierarchical and non-homogeneous. All these changes make parallel programming harder than before. A realistic bridging model is desirable for handling these heterogeneous hierarchical machines.

We then traversed in chapter 3 the state of the art of parallel programming models. We found that the multi-threaded concurrent programming is good for getting started, but it can be applied only on shared-memory architectures. The message-passing approach handles the distributed-memory architectures, but the management of communication is never an easy job. Actor models provide patterns for the communication, but their application is too hard to optimise without any algorithm-machine bridging. The BSP bridging model links software and hardware, offers a sequential view of a parallel program with supersteps, simplifies algorithm design and analysis with the barrier, but more and more nowadays parallel computers are not developed in BSP-proposed flat structure but in a hierarchical architecture. MapReduce simplifies large data set processing on distributed cluster with implicit communication, but how it handles a complex algorithm with a good performance, is still a question. All these observations lead us to propose a hierarchical bridging model for simplifying parallel programming.

We thus introduced in chapter 4 our Scatter-Gather parallel-programming and parallel execution model in the form of a simple imperative Scatter-Gather Language (SGL). Existing research has been surveyed to identify SGL's original and useful aspects. SGL has been given a precise operational semantics and cost model. The SGL cost model improves the clarity of algorithms performance analysis; it allows benchmarking machine performance and algorithm quality. We believe that the SGL computer can cover most of the modern parallel computers. The synchronisation cost of SGL algorithms for a massively parallel computer can be greatly reduced by its hierarchical structure. The communication cost of SGL between different levels is more realistic than the flat structure.

SGL was used in chapter 5 with its programming model to program basic parallel operations and more complex parallel skeletons such as parallel reduce, parallel scan (prefix), parallel sort and a butterfly algorithm distributable homomorphism. In both cases, the scalability and accuracy of the cost-model were measured. We have found that a subset of parallel communi-

cation patterns is not naturally covered by SGL's centralised operations. This is visible in a recursive implementation of parallel sample-sort.

Adding a general communications primitive to SGL may be a radical solution. But this would have hurt the practical and theoretical simplicity of SGL. Experience and research with BSML (BSP-CAML) shows that the semantics and practical use of such a primitive (**put**) is more complex and error-prone than pure SGL. This is obvious because the semantics of **put** is based on a communication matrix, while **scatter** and **gather** are described by 1-dimensional vectors of messages.

Further analysis has led us to propose in chapter 6 two elements of a general solution to this dilemma. The first one is the GPS theorem: a semantic equivalence between **gather**; `P`; **scatter** sub-programs (where `P` is a local sequential program in the *master*) and horizontal **put**-like operations. This result is the basis of future compiler optimisations whereby a family of clean but inefficient SGL programs can be compiled to lower-level but more efficient programs using horizontal communications. The second proposed solution is a simplified form of **put** that we have designed and experimented in BSML. With it, the Tiskin-McColl BSP sample-sort algorithm [Tis99] is programmed without having to encode a general communication matrix: only its sender-side 1-to-n and receiver-side n-to-1 communication relations.

7.2 FUTURE WORK

The definition and validation of SGL detailed in this dissertation are only a first step towards its safe and efficient application to high-level parallel programming. Relevant and related existing work is abundant and likewise, future work in this direction has many aspects.

SGL code compilation will be investigated by defining a complete language that includes the SGL syntax and semantics given here. This can be done from scratch or by extending an existing language's parser, static analysis, code generation etc. The target language might range from BSML to C+MPI but optimisations based on the GPS theorem should be investigated for their generality and efficiency. Native code generation via LLVM (formerly Low Level Virtual Machine) will also be considered as a portable and high-performance

alternative to BSML. Code generation for accelerators (like GPGPUs) has not been experimented yet and should be investigated since SGL does **not** require workers or sub-workers to be identical. Furthermore, GPGPUs' multi-processor – multi-core – multi-thread and shared-memory nature coincides with SGL's hierarchical tree structure.

Another direction that will be explored is the use of SGL as intermediate language for more abstract ones. For example EXQIL is a domain-specific language (DSL) developed at EXQIM S.A.S. for simplified expression of algorithmic trading strategies [HGL+]. EXQIL uses algorithmic skeletons but has no explicit parallel constructions. The current EXQIL version is 0.63 and its future versions will use SGL code generation for scalable and predictable acceleration on parallel hardware.

An even higher-level approach that has been started is automatic code generation whereby parallel programs are generated directly from domain-specific UML descriptions. This "no-programming" model-driven paradigm has been presented in [KHL12]. It currently generates multi-thread or multi-process code that is embarrassingly parallel i.e. without communications. Future work will generalise it to target SGL code so that more operations can be parallelised.

A

APPENDIX

In a bridging model, the only information we have of the hardware is the parameters of the cost model. However, since the GPS theorem (Section 6.1) has not yet been included in SGL's language semantics (Section 5.1), the parameters of SGL's cost model (Section 4.2.3) may not tell us whether a transformation based on this theorem would constitute an optimisation.

Before starting a new research topic on it, we propose here some ideas about how to measure whether inter-processor "horizontal" communication is supported by hardware, in order to detect the feasibility of this optimisation on a given machine.

A.1 COMMUNICATION THROUGHPUT

Network speed and network capacity are two different dimensions of communication performance. In the SGL bridging model, the network speed can be measured by SGL's cost model machine parameter g presented in Section 4.2.3. This parameter describes how fast the connection is for transmitting one word from *master* (resp. *child*) to its *child* (resp. *master*).

However, if inter-worker communication is allowed in SGL, the network capacity used to measure the amount of words that can be transferred at the same moment between *workers* of a *master*, should also be taken into account in order to recognize the network saturation. We thus propose a new machine parameter for the cost model of SGL:

- **t**: the throughput, maximum words that can be transferred simultaneously in a master-children sub-tree.

For example, $t = 1$ denotes that the hardware does not support the inter-processor communication (cf. SGL); $t < p$ denotes that the inter-processor communication is allowed with limited bandwidth (eg. most of current CPU-GPU architectures); and $t = p$ denotes that the network supports all-to-all communication with a satisfactory bandwidth (cf. BSP).

The communication cost also depends on the partition of data to transfer. The data partition can be reflected using the algorithm parameter k of SGL's cost model. For sending n words initially stored on one processor, or receiving n words by one processor, according to SGL's cost formulae where $comm = g \times k$, the communication cost is:

$$g \times n$$

which is the *upper bound* of communication cost in one SGL level.

In the case where if these n words are evenly partitioned over p SGLC components (children) in the same level (ie. n/p words per processor) before and after the data exchange (eg. in a perfect balancing algorithm), the communication cost is:

$$\frac{g \times n}{\min(p, t)}$$

where g represents the communication speed and t the communication capacity. This is the *lower bound* of communication cost in one SGL level.

The additional parameter t that reflects the cost of a horizontal exchange, allows to detect whether a transformation based on the GPS theorem presented in this thesis would constitute an optimisation.

BIBLIOGRAPHY

[AAA+01] F. Allen, G. Almasi, W. Andreoni, D. Beece, B. J. Berne, A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, P. Coteus, P. Crumley, A. Curioni, M. Denneau, W. Donath, M. Eleftheriou, B. Fitch, B. Fleischer, C. J. Georgiou, R. Germain, M. Giampapa, D. Gresh, M. Gupta, R. Haring, H. Ho, P. Hochschild, S. Hummel, T. Jonas, D. Lieber, G. Martyna, K. Maturu, J. Moreira, D. Newns, M. Newton, R. Philhower, T. Picunko, J. Pitera, M. Pitman, R. Rand, A. Royyuru, V. Salapura, A. Sanomiya, R. Shah, Y. Sham, S. Singh, M. Snir, F. Suits, R. Swetz, W. C. Swope, N. Vishnumurthy, T. J. C. Ward, H. Warren, and R. Zhou. Blue Gene: a vision for protein science using a petaflop supercomputer. *IBM Syst. J.*, 40(2):310–327, February 2001. (Cited on page 13.)

[ABB+03] George Almasi, Ralph Bellofatto, Jose Brunheroto, Calin Cascaval, Jose G. Castanos, José G, Luis Ceze, Paul Crumley, C. Christopher Erway, Joseph Gagliano, Derek Lieber, Xavier Martorell, José E. Moreira, and Alda Sanomiya. An Overview of the Blue Gene/L System Software Organization. In *In Proceedings of Euro-Par 2003 Conference, Lecture Notes in Computer Science*, pages 543–555. Springer-Verlag, 2003. (Cited on pages 3, 13, and 68.)

[Adv09] Sarita V. Adve. Memory models: a case for rethinking parallel languages and hardware. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 45–45, New York, NY, USA, 2009. ACM. (Cited on pages 1 and 66.)

[AISS95] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model — One step closer towards a realistic model for parallel computation. Technical report, Department of Computer

Science, University of California, Santa Barbara, CA, USA, 1995. (Cited on page 58.)

[AJS04] Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors. *Communicating Sequential Processes. The First 25 Years*. Springer, 2004. (Cited on page 41.)

[Alt07] M. Alt. *Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance*. PhD thesis, Universität Münster, 2007. (Cited on pages 60 and 104.)

[Bar95] Geoff Barrett. Model Checking in Practice: The T9000 Virtual Channel Processor. *IEEE Trans. Softw. Eng.*, 21(2):69–78, February 1995. (Cited on page 42.)

[BCC+06] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *Int. J. High Perform. Comput. Appl.*, 20(4):481–494, November 2006. (Cited on page 19.)

[BDH+08] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 1:1–1:11, Piscataway, NJ, USA, 2008. IEEE Press. (Cited on pages 3, 24, and 68.)

[BDH+09] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Michael Lang, Scott Pakin, and Jose Carlos Sancho. Using Performance Modeling to Design Large-Scale Systems. *Computer*, 42:42–49, 2009. (Cited on page 47.)

[BFMR96] Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors. *The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model*, volume 1124 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1996. (Cited on page 54.)

[BHC+93] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 102–111, New York, NY, USA, 1993. ACM. (Cited on page 39.)

[BHP+96] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul Spirakis. BSP vs LogP. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, SPAA '96, pages 25–32, New York, NY, USA, 1996. ACM. (Cited on page 58.)

[Bis95] Rob H. Bisseling. Sparse Matrix Computations on Bulk Synchronous Parallel Computers. In G. Alefeld, O. Mahrenholtz, and R. Mennicken, editors, *Proceedings ICIAM '95*, pages 127–130. Akademie Verlag, 1995. (Cited on page 53.)

[Bis04] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, 2004. (Cited on page 50.)

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM. (Cited on page 58.)

[BJOR99] Olaf Bonorden, Ben Juurlink, Ingo Von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) Library - Design, Implementation and Performance. In *In Proc. of 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP*, pages 99–104. Society Press, 1999. (Cited on pages 51, 65, and 66.)

[BJvOR03] Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) library. *Parallel Comput.*, 29:187–207, February 2003. (Cited on page 51.)

[BK96] George Horatiu Botorog and Herbert Kuchen. Efficient Parallel Programming with Algorithmic Skeletons. In *Euro-Par, Vol. I*, pages 718–731, 1996. (Cited on pages 2 and 66.)

[BM94] Rob H. Bisseling and William F. McColl. Scientific Computing on Bulk Synchronous Parallel Architectures. In *IFIP Congress (1)*, pages 509–514, 1994. (Cited on page 53.)

[Bou96] Luc Bougé. The data parallel programming model: A semantic perspective. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, pages 4–26. Springer Berlin Heidelberg, 1996. (Cited on pages 47 and 86.)

[CJvdPK07] Barbara Chapman, Gabriele Jost, Ruud van der Par, and David J. Kuck. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007. (Cited on page 32.)

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM. (Cited on page 57.)

[CL01] Hojung Cha and Dongho Lee. H-BSP: A Hierarchical BSP Computation Model. *J. Supercomput.*, 18(2):179–200, February 2001. (Cited on page 55.)

[Co.09] Nvidia Co. Nvidia Fermi Compute Architecture Whitepaper , 2009. (Cited on pages 22 and 23.)

[Col89] M. Cole. *Algorithmic Skeletons, Structural Management of Parallel Computation*. MIT Press, 1989. (Cited on pages 2, 60, and 66.)

[Col04a] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004. (Cited on page 60.)

[Col04b] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004. (Cited on page 94.)

[Cra91] Cray Computer Corporation. *CRAY-3 Software Introduction Manual*. Cray Computer Corp., Technical Publications Department, 1991. (Cited on page 12.)

[CRDI07] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM J. Res. Dev.*, 51(5):559–572, September 2007. (Cited on page 21.)

[CT94] D.K.G. Campbell and S.J. Turner. CLUMPS: a model of efficient, general purpose parallel computation. In *TENCON '94. IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994*, pages 723–727 vol.2, 1994. (Cited on page 54.)

[DD95] J. J. Dongarra and T. Dunigan. Message-Passing Performance of Various Computers. Technical report, University of Tennessee, Knoxville, TN, USA, 1995. (Cited on page 36.)

[DDH97] Frank Dehne, Wolfgang Dittrich, and David Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 106–115, New York, NY, USA, 1997. ACM. (Cited on page 55.)

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. (Cited on page 61.)

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, September 1965. (Cited on page 31.)

[DK96a] Dimitris C. Dracopoulos and Simon Kent. Speeding up genetic programming: a parallel BSP implementation. In *Proceedings of the First Annual Conference on Genetic Programming*, GECCO '96, pages 421–421, Cambridge, MA, USA, 1996. MIT Press. (Cited on page 53.)

[DK96b] Dimitris C. Dracopoulos Dracopoulos and Simon Kent. Bulk Synchronous Parallelisation of Genetic Programming. In *Proceedings of the third International Workshop, PARA '96*, pages 216–226. Springer Verlag, 1996. (Cited on page 53.)

[DM02] Narsingh Deo and Paulius Micikevicius. Coarse-Grained Parallelization of Distance-Bound Smoothing for the Molecular Conformation Problem. In SajalK. Das and Swapan Bhattacharya, editors, *Distributed Computing*, volume 2571 of *Lecture Notes in Computer Science*, pages 55–66. Springer Berlin Heidelberg, 2002. (Cited on page 53.)

[ELZ98] Jörn Eisenbiegler, Welf Löwe, and Wolf Zimmermann. BSP, LogP, and Oblivious Programs. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, Euro-Par '98, pages 865–874, London, UK, UK, 1998. Springer-Verlag. (Cited on page 58.)

[Fis03] Randall James Fisher. *General-purpose SIMD within a register: Parallel processing on consumer microprocessors*. PhD thesis, Purdue University, 2003. (Cited on page 39.)

[FK99] Ian Foster and Carl Kesselman, editors. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. (Cited on page 18.)

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, August 2001. (Cited on page 18.)

[FMM+13] Michael J. Flynn, Oskar Mencer, Veljko Milutinovic, Goran Rakocevic, Per Stenstrom, Roman Trobec, and Mateo Valero. Moving from petaflops to petadata. *Commun. ACM*, 56(5):39–42, May 2013. (Cited on page 47.)

[FSCL06] J. Falcou, J. Serot, T. Chateau, and J. T. Lapreste. QUAFF : Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32(7-8):604–615, 2006. (Cited on page 94.)

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM. (Cited on pages 47 and 70.)

[FYA+97] H. Fujii, Y. Yasuda, H. Akashi, Y. Inagami, M. Koga, O. Ishihara, M. Kashiyama, H. Wada, and T. Sumimoto. Architecture and

performance of the Hitachi SR2201 massively parallel processor system. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 233–241, 1997. (Cited on page 13.)

[Gav05] Frédéric Gava. *Approches fonctionnelles de la programmation parallèle et des méta-ordinateurs. Sémantiques, implantations et certification*. PhD thesis, Université Paris XII-Val de Marne, LACL, 2005. (Cited on pages 43 and 51.)

[Ger93] Alexandros V. Gerbessiotis. *Topics in Parallel and Distributed Computation*. PhD thesis, Harvard University, 1993. (Cited on page 53.)

[Ges09] Louis Gesbert. *Développement systématique et sûreté d'exécution en programmation parallèle structurée*. PhD thesis, Université Paris-Est, 2009. (Cited on page 124.)

[GG11] I. Garnier and F. Gava. CPS Implementation of a BSP Composition Primitive with Application to the Implementation of Algorithmic Skeletons. *Parallel, Emergent and Distributed Systems*, 2011. To appear. (Cited on page 94.)

[GGP09] Philipp Grabher, Johann Großschädl, and Dan Page. Selected areas in cryptography. chapter On Software Parallel Implementation of Cryptographic Pairings, pages 35–50. Springer-Verlag, Berlin, Heidelberg, 2009. (Cited on page 39.)

[Gla09] Peter N. Glaskowsky. *NVIDIA's Fermi: The First Complete GPU Computing Architecture*. Nvidia, September 2009. (Cited on pages 23 and 82.)

[GLS99] William Gropp, Ewing L Lusk, and Anthony Skjellum. *Using MPI : Portable Parallel Programming with the Message Passing Interface*, volume 1. MIT press, 1999. (Cited on page 43.)

[GMR94] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The QRQW PRAM: accounting for contention in parallel algorithms. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, SODA '94, pages 638–648, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics. (Cited on page 48.)

[Gor96] Sergei Gorlatch. Systematic Extraction and Implementation of Divide-and-Conquer Parallelism. In *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs*, PLILP '96, pages 274–288, London, UK, UK, 1996. Springer-Verlag. (Cited on page 104.)

[Gri74] Ralph Grishman. *Assembly Language Programming for the Control Data 6000 Series and the Cyber 70 Series*. Algorithmics Press, 1974. (Cited on page 9.)

[Gri12] Grid5000. Grid'5000: Hardware, 2012. (Cited on page 19.)

[Gro12] Michael Gross. Folding research recruits unconventional help. *Current Biology*, 22(2):R35 – R38, 2012. (Cited on page 20.)

[Hai98] G. Hains. Subset Synchronization in BSP Computing. In H. R. Arabnia, editor, *PDPTA'98 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 242–246, Las Vegas, July 1998. CSREA Press. (Cited on page 69.)

[HB99] Guy Horvitz and Rob H. Bisseling. Designing a BSP Version of ScaLAPACK. In *SIAM Conference on Parallel Processing for Scientific Computing 1999*, 1999. http://www.odysci.com/article/10101129880216 97. (Cited on page 53.)

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. (Cited on page 44.)

[HC02] Anthony Hall and Roderick Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Softw.*, 19(1):18–25, January 2002. (Cited on page 42.)

[HDH02] Per Brinch Hansen, Edsger W. Dijkstra, and C. A. R. Hoare. *The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002. (Cited on page 31.)

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323 – 364, 1977. (Cited on page 45.)

[HF93] G. Hains and C. Foisy. The Data-Parallel Categorical Abstract Machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, pages 56–67, Munich, June 1993. Springer. (Cited on page 67.)

[HGL+] Gaétan Hains, Jean-Guillaume Grebet, Chong Li, Mohamad Al Hajj Hassan, Aurélien Gonnay, and Frédéric Loulergue. *EXQIL Reference Manual*. EXQIM SAS, 24, rue de Caumartin, 75009 Paris. (Cited on pages x and 130.)

[Hin13] Pieter Hintjens, editor. *Code Connected Volume 1: Learning ZeroMQ*. CreateSpace Independent Publishing Platform, 2013. (Cited on pages 36, 37, and 38.)

[Hit97] Hitachi. HITACHI SR2201 Massively Parallel Processor, 1997. (Cited on page 13.)

[HJB98] David R. Helman, Joseph JáJá, and David A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. *J. Exp. Algorithmics*, 3, September 1998. (Cited on page 100.)

[HJS99] Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi. *Readings in computer architecture*. Morgan Kaufmann, 1999. (Cited on page 11.)

[HML+03] Th. Hauser, T. I. Mattox, R. P. LeBeau, H. G. Dietz, and P. G. Huang. Code Optimizations for Complex Microprocessors Applied to CFD Software. *SIAM J. Sci. Comput.*, 25(4):1461–1477, April 2003. (Cited on page 39.)

[HMS+98] Jonathan M.D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947 – 1980, 1998. (Cited on pages 51, 65, and 66.)

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, October 1974. (Cited on page 31.)

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. (Cited on page 41.)

[HOF+12] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, alan gara, George Chiu, Peter Boyle, Norman Chist, and Changhoan Kim. The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, 32(2):48–60, March 2012. (Cited on pages 3, 15, and 68.)

[HR92] Todd Heywood and Sanjay Ranka. A practical hierarchical model of parallel computation i. the model. *Journal of Parallel and Distributed Computing* , 16(3):212 – 232, 1992. (Cited on page 54.)

[HS98] Jonathan M. D. Hill and David B. Skillicorn. Practical Barrier Synchronisation. In *In 6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98*, pages 438–444. IEEE Computer Society Press, 1998. (Cited on page 49.)

[Inc77] Cray Research Inc. The Cray-1 Computer System, 1977. (Cited on page 11.)

[Int12] Intel Corp. An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors, 2012. (Cited on page 23.)

[JB07] C. R. Johns and D. A. Brokenshire. Introduction to the cell broadband engine architecture. *IBM J. Res. Dev.*, 51:503–519, September 2007. (Cited on pages 3, 21, and 68.)

[KB07] William J. Knottenbelt and Jeremy T. Bradley. Tackling large state spaces in performance modelling. In *Proceedings of the 7th international conference on Formal methods for performance evaluation*, SFM'07, pages 318–370, Berlin, Heidelberg, 2007. Springer-Verlag. (Cited on page 44.)

[KC02] H. Kuchen and M. Cole. The Integration of Task and Data Parallel Skeletons. *Parallel Processing Letters*, 12(2):141–155, 2002. (Cited on page 94.)

[KCD+08] Valentin Kravtsov, David Carmeli, Werner Dubitzky, Ariel Orda, Assaf Schuster, Mark Silberstein, and Benny Yoshpa. Quasi-opportunistic Supercomputing in Grid Environments. In *Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*, ICA3PP '08, pages 233–244, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on page 19.)

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49:589–604, July 2005. (Cited on pages 3, 21, and 68.)

[KDH11] Kamran Karimi, Neil Dickson, and Firas Hamze. High-performance Physics Simulations Using Multi-core CPUs and GPGPUs in a Volunteer Computing Context. *Int. J. High Perform. Comput. Appl.*, 25(1):61–69, February 2011. (Cited on page 19.)

[KHL12] Youry Khmelevsky, Gaétan Hains, and Chong Li. Automatic code generation within student's software engineering projects. In *Proceedings of the Seventeenth Western Canadian Conference on Computing Education*, WCCCE '12, pages 29–33, New York, NY, USA, 2012. ACM. (Cited on pages x and 130.)

[Kog81] Peter M. Kogge. *The Architecture of Pipelined Computers*. Taylor & Francis, 1981. (Cited on page 10.)

[KTJR05] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous Chip Multiprocessors. *Computer*, 38:32–38, 2005. (Cited on pages 3 and 68.)

[LG02] Isabelle Guérin Lassous and Jens Gustedt. Portable list ranking: an experimental study. *J. Exp. Algorithmics*, 7:7–, December 2002. (Cited on page 53.)

[LGB05] Frédéric Loulergue, Frédéric Gava, and David Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In Vaidy S. Sunderam, Geert Dick Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *Computational*

Science - ICCS 2005, volume 3515 of *Lecture Notes in Computer Science*, pages 1046–1054. Springer Berlin Heidelberg, 2005. (Cited on pages 51, 65, and 66.)

[LGG97] Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors. *Parallel priority Queue and list contraction: The BSP approach*, volume 1300 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1997. (Cited on page 53.)

[LGH12] Chong Li, Frederic Gava, and Gaetan Hains. Implementation of data-parallel skeletons: A case study using a coarse-grained hierarchical model. In *Proceedings of the 2012 11th International Symposium on Parallel and Distributed Computing*, ISPDC '12, pages 26–33, Washington, DC, USA, 2012. IEEE Computer Society. (Cited on pages ix and 85.)

[LH10] Chong Li and Gaétan Hains. A simple bridging model for high-performance computing. Technical Report TR-LACL-2010-12, Laboratoire d'Algorithmique, Complexité et Logique, Université Paris-Est, http://lacl.fr/Rapports/TR/TR-LACL-2010-12.pdf, 2010.

[LH11a] Chong Li and Gaétan Hains. SGL - programmation parallèle hétérogène et hiérarchique. In *Actes des troisièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel*, pages 93–96. 2011.

[LH11b] Chong Li and Gaétan Hains. A simple bridging model for high-performance computing. In *High Performance Computing and Simulation, 2011 International Conference on*, HPCS 2011, pages 249–256, Washington, DC, USA, 2011. IEEE Computer Society. (Cited on pages ix and 65.)

[LH12a] Chong Li and Gaétan Hains. SGL: towards a bridging model for heterogeneous hierarchical platforms. *Int. J. High Perform. Comput. Netw.*, 7(2):139–151, April 2012. (Cited on pages ix, 65, and 85.)

[LH12b] Chong Li and Gaétan Hains. SGL: vers la programmation parallèle hétérogène et hiérarchique. In *Actes des journées nationales de GDR-GPL, CIEL et EJCP 2012*, pages 127–130. 2012.

[LHF00] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000. (Cited on pages 2 and 68.)

[LHK+12] Chong Li, Gaétan Hains, Youry Khmelevsky, Brandon Potter, Jesse Gaston, Andrew Jankovic, Sam Boateng, and William Lee. Generating a real-time algorithmic trading system prototype from customized UML models (a case study). Technical Report TR-LACL-2012-09, Laboratoire d'Algorithmique, Complexité et Logique, Université Paris-Est, http://lacl.fr/Rapports/TR/TR-LACL-2012-09.pdf, 2012.

[Lib10] SGI Techpubs Library. SGI Altix ICE 8200 Series Hardware System User's Guide , 2010. (Cited on page 16.)

[LLN11] LLNL. Lawrence Livermore National Laboratory: BlueGene/L , 2011. (Cited on pages 14 and 15.)

[LLV12] LLVM. LLVM Documentation, 2012. (Cited on page 40.)

[Lou00] F. Loulergue. *Conception de langages fonctionnels pour la programmation massivement parallèle*. thèse de doctorat, Université d'Orléans, LIFO, 4 rue Léonard de Vinci, BP 6759, F-45067 Orléans Cedex 2, France, January 2000. (Cited on pages 2 and 68.)

[LZE97] Welf Löwe, Wolf Zimmermann, and Jörn Eisenbiegler. On Linear Schedules of Task Graphs for Generalized LogP-Machines. In *Proceedings of the Third International Euro-Par Conference on Parallel Processing*, Euro-Par '97, pages 895–904, London, UK, UK, 1997. Springer-Verlag. (Cited on page 58.)

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. (Cited on pages 53 and 66.)

[Mar12] Nathan Marz. Storm - Tutorial, 2012. (Cited on page 46.)

[MH05] Armelle Merlin and Gaétan Hains. A Generic Cost Model for Concurrent and Data-parallel Meta-computing. *Electron. Notes Theor. Comput. Sci.*, 128(6):3–19, May 2005. (Cited on page 44.)

[MH07] Armelle Merlin and Gaétan Hains. A bulk-synchronous parallel process algebra. *Comput. Lang. Syst. Struct.*, 33(3-4):111–133, October 2007. (Cited on page 44.)

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. (Cited on page 44.)

[Mil93] R. Milner. Action Structures and the Pi Calculus. Technical Report ECS-LFCS-93-264, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1993. (Cited on page 41.)

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40 and 41–77, September 1992. (Cited on page 41.)

[Mur97] Charles Murray. *The Supermen*. Wiley & Sons, 1997. (Cited on page 12.)

[Mur08] San Murugesan. Harnessing Green IT: Principles and Practices. *IT Professional*, 10(1):24–33, January 2008. (Cited on page 3.)

[MW98] William F. McColl and David Walker. Theory and Algorithms for Parallel Computation. In David J. Pritchard and Jeff Reeve, editors, *Euro-Par*, volume 1470 of *Lecture Notes in Computer Science*, pages 863–864. Springer, 1998. (Cited on pages 2 and 67.)

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6:40–53, March 2008. (Cited on page 23.)

[NHBY09] Rajesh Nishtala, Paul H. Hargrove, Dan O. Bonachea, and Katherine A. Yelick. Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society. (Cited on page 15.)

[Nor11] Norddeutsche Verbund für Hoch- und Höchstleistungsrechnen (HLRN). The SGI System - Hardware Overview , 2011. (Cited on page 21.)

[O'Ho7] John O'Hara. Toward a Commodity Enterprise Middleware. *Queue*, 5(4):48–55, May 2007. (Cited on page 36.)

[PGo4] Onil Nazra Persada and Thierry Goubier. Accelerating Raster Processing with Fine and Coarse Grain Parallelism in GRASS. In *Proceedings of the FOSS/GRASS Users Conference*, Bangkok, Thailand, 2004. (Cited on page 39.)

[PPdQN+01] Flavio L. C. Pádua, Guilherme A. S. Pereira, Jose P. de Queiroz Neto, Mario F. M. Campos, and Antonio O. Fernandes. Improving Processing Time of Large Images By Instruction Level Parallelism, 2001. (Cited on page 39.)

[Pur74] Charles J. Purcell. The control data STAR-100: performance measurements. In *Proceedings of the May 6-10, 1974, national computer conference and exposition*, AFIPS '74, pages 385–387, New York, NY, USA, 1974. ACM. (Cited on page 10.)

[RS98] R. O. Rogers and D. B. Skillicorn. Using the BSP cost model to optimise parallel neural network training. *Future Gener. Comput. Syst.*, 14(5-6):409–424, December 1998. (Cited on page 53.)

[sDo8] IBM journal of Research staff and Development. Overview of the IBM Blue Gene/P project. *IBM J. Res. Dev.*, 52:199–220, January 2008. (Cited on pages 3, 15, and 68.)

[SHD99] Andrew C. Simpson, Jonathan M.D. Hill, and Stephen R. Donaldson. BSP in CSP: Easy as ABC. In *Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 1299–1313. Springer Berlin Heidelberg, 1999. (Cited on page 44.)

[SHM97] David B. Skillicorn, Jonathan M. D. Hill, and William F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997. (Cited on page 49.)

[Sit99] Kragen Javier Sitaker. Beowulf mailing list FAQ, 1999. (Cited on page 17.)

[SK93] Holger Stoltze and Herbert Kuchen. Parallel Functional Programming Using Algorithmic Skeletons. In *PARCO*, pages 647–654, 1993. (Cited on pages 2 and 66.)

[SKP06] Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. High-performance and scalable MPI over InfiniBand with reduced memory usage: an in-depth performance analysis. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. (Cited on page 36.)

[SMD+10] A.C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh. Parallelism via Multithreaded and Multicore CPUs. *Computer*, 43(3):24 –32, March 2010. (Cited on page 69.)

[SPI] SPIRAL. Software Generation for the Cell Broadband Engine (Cell BE). (Cited on page 22.)

[Spro01] L.A. Spracklen. *SWAR Systems and Communications Applications*. University of Aberdeen, 2001. (Cited on page 39.)

[SS92] Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *J. Parallel Distrib. Comput.*, 14(4):361–372, April 1992. (Cited on pages 100 and 123.)

[Ste01] Thomas Sterling. *Beowulf Cluster Computing With Linux*. MIT Press, 2001. (Cited on page 17.)

[Sto12] Storm project. Storm - distributed and fault-tolerant realtime computation, 2012. (Cited on page 45.)

[Suj96] Ronald Sujithan. BSP Parallel Sorting by Regular Sampling - Algorithm and Implementation. Technical report, Computing Laboratory, University of Oxford, 1996. (Cited on page 100.)

[Ter07] Daniel Terdiman. Sony's Folding@home project gets Guinness record , 2007. (Cited on page 20.)

[TFM+11] Martin Thompson, Dave Farley, Barker Michael, Patricia Gee, and Andrew Stewart. Disruptor: high performance alternative to bounded queues for exchanging data between concurrent threads, 2011. (Cited on page 32.)

[Tho63] James E. Thornton. *Considerations in Computer Design - Leading up to the Control Data 6600*. Control Data Corporation, 1963. (Cited on page 9.)

[Tho70] James E. Thornton. *Design of a Computer - The Control Data 6600*. Scott Foresman & Co, 1970. (Cited on pages 9 and 10.)

[THS03] M. Osman Tokhi, M. Alamgir Hossain, and M. Hasan Shaheed. *Parallel Computing for Real-time Signal Processing and Control*. Springer, 2003. (Cited on page 11.)

[Tis98] Alexandre Tiskin. The bulk-synchronous parallel random access machine. *Theor. Comput. Sci.*, 196(1-2):109–130, April 1998. (Cited on pages 55, 72, 78, and 80.)

[Tis99] Alexandre Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD thesis, University of Oxford, 1999. (Cited on pages ix, 53, 65, 78, 100, 121, and 129.)

[TK96] Pilar de la Torre and Clyde P. Kruskal. Submachine Locality in the Bulk Synchronous Setting (Extended Abstract). In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, Euro-Par '96, pages 352–358, London, UK, UK, 1996. Springer-Verlag. (Cited on page 54.)

[Top] The Top500 List. (Cited on page 26.)

[Ung95] Stephen H. Unger. Hazards, Critical Races, and Metastability. *IEEE Trans. Comput.*, 44(6):754–768, June 1995. (Cited on page 31.)

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990. (Cited on pages viii, 2, 47, 48, 65, 67, 70, and 86.)

[Val11] Leslie G. Valiant. A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77(1):154–166, January 2011. (Cited on pages 56, 65, 69, 75, 77, and 86.)

[Wik] Wikipedia. Scope (computer science) - Lexical scoping and dynamic scoping. (Cited on page 43.)

[Wik08a] Wikipedia. Compute Unified Device Architecture , 2008. (Cited on page 23.)

[Wik08b] Wikipedia. IBM Roadrunner , 2008. (Cited on pages 24 and 25.)

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993. (Cited on page 86.)

[YBRM13] A. N. Yzelman, R. H. Bisseling, D. Roose, and K. Meerbergen. MulticoreBSP for C: a high-performance library for shared-memory parallel programming. *International Journal of Parallel Programming*, 2013. accepted for publication. (Cited on pages 51, 65, and 66.)

[YLL+11] Xue-Jun Yang, Xiang-Ke Liao, Kai Lu, Qing-Feng Hu, Jun-Qiang Song, and Jin-Shu Su. The TianHe-1A Supercomputer: Its Hardware and Software. *Journal of Computer Science and Technology*, 26(3):344–351, 2011. (Cited on page 26.)

Un Modèle de Transition Logico-Matérielle pour la Simplification de la Programmation Parallèle

Résumé :
La programmation parallèle et les algorithmes data-parallèles sont depuis plusieurs décennies les principales techniques de soutien l'informatique haute performance. Comme toutes les propriétés non-fonctionnelles du logiciel, la conversion des ressources informatiques dans des performances évolutives et prévisibles implique un équilibre délicat entre abstraction et automatisation avec une précision sémantique. Au cours de la dernière décennie, de plus en plus de professions ont besoin d'une puissance de calcul très élevée, mais la migration des programmes existants vers une nouvelle configuration matérielle ou le développement de nouveaux algorithmes à finalité spécifique dans un environnement parallèle n'est jamais un travail facile, ni pour les développeurs de logiciel, ni pour les spécialistes du domaine.

Dans cette thèse, nous décrivons le travail qui vise à simplifier le développement de programmes parallèles, en améliorant également la portabilité du code de programmes parallèles et la précision de la prédiction de performance d'algorithmes parallèles pour des environnements hétérogènes. Avec ces objectifs à l'esprit, nous avons proposé un modèle de transition nommé SGL pour la modélisation des architectures parallèles hétérogènes et des algorithmes parallèles, et une mise en œuvre de squelettes parallèles basés sur le modèle SGL pour le calcul haute performance. SGL simplifie la programmation parallèle à la fois pour les machines parallèles classiques et pour les nouvelles machines hiérarchiques. Il généralise les primitives de la programmation BSML. SGL pourra plus tard en utilisant des techniques de Model-Driven pour la génération de code automatique à partir d'une fiche technique sans codage complexe, par exemple pour le traitement de Big-Data sur un système hétérogène massivement parallèle. Le modèle de coût de SGL améliore la clarté de l'analyse de performance des algorithmes, permet d'évaluer la performance d'une machine et la qualité d'un algorithme.

Mots clés: Modèle de Transition, Programmation Parallèle, Prédiction de Performance, Sémantique de Langage, Communication Simplifiée, Architecture Hiérarchique, Machine Hétérogène, Bulk Synchronous Parallel, SGL.

A Software-Hardware Bridging Model for Simplifying Parallel Programming

Abstract :
Parallel programming and data-parallel algorithms have been the main techniques supporting high-performance computing for many decades. Like all non-functional properties of software, the conversion of computing resources into scalable and predictable performance involves a delicate balance of abstraction and automation with semantic precision. During the last decade, more and more professions require a very high computing power. However, migrating programs to a new hardware configuration or developing new specific-purpose algorithms on a parallel environment is never an easy work, neither for software developers nor for domain specialists.

In this thesis, we describe work that attempts to improve the simplicity of parallel program development, the portability of parallel program code, and the precision of parallel algorithm performance prediction for heterogeneous environments. With these goals in mind we proposed a bridging model named SGL for modelling heterogeneous parallel architectures and parallel algorithms, and an implementation of parallel skeletons based on SGL model for high-performance computing. SGL simplifies the parallel programming either on the classical parallel machines or on the novel hierarchical machines. It generalizes the BSML programming primitives. SGL can be used later with model-driven techniques for automatic code generation from specification sheet without any complex coding, for example processing Big Data on heterogeneous massively parallel systems. The SGL cost model improves the clarity of algorithms performance analysis; it allows benchmarking machine performance and algorithm quality.

Keywords: Bridging Model, Parallel Programming, Performance Prediction, Language Semantics, Simplified Communication, Hierarchical Architecture, Heterogeneous Machine, Bulk Synchronous Parallel, SGL.