An environment for peer-to-peer high performance computing

The Tung Nguyen

To cite this version:

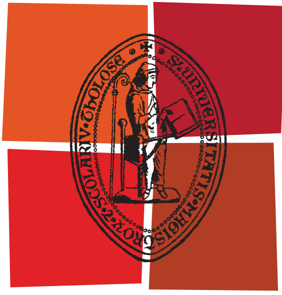
The Tung Nguyen. An environment for peer-to-peer high performance computing. Embedded Systems. Institut National Polytechnique de Toulouse - INPT, 2011. English. NNT : 2011INPT0105 . tel-04241121v2

HAL Id: tel-04241121
https://theses.hal.science/tel-04241121v2

Submitted on 13 Oct 2023

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

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



THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Systèmes informatiques critiques

Présentée et soutenue par :
The Tung NGUYEN

le : 16 Novembre 2011

Titre :

Un environnement pour le calcul intensif pair à pair

Ecole doctorale :
Systèmes (EDSYS)

Unité de recherche :
Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS)

Directeur(s) de Thèse :

Didier EL BAZ

Rapporteurs :

Frédéric MAGOULES
Olivier BEAUMONT

Membre(s) du jury :
Olivier BEAUMONT
Julien BOURGEOIS
Michel DIAZ
Didier EL BAZ
Frédéric MAGOULES
Toufik SAADI

UNIVERSITY OF TOULOUSE
DOCTORAL SCHOOL EDSYS

Ph.D. THESIS

to obtain the title of

Ph.D. of Science

of the University of Toulouse
Specialty : Computer Science

Defended by
The Tung Nguyen

An environment for peer-to-peer high performance computing

Thesis Advisor: Didier El Baz

prepared at LAAS-CNRS, Toulouse, CDA Team
defended on November 16, 2011

Jury :

Advisor :	Didier EL BAZ	-	LAAS-CNRS
Reviewers :	Olivier BEAUMONT	-	LaBRI-INRIA
	Frédéric MAGOULES	-	Ecole Centrale Paris
Examinators :	Julien BOURGEOIS	-	University of Franche-Comté
	Michel DIAZ	-	LAAS-CNRS
	Toufik SAADI	-	University of Picardie

Remerciement

Les travaux de thèse présentés dans ce mémoire ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS). Je tiens à remercier Raja CHATILA, Jean-Louis SANCHEZ, et Jean ARLAT, directeurs successifs du LAAS - CNRS, pour m'avoir accueilli au sein de ce laboratoire.

Je souhaite témoigner ma plus sincère gratitude à mon encadrant, Monsieur Didier EL BAZ, pour son écoute et ses conseils tout au long de ces trois années. Je le remercie tout particulièrement pour les nombreuses réflexions que nous avons pu mener ensemble et au travers desquelles il a partagé une partie de son expérience avec moi.

Je tiens également à remercier Messieurs Olivier BEAUMONT (LaBRI-INRIA), Julien BOURGEOIS (Université Franche Comté), Michel DIAZ (LAAS-CNRS), Frédéric MAGOULES (Ecole Centrale de Paris) et Toufik SAADI (Université Picardie) pour leur participation à mon jury de thèse. Particulièrement, je souhaite remercier Monsieur Olivier BEAUMONT et Monsieur Frédéric MAGOULES de m'avoir fait l'honneur d'être rapporteur de mes travaux. Leurs remarques et conseils avisés ayant permis d'améliorer la clarté de la présentation des idées véhiculées par le présent manuscrit.

Les travaux de la thèse présenté dans ce mémoire est dans le cadre du projet ARN-CIP. Je souhaite remercier les collaborateurs du projet pour les collaborations fort enrichissantes.

Je tiens également à remercier Monsieur Guillaume JOURJON et Monsieur Max OTT pour m'avoir invité et accueilli pour le séjour de quatre mois à NICTA, Sydney, Australie.

Mes remerciements vont naturellement à l'ensemble des membres du groupe CDA: Messieurs Jean-Michel ENJALBERT, Moussa ELKIHEL, Vincent BOYER et Mohamed LALAMI avec lesquels j'ai partagé ces trois années.

Un grand merci à mes amis vietnamiens, à mes frères Nam, Thach, Thanh et Van pour leur aide précieuse durant ces trois ans, y compris dans les moments difficiles.

Enfin, je souhaite remercier ma famille et Phuong Linh pour leur encouragement et leur soutien moral.

Abstract

The concept of peer-to-peer (P2P) has known great developments these years in the domains of file sharing, video streaming and distributed databases. Recent advances in microprocessors architecture and networks permit one to consider new applications like distributed high performance computing. However, the implementation of this new type of application on P2P networks gives raise to numerous challenges like heterogeneity, scalability and robustness. In addition, existing transport protocols like TCP and UDP are not well suited to this new type of application.

This thesis aims at designing a decentralized and robust environment for the implementation of high performance computing applications on peer-to-peer networks. We are interested in applications in the domains of numerical simulation and optimization that rely on tasks parallel models and that are solved via parallel or distributed iterative algorithms. Unlike existing solutions, our environment allows frequent direct communications between peers. The environment is based on a self adaptive communication protocol that can reconfigure itself dynamically by choosing the most appropriate communication mode between any peers according to decisions concerning the scheme of computation that are made at the application level or elements of context at transport level, like topology.

We present and analyze computational results obtained on several testeds like GRID'5000 and PlanetLab for the obstacle problem and nonlinear network flow problems.

Résumé

Le concept de pair à pair (P2P) a connu récemment de grands développements dans les domaines du partage de fichiers, du streaming vidéo et des bases de données distribuées. Le développement du concept de parallélisme dans les architectures de microprocesseurs et les avancées en matière de réseaux à haut débit permettent d'envisager de nouvelles applications telles que le calcul intensif distribué. Cependant, la mise en oeuvre de ce nouveau type d'application sur des réseaux P2P pose de nombreux défis comme l'hétérogénéité des machines, le passage à l'échelle et la robustesse. Par ailleurs, les protocoles de transport existants comme TCP et UDP ne sont pas bien adaptés à ce nouveau type d'application.

Ce mémoire de thèse a pour objectif de présenter un environnement décentralisé pour la mise en oeuvre de calculs intensifs sur des réseaux pair à pair. Nous nous intéressons à des applications dans les domaines de la simulation numérique et de l'optimisation qui font appel à des modèles de type parallélisme de tâches et qui sont résolues au moyen d'algorithmes itératifs distribués or parallèles. Contrairement aux solutions existantes, notre environnement permet des communications directes et fréquentes entre les pairs. L'environnement est conçu à partir d'un protocole de communication auto-adaptatif qui peut se reconfigurer en adoptant le mode de communication le plus approprié entre les pairs en fonction de choix algorithmiques relevant de la couche application ou d'éléments de contexte comme la topologie au niveau de la couche réseau.

Nous présentons et analysons des résultats expérimentaux obtenus sur diverses plateformes comme GRID'5000 et PlanetLab pour le problème de l'obstacle et des problèmes non linéaires de flots dans les réseaux.

Contents

1 Introduction **1**
- 1.1 Problem statement . . . 1
- 1.2 Contribution . . . 3
  - 1.2.1 Project ANR CIP . . . 3
  - 1.2.2 Contribution of the thesis . . . 3
- 1.3 Structure of the dissertation . . . 5

2 State of the art **7**
- 2.1 Introduction . . . 7
- 2.2 Peer-to-peer systems . . . 7
  - 2.2.1 Introduction . . . 8
  - 2.2.2 Characteristics . . . 8
  - 2.2.3 Architectures . . . 10
- 2.3 Distributed computing . . . 13
  - 2.3.1 Grid computing . . . 13
  - 2.3.2 Global computing . . . 13
  - 2.3.3 Peer-to-peer high performance computing . . . 14
- 2.4 High Performance Computing, parallel iterative methods . . . 17
  - 2.4.1 High Performance Computing . . . 17
  - 2.4.2 Parallel iterative methods . . . 17
- 2.5 Conclusion . . . 25

3 P2PSAP - A self-adaptive communication protocol **27**
- 3.1 Introduction . . . 27
- 3.2 State of the art in adaptive communication protocols . . . 28
  - 3.2.1 Micro-protocol approach . . . 29
  - 3.2.2 Cactus framework and CTP protocol . . . 31
- 3.3 P2PSAP Protocol architecture . . . 33
  - 3.3.1 Socket API . . . 33
  - 3.3.2 Data channel . . . 33
  - 3.3.3 Control channel . . . 34
- 3.4 Example of scenario . . . 36
- 3.5 Some modifications to Cactus . . . 37
- 3.6 Self-adaptive mechanisms . . . 37
  - 3.6.1 Choice of protocol features . . . 38
  - 3.6.2 New micro-protocols . . . 39
  - 3.6.3 (Re)Configuration . . . 43
- 3.7 Computational experiments . . . 47
  - 3.7.1 Network flow problems . . . 47
  - 3.7.2 Platform . . . 49

3.7.3 Computational results 50
3.8 Chapter summary 51

4 Centralized version of the environment for peer-to-peer high performance computing 53
4.1 Introduction 53
4.2 Global architecture 54
4.3 Programming model 55
4.3.1 Communication operations 55
4.3.2 Application programming model 56
4.4 Implementation 58
4.4.1 User daemon 58
4.4.2 Resource manager 58
4.4.3 Application repository 59
4.4.4 Task manager 59
4.5 Computational results 60
4.5.1 Obstacle problem 60
4.5.2 Implementation 62
4.5.3 NICTA testbed and OMF framework 65
4.5.4 Problems and computational results 65
4.6 Chapter summary 68

5 Decentralized environment for peer-to-peer high performance computing 71
5.1 Introduction 71
5.2 Hybrid resource manager 72
5.2.1 General topology architecture 73
5.2.2 IP-based proximity metric 74
5.2.3 Topology initialization 74
5.2.4 Tracker joins 74
5.2.5 Peer joins 75
5.2.6 Tracker leaves 75
5.2.7 Peer leaves 76
5.2.8 Peers collection 76
5.3 Hierarchical task allocation 77
5.4 Dynamic application repository 78
5.5 File transfer 78
5.6 New communication operations 79
5.7 Computational experiments 80
5.7.1 New approach to the distributed solution of the obstacle problem 80
5.7.2 Grid'5000 platform 85
5.7.3 Experimental results 87
5.8 Chapter summary 89

6 Fault-tolerance in P2PDC 91
6.1 Introduction 91
6.2 State of the art in fault-tolerance techniques 92
6.2.1 Replication techniques 92
6.2.2 Rollback-recovery techniques 93
6.3 Choices of fault-tolerance mechanisms 96
6.4 Worker failure 96
6.4.1 Coordinated checkpointing rollback-recovery for synchronous iterative schemes 97
6.4.2 Uncoordinated checkpointing rollback-recovery for asynchronous iterative schemes 99
6.5 Coordinator failure 100
6.6 Computational experiments 102
6.6.1 Coordinator replication overhead 102
6.6.2 Worker checkpointing and recovery overhead 102
6.6.3 Influence of worker failures on computational time 103
6.7 Chapter summary 104

7 Contribution to a web portal for P2PDC application deployment 107
7.1 Introduction 107
7.2 Background 107
7.2.1 OML 107
7.2.2 OMF and its Portal 109
7.3 Motivation 110
7.4 A new measurement channel for P2PDC 112
7.4.1 Hierarchical measurements collection 112
7.4.2 Application to task deployment 114
7.5 Chapter summary 117

8 Conclusions and perspectives 119

A OMF's Experiment Description Language 123
A.1 OMF's Experiment Description Language (OEDL) 123
A.2 Examples of experiment description 125

B How to write and run P2PDC applications 127
B.1 How to write a P2PDC application 127
B.2 Compile and run a P2PDC application 128
B.2.1 Compile a P2PDC application 128
B.2.2 Run a P2PDC application 128

C List of publications 129

Bibliography 131

List of Figures

2.1 Centralized peer-to-peer architecture 11
2.2 Unstructured decentralized peer-to-peer architecture 11
2.3 Hybrid peer-to-peer architecture 12
2.4 SETI@home architecture. 14
2.5 Peer groups hierarchy in JNGI framework. 15
2.6 Interactions between peers in ParCop. 16
2.7 Synchronous parallel iteration. 19
2.8 Asynchronous parallel iterations. 19

3.1 Example of x-kernel protocol graph configuration 30
3.2 FPTP compositional architecture 31
3.3 CTP - Configurable Transport Protocol 32
3.4 P2PSAP Protocol Architecture 33
3.5 Protocol session life cycle 35
3.6 Example of P2PSAP reconfiguration scenario 36
3.7 Synchronous communication mode. 40
3.8 Synchronous micro-protocol. 40
3.9 Asynchronous communication mode 41
3.10 Asynchronous micro-protocol. 41
3.11 Micro-protocol TCP New-Reno congestion avoidance 42
3.12 Micro-protocol DCCPAck 44
3.13 Micro-protocol DCCP Window Congestion Control 45
3.14 Network used for computational tests on the LAASNETEXP experimental network 49

4.1 General architecture of P2PDC 54
4.2 Activity diagram of a parallel application 57
4.3 Centralized topology of resource manager. 59
4.4 Slice decomposition of the 3D obstacle problem. 63
4.5 Basic computational procedure at node P_k. 64
4.6 Termination detection in the case of slice decomposition. 64
4.7 Computational results in the case of the obstacle problem with size $96 \times 96 \times 96$ 66
4.8 Computational results in the case of the obstacle problem with size $144 \times 144 \times 144$ 67

5.1 General topology architecture. 73
5.2 Trackers topology. 74
5.3 Trackers topology after a new tracker has joined. 75
5.4 Trackers topology after a tracker has disconnected. 76

5.5 Allocation graph. 77
5.6 Pillar decomposition of the 3D obstacle problem. 81
5.7 Basic computational procedure at node $P_{r,c}$ with pillar decomposition. 83
5.8 Termination detection in the case of pillar decomposition. 84
5.9 Behavior of workers implementing new termination method. 84
5.10 Evolution of the activity graph. 86
5.11 Efficiency of distributed algorithms in the cases 1 and 2. 88
5.12 Number of relaxations of asynchronous iterative algorithms in the cases 1 and 2. 88
5.13 Efficiency of distributed algorithms in the case 3 89

6.1 Passive replication. 93
6.2 Active replication. 93
6.3 Semi-active replication. 94
6.4 Coordinated checkpointing process for synchronous iterative schemes. 97
6.5 Recovery process upon a worker failure for synchronous iterative schemes. 98
6.6 Uncoordinated checkpointing process for asynchronous iterative schemes. 100
6.7 Recovery process upon a worker failure for asynchronous iterative schemes. 100
6.8 Replication of coordinators. 101
6.9 Computational time for number of worker failures from 0 up to 10. . 104

7.1 OML - the OMF Measurement Library 108
7.2 Overview of OMF architecture from the user's point of view 109
7.3 A web portal for P2PDC application deployment 111
7.4 Current measurement architecture 112
7.5 Maximum error measurement for the obstacle problem with 2 peers at NICTA and 2 peers on PlanetLab 113
7.6 Hierarchical measurement architecture 113
7.7 Multi-level hierarchical measurement architecture 114
7.8 Task deployment via OML 115
7.9 Computational results on PlanetLab 118

List of Tables

3.1 Choice of P2PSAP protocol features according to algorithmic and communication context 38
3.2 P2PSAP protocol composition according to algorithmic and communication context 46
3.3 Computational results for network flow problems on LAASNETEXP 50

5.1 Machine characteristics and sequential computational time 86

6.1 Worker checkpointing and recovery overhead 103

Chapter 1

Introduction

Contents

1.1 Problem statement **1**
1.2 Contribution **3**
1.2.1 Project ANR CIP 3
1.2.2 Contribution of the thesis 3
1.3 Structure of the dissertation **5**

1.1 Problem statement

The design of complex systems like aircrafts and space vehicles requires a very large amount of computational resources. The same remark can be made in the domain of services like meteorology and telecommunications. The most popular solutions use supercomputers that are composed of hundreds thousands of processors connected by a local high-speed computer bus. The system, called the K Computer, at the RIKEN Advanced Institute for Computational Science (AICS) in Kobe, Japan presently keeps the top position of TOP500 list of world's supercomputers [top]. However, supercomputers are very expensive and are only located in research laboratories and organizations funded by governments and big industrial enterprises. With the presence of high speed backbone networks, cost-effective solutions that share common resources like supercomputers have been proposed; they correspond to the so-called *Grid*. Grid permit users of an organization to collect more resources from other organizations. However, resources on the grid are generally managed by administrators with hard system configuration and centralized management that limit the flexibility and the availability. Conditions of authentication are also very restrictive for users who want to reserve resources and execute computations.

Recently, Peer-to-Peer (P2P) applications have known great developments. These applications were originally designed for file sharing, e.g. Gnutella [gnu] or FreeNet [fre] and are now considered to a larger scope from video streaming to system update and distributed database. Recent advances in microprocessors architecture and networks permit one to consider new applications like High Performance Computing (HPC) applications. Therefore, there is a real stake at developing new protocols and environments for HPC since this can lead to economic and attractive solutions.

Along with the advances in system architectures, many parallel or distributed numerical methods have been proposed. Among them, parallel or distributed iterative algorithms take an important part [El Baz 1998]. Nevertheless, task parallel model and distributed iterative methods for large scale numerical simulation or optimization on new architectures raises to numerous challenges. This is particularly true in the case of P2P computing where questions related to communication management, resource management, scalability, peer volatility and heterogeneity have to be addressed. In particular, the underlying transport protocols must be suited to the profile of the application. However, existing transport protocols are not well suited to HPC applications. Indeed, transport protocols like TCP [TCP 1981] and UDP [UDP 1980] were originally designed to provide ordered and reliable transmission to the applications and are no longer adapted to both real-time and distributed computing applications. In particular, P2P applications require a message based transport protocol whereas TCP only offers a stream-based communication. Recently, new transport protocols have been standardized such as SCTP [SCT 2000] and DCCP [Kohler 1999]. Nevertheless, these protocols still do not offer the complete modularity needed to reach an optimum solution pace in the context of HPC and P2P.

To the best of our knowledge, most of existing environments for peer-to-peer high performance computing are based on a centralized architecture where the centralized server may become a bottleneck that leads to a single failure point of the system. Moreover, they are only devoted to bag-of-tasks applications where the application is decomposed into independent tasks with no synchronization nor dependencies between tasks. Few systems consider connected problems where there are frequent communications between tasks like applications solved by parallel or distributed iterative algorithms. Most of them are developed in Java language that is not efficient for HPC applications. We note that the implementation of connected problem is more difficult than bag-of-tasks applications and believe that asynchronous iterative algorithms are well suited to the solution of HPC applications on peer-to-peer networks.

This thesis aims at designing an environment for the implementation of high performance computing on peer-to-peer networks. We are interested in applications in the domains of numerical simulation and optimization that rely on tasks parallel model and that are solved via parallel iterative algorithms. Our environment is built on a decentralized architecture whereby peers can communicate directly. Many aspects are considered like the scalability, resource collection, self-organization and robustness. We have followed a classical approach for the design of distributed computing environments, indeed, we have designed first a self-adaptive communication protocol dedicated to peer-to-peer computing in order to allow rapid message exchanges between peers. Then, we have designed our decentralized environment. Our approach is developed in C language that is more efficient for HPC applications than Java language.

1.2 Contribution

In this section, we shall enumerate our contributions. We note that this work was funded by ANR under project CIP (ANR-07-CIS7-011) [anr].

1.2.1 Project ANR CIP

The project ANR CIP coordinated by Dr. Didier El Baz, LAAS-CNRS, started January 2008, it aims at proposing innovative tools and demonstrators for the implementation of high performance computing applications over peer-to-peer networks. The project is composed of three sub-projects:

Sub-project P2PDC: *Environment for peer-to-peer high performance computing.*

The sub-project P2PDC, in charge of CDA team at LAAS-CNRS, aims at designing an environment for the implementation of high performance computing applications on peer-to-peer networks.

Sub-project P2PPerf: *Simulation tool for peer-to-peer high performance computing.*

P2PPerf developed by OMNI team at LIFC is a simulation tool for large scale peer-to-peer computing. P2PPerf permits one to simulate peer-to-peer computations involving thousands peers on several network architectures. The tool P2PPerf is constituted of two modules: the module CompPerf evaluates the computational time of sequential parts of a program; the module NetPerf allows to simulate the network part of a peer-to-peer application.

Sub-project P2PDem: *Demonstrators and applicative challenges.*

Sub-project P2PDem consists of two parts. P2PPro, developed by the team at MIS, aims at developing demonstrators for complex combinatorial applications that come from the domain of logistic. P2PSimul, developed by the team at ENSEEIHT-IRIT, aims at developing demonstrators for numerical simulation applications. Two problems related to domains of financial mathematics and process engineering are considered.

1.2.2 Contribution of the thesis

Our contributions concern works done in the framework of the sub-project P2PDC. They include the following points.

- The design and implementation of a self-adaptive communication protocol (P2PSAP) dedicated to P2P HPC applications. P2PSAP was developed by using the Cactus framework [Hiltunen 2000] that makes use of micro-protocols. P2PSAP protocol can reconfigure dynamically by choosing the most appropriate communication mode between any peers according to decisions made at

the application level or elements of context like topology at transport level. In particular, we have designed a set of micro-protocols like Synchronous, Asynchronous, DCCP Ack, DCCPCongestionAvoidance, respectively that permit one to implement synchronous or asynchronous communications and DCCP congestion control function [Kohler 1999].

- The design and implementation of a decentralized and robust environment (P2PDC) for peer-to-peer high performance computing that makes use of P2PSAP protocol in order to allow direct communications between peers. This contribution is divided into three phases.
  The first phase aims at defining the global architecture of P2PDC with mains functionalities and proposing programming model that is suited to peer-to-peer high performance computing applications. In this phase, we have developed a first version of P2PDC with centralized and simple functionalities. The goal of the implementation of the centralized version was to validate the programming model by a specific application. Moreover, this allowed us to provide to partners of the project CIP with a programming model and a first version of P2PDC environment so that they can quickly develop applications for P2PDC.
  In the second phase, we have developed a decentralized version of P2PDC that includes some features aimed at making P2PDC more scalable and efficient. Indeed, a hybrid resource manager manages peers efficiently and facilitates peers collection for computation; a hierarchical task allocation mechanism accelerates task allocation to peers and avoids connection bottleneck at submitter. Furthermore, a file transfer functionality was implemented that allows to transfer files between peers.
  The last phase deals with fault-tolerance aspects of P2PDC.
  We note that the main originalities of our approach are:

  - a decentralized and robust environment that permits frequent direct communications between peers;
  - an environment developed in C language that is more efficient for HPC applications;
  - an environment that aims at facilitating programming and which relies on the use of a limited number of communication operations, basically: *send*, *receive* and *wait* operations; moreover, the programmer does not need to specify the communication mode between any two peers, he rather chooses an iterative scheme of computation, i.e. a synchronous scheme or an asynchronous scheme or let the protocol choose according to elements of context like topology of the network.
  - the possibility to combine efficiently parallel or distributed asynchronous iterative schemes of computation with a peer-to-peer computing environment.

- The use of P2PDC environment for the solution of a numerical simulation problem, i.e. the obstacle problem [Spitéri 2002] and the test of this application on several platforms like Nicta testbed and GRID'5000 with up to 256 machines. Along with the evolution of P2PDC environment and scaling up experimental platforms, the code for the solution of the obstacle problem has also been modified in order to adapt to these evolutions and to improve the efficiency of parallel algorithms. In particular, we have consider several decomposition of the original problem.

1.3 Structure of the dissertation

This thesis is organized as follows:

- **Chapter 2** presents a state of the art in domains that inspire the contribution of this thesis. We concentrate first on peer-to-peer systems. Afterwards, we precise approaches related to distributed computing, i.e. grid computing, global computing and peer-to-peer high performance computing. An overview on existing environments for peer-to-peer high performance computing is also presented. Finally, we deal with HPC applications, fixed-point problems and parallel iterative algorithms. In particular, asynchronous iterative algorithms are considered.

- **Chapter 3** describes the Peer-To-Peer Self Adaptive communication Protocol, a self-adaptive communication protocol dedicated to peer-to-peer high performance computing. We display the architecture of P2PSAP and detail self-adaptive mechanisms of the protocol for peer-to-peer high performance computing. A first series of computational experiments for a nonlinear optimization problem is presented and analyzed in order to illustrate the behavior of the proposed protocol for HPC applications.

- **Chapter 4** presents the first version of the P2PDC environment. In this chapter, we define the global architecture of P2PDC with mains functionalities. Moreover, we propose a new programming model that is suited to peer-to-peer high performance computing applications and more particularly applications solved by iterative algorithms. A centralized implementation of P2PDC with simple functionalities is developed in order to validate the programming model. Computational results are displayed and analyzed for a numerical simulation problem solved on NICTA testbed.

- **Chapter 5** details the decentralized version of P2PDC that includes some features aimed at making P2PDC more scalable and efficient. Indeed, a hybrid resource manager manages peers efficiently and facilitates peers collection for computation; a hierarchical task allocation mechanism accelerates task allocation to peers and avoids connection bottleneck at submitter. Furthermore, a file transfer functionality is implemented in order to allow file

transfer between peers. Moreover, some modifications to the communication operation set are introduced. Experimental results for the obstacle problem on GRID'5000 platform with up to 256 peers are displayed and analyzed.

- **Chapter 6** deals with the fault-tolerance mechanisms in P2PDC to cope with peer volatility. The fault-tolerance mechanisms can adapt themselves according to peer role and computational scheme. Computational results are presented and analyzed for several cases with fault injection.
- **Chapter 7** presents the first ideas related to the use of OML [White 2010], OMF [Rakotoarivelo 2010] and its Web portal [Jourjon 2011] in order to facilitate the deployment of P2PDC applications on peer-to-peer networks. Some aspects related to measurements in P2P applications are also presented.
- **Chapter 8** gives some conclusions on our work and deals also with future work.

Chapter 2

State of the art

Contents

2.1 Introduction 7

2.2 Peer-to-peer systems 7

2.2.1 Introduction 8

2.2.2 Characteristics 8

2.2.3 Architectures 10

2.3 Distributed computing 13

2.3.1 Grid computing 13

2.3.2 Global computing 13

2.3.3 Peer-to-peer high performance computing 14

2.4 High Performance Computing, parallel iterative methods . 17

2.4.1 High Performance Computing 17

2.4.2 Parallel iterative methods 17

2.5 Conclusion 25

2.1 Introduction

This chapter presents a state of the art in domains that inspire the contribution of this thesis. Section 2.2 concentrates on peer-to-peer systems: the definition and characteristics of peer-to-peer systems are presented. We describe also in this section different architectures of peer-to-peer systems. In the section 2.3, we present an overview on existing environments for distributed computing. Sections 2.4 deals with High Performance Computing (HPC) applications and parallel or distributed iterative methods. We present in this section the definition as well as a comparison between synchronous and asynchronous iterative schemes. We concentrate on asynchronous iterative schemes since these schemes seem more attractive than synchronous iterative schemes in the case of heterogeneous architectures like peer-to-peer networks.

2.2 Peer-to-peer systems

Peer-to-Peer (P2P) systems have become well-known those last years thanks to file sharing systems on the Internet like Gnutella [gnu] or FreeNet [fre]. They are now

considered to a larger scope from video streaming to system update and distributed database.

In this section, we shall define peer-to-peer systems and present their essential characteristics. Afterwards, we shall describe the different architectures of peer-to-peer systems that may be encountered.

2.2.1 Introduction

In the literature, there are many definitions of peer-to-peer systems.

Definition 2.1 *[wik] Peer-to-peer computing or networking is a distributed application architecture that partitions tasks or workloads between peers. Peers are equally privileged, equipotent participants in the application.*

Definition 2.2 *[Oram 2001] P2P is a class of applications that take advantage of resources storage, cycles, content, human presence available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, peer-to-peer nodes must operate outside the DNS and have significant or total autonomy of central servers.*

Definition 2.3 *[Dejan 2003] The term peer-to-peer refers to a class of systems and applications that use distributed resources to perform a function in a decentralized manner.*

In principle, in peer-to-peer systems, all participants play a similar role. This differs from client/server architectures, in which some computers are dedicated to serving the others. For example in the case of file sharing on peer-to-peer networks, computers are taking part in turn to supply and demand, they can be client and server as well; they are peers.

2.2.2 Characteristics

We distinguish several characteristics of peer-to-peer systems.

2.2.2.1 Decentralization

A centralized entity may become a bottleneck and constitute a single failure point of the overall system. Peer-to-peer systems reduce less or more this drawback according to their architecture (see subsection 2.2.3). In Napster music sharing system [nap], there is a centralized directory of files but peers download file directly from each others. In the Gnutella 0.4 [gnu], there is no centralized entity. Nevertheless, the less the entities are centralized in the peer-to-peer systems, the more the implementation is difficult.

2.2.2.2 Scalability

The scalability of a P2P network is often described as the main quality of such a system. Scaling is often defined in relationship with *the size of the problem* and not in relationship with *the size of the system*. However, in networked systems, the problem of scaling is set, most of the time, along with the size of the network, i.e. the number of nodes and arcs of the graph representing the network according to a topology point of view.

In [Jourjon 2005] G. Jourjon and D. El Baz have proposed a definition of the principle of scalability for a computing system on a peer-to-peer network.

Definition 2.4 *[Jourjon 2005] The scalability of a P2P network designed for global computing is its capacity to maintain its efficiency when peers join or leave the system.*

Aspects related to efficiency of a global computing system over a P2P network are numerous, including the routing efficiency, the search effectiveness, the algorithm's speed, etc.

2.2.2.3 Transparency

Definition 2.5 *[Jourjon 2005] The transparency can be defined as the property to make undistinguished local or remote access to all parts of the task and data set needed for computation.*

The above definition means that, whatever happens to the network, each peer still online can have access to the entire set of components for the computation. This can be translated by the fact that we need to envision duplication and a good distribution of this set of data and tasks.

2.2.2.4 Robustness

Robustness, in a general point of view, is the system's ability to maintain stability when a fault occurs. Faults in a peer-to-peer network are the failures of peers or links. These failures may occur due to several reasons: attacks by viruses, machines turned off, congestion of the first IP router, etc. If we want to model this event with the help of graph theory, then a fault can be represented by the expulsion of a node and all its incoming and outgoing edges or the removal of an edge.

The robustness of a peer-to-peer network can be defined as follows.

Definition 2.6 *[Jourjon 2005] The robustness of a P2P network is its capacity to stabilize itself despite failure of some of its components (peers or links).*

2.2.2.5 Performance

The performance is a significant concern in peer-to-peer systems. These systems aim at improving their performance by aggregating new storage and computer cycles. However, due to the decentralized nature of the models, the performance is conditioned by three types of resources: processing, storage and network management.

In particular, communication delays can be very significant in large-scale networks. In this case, bandwidth is an important factor when it comes to spreading a large number of messages or share files between multiple peers. This also limits the scalability of the system.

2.2.3 Architectures

Since their emergence in the late 90s, peer-to-peer systems have evolved and diversified in their architecture. We can classify peer-to-peer networks into three major classes: centralized, decentralized and hybrid architectures [Bo 2003, Lua 2005]. In the sequel, we will detail these classes of architectures as well as their advantages and drawbacks.

2.2.3.1 Centralized architecture

The first class of peer-to-peer networks that corresponds to the first generation is the centralized architecture that is very similar to the client/server architecture. In this model, a stable central server indexes all the peers of the system and stores information about the content. When receiving a request from a peer, the central server selects another peer in its directory that matches the request. Then, communications are carried out directly between two peers. Examples of this generation are Napster [nap] and BitTorrent [bit]. Figure 2.1 shows a diagram of a centralized peer-to-peer architecture.

By centralizing information, this type of architecture makes exhaustive search algorithms particularly effective, with minimal communications; in addition, it is easier to implement. However, the centralized server may become a bottleneck that leads to a single failure point in the system: when the number of peers and requests increases, the server must be a very powerful machine and needs very high bandwidth; moreover, if the server crashes or is attacked successfully by a virus or a malicious person, then the whole system collapses.

2.2.3.2 Decentralized architectures

The second class of peer-to-peer networks corresponds to decentralized architectures that does not rely on any server. This type of architecture corresponds to the so-called second generation of peer-to-peer networks. Each peer has exactly the same possibilities as other peers and can act as client or server indistinctly. This class can be divided into two subclasses: unstructured and structured.

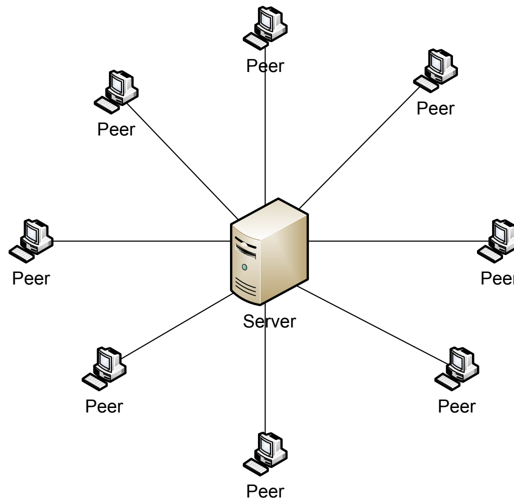


Figure 2.1: Centralized peer-to-peer architecture

In the first subclass, the logical topology is often random. Each peer indexes its own shared resources. A request from a peer is broadcasted directly to neighboring peers, which in turn broadcast the request to their neighbors. This is repeated until the application has received the answer or a maximum number of stages of flooding has been reached. One can find Gnutella 0.4 in this class [gnu]. Figure 2.2 shows a diagram of a unstructured decentralized peer-to-peer architecture.

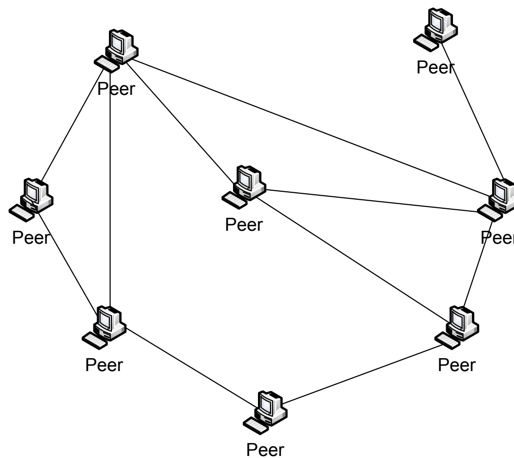


Figure 2.2: Unstructured decentralized peer-to-peer architecture

The advantage of this class of architecture is to provide a robust system: since each peer turning into client/server indistinctly, the disappearance of one or more of them will not lead to system crash down. In contrary, the communication traffic will be heavy and the search much longer. When scaling, the more peers in a network,

the more communication traffic.

In the second subclass, the logical topology is structured like for example in a ring (Chord [Stoica 2003]), d-dimension (CAN [Ratnasamy 2001]), etc. They are often structured topologies using Distributed Hash Tables (DHT). Each peer indexes some of the shared resources of the network and owns some of the hash table of the system. The request is transmitted according to the structured topology and is ensured of success after a specified number of steps has been reached under ideal conditions.

The second class is more robust than the first class and guarantees the anonymity of peers. It provides self-organization when scaling and offers search time reduction through the hash table. However, this class requires a fairly heavy protocol for maintaining the topology structure.

2.2.3.3 Hybrid architecture

The third class of peer-to-peer networks corresponds to hybrid architecture that combine elements of both centralized and decentralized architectures. This architecture is the third generation of peer-to-peer networks. This architecture makes use of multiple peers, called super-peers or super-nodes, that index and monitor a set of peers connected to the system. A super-peer is connected to other super-peers following the model of the decentralized architecture. The number of super-peers should remain large enough to avoid system shutdown in case of loss or stop of a super-peer. Therefore, if a search for a peer is not indexed by the super-peer which is attached to it, then it sends the request to another super-peer. The system KaZaA [kaz] is an example of peer-to-peer network of this generation. Figure 2.3 shows a diagram of a hybrid peer-to-peer architecture.

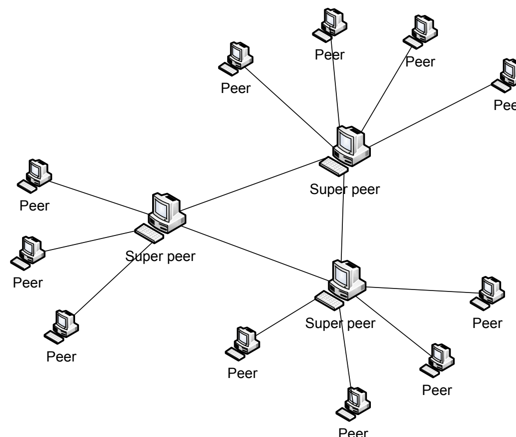


Figure 2.3: Hybrid peer-to-peer architecture

There are two types of hybrid architectures: the static and dynamic hybrid architectures. In the first case, a machine can become a super-peer according to the

choice of users. In the second case, a peer can automatically become a super-peer under certain conditions.

This class of architecture has advantages of both two previous classes, i.e. fault-tolerance and query traffic and search time reduction. However, it is more complex to implement.

2.3 Distributed computing

In this section, we precise some approaches related to distributed computing, i.e. grid computing, global computing and peer-to-peer high performance computing that share the goal of better utilizing computing resources connected to the network. We present also an overview on existing middlewares and environments for each approach.

2.3.1 Grid computing

When the need for high performance computing has increased, grid computing has emerged as a solution for resources sharing between organizations. Grid computing [Magoulès 2009] makes use of supercomputers, clusters and park of workstations owned by universities, research labs inter-connected by high bandwidth network links in order to form a super virtual computer. Resources inside an organization are generally turned on all the time and are connected by reliable high bandwidth network. Several middlewares have been proposed to facilitate the implementation of HPC applications on grid environments like Globus [Foster 1996], Condor [Litzkow 1988]. However, resources on the grid are generally managed by administrators of organizations with hard system configuration and centralized management. Users have to authenticate in order to use resources on the grid. Thus, grid computing middlewares provides only limited reconfigurability and scalability.

2.3.2 Global computing

A number of systems that attempt to use idle computing power of volunteer computers or institutional computers connected to the Internet in order to solve some large granularity applications have also been proposed. These systems are called global computing systems. Global computing systems are generally based on a centralized architecture where jobs are submitted to a centralized server and workers consult the server to get job. The central server in these systems may become a bottleneck that leads to a single point of failure.

Projects SETI@home [set] and GENOME@home [gen] are pioneers of global computing. These systems are often restricted to a specific application. SETI@home [set] uses volunteer computers around the world to analyze radio signals from space, whose goal is to detect intelligent life outside Earth. In GENOME@home [gen] and its successor Folding@home [fol], volunteer computers are used to perform computationally intensive simulations of protein folding and other molecular dynamics

whose goal is to design new genes and proteins for the purpose of better understanding how genomes evolve, and how genes and proteins operate. The model of these systems has been used to create the general global computing platform BOINC [Anderson 2004] that is now used in many projects. Volunteer computers are general PCs and workstations connected to the Internet with low bandwidth. Moreover, they are turned off or disconnected frequently at unpredictable rate. In these systems, data are split into small work units which are stored in a database. A central server then assigns work units to volunteer computers asking for work. Figure 2.4 presents the architecture of SETI@home.

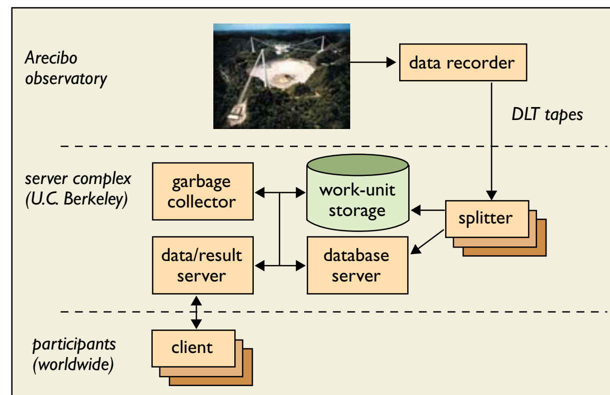


Figure 2.4: SETI@home architecture.

XtremWeb [xtr] provides a platform for global computing that collects not only volunteer computers but also institutional computers connected to LAN or to the Internet. Moreover, XtremWeb allows multi-users, multi-applications, i.e. some specific users can submit their applications to servers and workers can get jobs of different applications from servers. To the best of our knowledge, XtremWeb does not implement yet direct communication between workers.

2.3.3 Peer-to-peer high performance computing

In peer-to-peer high performance computing, all participants, i.e. peers, can carry out their application. Peers can be workstations at companies and organizations or even individual PCs at home connected to the Internet. Moreover, peer-to-peer computing systems try to eliminate centralized entities and allow the reconfiguration in the case of peer disconnection or failure.

Several middlewares and environments for peer-to-peer high performance computing have been proposed.

JNGI [Verbeke 2002] is a decentralized framework for peer-to-peer distributed computing that makes use of JXTA [jxt] in order to build a virtual network of peers on top of physical network. JNGI uses the concept of peer group in JXTA in order to divide peers into groups according to functionality. In JNGI, there are three group

type: monitor groups, worker groups and task dispatcher groups. Monitor groups handle peers joining the framework and high-level aspects of the job submission process. Each worker group is composed of a task dispatcher group and workers. Task dispatcher group distributes tasks to workers and workers perform received tasks. In [Ernst-Desmulier 2005], JNGI has been extended to permit one to constitute similarity worker groups that contain workers with similar characteristics like CPU speed or memory size in order to improve task dispatching efficiency. In order to cope with the scalability problem, JNGI enables one to have a hierarchy of monitor groups (see Figure 2.5). Job code and job data are submitted to a code repository manager. Upon receiving a job submission, the task dispatcher groups consult the code repository manager for tasks to be performed and distribute tasks to workers. JNGI considers only bag-of-tasks applications that does not need any synchronization and have no dependencies between tasks leading to no communication between tasks.

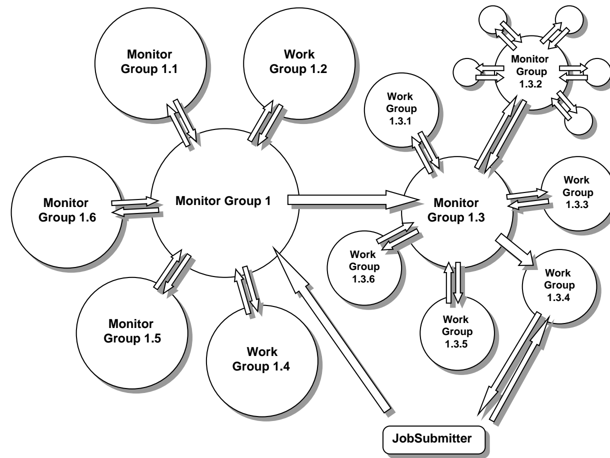


Figure 2.5: Peer groups hierarchy in JNGI framework.

Ourgrid [Andrade 2003] is a peer-to-peer middleware for sharing computing cycles through different companies or organizations. The main motivation of the Ourgrid project is to develop a middleware that automatically gathers resources across multiple organizations and to provide easy access to resources. Ourgrid is devoted to bags-of-tasks application class.

ParCop [Al-Dmour 2004] is a decentralized peer-to-peer computing system. ParCop is characterized by the integration of various features such as scalability, adaptive parallelism, fault tolerance, dynamic resource discovery, and ease of use. ParCop supports Master/Worker style of applications which can be decomposed into non-communicating and independent tasks. A peer in ParCop can be a Master or a Worker, but not both at the same time (see Figure 2.6). A Master distributes tasks to workers, collects computed results and returns the results to the user. There are two kinds of communication pathways: permanent pathways that are used to

maintain the topology of P2P overlay; temporary pathways that are established between Master and Workers for sending tasks and results that will be closed when the computation finishes. Peers in ParCop are organized according to an unstructured topology that makes idle peer collection for a computation slow and that leads to high resources consumption.

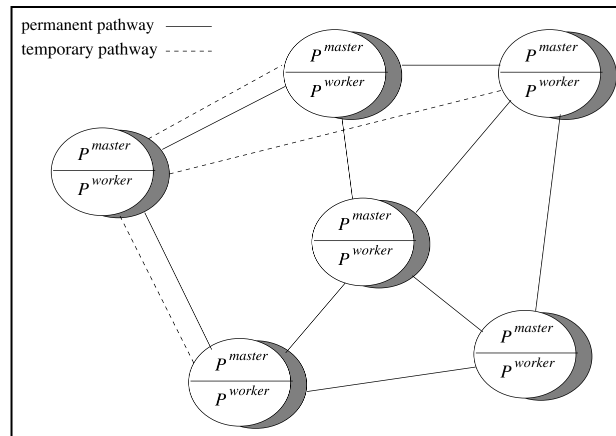


Figure 2.6: Interactions between peers in ParCop.

MapReduce [Dean 2004] is a programming model and an associated implementation for processing and generating large data sets on large clusters. It is extended in [Lee 2011] to be used in peer-to-peer network. In MapReduce programming model, users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pair and a *reduce* function that merges all intermediate values associated with the same intermediate key. This programming model is not appropriate to parallel iterative algorithms (see Section 2.4).

Vishwa [Reddy 2006] is a dynamically reconfigurable middleware that provides a reliable peer-to-peer environment for grid applications. Vishwa supports not only *bag-of-tasks* application class but also *connected problem* application class that involves inter-tasks communication. Vishwa is based on a two-layer architecture that includes a task management layer and reconfiguration layer. Task management layer organizes peers into zones based on the proximity in order to facilitate inter-task communication. Each peer can have neighbors in its zone and other zones that construct an unstructured topology. The reconfiguration layer handles nodes/network failures. Inter-tasks communication is built on the Distributed Pipes (DP) abstraction. However, Vishwa considers only connected problems solved by synchronous iterative schemes, asynchronous iterative (see Section 2.4) schemes are not taken in account.

P2P-MPI [Genaud 2009] is a framework aimed at the development of message-passing programs in large scale distributed networks of computers. P2P-MPI is developed in Java and makes use of Java TCP socket to implement the MPJ (Message Passing for Java) communication library. P2P-MPI uses a single super-node to

manage peer registration and discovery; this node may become a bottleneck. P2P-MPI implements a fault tolerance approach using peer replication that may not be efficient and appropriate to P2P context and connected problems since the number of peers involved in the computation will multiply; furthermore, the coordination protocol insuring coherence between replicas has great overhead.

In summary, existing middewares and environments for grid computing and volunteer computing can not be used easily for peer-to-peer high performance computing. Most of existing environments for peer-to-peer high performance computing are devoted only to bag-of-tasks applications where the applications are decomposed into independent tasks with no synchronization nor dependencies between tasks. Few systems consider connected problem application class where there are frequent communications between tasks like applications solved by parallel iterative algorithms; however, asynchronous iterative algorithms are not taken in account. We recall that we aim at designing a decentralized and fault-tolerant environment for peer-to-peer high performance computing that allow direct and frequent communications between peers. We are interested in applications in the domains of numerical simulation and optimization that can be solved via parallel or distributed iterative method. In particular, we think that the combination of asynchronous iterative algorithms with our environment on peer-to-peer networks are well suited to HPC applications. We note that the implementation of connected problem is more difficult than the bag-of-tasks applications.

2.4 High Performance Computing, parallel iterative methods

2.4.1 High Performance Computing

In this study, we concentrate on High Performance Computing (HPC) applications relevant to the domains of numerical simulation and optimization. These applications lead to complex or large scale problems that can often be solved efficiently via parallel or distributed iterative methods. Thus, we are mainly interested in task parallel models. In the sequel, we shall present some of these problems, e.g. nonlinear optimization problem, the so-called nonlinear network flow problems (see subsection 3.7.1) and a numerical simulation problem: the obstacle problem (see subsection 4.5.1).

2.4.2 Parallel iterative methods

Iterative methods play an important part in optimization and numerical simulation [Luenberger 1973, Bertsekas 1998, Ortega 1970]. The need for intensive computation has emphasized the interest for parallel and distributed iterative algorithms for solving systems of equations or fixed-point problems (see [Bertsekas 1989]). In this thesis, we concentrate on the fixed-point formulation.

2.4.2.1 Fixed-point problems and successive approximation methods

Consider the following fixed-point problem:

$$x^* = F(x^*) \tag{2.1}$$

where x^* is a solution vector of R^n and F is a given mapping from R^n to R^n.

The problem (2.1) can be solved by means of the following successive approximation method starting from x^0:

$$x^{j+1} = F(x^j), j = 0, 1, 2, \dots \tag{2.2}$$

The convergence of this type of algorithm has been studied in particular in [Ortega 1970].

Parallel iterative methods aim at solving problem (2.1) by means of iterative schemes carried out on several processors. The iterate vector x can be decomposed into p components $x_1, x_2, \dots, x_p$ where p is a given natural number related to the number of available machines. Similarly, the fixed-point mapping is decomposed into p components $F_1, F_2, \dots, F_p$. Let x_i^j denote the ith component of x^j and let F_i denote the ith component of the fixed-point mapping F. Then, the mathematical formulation of a simple example of parallel synchronous iterative algorithm can be written as follows:

$$x_i^{j+1} = F_i(x_1^j, x_2^j, \dots, x_p^j), i = 1, 2, \dots, p. \tag{2.3}$$

If p components of x are assigned to p processors, then each processor can update a different component of x according to (2.3) in parallel. The particular model (2.3) corresponds to *Jacobi-type* iterative scheme. The ith processor denoted by P_i has to receive the value of all components of x^j on which P_i depends from others processors in order to start next iteration $j + 1$. Moreover it has to send the value x_i^j to processors that depend on x_i^j. Thus, in order to implement a Jacobi parallel iterative scheme, it is necessary to update components of iterate vector in a certain order with some synchronizations. Figure 2.7 illustrates an example of synchronous scheme of computation whereby two processors cooperate to solve a problem. In the Figure 2.7, numbered rectangles correspond to updating phases, hatched rectangles correspond to communication and waiting phases, arrows delimit the beginning and the end of communications and the number in boxes correspond to the number of times the component has been updated. It is noted that the need to respect a strict order of computation (steering) and to synchronize processors can cause significant loss of time and lead to inefficiency.

Asynchronous parallel iterative algorithms have been proposed in order to generalize parallel iterative algorithms. In asynchronous iterative algorithms, components of iterate vector are updated in arbitrary order and without any synchronization. As a consequence, processors implementing parallel asynchronous iterative algorithms can go at their own pace according to their characteristics and computational load [El Baz 1998]. Restrictions imposed to asynchronous iterative algorithms are very

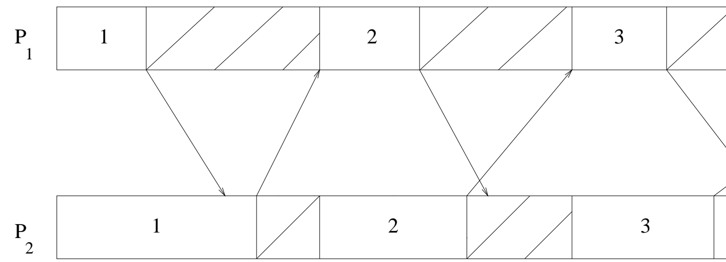


Figure 2.7: Synchronous parallel iteration.

weak: no component of the iterate vector must be abandoned forever and old values of components of the iterate vector must be discarded as the computation progresses. The Figure 2.8 displays an example of progress of an asynchronous iterative algorithm. The number in boxes corresponds here to iteration number and is increased at the start of each new update phase. It is noted that there is no idle time and updating phases are chained more rapidly.

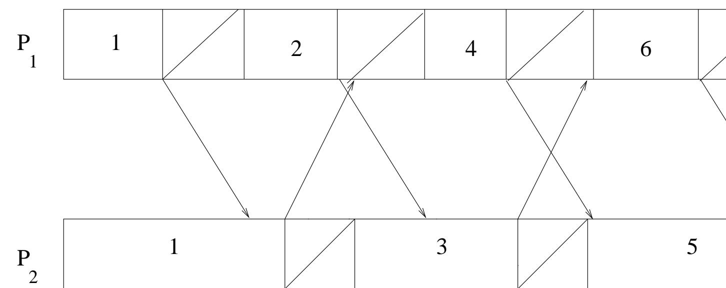


Figure 2.8: Asynchronous parallel iterations.

The concept of asynchronous iterative schemes has many advantages as compared with the one of synchronous iterative schemes. First, the lack of idle time due to synchronization as well as the lack of synchronization permits asynchronous iterative schemes to be more efficient, particularly when the loads are unbalanced or the system is heterogeneous which is a characteristic of peer-to-peer systems. Secondly, asynchronous iterative schemes scale better than synchronous iterative schemes since the synchronization overhead increases when the number of processors increases. Finally, asynchronous iterative algorithms tolerate temporary failures and message loss. Thus, asynchronous iterations seem better suited to high performance computing on peer-to-peer networks than synchronous iterative schemes.

However, programmers using asynchronous iterative algorithms have to face some challenges. The study of the convergence of parallel asynchronous iterations is generally more complicated than the one of synchronous iterations, particularly in the non-linear case. Moreover, non synchronization raises difficulties in terms of convergence detection and termination of algorithms. We give some details on these

topics in the sequel.

2.4.2.2 A general model of asynchronous iterations

In this subsection, we present briefly classical parallel asynchronous iterative schemes. The reader is referred to [El Baz 1996b, Miellou 1998, El Baz 2005, Chau 2007] for new extensions of the class of asynchronous iterative algorithms.

We consider the fixed-point problem (2.1).

Definition 2.1. Let N be the set of natural numbers, $n, \alpha \in N, \alpha \leq n$ the decomposition of R^n into $\prod_{i=1}^{\alpha} R^{n_i}, \sum_{i=1}^{\alpha} n_i = n$. An asynchronous iteration associated to the mapping F from $\prod_{i=1}^{\alpha} R^{n_i}$ to $\prod_{i=1}^{\alpha} R^{n_i}$ and initial point $x^0 \in \prod_{i=1}^{\alpha} R^{n_i}$ is a sequence $x^j, j = 0, 1, \ldots$ of vectors of $\prod_{i=1}^{\alpha} R^{n_i}$ defined recursively as follows for $i = 1, \ldots, \alpha$:

$$\begin{cases} x_i^j = F_i(x_1^{\rho_1(j)}, \ldots, x_\alpha^{\rho_\alpha(j)}) \ if \ i \in s(j), \\ x_i^j = x_i^{j-1} \ if \ i \notin s(j), \end{cases} \tag{2.4}$$

where $x_i \in R^{n_i}$ represents the ith sub-vector of vector x and F_i represents the ith block-component of mapping F, $S = \{s(j) | j = 1, 2, \ldots\}$ is a sequence of non-empty subsets of $1, \ldots, \alpha$ and $\rho = \{\rho(j) = (\rho_1(j), \ldots, \rho_\alpha(j)) | j = 1, 2, \ldots\}$ is a sequence of elements of N^α. Moreover, S and ρ satisfy following conditions for $i = 1, \ldots, \alpha$:

- $0 \leq \rho_i(j) \leq j - 1, j = 1, 2, \ldots$
- $\rho_i(j)$ tends to infinity when j tends to infinity.
- i appears an infinite number of times in the set S.

The above conditions can be interpreted respectively as follows:

- The value of the components of the iterate vector used during the computations at iteration j comes at most from iteration $j - 1$.
- Old values of the components of the iterate vector must be eliminated definitely as the computation progresses.
- No sub-vector of the iterate vector ceases to be updated during computations.

An asynchronous iteration associated with fixed-point mapping F, initial point x^0 and sequences s and ρ is denoted (F, x^0, S, ρ).

An asynchronous iterative algorithm (F, x^0, S, ρ) can be interpreted as follows. Let $\{P_1, \ldots, P_\alpha\}$ be a set of α processors. Let $\{t(j), j = 1, 2, \ldots\}$ be an increasing sequence of times. At the time $t(j)$, processors $P_i, i \in s(j)$ that are inactive are assigned to an evaluation of x^j that is different from x^{j-1} only by values of sub-vector x_i (see Figure 2.8). A processor P_i starts to update sub-vectors x_l using values $x_l^{\rho_l(j)}, l = 1, \ldots, \alpha$ of sub-vectors x_l that are available at the start of computations

and that come from previous iterations; a natural strategy is to take the most recent value of components. At an ulterior instant denoted by $t(j+k)$, with $k \in N$ and $k > 0$, the processor P_i will terminate the computations and will be assigned to the evaluation of x_i^{j+k}.

2.4.2.3 Convergence of asynchronous iterations

The study of the convergence of asynchronous iterations is a complex problem. However, a large number of results have been established in various contexts.

In the linear case, a necessary and sufficient condition of convergence for asynchronous iterative algorithms has been given in [Chazan 1969].

In the nonlinear case, sufficient condition under partial ordering have been established by [El Baz 1990] (see also [El Baz 1994]). These results can be applied to a large class of problems including systems issued from the discretization of partial differential equations and optimization problems. In particular, results proposed in [El Baz 1994] generalize a first result for asynchronous relaxation methods for the solution of convex network flow problems (see [Bertsekas 1987]).

Miellou and Spitéri have established results in the nonlinear case for H-accretive mappings (see [Miellou 1985b]). For nonlinear fixed point problems, convergence results have been established by Miellou and his team at Scientific Computing Laboratory (LCS) of Besançon. In particular, a sufficient condition of convergence has been given in [Miellou 1975] in the case of contractant operators (see also [Baudet 1978], [El Tarazi 1981] and [El Tarazi 1982]). Reference is also made to [Venet 2010] for recent convergence results concerning asynchronous sub-structuring methods.

The asynchronous convergence theorem of Bertsekas [Bertsekas 1983] (see also [Bertsekas 1989]) is an original and general result of convergence. It is also a powerful tool to prove the convergence of asynchronous iterative algorithms for various applications. The asynchronous convergence theorem of Bertsekas gives a set of sufficient conditions that ensure the convergence of asynchronous algorithms for fixed point problems. Unlike previous results which are based on the study of a sequence of vectors, this result is based on the study of a sequence of level sets. This approach has its origin in the theory of stability of Lyapunov; its advantage is to provide a more abstract framework for the analyzis of the convergence of asynchronous iterations. It encompasses also contractive and partial ordering aspects. The approach developed by Bertsekas is particularly interesting. However, this approach can not be applied in a direct manner. Obtaining a particular result of convergence for a given problem requires a detailed study of level sets [Bertsekas 1989].

Lubachevski et Mitra [Lubachevsky 1986] have also established a sufficient result of convergence for asynchronous bounded delay iterations applied to the solution of singular systems of Markovian type. Their asynchronous iterative algorithm model is close to partial asynchronous iterations of Bertsekas with bounded delay [Bertsekas 1989].

Reference is made to [Frommer 1997] and [Szyld 1998] for what concerns asynchronous multisplitting methods. The reader is also referred to [El Baz 1996b,

Miellou 1998, El Baz 1998] for new results related to a general class of asynchronous iterative algorithms with order intervals that generalize classical asynchronous iterations.

Finally, we note the analogy between iterative schemes and dynamic discrete systems, and more particularly between asynchronous iterative algorithms and discrete systems with delays which can vary over the time. In some cases, the convergence study of numerical schemes can learn from the study of the stability or from the asymptotic stability of corresponding dynamic discrete systems. Results based on the theorem of stability of Lyapunov have been presented in this scope in the following references: [Tsitsiklis 1987], [Kaszkurewicz 1990] and [Bhaya 1991].

2.4.2.4 Convergence detection and termination of asynchronous iterations

The convergence detection and termination of asynchronous iterative algorithms raises several problems related to applied mathematics since the termination of iterative algorithms must happen when the iterate vector is sufficiently close to a solution of the problem as well as problems related to computer science since a special procedure must be designed in order to detect convergence and to terminate the computation.

This problem has a strong connection with the termination of distributed processes, although the number of iterations of the algorithm can be infinite and computing processes can never be inactive.

Convergence detection and termination presents several difficulties particularly in the case of message passing architectures since processes have only local information, there is no global clock and the communication time may be arbitrarily long. As a consequence, there are few efficient termination methods for asynchronous iterative algorithms.

In a general context, the global state related to the termination of an asynchronous iterative algorithm can be inferred from a motley set of local informations of type $||x_i^j - x_i^{j-1}||_i \leq \varepsilon$, if we consider the difference between two successive values of the same sub-vector x_i of the iterate vector or related to residual. It appears that local informations can not be assembled in an given order if we want to establish formally that the termination has well happened.

In the sequel, we present briefly existing solutions for convergence detection and termination.

Empirical methods Termination methods for asynchronous iterations are usually designed according to an empirical manner. An usual method consists in observing with the help of a particular processor the local termination condition at each processor. The algorithm is arbitrarily terminated when all local conditions are satisfied. We can see easily that this type of method can give satisfying results only in the case where the asynchronism degree related to the value of delays in the mathematical model of asynchronous iterations is relatively small. When the delay

due to communication or due to unbalanced task is important, this method may cause an early termination.

Another method [Bertsekas 1989] consists in sending termination messages and restart messages by each processor and using a special processor that collects and centralizes these messages.

In another approach [Miellou 1989], the termination scheme samples periodically the state of processors and associates to each processor a Boolean value according to the satisfaction of the local termination criteria. This local value is then communicated to other processors. The global state is inferred in computing the fixed point of a Boolean operator via an asynchronous iterative algorithm. However, this approach needs for each processor to have an estimation of start time and end time of the asynchronous algorithm that finds the fixed point of the Boolean operator.

Another termination method [Chajakis 1991] uses termination messages and acknowledgments of termination messages. In this termination scheme, a processor terminates its computation if its local termination condition is satisfied and if it has received termination messages as well as acknowledgments of all termination messages from all processors.

There are no formal proof of validity for termination methods cited above in the general case. Furthermore, as we have mentioned above, for message passing architectures, all processors have only local information, there is no global clock and some messages may be delayed or arrive out of order.

Method of Bertsekas and Tsitsiklis One of the most interesting methods for detecting convergence and terminating asynchronous iterative algorithm has been proposed by Bertsekas and Tsitsiklis in [Bertsekas 1989] and [Bertsekas 1991]. Assumption is made that *each communicated data on a link is correctly received with a finite delay that is however non specified*. This method is based on the decomposition of the problem into two parts. First, the asynchronous iterative algorithm is modified so that it terminates in finite time and converges to a fixed point sufficiently close to the solution of the problem. Secondly, a procedure of convergence detection and termination is applied.

Bertsekas and Tsitsiklis have proposed to modify the asynchronous iterative algorithms as follows. If the update of a component of the iterate vector does not alter significantly its value, then the value of the iterate vector is not modified nor communicated to other processors. The termination of the modified asynchronous iterative algorithm happens when an update does not modify the value of components of iterate vector at all processors (i.e. all local termination conditions are satisfied) and no message is in transit in the communication network.

Several procedure can be used in order to detect the termination of the modified asynchronous iterative algorithm. We can quote for instance the procedure of Dijkstra and Scholten (see [Dijkstra 1980] and [Bertsekas 1989]) which is based on acknowledgements of all messages and the generation of an activity graph.

The method of Bertsekas and Tsitsiklis is one of rare methods in the literature

for which we have a formal proof of validity. However, this method presents some weaknesses. It requires first the use of a complex protocol as well as twice communications as the original asynchronous iterative algorithm. Moreover, conditions that are more restrictive than conditions of the asynchronous convergence theorem of Bertsekas must be satisfied in order to ensure the convergence of the modified asynchronous iterative algorithm.

In [Bertsekas 1991], Bertsekas and Tsitsiklis suggest to use another termination procedure, namely the snapshot algorithm of Chandy and Lamport [Chandy 1985]. This method is based on a procedure of marked messages and records of states of links and processors when all marked messages are delivered. We note that recorded states in a snapshot do not necessarily correspond to a true global state of the system at a given instant. However, the information contained in the snapshot is sufficient to detect certain properties of the global state of the system and in particular the termination.

In [El Baz 1998], El Baz has proposed a variant of the termination method of Bersekas and Tsitsiklis that reduces the number of exchanged messages by eliminating acknowledgments of messages.

Method of Savari and Bertsekas Another interesting termination method has been proposed by Savari and Bertsekas in [Savari 1996]. In this method, asynchronous iterations are slightly modified: the result of each new update of a component of the iterate vector is taken into account and communicated to other processors if it is different from the latest value of the component. In addition, queries are sent to all processors of the system whenever a termination condition is not satisfied. A processor performs computations and sends messages and queries to other processors as long as its local termination condition is not satisfied or it receives queries from other processors. The termination happens when all processors have satisfied their local termination condition and no message related to a query or to the result of an update is in transit in the system. The termination is detected using a standard protocol (see [Dijkstra 1980] and [Chandy 1985]).

Savari and Bertsekas have given a formal proof of validity of this termination algorithm. The principal advantage of this method is that it can be applied successfully to a larger class of iterative algorithms than the method of Bertsekas and Tsitsiklis. Its principal weakness is the necessity of a large number of communication of query type.

Method of level sets In [El Baz 1996a], El Baz has proposed an approach that relies on the use of the sequence of level set. The principle of this method consists in terminating asynchronous iterative algorithm when the iterate vector penetrates into level set $X(\hat{q})$ where $\hat{q}$ is a natural integer fixed a priori in function of the problem and no message is in transit in the network. The asynchronous iterative algorithm is slightly modified. A simple computing procedure related to level sets is added. Reference is made to [El Baz 1998] for more details about this method.

Other termination methods Savari and Bertsekas have proposed in [Savari 1996] several schemes of supervised termination.

Miellou has proposed in [Miellou 1975] and [Miellou 1990] a method based on the use of secondary algorithm or the error control which is derived from algorithm of F.Robert and G. Schroeder (see [Robert 1969] [Robert 1975]). This secondary algorithm consumes less computational resources than the initial (or principal) asynchronous algorithm. However, sequences S and I must be necessarily identical for both main and secondary algorithms.

2.5 Conclusion

The raise of the parallelism concept in microprocessor architectures together with progress in high bandwidth network has made possible high performance computing applications on peer-to-peer networks. This solution seems economic and attractive. Among the different problems that can be treated, HPC applications related to task parallel model that can be solved in particular via asynchronous iterative algorithms constitute an important field with possible relevance to many engineering specialties and services like mechanics, telecommunications and finance. In the sequel, we present our contributions to this domain.

Chapter 3

P2PSAP - A self-adaptive communication protocol

Contents

3.1 Introduction 27

3.2 State of the art in adaptive communication protocols 28

3.2.1 Micro-protocol approach 29

3.2.2 Cactus framework and CTP protocol 31

3.3 P2PSAP Protocol architecture 33

3.3.1 Socket API 33

3.3.2 Data channel 33

3.3.3 Control channel 34

3.4 Example of scenario 36

3.5 Some modifications to Cactus 37

3.6 Self-adaptive mechanisms 37

3.6.1 Choice of protocol features 38

3.6.2 New micro-protocols 39

3.6.3 (Re)Configuration 43

3.7 Computational experiments 47

3.7.1 Network flow problems 47

3.7.2 Platform 49

3.7.3 Computational results 50

3.8 Chapter summary 51

3.1 Introduction

In this chapter, we present the Peer-To-Peer Self Adaptive communication Protocol (P2PSAP), a self-adaptive communication protocol dedicated to Peer-to-Peer (P2P) High Performance Computing (HPC) [El Baz 2010]. As explained in Chapter 1, the design of this protocol is the first step of a classical approach used to design distributed computing environments. The P2PSAP protocol is designed to allow rapid update exchanges between peers in the case of the solution of numerical simulation

problems and optimization problems via distributed iterative algorithms. The protocol can configure itself automatically and dynamically in function of application requirements like scheme of computation and elements of context like topology by choosing the most appropriate communication mode between peers. The protocol is an extension of CTP [Wong 2001] and makes use of the Cactus framework [Hiltunen 2000]. We note that our contribution differs from existing communication libraries for high performance computing like MPICH/Madeleine [Aumage 2001] in allowing the modification of internal transport protocol mechanism in addition to switching between networks. A first series of computational experiments for an optimization problem illustrate the behavior of the proposed protocol for HPC applications.

This chapter is organized as follows. Next section presents existing work in adaptive communication protocols. Section 3.3 describes the architecture of P2PSAP protocol. An example of scenario that shows the automatic and dynamic configuration capability of P2PSAP is displayed in section 3.4. Section 3.5 describes some modifications we have made to the Cactus framework in order to improve protocol performance and to facilitate the reconfiguration. In the section 3.6, we detail the choice of self-adaptive mechanisms for distributed peer-to-peer HPC applications. A first series of computational experiments for an optimization problem, i.e. a network flow problem is displayed and analyzed in the section 3.7. Finally, a summary of P2PSAP protocol concludes this chapter.

3.2 State of the art in adaptive communication protocols

Early communication protocols such as TCP [TCP 1981] and UDP [UDP 1980] has been designed to fulfill simple requirements regarding the reliability and order of data. Nowadays, new applications over the Internet like VoIP, VoD and P2P HPC require communication protocols to adapt to context as well as to application profile. In the literature, several solutions have been proposed. One can classify them into two classes: behavioral and structural adaptation [Van Wambeke 2008]:

- Behavioral adaptation relies on the capacity of the algorithm to change the behavior of the protocol without modifying its structure. One can find this adaptation property in standard TCP protocol in the case of network congestion. Behavioral adaptation is easy to implement but limits the adaptability.

- Structural adaptation can change the internal structure of the protocol, thus changing the provided services. Structural adaptation is based on modular programming where software is composed of separate, interchangeable components. The implementation of this adaptation is complicated but it allows the flexible adaptability. Structural adaptation is known as micro-protocol approach.

In the next subsection, we shall present in detail the micro-protocol approach.

3.2.1 Micro-protocol approach

Micro-protocols are an interesting approach to design and implement self-adaptive communication protocols. Micro-protocols were first introduced in x-kernel [Hutchison 1991]. They have been widely used since in several systems. A micro-protocol is a primitive building block that implements merely a functionality of a given protocol such as error recovery, ordered delivery and so on. A protocol then results from the composition of a given set of micro-protocols. This approach permits one to reuse the code, facilitate the design of new protocols and give the possibility to configure the protocol dynamically.

Protocol composition frameworks provide the infrastructure that allows programmers to build communication protocols according to micro-protocol approach. In the literature, several protocol composition frameworks have been proposed. Based on the composition model, we can divide these frameworks into three models: the hierarchical, non-hierarchical and hybrid models.

3.2.1.1 Hierarchical model

In the hierarchical model, a stack of micro-protocols composes a given protocol, similarly to the ISO model. This model can be found in the x-kernel [Hutchison 1991] and APPIA [Miranda 2001, Mocito 2005] frameworks.

X-kernel. The x-kernel [Hutchison 1991] is an operating system kernel that provides architecture for constructing and composing network protocols. In the x-kernel framework, a protocol is considered as an abstraction object with an uniform interface that allows protocols to invoke operations on one another (i.e., to send a message to and receive a message from an adjacent protocol). The suite of protocols in x-kernel is statically configured at initialization time onto a protocol graph (see Figure 3.1). Based on the protocol graph, users can plug protocols together in difference ways.

APPIA. Appia [Miranda 2001, Mocito 2005] is a protocol kernel that supports applications requiring multiple coordinated channels and offers a clean and elegant way for the application to express inter-channel constraints. In Appia, micro-protocols are defined as layers that exchange informations using events. A session is an instance of a micro-protocol; it maintains state variables used to process events. A Quality of Service (QoS) is defined as a stack of layers. The QoS specifies which protocols must act on the messages and the order they must be traversed thus defining a quality of service by enumerating the properties it will provide. A channel is an instantiation of a QoS and is characterized by a stack of sessions of the corresponding layers. Inter-channel coordination can be achieved by letting different channels share one or more common sessions.

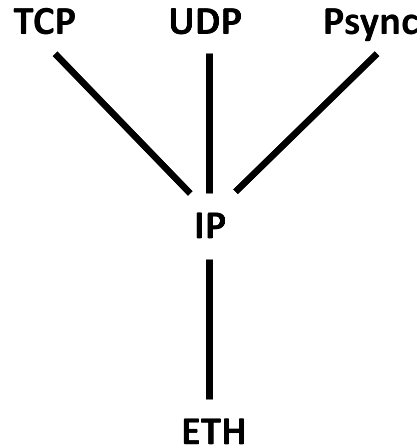


Figure 3.1: Example of x-kernel protocol graph configuration

3.2.1.2 Non-hierarchical model

In the non-hierarchical model, there is no particular order between micro-protocols; the SAMOA [Paweł 2004] framework corresponds to this model.

SAMOA. SAMOA [Paweł 2004] is a protocol framework that ensures the isolation property. It has been designed to allow concurrent protocols to be expressed without explicit low-level synchronization, thus making programming easier and less error-prone. In SAMOA, a micro-protocol is composed of a set of event handlers and a local state. A local state of a given micro-protocol can be modified only by event handlers of this micro-protocol. Each event handler has to be bound to a predefined event type. When an event of a given event type is triggered, all event handlers that have been bound to this event type are executed. There are two kinds of events: internal and external. An internal event is generated during a handler's execution. External event are requests from the network layer (or application) to inject messages to the protocol.

3.2.1.3 Hybrid model

The hybrid model is a combination of the two previous models; micro-protocols are composed here hierarchically and non-hierarchically. One can find this last model in the FPTP [Exposito 2003] and Cactus [Hiltunen 2000] frameworks.

FPTP. FPTP (Full Programmable Transport Protocol) is a connection oriented and message oriented transport protocol that has been designed to be statically or dynamically configured according QoS requirements [Exposito 2003]. FPTP is constructed by the composition of configurable mechanisms suited to control and manage the QoS. FPTP architecture follows a hierarchical model for the composition of services related to QoS control mechanisms (i.e. rate control, flow control,

time control, loss detection) and a non-hierarchical model for the QoS management mechanisms (i.e. congestion control, error recovery, inter-flow synchronization) (see Figure 3.2). FPTP has been implemented in Java for multimedia applications with different QoS requirements in terms of time constraints.

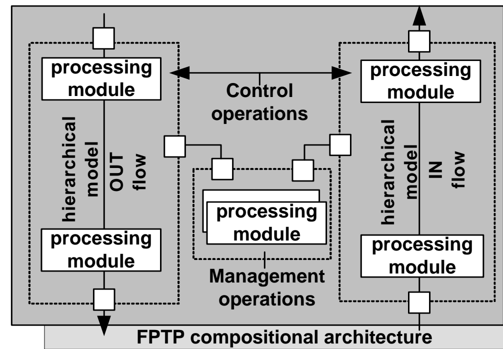


Figure 3.2: FPTP compositional architecture

Cactus. The Cactus frameworks [Hiltunen 2000] extends x-kernel in providing a finer granularity of composition. In addition to layered composition in x-kernel, intra-layer composition following non-hierarchical model is allowed.

We have concentrated on the Cactus framework since this approach is flexible and efficient. In the next subsection, we shall detail the Cactus framework and one example, the CTP protocol.

3.2.2 Cactus framework and CTP protocol

Cactus [Hiltunen 2000] is a system for constructing highly-configurable protocols for networked and distributed system. Cactus has two grain levels of composition. Individual protocols, termed composite protocols, are constructed from micro-protocols. Composite protocols are then layered on top of each other to create a protocol stack using an interface similar to the standard x-kernel API [Hutchison 1991].

Cactus is an event-based framework. Events are used to signify state changes, such as arrival of messages from the network. Each micro-protocol is structured as a collection of event handlers, which are procedure-like segments of code and are bound to events. When an event occurs, all handlers bound to that event are executed. Events can be raised in different ways, explicitly by micro-protocols or implicitly by the runtime system, with either blocking or non-blocking semantics, with a specific delay and a priority execution number. Arguments can be passed to handlers in two ways, statically when a handler is bound to an event and dynamically when an event is raised. The runtime system also provides operations for unbinding handlers, creating and deleting events, halting event execution, and canceling a delayed event.

Handler execution is atomic with respect to concurrency, i.e. a handler is executed till completion before another handler is started unless it voluntarily yields the CPU.

The Cactus framework provides a message abstraction named dynamic messages, which is a generalization of traditional message headers. A dynamic message consists of a message body and an arbitrary set of named message attributes. Micro-protocols can add, read, and delete message attributes. When a message is passed to a lower-level protocol, a pack routine combines message attributes with the message body; while an analogous unpack routine extracts message attributes when a message is passed to a higher-level protocol. Cactus also supports shared data that can be accessed by all micro-protocols configured in a composite protocol.

The CTP Configurable Transport Protocol [Wong 2001] is designed and implemented using the Cactus framework. The Figure 3.3 shows the CTP implementation with events on the right side and micro-protocols on the left side. An arrow from a micro-protocol to a given event indicates that the micro-protocol binds a handler to this event.

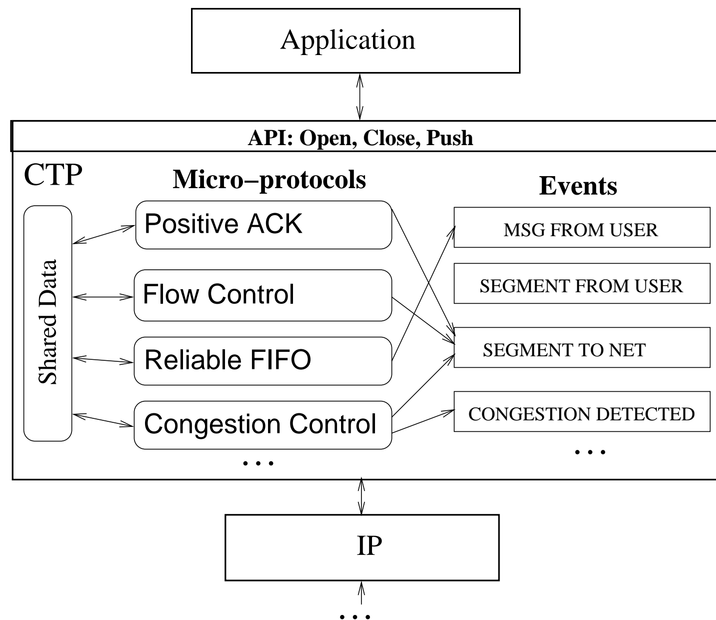


Figure 3.3: CTP - Configurable Transport Protocol

The CTP protocol includes a wide range of micro-protocols including: a small set of basic micro-protocols like Transport Driver, Fixed Size or Resize and Checksum that are needed in every configuration and a set of micro-protocols implementing various transport properties like acknowledgments, i.e. PositiveAck, NegativeAck and DuplicateAck, retransmissions, i.e. Retransmit, forward error correction, i.e. ForwardErrorCorrection, and congestion control, i.e. WindowedCongestionControl and TCPCongestionAvoidance.

We have extended the CTP protocol in order to build the self-adaptive commu-

nication protocol dedicated to peer-to-peer high performance computing that will be presented in the sequel.

3.3 P2PSAP Protocol architecture

Figure 3.4 shows the architecture of the P2PSAP protocol; this protocol has a Socket interface and two channels: a control channel and a data channel. We present now in detail those components.

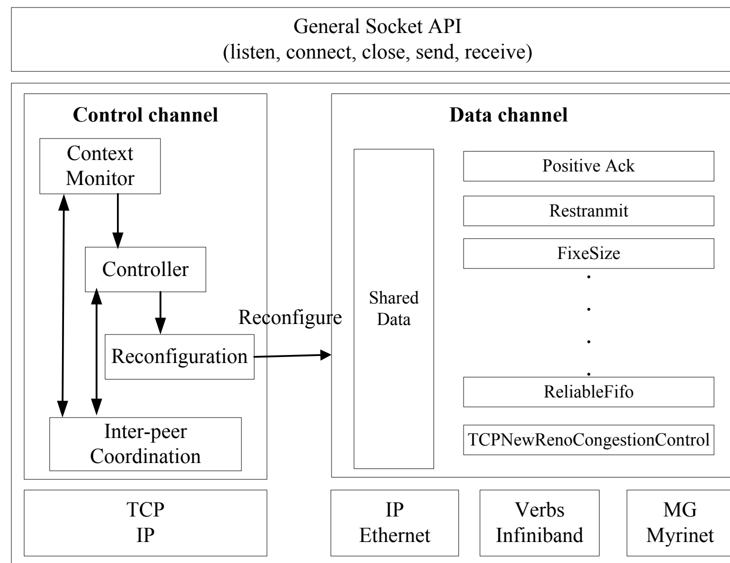


Figure 3.4: P2PSAP Protocol Architecture

3.3.1 Socket API

A main lack of Cactus CTP [Wong 2001] is that it has no Application Programming Interface (API). Application has to use an interface as though it was just another composite protocol. So, we have placed on the top of our protocol a socket-like API. Application can open and close connection, send and receive data. Furthermore, application can get session state and change session behavior or architecture through socket options, which are not available in Cactus. Session management commands like listen, open, close, setsockoption and getsockoption are directed to Control channel; while data exchanges commands, i.e. send and receive commands are directed to Data channel.

3.3.2 Data channel

The Cactus built data channel transfers data packets between peers. The data channel has two levels: the physical layer and the transport layer; each layer corresponds

to a Cactus composite protocol. We encompass the physical layer to support communications on different networks, i.e. Ethernet, InfiniBand and Myrinet. Each communication type is carried out via a composite protocol. The data channel can be triggered between the different types of networks; one composite protocol is then substituted to another. The transport layer is constituted by a composite protocol made of several micro-protocols, which is an extension of CTP. At this level, data channel reconfiguration is carried out by substituting or removing and adding micro-protocols. The behavior of the data channel is triggered by the control channel.

3.3.3 Control channel

The Control channel manages session opening and closure; it captures context information and (re)configures the data channel at opening or operation time; it adapts itself to these informations and their changes; it is also responsible for coordination between peers during reconfiguration process. Note that we use the TCP protocol to exchange control messages since these messages cannot be lost.

Before describing the main components of the control channel, we present first a session life cycle (see Figure 3.5). Suppose process A wants to exchange data with process B, it opens a session through socket create and connect command. Then, a TCP connection is opened between 2 processes. Process B accepting connection must send its context information to process A. Process A chooses the most appropriate configuration for data channel and send configuration command to process B based on its context information and those of B. After that, the two processes carry out the configuration of data channel. When the configuration is done, each process has to inform the other process and waits for the notification of other process. Data is exchanged only when both processes have finished data channel configuration. During the communication, a process can decide to change configuration of data channel due to context changes or user requirements, like process A in Figure 3.5. Then, a procedure similar as the one implemented for configuration at session opening will be realized. When session is closed, the data channel is closed first; the control channel with TCP connection is closed later on.

We describe now the main components of the control channel.

- **Context monitor:** the context monitor collects context data and their changes. Protocol adaptation is based on context acquisition, data aggregation and data interpretation. Context data can be requirements imposed by the user or the algorithm at the application level, i.e. asynchronous algorithms, synchronous algorithms or hybrid methods. Context data can also be related to peers location and machine loads. Context data are collected at specific times or by means of triggers. Data collected by the context monitor can be referenced by the controller.

- **Controller:** the controller is the most important component in the control channel; it manages session opening and closure through TCP connection

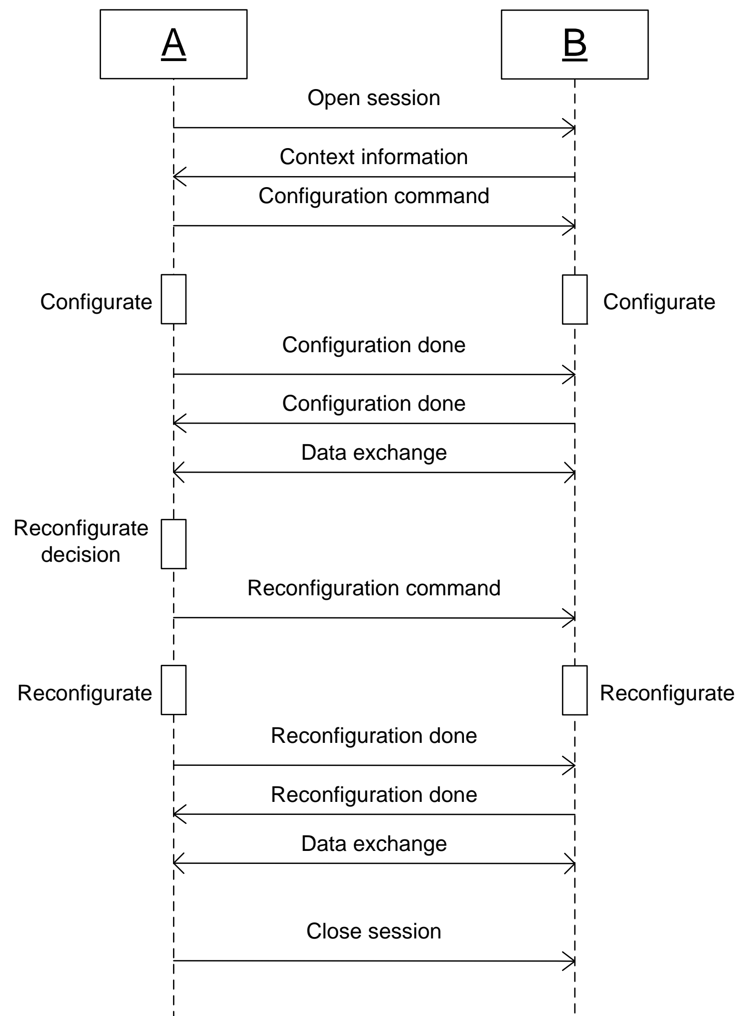


Figure 3.5: Protocol session life cycle

opening and closure; it also combines and analyzes context information provided by the context monitor so as to choose the configuration (at session opening) or to take reconfiguration decision (during session operation) for data channel. The choice of the most appropriate configuration is determined by a set of rules (this point will be detailed in the sequel). These rules specify new configuration and actions needed to achieve it. The (re)configuration command along with necessary information is sent to component Reconfiguration and to other communication end point.

- **Reconfiguration:** reconfiguration actions are made by the reconfiguration component via the dedicated Cactus functions. Reconfiguration is done at the physical layer by substituting a composite protocol supporting communication on a network board to a composite protocol supporting communication on another type of network board or at the transport layer by removing and adding or substituting micro-protocols.

- **Inter-peer coordination:** the coordination component is responsible of context information exchanges and coordination of peers reconfiguration processes so as to ensure proper working of the protocol.

3.4 Example of scenario

We present now a simple scenario for the P2PSAP protocol so as to illustrate its behavior.

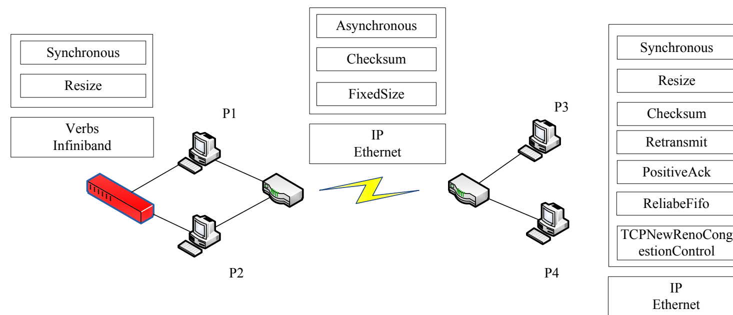


Figure 3.6: Example of P2PSAP reconfiguration scenario

We consider a high performance computing application, like for instance a large scale numerical simulation application or a complex optimization problem, solved on the network composed of two simple clusters shown in Figure 3.6. The first cluster is composed of two similar machines: P_1 and P_2 that can be connected via Ethernet or InfiniBand. The second cluster is made of two similar machines: P_3 and P_4 connected only via Ethernet. The communication protocol between machines P_3 and P_4 is based on synchronous communication (since machines have similar characteristics and loads) via micro-protocols ensuring e.g. reliability and order, i.e. ReliableFifo,

and congestion control, i.e. TCPNewRenoCongestionControl. The communication protocol between machines P_1 and P_2 is based on synchronous communication (for the same reasons) via Infiniband since Infiniband is faster than Ethernet. Moreover, InfiniBand insures reliability and message order; as a consequence, the data channel needs only micro-protocols ensuring synchronous communication (Synchronous) and segment size management (Resize). Communications between machines of the different clusters are asynchronous; as a consequence, in this case we need no order, nor reliability micro-protocols.

3.5 Some modifications to Cactus

In order to achieve the reconfiguration capability of P2PSAP presented in previous sections as well as to improve protocol performance, we have introduced some modifications to the Cactus framework:

- Firstly, Cactus does not allow concurrent handler execution; this means that a handler must wait for current executed handler completion before being executed. But nowadays, almost all PCs have more than one core and concurrent handler execution is necessary in order to improve performance. So, we have modified Cactus to allow concurrent handler execution. Each thread has its own resources and its handler execution is independent of others.

- Secondly, we have eliminated unnecessary message copies between layers. In the Cactus framework, when a message is passed to upper or lower layers, Cactus runtime creates a new message that is sent to upper or lower layers. Hence, a significant number of CPU cycles and memories are consumed in multiple-layers systems. In our protocol, message copies occur between Socket API layer and Data channel, and within the Data channel. In order to eliminate message copies, we have modified the pack and unpack functions so that only a pointer to message is passed between layers. Therefore, no message copy is made within the stack.

- Finally, Cactus provides operations for unbinding handlers but it has no explicit operation for removing a micro-protocol. In order to facilitate protocol reconfiguration, we have added to Cactus API an operation for micro-protocol removing. In addition to the micro-protocol initiating function, each micro-protocol must have a remove function, which unbinds all its handlers and releases its own resources. This function will be executed when the micro-protocol is removed.

3.6 Self-adaptive mechanisms

In this section, we shall present and explain our choices of P2PSAP's self-adaptive mechanisms for distributed peer-to-peer computing. We plan to support the com-

munication on several networks. So far, we have concentrated on Ethernet network that is widely used. Thus, the self-adaptive of the protocol is only at transport level.

Similar machines connected via a local network with small latency, high bandwidth and reliable data transfer can be gathered into a cluster. The reader is referred to [Beaumont 2011] for recent study dealing with grouping peers on the Internet into clusters based on latency metric. During solution, the transport protocol is configured according to the following context data: schemes of computation (i.e. synchronous, asynchronous or hybrid iterative schemes) and topology parameters like type of connection (i.e. intra or inter cluster). Firstly, we determine required protocol features in each considered context. A context corresponds to the combination of elements from network layer like topology and application layer like a given iterative scheme, e.g. synchronous or asynchronous.

3.6.1 Choice of protocol features

The choice of protocol features in each context is summarized in Table 3.1. In the sequel, we explain our choices.

	Synchronous		**Asynchronous**		**Hybrid**	
	Intra	**Inter**	**Intra**	**Inter**	**Intra**	**Inter**
Synchronous	Yes	Yes	No	No	Yes	No
Reliable transport	Yes	Yes	No	No	Yes	No
Ordered delivery	Yes	Yes	No	No	Yes	No
Congestion control	No	Yes	No	Yes	No	No

Table 3.1: Choice of P2PSAP protocol features according to algorithmic and communication context

Sometimes, communication mode must fit a computational scheme requirement (e.g. a special requirement related to the convergence of the implemented numerical method) as in the case where synchronous computational schemes are imposed. Then, synchronous communications are imposed in both intra-cluster and inter-cluster data exchanges. In this case, reliable transport and ordered delivery are required in order to ensure that the application is not going to be blocked by a message loss or unordered message delivery. Moreover, in synchronous communication, after sending a message, the sender is blocked until it receives an acknowledgement about the delivery of this message to application at receiver. Thus the receiver buffer can not be overwhelmed and flow control is not necessary in both intra and inter-cluster communication. In intra-cluster with low latency, high bandwidth and reliable links, congestion control is not really necessary. Whereas, congestion control is required in inter-cluster with high latency, low bandwidth and unreliable link in order to behave fairly with others flows and to reduce retransmission overhead. In this case, we have chosen TCP New-Reno congestion avoidance algorithm [Floyd 1999] which is the most commonly implemented RFC-based one.

Likely, in the case where asynchronous iterative schemes of computation are required by user, asynchronous communication must be preferably implemented in both intra-cluster and inter-cluster data exchanges. We note that asynchronous schemes of computation are fault tolerant in some sense since they allow messages losses. For this reason, reliable transport and ordered delivery as well as flow control are not needed in both intra-cluster and inter-cluster communication. While congestion control is not necessary in intra-cluster communication, it is required in inter-cluster communication in order to ensure a fair behavior with others flows. In our opinion, DCCP congestion mechanisms [Kohler 1999] are the most appropriate one for unreliable datagram flow.

There are also some situations where a given problem can be solved by using any combination of computational schemes. In this latter case, users can leave the system to freely choose communication mode. As a consequence, the most appropriate communication mode according to topology parameters ,i.e. inter-cluster or extra cluster connection should be chosen. This corresponds to the so-called *Hybrid* scheme of computation. In this case, if computational loads are well balanced on machines inside a cluster that are identical, then synchronous communication between peers are appropriate. The communication protocol in this context has the same features as in the case of synchronous iterative scheme and intra-cluster communication. On the other hand, synchronization may be an obstacle to efficiency and robustness in inter-cluster data exchanges situations where there may be some heterogeneities in terms of processors, OS, bandwidth, and communications may be unreliable and have high latency. Thus, asynchronous communication seems more appropriate in this latter context. The communication protocol in this latter context has the same features as in the case of asynchronous scheme and inter-cluster communication.

According to the choices of protocol features for each context, there are some functionalities that are needed to achieve those features but are not implemented by any micro-protocol in CTP. Thus, we have designed and developed some new micro-protocols as we shall present in the next subsection.

3.6.2 New micro-protocols

3.6.2.1 Micro-protocols synchronization

CTP supports only asynchronous communication. Distributed applications may nevertheless use plural communication modes. Hence, we have implemented two micro-protocols corresponding to two communication modes: synchronous and asynchronous. These micro-protocols introduce new events, *UserSend* and *UserReceive*, that will be raised when send and receive socket commands are called by an application.

The *synchronous* micro-protocol implements blocked synchronous communication mode as presented in the Figure 3.7. Synchronous micro-protocol consists of 3 handlers for 3 events: *UserSend*, *SegmentToNet* and *UserReceive*. Figure 3.8

displays the pseudo-code of synchronous micro-protocol.

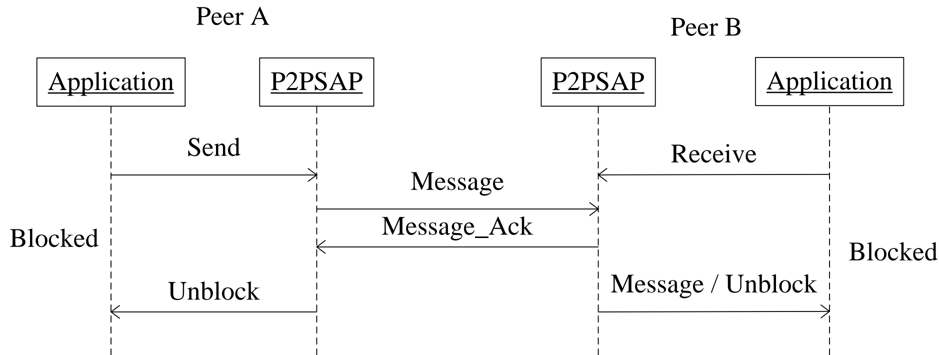


Figure 3.7: Synchronous communication mode.

```
 1: procedure SynchronousUserSend
 2:     Push message into sender buffer
 3:     Wait_1: Wait for acknowledgment
 4: end procedure

 5: procedure SynchronousUserReceive
 6:     if receiver buffer is empty then
 7:         Wait_2: Wait for message
 8:     end if
 9:     Pop message from receiver buffer
10: end procedure

11: procedure SynchronousSegmentToUser
12:     if acknowledgment then
13:         Unblock Wait_1
14:     end if
15: end procedure
```

Figure 3.8: Synchronous micro-protocol.

The asynchronous mode implemented by the asynchronous micro-protocol is presented in the Figure 3.9. Asynchronous micro-protocol consists of 2 handlers for 2 events: *UserSend* and *UserReceive*. Figure 3.10 displays the pseudo-code of asynchronous micro-protocol.

3.6.2.2 Micro-protocol TCP New-Reno congestion avoidance

CTP has micro-protocols implementing SCP and TCP-Tahoe congestion avoidance algorithm. However, to the best of our knowledge, TCP New-Reno [Floyd 1999] is the most commonly implemented RFC-based congestion avoidance algorithm.

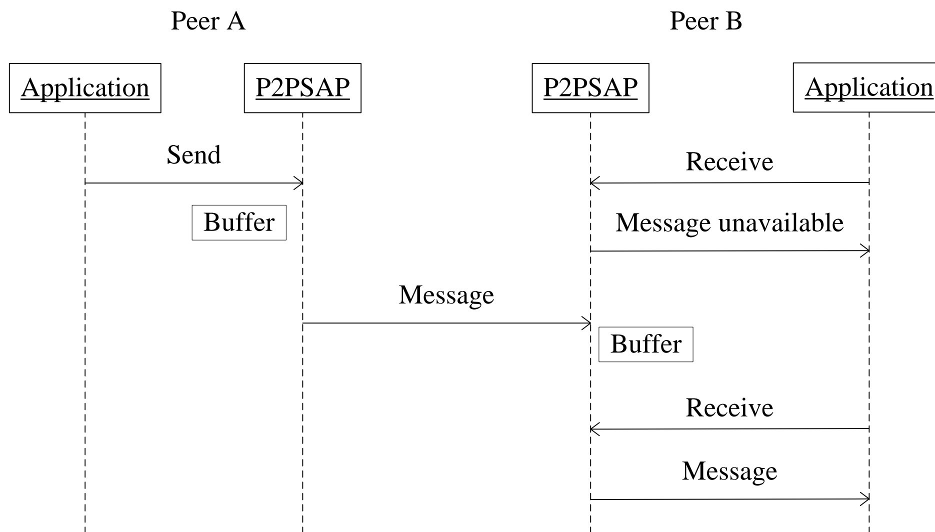


Figure 3.9: Asynchronous communication mode

```
1: procedure SynchronousUserSend
2:     Push message into sender buffer
3: end procedure

4: procedure SynchronousUserReceive
5:     if receiver buffer is not empty then
6:         Pop message from receiver buffer
7:     end if
8: end procedure
```

Figure 3.10: Asynchronous micro-protocol.

Thus, we have developed a new micro-protocol implementing the TCP New-Reno congestion avoidance algorithm. This micro-protocol must be used in combination with *PositiveAck*, *Restransmit*, *DuplicateAck* and *WindowCongestionControl* micro-protocols that are already available in CTP. It consists of 4 handlers of events: *SegmentLost*, *SegmentTimeout*, *AddDupAck* and *SegmentAcked*. *SegmentLost* event is raised by *DuplicateAck* micro-protocol when a third duplicate ACK is received. *AddDupAck* event is raised by *DuplicateAck* micro-protocol when a additional duplicate ACK is received. Figure 3.11 displays the pseudo-code of TCP New-Reno congestion avoidance micro-protocol.

```
 1: procedure TCPNewRenoCongestionAvoidanceSegmentLost
 2:     ssthresh ← min(FlightSize/2, 2)
 3:     crwd ← crwd + 3
 4:     fast_recovery ← TRUE
 5: end procedure

 6: procedure TCPNewRenoCongestionAvoidanceSegmentTimeout
 7:     ssthresh ← min(FlightSize/2, 2)
 8:     crwd ← crwd + 1
 9:     fast_recovery ← FALSE
10: end procedure

11: procedure TCPNewRenoCongestionAvoidanceAddDupAck
12:     crwd ← crwd + 1
13: end procedure

14: procedure TCPNewRenoCongestionAvoidanceSegmentAcked
15:     if fast_recovery = FALSE then
16:         if cwnd < ssthresh then
17:             crwd ← crwd + 1
18:         else
19:             cwnd ← 1/cwnd
20:         end if
21:     else
22:         if full acknowledgement then
23:             ssthresh ← min(FlightSize/2, 2)
24:             fast_recovery ← FALSE
25:         else                                        ▷ Partial acknowledgement
26:             cwnd ← cwnd − n_acked
27:         end if
28:     end if
29: end procedure
```

Figure 3.11: Micro-protocol TCP New-Reno congestion avoidance

3.6.2.3 Micro-protocols DCCP congestion control

The Datagram Congestion Control Protocol (DCCP) [Kohler 1999] is an unreliable datagram transport protocol that provides congestion control mechanisms in order to behave fairly with others TCP flows. DCCP has plural variants identified by a Congestion Control Identifier (CCID). In CCID 2 [Floyd a], a window-based congestion control is implemented that is similar to TCP Congestion Control. CCID 3 [Floyd c] implements a rate-based congestion control that is based on TCP-Friendly Rate Control (TFRC). CCID 4 [Floyd b] propose TFRC-SP, a Small-Packet (SP) variant of TFRC, that is designed for applications that send small packets.

As remarked in subsection 3.6.1, in the context of asynchronous iterative scheme and inter-cluster connexion, the transport protocol is unreliable but needs a congestion control mechanism in order to ensure a fair behavior with others flows. Thus, we have developed micro-protocols implementing the congestion control mechanism of DCCP, i.e. CCID 2. Since the adjustment of the congestion window in DCCP is the same as the one in TCP, we can reuse TCP Congestion Avoidance micro-protocol that is already available in CTP. Thus, we have developed only two news micro-protocols. Micro-protocol *DCCP Ack* implements acknowledgments of DCCP. Micro-protocol *DCCP Window Congestion Control* adjusts the *pipe* value (i.e. number of packets outstanding in the network) and sends queued packet if the *pipe* value is less than the congestion window (cwnd). Figure 3.12 and Figure 3.13 display the pseudo-code of DCCPACK and DCCP Window Congestion Control micro-protocols.

3.6.3 (Re)Configuration

Based on the choices of protocol features (see subsection 3.6.1) and with new developed micro-protocols (see subsection 3.6.2), we can determine the protocol composition, i.e. the set of micro-protocols for each considered context as in the Table 3.2.

At session opening, based on the context data, the Control Channel configures the composition of Data Channel as in the Table 3.2, i.e. it adds chosen micro-protocols to Data Channel. For example, in the case of asynchronous iterative scheme of computation, only minimum set of micro-protocols including Transport-Driver and Resized are added to Data Channel for intra-cluster communication; while for inter-cluster communication, in additional to minimum set, DCCPAck, DCCPWindowedCongestionControl and TCPCongestionAvoidance, that provide the congestion control mechanism of DDCP, are added to Data Channel.

During execution, context data can be changed. Then the Control Channel can determine new composition for Data Channel according to the Table 3.2. Comparing new composition with the old one, the Control Channel can determine operation needed to be carried out in order to reconfigure the Data Channel from the old composition to obtain the new one. For example, in an evolution application of numerical simulation, the computation scheme can be changed during execution, e.g. from asynchronous iterative scheme to synchronous iterative scheme. In this case,

```
 1: procedure DCCPAckSegmentToNet
 2:     if need Ack then
 3:         Insert Ack option
 4:     end if
 5:     if need Ack-of-Ack then
 6:         Insert Ack-of-Ack option
 7:     end if
 8: end procedure

 9: procedure DCCPAckSegmentFromNet
10:     if segment has Ack option then
11:         Check remote Ack vector
12:         if segments dropped or ECN-marked then
13:             Raise SegmentLost event
14:         end if
15:         if segments acked then
16:             Raise SegmentAcked event
17:         end if
18:     end if
19:     if segment has Ack-of-Ack option then
20:         Update local Ack vector
21:     end if
22:     if segment is new data then
23:         Schedule AckTimeout event                    ▷ Delayed raise
24:     end if
25: end procedure

26: procedure DCCPAckAckTimeout
27:     Create a segment
28:     Raise SegmentToNet event
29: end procedure
```

Figure 3.12: Micro-protocol DCCPAck

```
1: procedure DCCPWindowedCongestionControlSegmentToNet
2:     while pipe > cwnd do
3:         Wait
4:     end while
5: end procedure

6: procedure DCCPWindowedCongestionControlSegmentAcked
7:     pipe ← pipe − n_segments_acked
8:     Unblock Wait
9: end procedure

10: procedure DCCPWindowedCongestionControlSegmentLost
11:     pipe ← pipe − n_segments_lost
12:     Unblock Wait
13: end procedure

14: procedure DCCPWindowedCongestionControlSegmentTimeout
15:     pipe ← 0
16:     Unblock Wait
17: end procedure
```

Figure 3.13: Micro-protocol DCCP Window Congestion Control

for intra-cluster communication, Asynchronous micro-protocol will be removed and Synchronous, SequencedSegment, PositiveAck, Retransmit, RTTEstimation micro-protocols will be added; for inter-cluster communication, Asynchronous, DDCPAck, DCCPWindowedCongestionControl, TCPCongestionAvoidance micro-protocols will be removed and Synchronous, SequencedSegment, PositiveAck, Retransmit, RTTEstimation, DuplicateAck, WindowedCongestionControl, TCPNewRenoCongestion-Avoidance micro-protocols will be added.

	Synchronous		Asynchronous		Hybrid	
	Intra	Inter	Intra	Inter	Intra	Inter
TransportDriver	X	X	X	X	X	X
Resize	X	X	X	X	X	X
Synchronous	X	X			X	
Asynchronous			X	X		X
SequencedSegment	X	X			X	
Retransmit	X	X			X	
PositiveAck	X	X			X	
RTTEstimation	X	X			X	
ReliableFifo	X	X			X	
DuplicateAck		X				
WindowedCongestionControl		X				
TCPNewRenoCongestionAvoidance		X				
DCCPAck				X		X
DCCPWindowedCongestionControl				X		X
TCPCongestionAvoidance				X		X

Table 3.2: P2PSAP protocol composition according to algorithmic and communication context

3.7 Computational experiments

This section presents experiments with P2PSAP protocol. In order to show the dynamic configuration capability as well as the efficiency of P2PSAP, we have applied P2PSAP protocol to the solution of an optimization problem, i.e. the network flow problem. P2PSAP protocol is used in the code of the network flow problem in order to exchange updates between machines. We have used the C implementation of Cactus 2.2 for micro-protocol composition over Linux UDP sockets.

3.7.1 Network flow problems

Network flow problems [El Baz 1996b] consist in distributing the flows in a network, from a source to a destination, in a way that minimizes the total traffic cost. The problems occur in many domains: electrical networks, gas or water distribution, financial models, communication and transportation networks. The solution of nonlinear network flow problems requires intensive computations, thus a distributed or parallel solution of these problems is very attractive.

3.7.1.1 Problem formulation

Let $G = (N, A)$ be a connected directed graph. N is referred to as the set of nodes, $A \subseteq N \times N$ is referred to as the set of arcs, and the cardinal number of N is denoted by n. Let $c_{ij} : R \to (-\infty, +\infty]$ be the cost function associated with each arc $(i, j) \in A$, c_{ij} is a function of the flow of the arc (i, j) which is denoted by $f_{i,j}$. Let b_i be the supply of demand at node $i \in N$, we have $\sum_{i \in N} b_i = 0$. The problem is to minimize total cost subject to a conservation of flow constraint at each node:

$$\min \sum_{(i,j)\in A} c_{ij}(f_{ij}),\ \textit{subject to} \sum_{(i,j)\in A} f_{ij} - \sum_{(m,i)\in A} f_{mi} = b_i, \forall i \in N \tag{3.1}$$

We assume that problem (3.1) has a feasible solution. We consider the following standing assumptions on c_{ij}.

Assumption 3.1 c_{ij} is strictly convex.

Assumption 3.2 c_{ij} is lower semicontinuous.

Assumption 3.3 The conjugate convex function of c_{ij}, defined by

$$c^*_{ij}(t_{ij}) = \sup\{t_{ij}.f_{ij} - c_{ij}(f_{ij})\}$$

is real valued, i.e., $-\infty < c^*_{ij}(t_{ij}) < +\infty, \forall t_{ij} \in R$.

3.7.1.2 The dual problem

A dual problem for 3.1 is given by

$$\min_{p \in R^*} q(p), \tag{3.2}$$

subject to no constraint on the vector $p = \{p_i | i \in N\}$, where q is the dual functional given by

$$q(p) = \sum_{(i,j) \in A} c^*_{ij}(p_i - p_j) - \sum_{i \in N} b_i . p_i$$

We refer to p as a price vector and its components as prices. The $i-th$ price p_i is a Lagrange multiplier associated with the $i-th$ conservation of flow constraint. The necessary and sufficient condition for optimality of a pair (f,p) is given as follows: a feasible flow vector $f = \{f_{ij} | (ij) \in A\}$ is optimal for (3.1) and a price vector $p = \{p_i | i \in N\}$ is optimal for (3.2) if and only if for all $(i,j) \in A$, $p_i - p_j$ is a sub-gradient of c_{ij} at f_{ij}. An equivalent condition is $f^*_{ij} = \nabla c^*_{ij}(p_i - p_j)$ where $\nabla c^*_{ij}(x)$ denotes the gradient of $c^*_{ij}(x)$.

3.7.1.3 The dual optimal solution set

The optimal solution of the dual problem is never unique since adding the same constant to all coordinates of a price vector leaves the dual cost unaffected. We can remove this degree of freedom by constraining the price of one node, say the destination node d, to be zero. Consider the set $P = \{p \in R^n | p_d = 0\}$. We concentrate on the reduced dual problem:

$$\min_{p \in P} q(p) \tag{3.3}$$

The reduced optimal solution set is defined by:

$$P^* = \{p' \in P | q(p') = \min_{p} q(p)\}$$

Assumption 3.4 The reduced dual optimal solution set P^* is nonempty and compact.

We note that assumption 3.4 is not very restrictive (see [El Baz 1996b]). In the sequel, $g(p)$ will denote the gradient of the dual functional, the components $g_i(p)$ of $g(p)$ are given by:

$$g_i(p) = \frac{\partial q(p)}{\partial p_i} = \sum_{(i,j) \in A} \nabla c^*_{ij}(p_i - p_j) - \sum_{(m,i) \in A} \nabla c^*_{mi}(p_m - p_i) - b_i, i \in N$$

3.7.1.4 Parallel iterative algorithms

One can implement several parallel iterative methods for the solution of the reduced dual problem. We present a gradient type method. The components F'_i of the gradient mapping F' are defined by $F'_i = p_i - \frac{1}{\alpha} g_i(p)$ for all $i \in N - \{d\}$ and $p \in P$, where α is a positive constant. Clearly F' is continuous since g is continuous. We introduce the following assumption.

Assumption 3.5 c_{ij} is strongly convex with modulus $\frac{1}{\beta}$.

Under Assumptions 3.1 to 3.5, there exists a constant $\alpha = \beta. \max_{i \in N} a_i$, where a_i denotes the degree of node $i \in N$, such that for all $p, p' \in P$ satisfying $p' \leq p$, we have $g(p) - g(p') \leq \alpha.(p - p')$. Therefore, the gradient mapping F' is monotone on P if $\alpha = \beta. \max_{i \in N} a_i$ (see [El Baz 1996b]). The gradient type method consists in iterating on the $i - th$ component of the vector of prices as follows:

$$p_i^q = p_i^{q-1} - \frac{1}{\alpha} g_i(p_i^{q-1}, p), i = 1, ..., n, q \in \{1, 2, ..., q'\}$$

where $p_i^0 = p_i$ and q' is the number of iterations such that $|g_i(p_i^{q'})| \leq \varepsilon$ where $p_i^0 = P_i$ and ε is the research accuracy.

3.7.2 Platform

Experiments have been carried out on the LAASNETEXP experimental network [Owezarski 2008]. The topology of the toy network used for this first set of computational experiments is shown on the Figure 3.14 where peers are connected via a Gigabit Ethernet network. Machines P_1, P_2, P_3 and P_4 are similar, i.e. Dual Core Xeon 3050, 2.13GHz with Linux Debian.

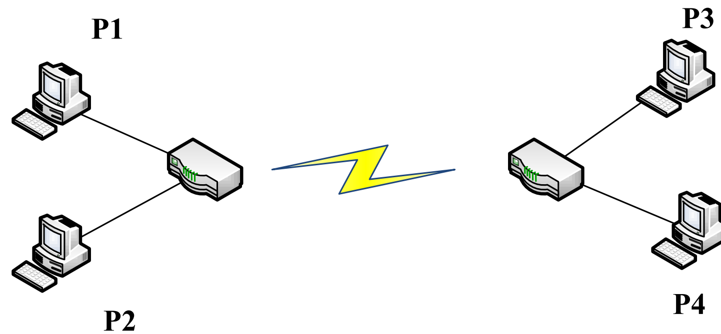


Figure 3.14: Network used for computational tests on the LAASNETEXP experimental network

3.7.3 Computational results

We have considered gas distribution problems solved via gradient type methods [El Baz 1996b]. The network topology corresponds to a grid-like network with 20×200 nodes. Computations have been carried out on 1, 2 and 4 machines. In the distributed case, i.e. for several machines, the original network flow problem is decomposed into several equal sub-networks. In the particular case of 2 machines, computations are made within the same cluster. We have carried out experiments with different computational schemes and communication scenarios i.e. synchronous, asynchronous and hybrid (synchronous / asynchronous). A set of computational results is displayed in Table 3.3. Here, the speedup s is computed as follows:

$$s = \frac{t_s}{t_p}$$

where t_s is the sequential computational time and t_p is the parallel computational time; the efficiency e is computed as follows:

$$e = \frac{s}{\alpha}$$

where α is the number of machines.

PCs	Scheme	Number of relaxation				Times	s	e
		P_1	P_2	P_3	P_4	(s)		
1	-	399813	-	-	-	2135	-	-
2	Syn	400694	400694	-	-	1481	1,44	0,72
2	Asyn	385780	583735	-	-	1209	1,76	0,88
4	Syn	402056	402056	402056	402056	1241	1,72	0,43
4	Asyn	419175	389144	464128	743636	656	3,25	0,81
4	Hybrid	449372	449372	398421	398421	935	2,28	0,57

Table 3.3: Computational results for network flow problems on LAASNETEXP

For the application and topology considered, we note that asynchronous iterative schemes have performed better than the synchronous ones. Moreover, the efficiency of synchronous iterative schemes deteriorates greatly when the number of processors increases, i.e. 0.72 with 2 machines and 0.44 with 4 machines. The efficiency of asynchronous iterative schemes decreases slowly with the number of processors, i.e. 0.88 with 2 machines and 0.81 with 4 machines. This is mainly due to waiting time due to synchronization and synchronization overhead.

When using asynchronous iterative schemes of computation, some processors may iterate faster than others; this is particularly the case when loads are unbalanced as for the application considered here. We note that the parallel gradient type algorithms led to nondeterministic load unbalancing although all machines receive a sub-network of the same size. Furthermore, in the synchronous case, the more

unbalanced the machine loads are, the greater the idle times due to synchronization are. This is the reason why the efficiency of the synchronous case is small (0.44) with a small number of machines (4 machines). The asynchronous iterative schemes are well suited to load unbalancing.

The use of hybrid iterative schemes, i.e. synchronous communication between peers in the same cluster and asynchronous communication between peers in different clusters gave in this case efficiency in between pure synchronous and asynchronous cases.

3.8 Chapter summary

In this chapter, we have proposed P2PSAP, a self-adaptive communication protocol for peer-to-peer high performance computing. P2PSAP protocol is designed in order to allow rapid update exchange between peers in the solution of numerical simulation problems via distributed iterative algorithms. The protocol can configure itself automatically and dynamically in function of application requirements like scheme of computation and elements of context like topology by choosing the most appropriate communication mode between peers. We note that this approach is different from existing communication libraries for high performance computing like MPICH/Madeleine [Aumage 2001] in allowing the modification of internal transport protocol mechanism in addition to switch between networks.

P2PSAP protocol has been implemented on a small network for the solution of nonlinear optimization problems, i.e. network flow problems. A set of computational experiments shows that the protocol permits one to obtain good efficiency particularly when using asynchronous communications or a combination of synchronous and asynchronous communications.

In next chapter, we shall present the centralized version of environment for peer-to-peer high performance computing that makes use of P2PSAP protocol in order to exchange updates between peers.

Chapter 4

Centralized version of the environment for peer-to-peer high performance computing

Contents

4.1 Introduction **53**
4.2 Global architecture **54**
4.3 Programming model **55**
4.3.1 Communication operations 55
4.3.2 Application programming model 56
4.4 Implementation **58**
4.4.1 User daemon 58
4.4.2 Resource manager 58
4.4.3 Application repository 59
4.4.4 Task manager 59
4.5 Computational results **60**
4.5.1 Obstacle problem 60
4.5.2 Implementation 62
4.5.3 NICTA testbed and OMF framework 65
4.5.4 Problems and computational results 65
4.6 Chapter summary **68**

4.1 Introduction

In the previous chapter, we have proposed P2PSAP, a self-adaptive communication protocol for peer-to-peer high performance computing. The self-adaptability of P2PSAP allows programmers not to care about the choice of communication mode and leave it to communication protocol.

In this chapter, we shall present the first version of P2PDC, an environment for peer-to-peer high performance computing that makes use of P2PSAP to allow direct communication between peers [Nguyen 2010]. We define the global architecture of P2PDC with mains functionalities. Moreover, we propose a new programming

model that is suited to P2P High Performance Computing (HPC) applications and more particularly applications solved by iterative algorithms. This programming model facilitates the work of programmers and allows them to develop easily a P2P HPC application. A centralized implementation of P2PDC with simple functionalities is developed in order to validate the programming model. Computational results are presented for a simulation problem at NICTA testbed. We note that the goal of the implementation of the centralized version with simple functionalities is to validate the programming model for a given application. Moreover, this allows us to provide to our partners in the project ANR-CIP with the programming model and a centralized version of the environment P2PDC so that they can develop applications for P2PDC in parallel with new developments of our environment and test applications with the centralized version of P2PDC. The evolution of P2PDC toward a decentralized, more complete (see Chapter 5) and fault-tolerant (see Chapter 6) version requires small changes in the code of applications.

This chapter is structured as follows. Next section describes the global architecture of P2PDC; the main functionalities of P2PDC are also presented. Section 4.3 presents the programming model with a reduced set of communication operations and explains how it facilitates the work of programmers. The first implementation of P2PDC with centralized and simplified functions is detailed in section 4.4. Section 4.5 displays and analyzes a set of computational experiments for a simulation problem, i.e. the obstacle problem. Finally, a summary of the centralized version of P2PDC is presented.

4.2 Global architecture

Figure 4.1 illustrates the architecture of our environment. We describe now its main components.

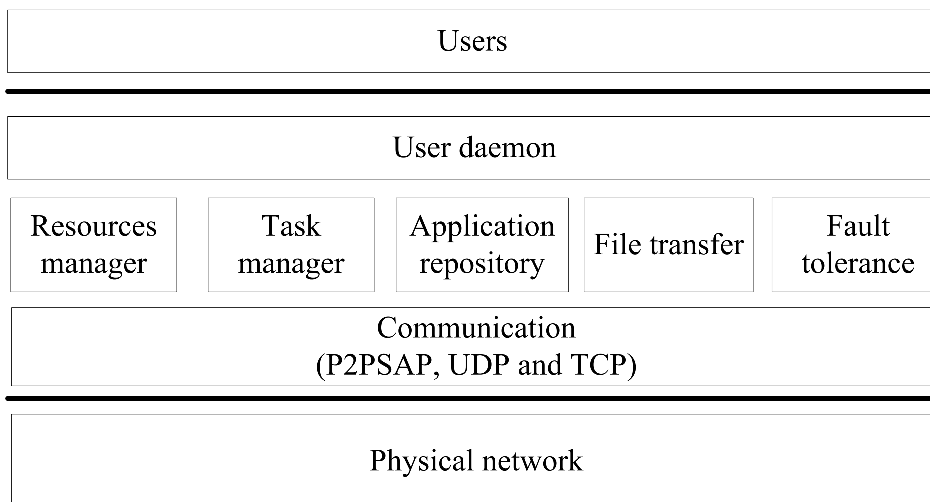


Figure 4.1: General architecture of P2PDC

- *User daemon* is the interaction interface between users and the environment. It allows users to submit their tasks and retrieve final results.
- *Resource manager* organizes peers connected to overlay network with a topology that facilitates peer discovery for a computation.
- *Task manager* is responsible for subtasks distribution, subtasks execution and results collection and aggregation.
- *Application repository* contains the code of all applications that can be run with the environment.
- *File transfer* transfers files between peers.
- *Fault-tolerance* ensures the integrity of the computation in case of peer failures.
- *Communication* provides support for data exchange between peers. We note that P2PSAP is used for data exchange of a given application; control messages of environment like messages used by resource manager to maintain the topology of connected peers or messages used to send subtasks to workers are exchanged using UDP and TCP.

4.3 Programming model

Programming model is the way programmers develop their application. We have proposed a programming model that allows all programmers to develop their own application easily.

4.3.1 Communication operations

The set of communication operations is reduced. There are only a send and a receive operations (*P2P_Send* and *P2P_Receive*). The idea is to facilitate programming of large scale peer-to-peer applications and hide complexity of communication management as much as possible. Contrarily to MPI communication library where communication mode is fixed by the semantics of communication operations, the communication mode of a given communication operation which is called repetitively can vary with P2PDC according to the context; e.g. the same *P2P_Send* from peer A to peer B, which is implemented repetitively, can be first synchronous and then become asynchronous. As a consequence, the programmer does not fix directly the communication mode; he rather selects the type of scheme of computation he wants to be implemented, e.g. synchronous or asynchronous iterative scheme or let the protocol free of choosing communication mode, this corresponds to a hybrid scheme. When the system is set free, the choice of communication mode depends only on elements of context like topology change and is thus dynamic.

Here are the prototype of two communication operations:

*int P2P_Send(P2PSubtask *pSubtask, uint32_t dest, char *buffer, size_t size)*

*int P2P_Receive(P2PSubtask *pSubtask, uint32_t dest, char *buffer, size_t size)*

where

- *pSubtask* is the current subtask.
- *dest* is the rank of destination subtask.
- *buffer* is the initial address of send buffer.
- *size* is the size of data to be sent or received.

4.3.2 Application programming model

Figure 4.2 shows the activity diagram that a parallel application must follow in order to be deployed. The so-called submitter is the peer where the task is initiated and submitted to environment. Workers are peers that receive and execute subtasks.

- *Task definition*: first, the task is defined at the submitter, i.e. setting task parameters such as computational scheme, number of peers necessary as well as the number of subtasks and subtask parameters.
- *Collect peers*: based on the task definition, the submitter collects free peers in the overlay network.
- *Enough peers*: the submitter verifies if there are enough free peers to carry out the task. If there are not enough free peers, then the computation is terminated.
- *Send subtask*: if there are enough free peers, then the submitter sends subtask to those peers.
- *Receive subtask*: peers receive subtask from submitter, so they become workers.
- *Calculate*: all workers execute received subtask. Depending on the choice of the user, the submitter can also execute a subtask. We note that in the case of applications solved by iterative algorithms, a worker has to carry out many relaxations; after each relaxation, it has to exchange updates with others workers.
- *Send results*: when a worker has finished a subtask, it sends subtask's result to submitter.
- *Receive results*: the submitter receives subtask's results from workers.
- *Results aggregation*: subtask's results are aggregated into final result and are written to an output such as a console or a file.

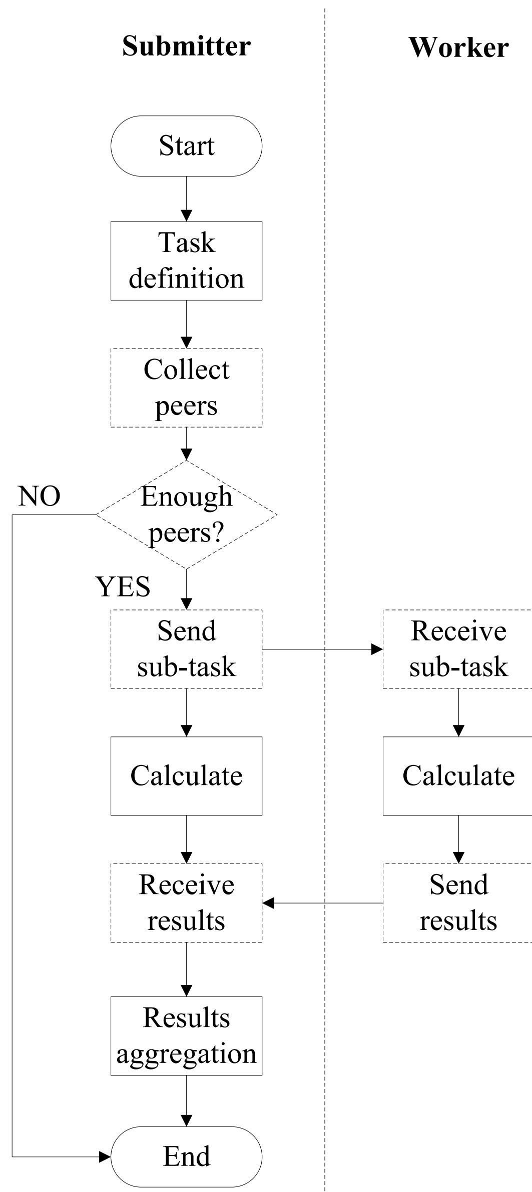


Figure 4.2: Activity diagram of a parallel application

In order to facilitate the work of programmers, we want the environment to carry out most of those activities automatically. Hence we propose a programming model based on this diagram. Only activities with solid line boundary, i.e. *Task definition*, *Calculate* and *Results aggregation*, are taken into account by the programmers. Activities with broken line boundary, i.e. *Collect peer*, *Send subtask*, *Receive subtask*, *Send results*, *Receive results*, are taken into account by the environment and are transparent to programmers. Thus, in order to develop an application, programmers have to write code for only three functions corresponding to the following three activities: *Task_Definition()*, *Calculate()* and *Results_Aggregation()*. In the *Task_Definition()* function, programmers define the task in indicating the number of subtasks and subtask data. The computational scheme and number of peers necessary can also be set in this function but they can be overridden at start time in command line. On what concerns the *Calculate()* function, programmers write subtasks code; they can use *P2P_Send()* and *P2P_receive()* to send or receive updates at each relaxation. In the *Results_Aggregation()* function, programmers define how subtasks results are aggregated into final result and write the final result to an output, i.e. a console or a file. *Task_Definition()* and *Results_Aggregation()* functions are called on submitter. Depending on the choice of the user, the *Calculate()* function is called only on workers or on both workers and submitter.

We note that this programming model not only carries out automatically most of support activities to execute computations but also manages advance tasks such as fault tolerance, then reducing the work of programmers.

4.4 Implementation

In this section, we present the implementation of a first version of P2PDC with centralized resource manager and simplified functionalities.

4.4.1 User daemon

In the centralized version, the User daemon component constitutes the command line interface between user and environment. We outline here some principal commands:

- *run*: run an application. Parameters are application name and application owner parameters that will be passed to *Task_Definition()* function.
- *stat*: return actual state of node.
- *exit*: quit the environment.

4.4.2 Resource manager

The resource manager organizes connected peers in a centralized manner as in the Figure 4.3. A server is used in order to store information about all peers in the network. When a node joins the overlay network, it sends to the server a "join"

message. Upon the reception of a "join" message from a peer, the server adds the new peer-to-peer list and sends to the peer an "accept" message. Peers must send ping messages periodically to server to inform it that they are alive. If the server does not receive any ping message from a peer after a time T, then the server considers that this peer is disconnected and removes it from the peer list.

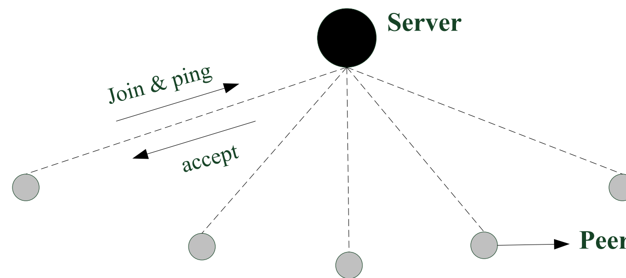


Figure 4.3: Centralized topology of resource manager.

Peer collections for a task execution is done as follows. When an user submits a task to environment, the task manager of the submitter sends a request message to the server with number of peers needed NP. The server checks if there are enough free peers in its peer list to meet this request. If there are not enough free peers, then the server sends an error message to the task manager of the submitter. In the contrary case, the server choose NP free peers from peer list and sends their address to the task manager of submitter.

When a peer is assigned to a task, the server marks that this peer is busy and not available to others tasks. A busy peer does not need to send ping message to server. When a peer has finished a task, it sends a ping message to server to inform that it is free and can receive another task.

4.4.3 Application repository

Application, in order to be run with P2PDC environment, needs to be developed according to the programming model presented in the section 4.3. Moreover, application needs to be added to the application repository. Each application is identified by a name that will be used to search and run application. In this version of P2PDC, application is added manually to application repository and are compiled at the same time with the environment.

4.4.4 Task manager

Task manager is the main component that calls functions of the application and carries out necessary actions to support execution of the application. When an user starts an application using the run command on a submitter, Task manager of the submitter finds the corresponding application in the application repository via application name and calls the *Task_Definition()* function. Afterward, it requests

peers from Resource manager on the basis of number of peers needed by application and sends subtasks along with their data to collected peers.

At peer side, when a peer receives a subtask, the Task manager finds the corresponding application on the application repository via application name and calls the *Calculate()* function. When the *Calculate()* has finished, the Task manager sends the result to submitter.

When the submitter has received results from all peers, Task manager of the submitter calls the *Results_Aggregation()* function.

File transfer and **Fault-tolerance** components are not developed in this version.

4.5 Computational results

We present now and analyze a set of computational experiments with the centralized version of P2PDC for the obstacle problem.

4.5.1 Obstacle problem

The application we consider, i.e. the obstacle problem, belongs to a large class of numerical simulation problems (see [Spitéri 2002] and [Lions 2002]). The obstacle problem occurs in many domains like mechanics and financial mathematics, e.g. options pricing.

4.5.1.1 Problem formulation

In the stationary case, the obstacle problem can be formulated as follows:

$$\begin{cases} Find\ u^* \ such\ that \\ A.u^* - f \geq 0, u^* \geq \phi \ everywhere\ in\ \Omega, \\ (A.u^* - f)(\phi - u^*) = 0 \ everywhere\ in\ \Omega, \\ B.C., \end{cases}$$

where $\phi \in \mathbb{R}^2 (or\ \mathbb{R}^3)$ is an open set, A is an elliptic operator, ϕ a given function and $B.C.$ denotes the boundary conditions on $\partial\Omega$.

There are many equivalent formulations of the obstacle problem in the literature like complementary problem, variational inequality and constrained optimization problem. Reference is made to [Lions 2002], [Spitéri 2002] and [Miellou 1985a] for more details. We concentrate here on the following variational inequality formulation:

$$\begin{cases} Find\ u^* \in K such\ that \\ \forall v \in K, \langle A.u^*, v - u^* \rangle \geq \langle f, v - u^* \rangle, \end{cases}$$

where K is a closed convex set defined by

$$K = v | v \geq \phi \ everywhere\ in\ \Omega,$$

and $\langle ., . \rangle$ denotes the dot product $\langle u, v \rangle = \int uvdx$.

4.5.1.2 Fixed point problem and projected Richardson method

The discretization of the obstacle problem leads to the following large scale fixed point problem whose solution via distributed iterative algorithms (i.e. successive approximation methods) presents many interests.

$$\begin{cases} Find\ u^* \in V\ such\ that \\ u^* = F(u^*), \end{cases} \tag{4.1}$$

where V is an Hilbert space and the mapping $F : v \to F(v)$ is a fixed point mapping from V into V. Let α be a positive integer, for all $v \in V$, we consider the following block-decomposition of v and the associated block-decomposition of the mapping F for distributed implementation purpose:

$$\begin{aligned} v &= (v_1, \dots, v_\alpha) \\ F(v) &= (F_1(v), \dots, F_\alpha(v)). \end{aligned}$$

We have $V = \Pi_{i=1}^{a} V_i$, where V_i are Hilbert spaces; we denote by $\langle .,. \rangle_i$ the scalar product on V_i and $|.|_i$ the associated norm, $i \in \{1, \dots, \alpha\}$; for all $u, v \in V$, we denote by $\langle u, v \rangle = \sum_{i=1}^{\alpha} \langle u_i, v_i \rangle_i$, the scalar product on V and $|.|$ the associated norm on V. In the sequel, we shall denote by A a linear continuous operator from V onto V, such that $A.v = (A_1.v, \dots, A_\alpha.v)$ and which satisfies:

$$\forall i \in \{1, \dots, \alpha\}, \forall v \in V, \langle A_i.v, v_i \rangle \geq \sum_{j=1}^{\alpha} n_{i,j} |v_i|_i |v_j|_j, \tag{4.2}$$

where

$$N = (n_{i,j})_{i \leq i,j \leq \alpha}\ is\ an\ M - matrix\ of\ size\ \alpha \times \alpha \tag{4.3}$$

The reader is referred to [Varga 1962] for the definition of $M - matrix$. Similarly, we denote by K_i, a closed convex set such that $K_i \subset V_i, \forall i \in \{1, \dots, \alpha\}$, we denote by K, the closed convex set such that $K = \Pi_{i=1}^{a} K_i$ and b, a vector of V that can be written as: $b = (b_1, \dots, b_\alpha)$. For all $v \in V$, let $P_K(v)$ be the projection of v on K such that $P_K(v) = (P_{K_1}(v_1), \dots, P_{K_\alpha}(v_\alpha))$, where P_{K_i}denotes the mapping that projects elements of V_i onto $K_i, \forall i \in \{1, \dots, \alpha\}$. For any $\delta \in \mathbb{R}, \delta > 0$, we define the fixed point mapping F_δ as follows (see [Spitéri 2002]).

$$\forall v \in V, F_\delta(v) = P_K(v - \delta(A.v - b)), \tag{4.4}$$

The mapping F_d can also be written as follows.

$$\begin{aligned} F_\delta(v) &= (F_{1,\delta}(v), \dots, F_{\alpha,\delta}(v))\ with \\ F_{i,\delta}(v) &= P_{K_i}(v_i - \delta(A_i.v - b_i)), \forall v \in V, \forall i \in \{1, \dots, \alpha\}. \end{aligned}$$

4.5.1.3 Parallel projected Richardson method

We consider the distributed solution of fixed point problem 4.1 via projected Richardson method combined with several schemes of computation, e.g. a Jacobi like synchronous scheme: $u^{p+1} = F_\delta(u^p), \forall p \in N$ or asynchronous schemes of computation that can be defined as follows (see [Spitéri 2002]).

$$\left\{ \begin{array}{l} u_i^{p+1} = F_{i,\delta}(u_1^{\rho_1(p)}, \ldots, u_j^{\rho_j(p)}, \ldots, u_\alpha^{\rho_\alpha(p)}) \; if \; i \in s(p), \\ u_i^{p+1} = u_i^p \; if \; i \notin s(p), \end{array} \right. \tag{4.5}$$

where

$$\left\{ \begin{array}{l} s(p) \subset \{1, \ldots, \alpha\}, s(p) \neq \phi, \forall p \in N, \\ \{p \in N | i \in s(p)\}, \; is \; infinite, \forall i \in \{1, \ldots, \alpha\}, \end{array} \right. \tag{4.6}$$

and

$$\left\{ \begin{array}{l} j(p) \in N, 0 \leq \rho_j(p) \leq p, \forall j \in \{1, \ldots, \alpha\}, \forall p \in N, \\ \lim_{p \to \infty} \rho_j(p) = +\infty, \forall j \in \; \{1, \ldots, \alpha\}. \end{array} \right. \tag{4.7}$$

The above asynchronous iterative scheme can model computations that are carried out in parallel without order nor synchronization. In particular, it permits one to consider distributed computations whereby peers go at their own pace according to their intrinsic characteristics and computational load. Finally, we note that the use of delayed components in 4.5 and 4.7 permits one to model nondeterministic behavior and does not imply innefficiency of the considered distributed schemes of computation. The convergence of asynchronous projected Richardson method has been established in [Spitéri 2002] (see also [Miellou 1985a]), [Giraud 1991] and [Miellou 1985b].

The choice of scheme of computation, i.e. synchronous, asynchronous or any combination of both schemes will have important consequences on the efficiency of distributed solution. The interest of asynchronous iterations for high performance computing in various contexts including optimization and boundary value problems have been shown in [Spitéri 2002], [El Baz 1990], [Bertsekas 1987], [Bertsekas 1989], [El Baz 1994] and [El Baz 1998].

4.5.2 Implementation

We have considered 3D obstacle problems. Let n^3 denote the number of discretization points. In the *Task_Definition()* function, the iterate vector is decomposed into n sub-blocks of n^2 points. The sub-blocks are assigned to α subtasks with $\alpha \leq n$. Subtasks are then allocated to α nodes. This decomposition is called *slice decomposition*. Figure 4.4 illustrates the decomposition of the iterate vector in the case where $n = 32$ and $\alpha = 8$.

The sub-blocks are computed sequentially at each node. The code for sequential computation of sub-blocks at each node is written in the *Calculate()* function. For simplicity of presentation and without loss of generality, we have displayed in Figure

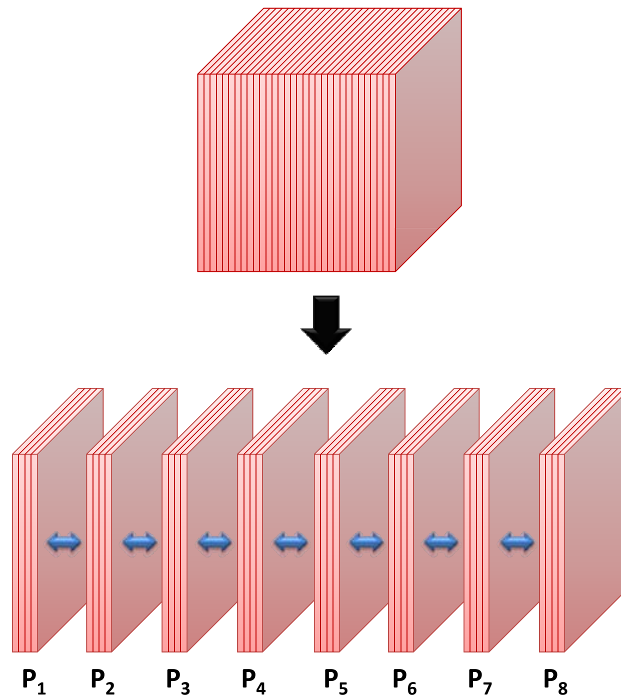


Figure 4.4: Slice decomposition of the 3D obstacle problem.

4.5 the basic computational procedure at node P_k with $k \neq 1, k \neq \alpha$. We note that in our experiments, the scheme of computation (synchronous, asynchronous or hybrid, i.e. combination of both schemes) is chosen at the beginning of the resolution; whereas, the communication mode is decided at runtime by the P2PSAP protocol according to Table 3.1.

The node P_k updates the components of the sub-blocks of the iterate vector denoted by $U_{f(k)}, U_{f(k)+1}, ..., U_{l(k)}$, where $U_{f(k)}$ stands for the first sub-block assigned to the node P_k and $U_{l(k)}$ stands for the last sub-block assigned to the node P_k. We note that the transmission of $U_f(k)$ to node P_{k-1} is delayed so as to reduce the waiting time in the synchronous case.

The convergence test is based on the error between components of iterate vector of two consecutive relaxations (see [Spitéri 2002]). The convergence is detected if $\delta = \max(|u^{p+1} - u^p|) < \varepsilon$ where ε is a positive constant. In our experiments, $\varepsilon = 1e - 11(10^{-11})$. In the distributed cases of all three computational schemes, the termination is detected as follows. Two tokens are appended to updates exchanged between nodes: token $tok_conv_{k,k+1}$ is appended to updates sent from node P_k to P_{k+1} in order to collect information about local termination test; token $tok_term_{k,k-1}$ is appended to the updates sent from P_k to P_{k-1} in order to propagate the termination (see Figure 4.6). Both tokens have type *boolean* and their default value is *FALSE*. $tok_conv_{k,k+1} = TRUE$ if and only if $tok_conv_{k-1,k} = TRUE$ and the local termination test at node P_k is satis-

1: **send** $U_{l(k)}$ to node $k+1$
2: **repeat**
3: $\quad i \leftarrow f(k)$
4: $\quad$ **receive** U_{i-1} from node $k-1$
5: $\quad U_i \leftarrow F_{i,\delta}(U_{i-1}, U_i, U_{i+1})$

6: $\quad$ **for** $i = f(k)+1 \rightarrow l(k)-1$ **do**
7: $\quad\quad U_i \leftarrow F_{i,\delta}(U_{i-1}, U_i, U_{i+1})$
8: $\quad$ **end for**

9: $\quad$ **send** $U_{f(k)}$ to node $k-1$
10: $\quad i \leftarrow l(k)$
11: $\quad$ **receive** U_{i+1} from node $k+1$
12: $\quad U_i \leftarrow F_{i,\delta}(U_{i-1}, U_i, U_{i+1})$
13: $\quad$ **send** U_i to node $k+1$
14: **until** convergence

Figure 4.5: Basic computational procedure at node P_k.

fied. It means that if $tok_conv_{k,k+1} = TRUE$, then the local termination test at nodes $1, \dots, k$ is satisfied. When $tok_conv_{\alpha-1,\alpha} = TRUE$ and the local termination at node P_α is satisfied, node P_α detects the termination. Then, node P_α sets $tok_term_{\alpha,\alpha-1} = TRUE$, sends update to node $P_{\alpha-1}$, sets values of components of sub-blocks of the iterate vector as result of the subtask and terminates the computation. When node P_k receives $tok_term_{k+1,k} = TRUE$, it sets $tok_term_{k,k-1} = TRUE$, sends update to node P_{k-1}, sets values of components of sub-blocks of the iterate vector as result of the subtask and terminates the computation.

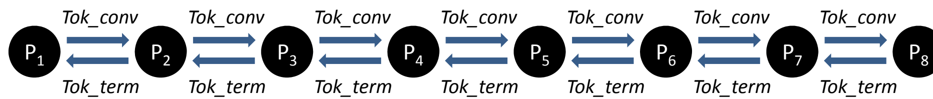


Figure 4.6: Termination detection in the case of slice decomposition.

In the *Results_Aggregation()* function, the final result of the task, i.e. final values of components of the iterate vector is built from final values of components of sub-blocks extracted from result field of subtasks. The final result is then written to a file.

4.5.3 NICTA testbed and OMF framework

Computational experiments have been carried out on the NICTA testbed [nic], Sydney, Australia. This testbed is constituted of 38 machines having the same configuration, i.e. processor speed 1GHz, memory 1GB based on Voyage Linux distribution. Those machines are connected via 100MBits Ethernet network.

NICTA testbed uses OMF (cOntrol and Management Framework) to facilitate the control and management of the testbed (see [Rakotoarivelo 2009, omf]). OMF provides a set of tools to describe and instrument an experiment, execute it and collect its results; OMF provides also a set of services to efficiently manage and operate the testbed resources (e.g. resetting nodes, retrieving their status information, installing new OS image). Furthermore, NICTA has developed OML (Orbit Measurement Library), a stand-alone software which could be used to collect and store any type of measurements from any type of application. More details about OMF and OML will be presented in section 7.2.

In order to perform our experimentations, we have written plural experiment descriptions files, using OMF's Experiment Description Language (OEDL), corresponding to different scenarios. Each experiment description file contains: configuration of the network topology, i.e. peer's IP address assignment so that they are in the desired cluster; network parameters, i.e. communication latency and path to application with appropriate parameters. Further details about OEDL and our descriptions files will be presented in Appendix A.

4.5.4 Problems and computational results

In this chapter, we present a set of computational experiments obtained with $n = 96$ and $n = 144$. Experiments have been carried out on 1, 2, 4, 8, 16 and 24 machines of the NICTA testbed. In the distributed context, i.e. for several machines, we have considered the case where machines either belong to a single cluster or are divided into 2 clusters connected via Internet. We used the Netem tool to simulate the Internet context; the latency between 2 clusters is set to 100ms. We have carried out experiments with different schemes of computation, i.e. synchronous, asynchronous and hybrid.

Figures 4.7 and 4.8, respectively, show the time, number of relaxations, speedup and efficiency of the different parallel schemes of computation in the case where $n = 96$ and $n = 144$, respectively. For the application and topologies considered, we note that asynchronous schemes of computation have performed better than the synchronous ones.

The efficiency of asynchronous schemes of computation decreases slowly with the number of processors; while the efficiency of synchronous schemes of computation deteriorates greatly when the number of processors increases (this is particularly true in the case of 2 clusters); this is mainly due to synchronization overhead and waiting time.

The speedup of synchronous schemes of computation is very small for 24 nodes.

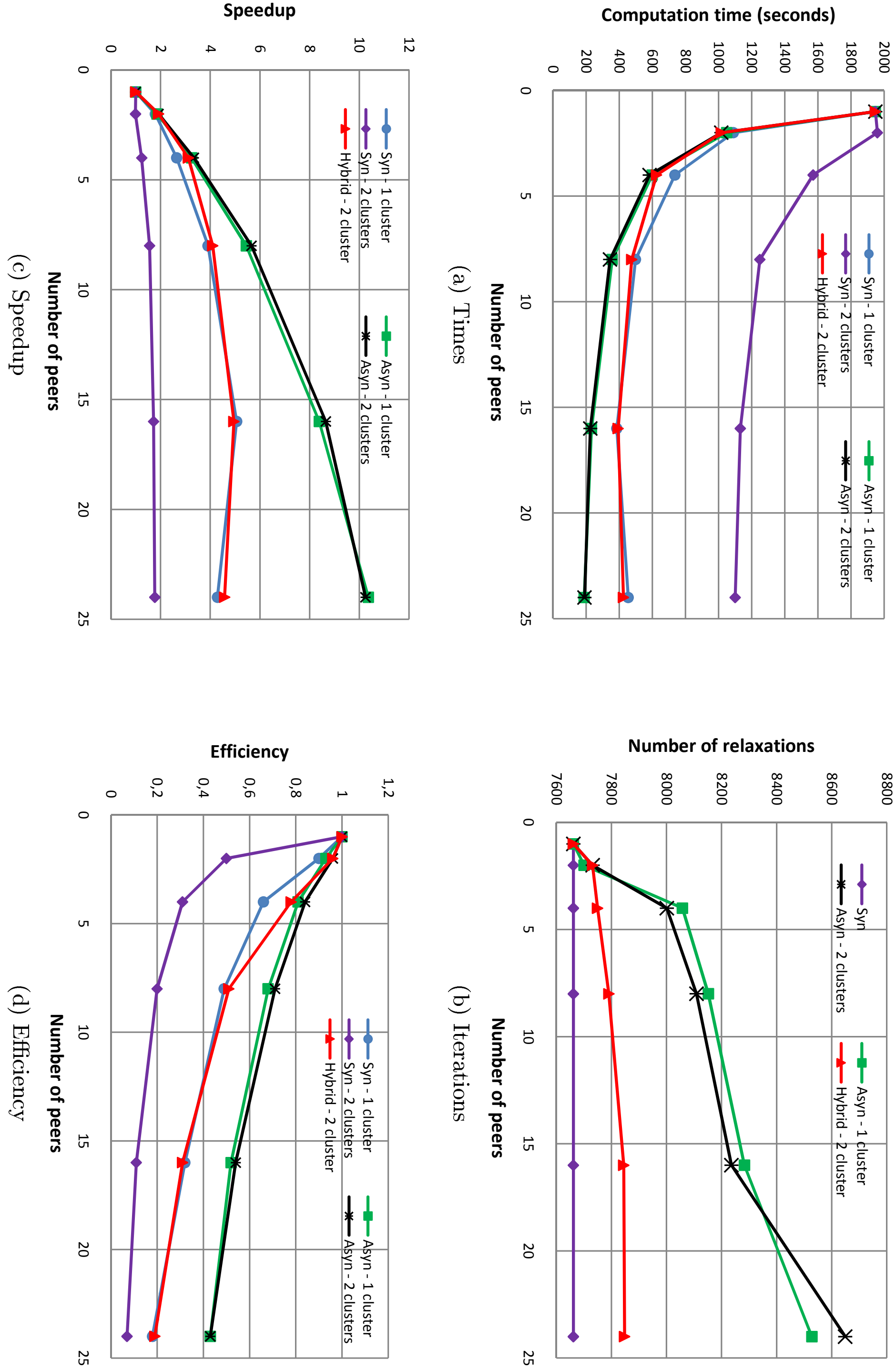
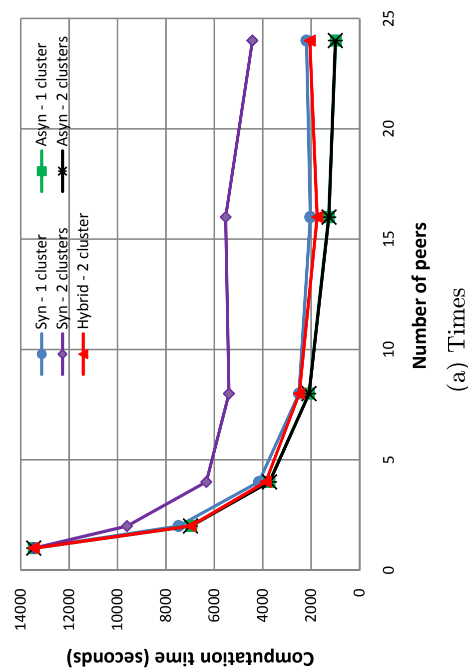
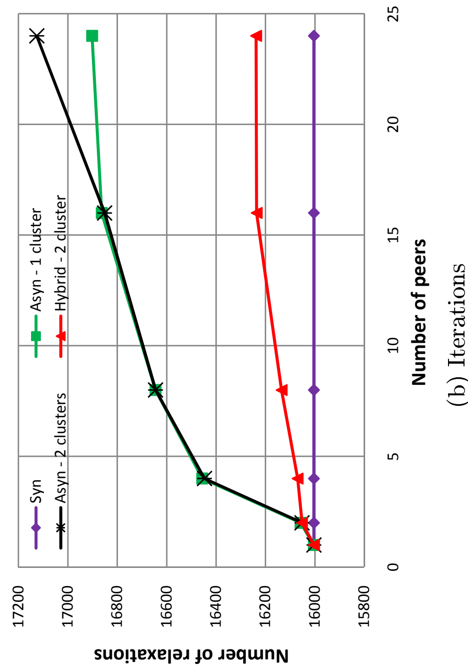


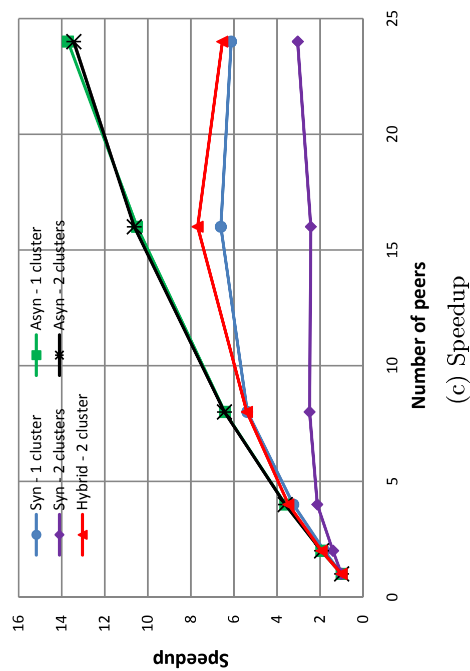
Figure 4.7: Computational results in the case of the obstacle problem with size $96 \times 96 \times 96$



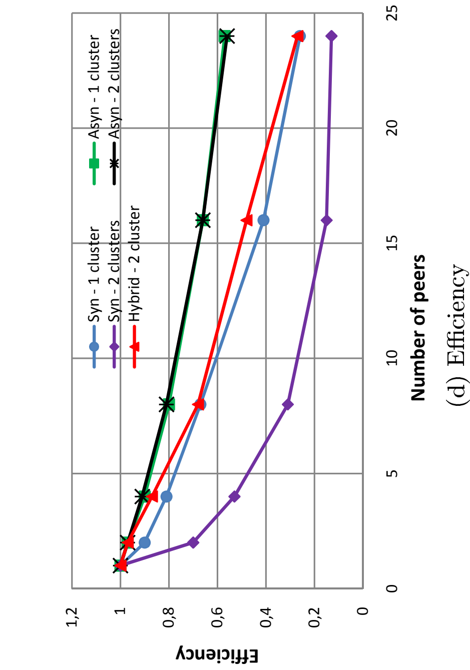
(a) Times



(b) Iterations



(c) Speedup



(d) Efficiency

Figure 4.8: Computational results in the case of the obstacle problem with size $144 \times 144 \times 144$

This can be explained as follow: when 24 nodes are used, each node calculates only a small number of sub-blocks; since exchanged messages and sub-blocks have the same size, communication overhead and waiting time reach a significant proportion.

When we compare the computational results with 1 and 2 clusters, we can see that there is not much difference with regard to the asynchronous schemes; while in the synchronous cases, 1 cluster results are better than 2 clusters results. This is due to the fact that communication latency between 2 clusters (100ms) increases the waiting time due to synchronization; this means that synchronous communication is sensible to latency increase and not appropriate for the communication between clusters.

When the problem size increases from $n = 96$ to $n = 144$, the efficiency of distributed methods increases since granularity increases.

The number of relaxations performed by synchronous schemes remains constant although the sub-block processing order is changed by the distribution of computation.

In the case of asynchronous schemes of computation, some nodes may iterate faster than others; this is particularly true when nodes have fewer neighbors than others, like nodes 1 and α that have only one neighbor. Then, the average number of relaxations increases with the numbers of machines, as depicted in Figure 4.7b and 4.8b.

The efficiency of hybrid schemes of computation is situated in between efficiencies of synchronous and asynchronous schemes.

It follows from the computational experiments that the choice of communication mode has important consequences on the efficiency of the distributed methods. The ability for the protocol P2PSAP to choose the best communication mode in function of network topology and context appears as a crucial property. We note also that the choice of communication mode has important consequences on the reliability of the distributed method and everlastingness of the high performance computing application. With regards to these topics, we note that asynchronous communications are more appropriate in the case of communications between clusters.

4.6 Chapter summary

In this chapter, we have described the general architecture of P2PDC with its main functionalities. Afterward, we have proposed a programming model for P2PDC that facilitates the work of programmer. Indeed, in order to develop an application, programmers have to write code for only three functions; all others support activities are carried out automatically by the environment. Moreover, the communication operations set is reduced with only two operations, thus programmers do not have to care about the choice of communication mode as well as communication operation to achieve it. The development of an application with P2PDC takes less effort of programmers than with MPI and PVM. The first implementation of P2PDC with centralized and simplified functionalities has been also presented. Finally, we

have displayed and analyzed computational results on the NICTA platform with up to 24 machines for numerical simulation problem, i.e. the obstacle problem. Computational results show that the combination of P2PSAP and P2PDC allows to solve efficiently large scale numerical simulation problems via distributed iterative methods, in particular when using asynchronous or hybrid schemes of computation.

In the next chapter, we shall present the decentralized version of P2PDC with some new features that make P2PDC more scalable and efficient.

Chapter 5

Decentralized environment for peer-to-peer high performance computing

Contents

5.1 Introduction **71**

5.2 Hybrid resource manager **72**

5.2.1 General topology architecture 73

5.2.2 IP-based proximity metric 74

5.2.3 Topology initialization 74

5.2.4 Tracker joins 74

5.2.5 Peer joins 75

5.2.6 Tracker leaves 75

5.2.7 Peer leaves 76

5.2.8 Peers collection 76

5.3 Hierarchical task allocation **77**

5.4 Dynamic application repository **78**

5.5 File transfer **78**

5.6 New communication operations **79**

5.7 Computational experiments **80**

5.7.1 New approach to the distributed solution of the obstacle problem 80

5.7.2 Grid'5000 platform 85

5.7.3 Experimental results 87

5.8 Chapter summary **89**

5.1 Introduction

In the previous chapter, we have presented a first version of P2PDC which is centralized with simplified functionalities. In this chapter, we present the decentralized version of P2PDC that includes new features aimed at making P2PDC more scalable and efficient [Cornea 2011]. Indeed, a hybrid resource manager manages peers efficiently and facilitates peers collection for computation; a hierarchical task allocation

mechanism accelerates task allocation to peers and avoids connection bottleneck at submitter. Furthermore, a file transfer functionality is implemented that allows to transfer files between peers. Moreover, some modifications to the communication operation set are introduced. Experiments for the obstacle problem are carried out on GRID'5000 platform with up to 256 peers.

This chapter is organized as follows. In the next section, we describe the hybrid resource manager and peer collection procedure for a computation. The section 5.3 deals with hierarchical task allocation. The section 5.4 presents the dynamic application repository. The implementation of file transfer functionality is detailed in the section 5.5. The section 5.6 presents new communication operations. The experiments for the obstacle problem on Grid'5000 are displayed and analyzed in the section 5.7. Finally, a summary of the decentralized version of P2PDC concludes this chapter.

5.2 Hybrid resource manager

In the centralized version of P2PDC (see Chapter 4), a server manages informations regarding peers and allocates peers to a task. This centralized architecture is not scalable since the topology server is overloaded when the number of peer increases. Furthermore, when the server fails, no task can be carried out. Thus, topology architecture of resource manager must be improved so that it becomes scalable, fault tolerant and it facilitates peers collection for computation.

In the literature, peer-to-peer topologies are designed most of the time for content sharing systems like Chord [Stoica 2003], Pastry [Rowstron 2001] or CAN [Ratnasamy 2001]. Thus, they are aimed at proposing an efficient object search algorithm with low cost in terms of query hop and messages. An object is usually identified by a key and keys are replicated in the overlay network. A query is matched when it reaches a peer having this key; the address of peer storing the object is then returned. Computational resource discovery is quite different. Computational resources are specified by peer characteristics such as CPU, memory, network bandwidth and so on. Hence, search query in P2P HPC applications may have some specific requirements about peer characteristics. The requirements may be exact (e.g. CPU speed equals to 3.0 GHz) or in range (e.g. having more than 2Gb of memory). The query will then return the address of α peers required to perform a given task. Moreover, we note that the latency is an important factor that influences the efficiency of a computation when using distributed iterative algorithms with frequent communications between peers. Thus, it is better for returned peers to be close to each others and to the submitter.

In the sequel, we propose a new resource manager for P2PDC that is based on a hybrid architecture. This hybrid architecture is simple but ensures the scalability, the fault tolerance and efficient peers collection for computation.

5.2.1 General topology architecture

Figure 5.1 illustrates the general topology architecture. It consists of a Server, Trackers and Peers.

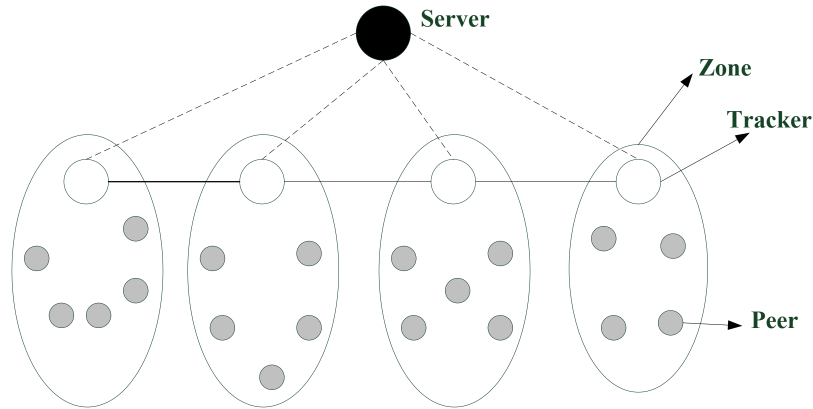


Figure 5.1: General topology architecture.

- Server manages informations regarding trackers connection/disconnection; it is the contact point of new nodes joining overlay network for the first time. When trackers or peers have no contact to join overlay network, they contact the server in order to receive a list of closest connected trackers, then they connect to trackers in the received list. The server can also store statistic information regarding connection/disconnection time, resources donated/consumed of all nodes in the overlay network.

- A tracker manages informations regarding a set of peers, called a zone. It collects statistical information regarding connection/disconnection time, resources donated/consumed of peers in its zone and periodically sends these data to server.

- Peers are donors of computational resources. Peers are grouped in zones and managed by the tracker of zone.

Trackers topology is a line, see Figure 5.2. Each tracker T_i maintains a set of closest trackers N_i. In order to get rid of the case where some trackers can be isolated, there are, in the set N_i, $|N_i|/2$ closest trackers having IP address greater than IP address of owner tracker and $|N_i|/2$ closest trackers having IP address smaller than IP address of owner tracker. Moreover, each tracker maintains connection with the closest tracker on right side and the closest tracker on left side.

In a zone, peers publish their information regarding processor, memory, hard disk and current usage state to tracker of zone and wait for works. Peers have to update periodically their usage state to tracker.

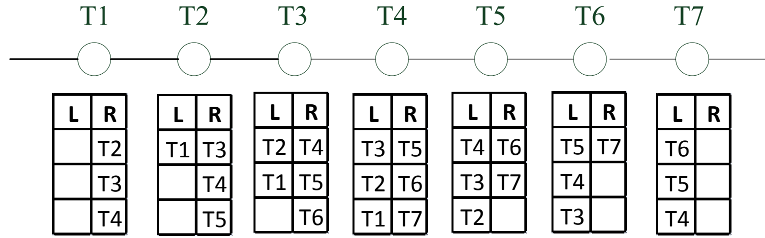


Figure 5.2: Trackers topology.

5.2.2 IP-based proximity metric

In the literature, there are several proximity metrics that can be used in order to calculate the proximity between peers in the network such as IP path length, AS path length, geographic distance, and measures related to Round Trip time (RTT) and so on (see [Huffaker 2002]). Each metric has its own advantages and weakness. We have chosen IP-based proximity metric because it makes use of local information (IP address) to calculate the proximity, hence it does not consume network resource and is faster than other metrics.

IP-based proximity metric [Zhao 2006] makes use of the longest common IP prefix length as the measure of proximity between peers. For example, in the case of 3 peers: P_1 having IP address 145.82.1.1, P_2 having IP address 145.82.1.129 and P_3 having address 145.83.56.74. The longest common prefix between P_1 and P_2 is 24 bits, while the longest common prefix between P_1 and P_3 is 15 bits. So P_1 considers that P_2 is closer than P_3.

5.2.3 Topology initialization

Initially, we suppose that the system has a server and some trackers managed by system administrator. These nodes are cores of the system and are on-line permanently. When the number of peers increases, system administrator chooses some trust volunteers peers to become trackers. Trackers are chosen based on on-line time, i.e. volunteers peers with largest on-line time will be chosen; moreover, trackers are chosen spearing on the IP range in order to ensure that the number of peers in a zone is balanced between zones. When P2PDC environment is downloaded and installed at a node, IP address of server and a list of trackers are set and stored in local memory. This tracker list will be updated when node joins to overlay network.

5.2.4 Tracker joins

When a new tracker connects to overlay network, it sends a join message to the closest tracker in tracker list stored in local memory. If this tracker does not answer, then it sends join message to next closest trackers in tracker list. In the case where all trackers in the tracker list do not answer, new tracker will contact the server; then the server sends to it a new tracker list. The tracker, when receiving a join message, calculates and compares the proximity between itself and new tracker with

proximity between trackers in its closest tracker set N and new tracker. If contacted tracker found in its set N a tracker that is closer to new tracker, then it transfers join message to this tracker. This step repeats until the closest tracker to new tracker is found in the overlay network. The closest tracker firstly informs all trackers in set N about new tracker. Secondly, it removes the farthest tracker along the same side as new tracker in the set N and adds new tracker to the set N. Others trackers in the set N of closest trackers must adjust their set N along the same way. The closest tracker sends also its set N to new tracker so that new tracker can build its own set N. Finally, new tracker establishes connections with two closest trackers along the two sides in his set N. Figure 5.3 shows state of trackers topology after new tracker T_8 has joined overlay network.

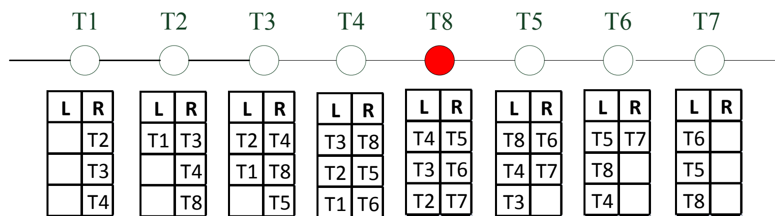


Figure 5.3: Trackers topology after a new tracker has joined.

5.2.5 Peer joins

When a new peer joins overlay network, it sends a join message to the closest tracker in tracker list stored in local memory; the message is transferred to the tracker which is closest to the new peer. The closest tracker adds this peer to its peer list and sends an accept message to new peer along with its neighbor set N_i. New peer updates its tracker list and sends to tracker of zone information regarding resources such as processor, memory, hard disk and current usage state. After joining a zone, peers have to update periodically their resources usage state to tracker. When tracker receives state update from a peer, it sends an answer message to this peer.

5.2.6 Tracker leaves

As a tracker maintains connection with the two closest trackers along the two sides in the set N_i, a tracker disconnection can be detected by direct neighbors when connection is broken. Suppose that tracker T_4 in Figure 5.2 crashes, its direct neighbors T_3 and T_5 detect disconnection of T_4. T_3 informs trackers along the left side of its set N_3 and the server about T_4 disconnection. T_3 sends also tracker list on the right side of its set N so that trackers on the left side of T_3 can rebuild their set N_i. These trackers then replace T_4 by the closest tracker that was received. Similarly, T_5 informs trackers on right side of its set N_5 and the server about T_4 disconnection and sends to them trackers on left side of its set N_5. Afterwards, T_3 establishes a connection with T_5 and the two trackers send to each other the farthest

trackers so that they can rebuild their set N_i. Figure 5.4 presents trackers topology after tracker T_4 has disconnected.

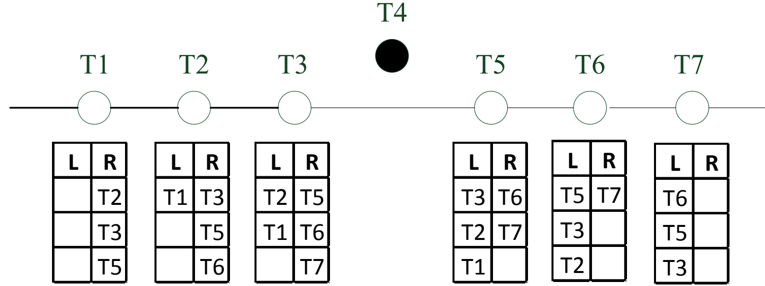


Figure 5.4: Trackers topology after a tracker has disconnected.

On the other hand, when a tracker disconnects, peers of this zone do not receive acknowledgment message in response to state update message. If peers do not receive acknowledgment message from tracker after a time T, then peers consider that this tracker is disconnected; then peers will send join message to closest tracker in their tracker list, i.e. they will join to neighbor zone.

5.2.7 Peer leaves

When a peer disconnects, tracker does not receive resources usage state update from this peer. If tracker does not receive state update of a peer after a time T, then tracker considers that this peer is disconnected.

We note that when the server disconnects, the system continues working; topology of trackers and peers are maintained; new trackers and new peers can join overlay network through their tracker list in local memory; Trackers store statistical information in local memory and send them to the server when the server comes back.

5.2.8 Peers collection

When a node, the so-called submitter, wants to submit a task, it has to join the overlay network firstly; i.e. it finds a closest tracker and joins this zone. Then the submitter sends peer request message to its tracker; this message contains information regarding computation like task description, number of peers needed initially, peers requirements; the tracker filters connected peers in its zone which satisfy requirements of the request and sends the address of these peers back to submitter. If number of peers collected by this tracker is not enough, then submitter requests peer from trackers in its local tracker list. If number of collected peers is not enough after having sent requests to all trackers in its local tracker list, then submitter requests more trackers address from the two farthest trackers on the two sides in its local tracker list. These two farthest trackers send to submitter trackers in their tracker list in other side with submitter. Then, submitter requests peers from new

received trackers. This step repeats until enough peers have been collected. Peers reserved for a computation are considered busy and cannot be reserved for another computation.

We note that with this peer collection algorithm, closest peers to the submitter are always collected. This reduces the latency between submitter and peers and between peers, ensuring an efficient computation.

This hybrid topology architecture is simple as compared with existing structured topology architectures like Chord [Stoica 2003] or Pastry [Rowstron 2001] but it is scalable, fault tolerant and efficient for both topology maintenance and peers collection. Each node is aware of a few others nodes: trackers are aware of peers in their zone and their neighbor set, peers are aware of their tracker and their local trackers list. The server manages all trackers but in an indirected manner, i.e. neighboring trackers monitor each others and only notifications about tracker joining or leaving are sent to server. When a tracker or a peer joins the overlay network, closest tracker finding may take, in the worst case, $\frac{|T|}{|N|/2}$ steps where $|T|$ is the number of trackers and $|N|$ is the size of neighbors set N. However, a node stores a list of closest trackers in its local memory that is updated over the time. Thus, a node joining the overlay network always contacts a tracker which is close. Peer leaving influences only its tracker, while tracker leaving influences peers in its zone and its neighbors. In particular, the cost of peers collection depends on the number of peers needed rather than the number of peers in the overlay network.

5.3 Hierarchical task allocation

When submitter has collected enough peers, it divides peers into groups based on proximity; in each group, a peer is chosen by submitter to become coordinator which will manage others peers in the group. The number of peers in a group cannot exceed $C_{\max}$ in order to ensure efficient management of coordinator. We have chosen $C_{\max} = 32$. Submitter sends peers list of a group to coordinator. Then, the coordinator connects to all peers in a group and sends a *reverse* message to peers. When a peer is reserved for a computation, it sends a message to its tracker to inform that it is not free any more. Figure 5.5 illustrates created peers graph.

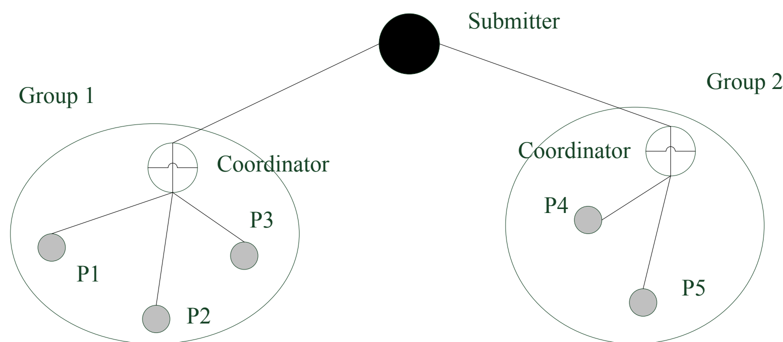


Figure 5.5: Allocation graph.

Submitter calls the *Task_Definition()* function where a given task is decomposed into subtasks. Afterward, submitter sends subtasks to groups coordinators. Subtasks are then sent by coordinators to peers. Subtasks results are sent in inverse direction, i.e. peers send their subtask result to coordinator, then coordinator transfers results to submitter.

We note that hierarchical task allocation has many advantages as compared with the case where there are not coordinator. Firstly, hierarchical task allocation is faster because submitter does not have to connect in succession to all peers in order to reserve peers and send subtasks; submitter has only to connect to coordinators and peer reservation and subtask sending are carried out in parallel by coordinators; moreover, peers grouping is based on proximity, hence communication between coordinator and peers is faster than directed communication between submitter and peers. Secondly, sending result to submitter via coordinators avoids bottleneck at submitter because if all peers would send results directly to submitter, then there could be a bottleneck at submitter.

5.4 Dynamic application repository

In the centralized version of P2PDC, applications are added manually to application repository and compiled at the same time with P2PDC. When users want to add a new application to P2PDC, they have to recompile P2PDC as well as redeploy P2PDC on every machines. This takes time and efforts of users. Thus, we have implemented a dynamic application repository in order to overcome this weakness. Then, applications of P2PDC are compiled independently with P2PDC as dynamic libraries, i.e. file .so in Linux or file .dll in Windows with file name being the application name. The library files are stored at a specific place.

When a task is submitted with an application name at a given submitter, the application repository will check if there is a library file having the same name as the application name in the the specific place. If this library exists, then the application repository will load this library, extract three principal functions and return pointers of those functions to Task manager. If this library file does not exist, then an error message will be returned to user.

At peer side, when a peer receives a subtask with an application name, the application manager finds and loads the library file from the application repository in a way similar to what is done at the submitter. However, if this library file does not exist in the repository, then the application manager downloads the library file from the submitter or from the coordinator via file transfer component (see section 5.5).

5.5 File transfer

File transfer system is responsible of application library files transfer as well as transfer of task input data files and result files between peers. Application library

files are transfered automatically from submitter to workers. On what concerns task input data, programmers can choose between two ways. In the first way, programmers read input data from file and set them as task parameters in the *Task_definition()* function. Those parameters will be sent along with subtasks to peers. In the second way, programmers set the input data file path as a task parameter in the *Task_definition()* function. Then, input data file will be transfered automatically to workers. Although programmers also have to read the data input file in *Calculate()* function, the second way has advantage as compared with the first one to avoid memory leak in very large application. Similarly, sending results via file transfer component instead of setting results as subtask parameters is also a solution to avoid memory leak.

Files transfered from submitter to workers are divided into two types. Common files like application library files and input data file need to be sent to all workers. Private files like private subtask input data file need to be sent to only one subtask. Private files are transfered directly from submitter to workers. Whereas, common files are transfered via the hierarchical allocation architecture, i.e. common files are transfered first from submitter to coordinators and then from coordinators to workers. The transfer of common file following the hierarchical architecture avoids the bottleneck at submitter and then is much faster than the direct transfer from submitter to workers. For example, in the case of 120 workers divided into 4 groups(4 coordinators), transfer of common file via the hierarchical architecture of common file is about 4 times faster than the direct transfer from submitter to workers.

5.6 New communication operations

In the centralized version of P2PDC, there are only 2 communication operations: *P2P_Send* and *P2P_Receive*. The communication mode is decided by P2PSAP protocol according to context, e.g. topology at the network layer or computational scheme at application layer. But some special messages need to be exchanged in a reliable mode like messages for termination detection, termination propagation, etc. Therefore, we have divided messages into 2 types: data message and control message. While data messages are used to exchange updates between peers after each relaxation, control messages are used for computation state exchange like data related to local termination criteria, termination command. Communication mode for data message is chosen according to the context by P2PSAP; while communication mode for control message is always asynchronous and reliable using control channel of P2PSAP. A *flags* parameter is added to 2 communication operations to distinguish 2 types of messages: *CTRL_FLAG* indicates control message and *DATA_FLAG* indicates data message. The prototype of the two communication operations now becomes:

*int P2P_Send(P2PSubtask *pSubtask, uint32_t dest, char *buffer, size_t size, int flag)*

*int P2P_Receive(P2PSubtask *pSubtask, uint32_t dest, char *buffer, size_t*

size, int flag)

Moreover, we have added a new operation *P2P_ Wait* that waits for a message from another peer. This new operation facilitates implementation of some asynchronous schemes and termination algorithm.

int P2P_ Wait(P2PSubtask pSubtask, uint32_ t *iSubtaskRank, int *flags)*

The use of control messages and operation *P2P_ Wait* will be detailed in subsection 5.7.1.

5.7 Computational experiments

In this section, we concentrate on the decomposition of the obstacle problem. We consider mainly a 3D obstacle problem with size $256 \times 256 \times 256$. We propose a decomposition that permits one to improve the efficiency of distributed algorithms when a large number of peers is used. Experimental results with P2PDC on the Grid'5000 platform [gri] with up to 256 workers are displayed and analyzed.

5.7.1 New approach to the distributed solution of the obstacle problem

The decentralized version of P2PDC aims at using hundreds of peers distributed over several clusters. However, the distributed algorithm described in the previous chapter may not scale well with large number of machines in peer-to-peer context. Hence, we have introduced a new problem decomposition and use a different termination method.

5.7.1.1 New decomposition of the obstacle problem

In the previous chapter, the iterate vector of the 3D obstacle problem $n \times n \times n$ was decomposed into n sub-blocks of size $n \times n$; sub-blocks are then assigned to α workers. This decomposition is called a *slice decomposition*. The worker P_k (excluding the first and the last worker) has then to send a message of size n^2 to worker P_{k-1} and a message of size n^2 to worker P_{k+1} after each relaxation. The workers P_1 and P_α have to send respectively only a message to workers P_2 and $P_{\alpha-1}$, respectively. When the number of workers increases, the computational load of workers decreases; whereas the total size of messages a typical worker has to send to other workers after each relaxation remains unchanged: $\sum S_{ms} = 2 \times n^2$. Therefore, the algorithm efficiency can deteriorate greatly.

In order to reduce the size of messages exchanged between workers after each relaxation, we have proposed the following decomposition. The iterate vector of the 3D obstacle problem is decomposed into $n \times n$ sub-blocks of size n. The sub-blocks are then assigned to α workers according to two axes, i.e. $n \times n$ sub-blocks are assigned to $p \times q$ workers, each worker is assigned $m \times k$ sub-blocks, where $p \times q = \alpha$ and $p \times m = q \times k = n$. This decomposition is called *pillar decomposition*. Figure

5.6 illustrates the *pillar decomposition* of the iterate vector in the case where $n = 32$, $p = 2$ and $q = 4$ ($\alpha = 8$).

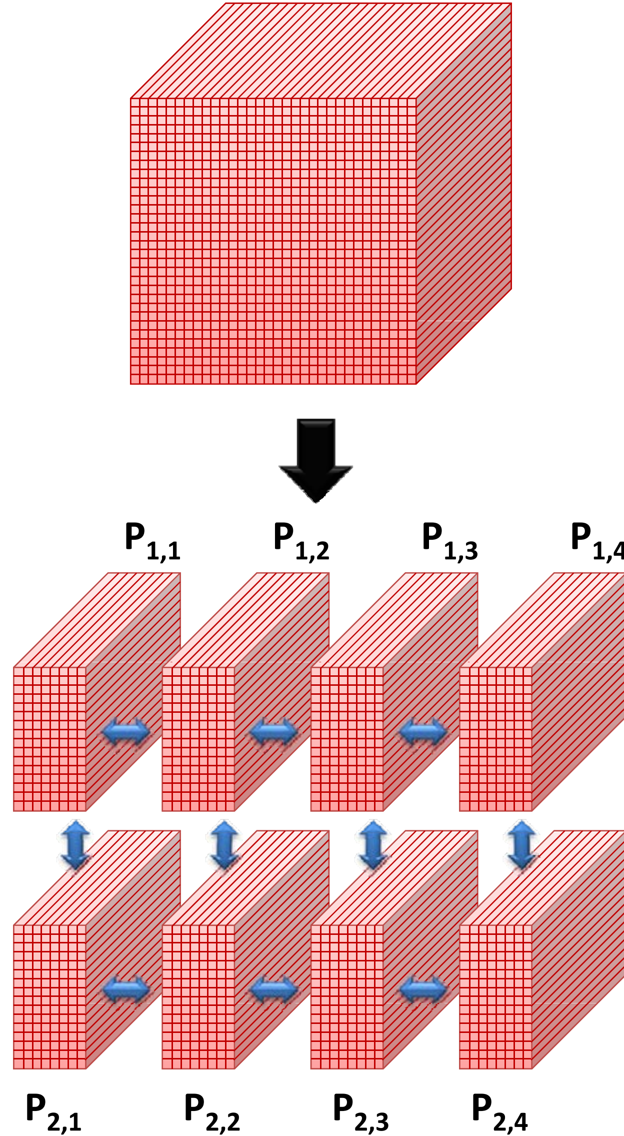


Figure 5.6: Pillar decomposition of the 3D obstacle problem.

Then, the message exchange topology is a grid where a typical worker (excluding workers on the boundary of the grid) has to send 4 messages after each relaxation (4 workers at 4 corners of the grid send two messages, others workers on the boundary of the grid send three messages). Thus, the total size of messages a typical worker has to send after each relaxation is:

$$\sum S_{ms} = 2 \times (m + k) \times n = 2 \times (\frac{n}{p} + \frac{n}{q}) \times n = (\frac{1}{p} + \frac{1}{q}) \times 2 \times n^2.$$

Since $\frac{1}{p} + \frac{1}{q} \leq 1, \forall p, q \geq 2$, $\sum S_{ms}$ with *pillar decomposition* is smaller than with *slice decomposition*. Moreover, when the number of peers increases, p and q will increase, then $\frac{1}{p} + \frac{1}{q}$ decreases. Thus, $\sum S_{ms}$ decreases when the number of peers increases. However, with *pillar decomposition*, a worker has to exchange messages with more workers than with *slice decomposition*. This leads to the enlargement of synchronization time in case of synchronous computational scheme. That is the reason why we do not decompose the iterate vector into points and assign points to workers according to all three axes.

For example, in the case where there are 64 workers and the problem size is $256 \times 256 \times 256$, the total size of messages a worker has to send after each relaxation with *slice decomposition* is $\sum S_{ms} = 2 \times 256^2$. If the problem is decomposed according to *pillar decomposition* with $p = 8$ and $q = 8$, then each worker is assigned 32×32 sub-blocks of size 256; the total size of messages a worker has to send after each relaxation is $\sum S_{ms} = (\frac{1}{8} + \frac{1}{8}) \times 2 \times 256^2 = \frac{1}{4} \times 2 \times 256^2$. Thus, $\sum S_{ms}$ with *pillar decomposition* is four time smaller than with *slice decomposition*.

Figure 5.7 displays the basic computational procedure with pillar decomposition at node $P_{r,c}$ which is at row r and column c and which is not on the boundary of the grid (the topology of update exchange between workers).

The node $P_{r,c}$ updates the sub-blocks of components of the iterate vector denoted by $U_{i,j}, f(r) \leq i \leq l(r), f(c) \leq j \leq l(c)$, where $f(r)$ and $l(r)$ stands for the first and the last sub-block row of the node $P_{r,c}$ and $f(c)$ and $l(c)$ stands for the first and the last sub-block column of the node $P_{r,c}$.

5.7.1.2 Termination

According to the change from *slice decomposition* to *pillar decomposition*, the termination detection is modified as follows. Token *tok_conv* is appended to updates from node $P_{r,c}$ to two nodes $P_{r+1,c}$ and $P_{r,c+1}$. Moreover, with the presence of control message (see section 5.6), token *tok_term* is not appended to updates but is sent as control messages from a given node $P_{r,c}$ to nodes $P_{r-1,c}$ and $P_{r,c-1}$ (see Figure 5.8). The reliability of control messages avoids loss of token *tok_term* in asynchronous and hybrid cases.

Furthermore, we have noticed that the termination described above is not efficient for asynchronous iterative algorithms in the case where a large number of peers is used and the architecture is heterogeneous. Thus, we have implemented a different termination method for the obstacle problem in asynchronous computational scheme that detects exactly the termination and reduces unnecessary relaxations. This termination method has been proposed in [El Baz 1998]; it is a variant of the termination method of Bersekas and Tsitsiklis [Bertsekas 1989, Bertsekas 1991]. This method is based on activity graph and acknowledgement of messages.

The behavior of workers implementing asynchronous iterative algorithms is presented by the finite state machine in Figure 5.9 where each worker can have three states: active (A), inactive (I) and terminal (T).

Initially, only the worker $P_{1,1}$ is active. This worker is call the root and is

1: **repeat**
2:     **if** r is even **then**
3:         **send** $U_{l(r),[f(c),\dots,l(c)]}$ to node $P_{r+1,c}$
4:         **receive** $U_{l(r)+1,[f(c),\dots,l(c)]}$ from node $P_{r+1,c}$
5:         **send** $U_{f(r),[f(c),\dots,l(c)]}$ to node $P_{r-1,c}$
6:         **receive** $U_{f(r)-1,[f(c),\dots,l(c)]}$ from node $P_{r-1,c}$
7:     **else**
8:         **receive** $U_{f(r)-1,[f(c),\dots,l(c)]}$ from node $P_{r-1,c}$
9:         **send** $U_{f(r),[f(c),\dots,l(c)]}$ to node $P_{r-1,c}$
10:         **receive** $U_{l(r)+1,[f(c),\dots,l(c)]}$ from node $P_{r+1,c}$
11:         **send** $U_{l(r),[f(c),\dots,l(c)]}$ to node $P_{r+1,c}$
12:     **end if**

13:     **if** c is even **then**
14:         **send** $U_{[f(r),\dots,l(r)],l(c)}$ to node $P_{r,c+1}$
15:         **receive** $U_{[f(r),\dots,l(r)],l(c)+1}$ to node $P_{r,c+1}$
16:         **send** $U_{[f(r),\dots,l(r)],f(c)}$ to node $P_{r,c-1}$
17:         **receive** $U_{[f(r),\dots,l(r)],f(c)-1}$ to node $P_{r,c-1}$
18:     **else**
19:         **receive** $U_{[f(r),\dots,l(r)],f(c)-1}$ to node $P_{r,c-1}$
20:         **send** $U_{[f(r),\dots,l(r)],f(c)}$ to node $P_{r,c-1}$
21:         **receive** $U_{[f(r),\dots,l(r)],l(c)+1}$ to node $P_{r,c+1}$
22:         **send** $U_{[f(r),\dots,l(r)],l(c)}$ to node $P_{r,c+1}$
23:     **end if**

24:     **for** $i = f(r) \to l(r)$ **do**
25:         **for** $j = f(c) \to l(c)$ **do**
26:             $U_{i,j} \leftarrow F_{i,j,\delta}(U_{i-1,j}, U_{i,j-1}, U_i, U_{i+1,j}, U_{i,j+1})$
27:         **end for**
28:     **end for**
29: **until** convergence

Figure 5.7: Basic computational procedure at node $P_{r,c}$ with pillar decomposition.

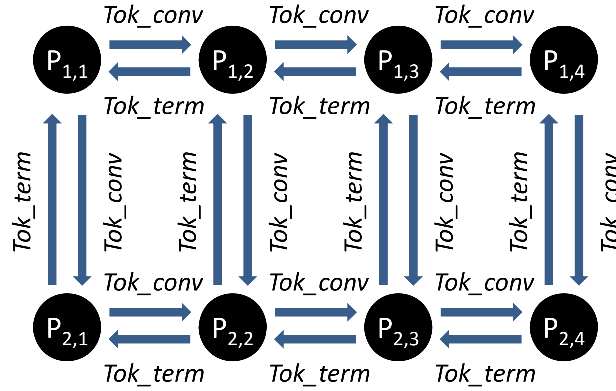


Figure 5.8: Termination detection in the case of pillar decomposition.

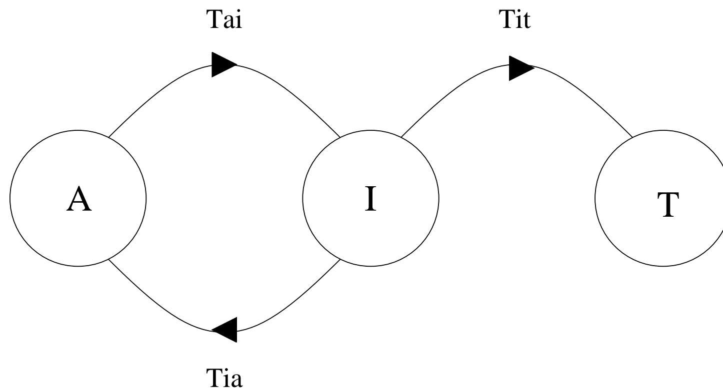


Figure 5.9: Behavior of workers implementing new termination method.

denoted R. All others workers are inactive. We denote in the sequel T, a tree of root R covering worker set.

Four types of messages may be issued by each worker:

- Updates of sub-blocks.
- Activate messages.
- Inactivate messages.
- Termination messages.

The first message type is data message. Three others message types are control messages.

Each worker $P_{r,c}$ has to store following additional data:

- The identity of the worker that has activated $P_{r,c}$ (which is also called parent of $P_{r,c}$).

- The list of workers activated by $P_{r,c}$ (which are also called children of $P_{r,c}$)

State A. In active state, a worker $P_{r,c}$ evaluates the local termination test. If the local termination test is satisfied, then $P_{r,c}$ does not execute update; otherwise, $P_{r,c}$ updates components of sub-blocks assigned to it and sends updates to adjacent workers. If $P_{r,c}$ receives an activate message from a worker $P_{r',c'}$, then $P_{r,c}$ adds $P_{r',c'}$ to its list of children. If $P_{r,c}$ receives an inactivate message from a worker $P_{r',c'}$, then $P_{r,c}$ removes $P_{r',c'}$ from its list of children.

State I. In inactive state, a worker is waiting for messages (using *P2P_ Wait* operation).

State T. In terminal state, the computation has been terminated, workers do nothing.

Transition Tia. An inactive worker $P_{r,c}$ becomes active when it receives a new update from an adjacent worker $P_{r',c'}$; then the worker $P_{r,c}$ sends an active message to $P_{r',c'}$ and $P_{r',c'}$ becomes parent of $P_{r,c}$.

Transition Tai. An active worker becomes inactive if its list of children is empty and its local termination test is satisfied; then the worker sends an inactive message to its parent.

Transition Tit. The root worker R changes immediately from inactive state to terminal state. Termination messages then are sent to adjacent workers in the tree T recovering workers. A worker $P_{r,c}$ different from R changes from inactive state to terminal state when it receives a termination message from an adjacent lower level node in the tree T. $P_{r,c}$ then sends termination messages to adjacent upper level nodes in the tree T.

The behavior of this method can be summarized as follows: initially, only the root worker R ($P_{1,1}$) is active and all other workers are inactive. All other workers become progressively active upon the receipt of an update from another worker. An activity graph is created; the topology of the graph changes progressively as the various messages are received and the local termination tests are satisfied. Figure 5.10 presents an example of the evolution of activity graph in the case of 8 workers.

5.7.2 Grid'5000 platform

Computational experiments have been carried out on the Grid'5000 platform [gri]. The French grid platform is composed presently of 2970 processors with a total of 6906 cores distributed over 9 sites in France. All of them have at least a Gigabyte Ethernet network for local machines. Nodes between the different sites range from 2.5 Gflops up to 10 Gflops. Sites of Grid'5000 have several clusters with different performances.

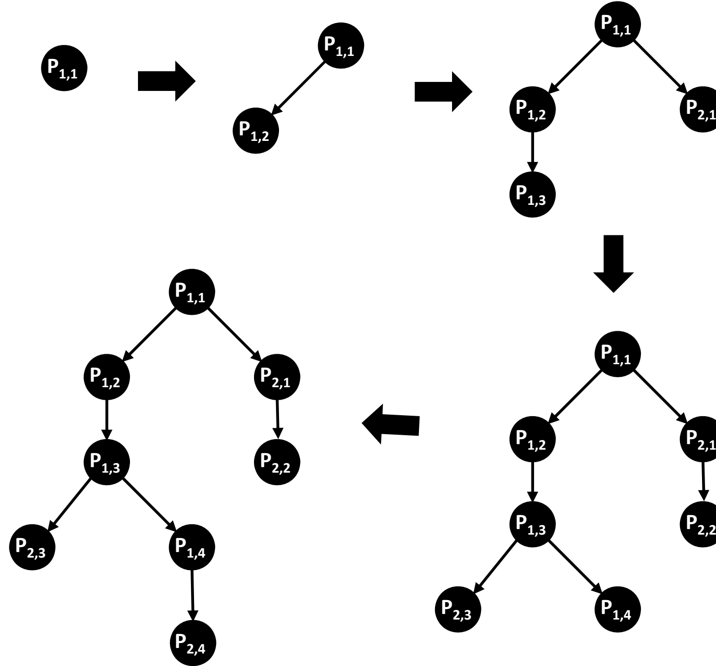


Figure 5.10: Evolution of the activity graph.

We have used machines over 8 clusters of 5 sites of the Grid'5000 testbed. Machine characteristics on each cluster we have used and corresponding sequential computational time are presented in Table 5.1 for the obstacle problem with size $256 \times 256 \times 256$.

Table 5.1: Machine characteristics and sequential computational time

Site	Cluster	Processor	Memory	Seq time
Lyon	Sagittaire	AMD 2.4 GHz	2 Gb	32166 s
	Capricorne	AMD 2.0 GHz	2 Gb	33942 s
Sophia	Helios	AMD 2.2 GHz	4 Gb	33178 s
	Sol	AMD 2.6 GHz	4 Gb	29400 s
Toulouse	Pastel	AMD 2.6 GHz	8 Gb	27843 s
Nancy	Grelon	Intel Xeon 1.6 GHz	2 Gb	32476 s
Orsay	Gdx	AMD 2.0/2.4 GHz	2 Gb	34636 s
	Netgdx	AMD 2.0	2 Gb	34711 s

The topology server is placed at the site of Toulouse. At each site, a tracker is launched in order to manage peers of the site. The submitter is a machine of the

cluster Sagittaire at Lyon.

5.7.3 Experimental results

Experiments have been carried out in the following contexts.

- Case 1: The slice decomposition and termination method presented in subsection 4.5.2 are used, computations are carried out on the cluster Gdx at Orsay with up to 128 workers.
- Case 2: The pillar decomposition and termination method presented in subsection 5.7.1.2 are used, computations are carried out on the cluster Gdx at Orsay with up to 128 workers.
- Case 3: The pillar decomposition and termination method presented in subsection 5.7.1.2 are used, computations are carried out on several clusters with up to 256 workers. In the cases where the number of nodes is less than 256 workers, computations are carried out on 4 clusters at 4 locations: cluster Pastel at Toulouse, cluster Sagittaire at Lyon, cluster Grelon at Nancy and cluster Gdx at Orsay. For each experiment, an equal number of nodes is used on each site. For example in experiment with 8 nodes, 2 nodes at Toulouse, 2 nodes at Orsay, 2 nodes at Nancy and 2 nodes at Lyon, respectively, are used. In the case where the number of nodes is 256, nodes of others clusters are used.

The efficiency of cases 1 and 2 are presented in Figure 5.11. We can see, in Figure 5.11, that the efficiency deteriorates more rapidly in the case 1 than in the case 2 for both synchronous and asynchronous computational schemes. This is due to the fact that, when the number of workers increases, the problem decomposition in the case 2 reduces the total size of messages sent by a worker after each relaxation while the total size of messages sent by a worker after each relaxation remains unchanged in the case 1.

Figure 5.12 displays the number of relaxations in function of number of workers for asynchronous algorithm. We note that the number of relaxations in the the case 2 is lower than the number of relaxations in the case 1. This is due to the fact that with the termination method in the case 2, a worker does not execute update if the local termination test is satisfied; whereas, with the termination method in the case 1, a worker still executes update when the local termination is satisfied.

Computational results are presented in the figure 5.13 for the case 3. We note that the results are computed by using sequential computational time on the most performant cluster, i.e. cluster Pastel at Toulouse. As compared with Figure 5.11, we note that the efficiency of synchronous algorithms deteriorates more rapidly in the case 3 than in the case 2. This is due to the fact that machines are distributed over 4 sites and the latency between clusters (from 11,5 ms to 18,9 ms) is greater than the latency inside a cluster (about 0,1 ms) in the case 3,. Thus the synchronization time is greater in the case 3 than in the case 2 for synchronous schemes. Moreover, the

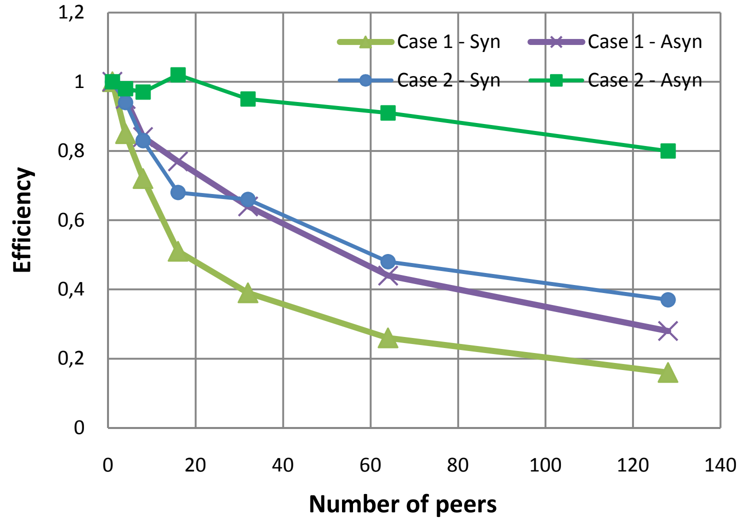


Figure 5.11: Efficiency of distributed algorithms in the cases 1 and 2.

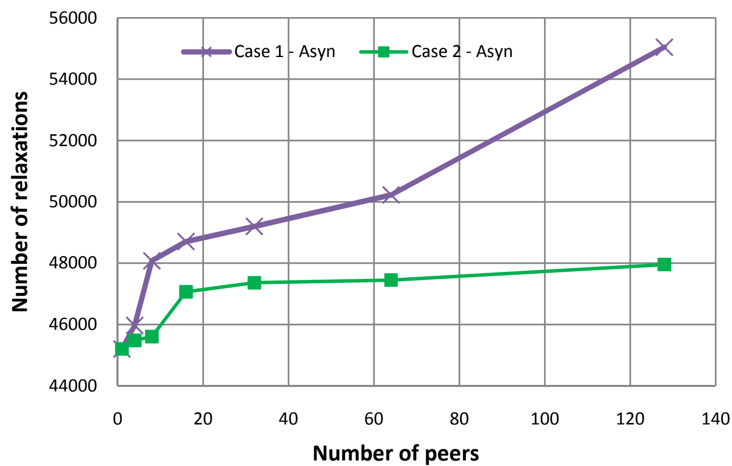


Figure 5.12: Number of relaxations of asynchronous iterative algorithms in the cases 1 and 2.

architecture is heterogeneous. In the synchronous case, faster workers have to wait for slower worker through messages exchanges; whereas, the results are computed by using the sequential computational time on the most performant cluster. In the asynchronous scheme, there is not much difference between the case 2 and 3. This means that the asynchronous scheme is less sensitive to latency increase and more appropriate for computations in interconnected clusters context than synchronous schemes. The efficiency of hybrid schemes of computation is situated in between efficiencies of synchronous and asynchronous schemes.

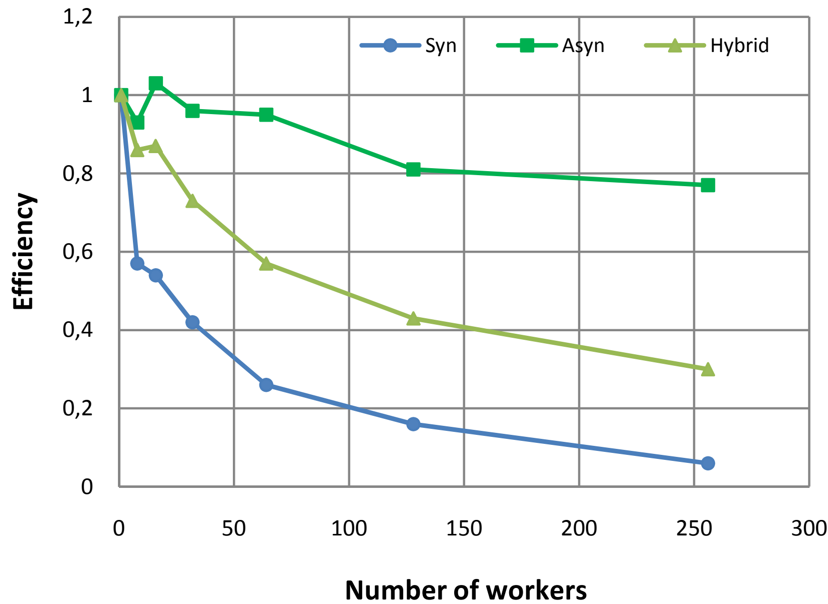


Figure 5.13: Efficiency of distributed algorithms in the case 3

5.8 Chapter summary

In this chapter, we have presented the decentralized version of P2PDC that includes new features aimed at making P2PDC more scalable and efficient. Indeed, the resources manager is based on a hybrid topology that is simple but efficient and facilitates peers collection for computation. The hierarchical task allocation mechanism accelerates task allocation to peers and avoids connection bottleneck at submitter. Furthermore, a file transfer functionality is implemented that allows to transfer files between peers. Moreover, the communication operation set has been extended in order to facilitate the implementation of some asynchronous algorithms and termination detection, in particular for evolution problems.

Experiments for the obstacle problem have been carried out on GRID'5000 platform with up to 256 peers. A *pillar decomposition* has been proposed that reduces the total size of messages sent by a worker after each relaxation as compared with *slice decomposition* presented in previous chapter. A different termination method has been implemented for asynchronous iterative schemes that detects exactly the termination and reduces unnecessary relaxations. Computational results show that the *pillar decomposition* improves significantly the efficiency of computations, e.g. in the case of 128 machines at Orsay, the efficiency of distributed algorithm with *pillar decomposition* is about twice as much as with *slice decomposition* in both synchronous and asynchronous schemes. Moreover, we have obtained a good efficiency for asynchronous iterations (0.78) in the case where up to 256 machines distributed over 8 clusters at 5 sites are used. This shows the interest of combining asynchronous schemes of computation with the decentralized environment P2PDC.

In the next chapter, we shall consider fault-tolerance functionalities of P2PDC that ensure the robustness of the computation in the case of peer failures.

Chapter 6

Fault-tolerance in P2PDC

Contents

6.1 Introduction **91**

6.2 State of the art in fault-tolerance techniques **92**

6.2.1 Replication techniques 92

6.2.2 Rollback-recovery techniques 93

6.3 Choices of fault-tolerance mechanisms **96**

6.4 Worker failure **96**

6.4.1 Coordinated checkpointing rollback-recovery for synchronous iterative schemes 97

6.4.2 Uncoordinated checkpointing rollback-recovery for asynchronous iterative schemes 99

6.5 Coordinator failure **100**

6.6 Computational experiments **102**

6.6.1 Coordinator replication overhead 102

6.6.2 Worker checkpointing and recovery overhead 102

6.6.3 Influence of worker failures on computational time 103

6.7 Chapter summary **104**

6.1 Introduction

Peer volatility is one of the great challenges of peer-to-peer applications and more particular for peer-to-peer High Performance Computing (HPC) applications. In peer-to-peer networks, peers may join and leave the network at unpredictable rate. If a peer executing a subtask of a given task, e.g. the solution of a numerical simulation problem leaves the network, then the task may not terminate or may produce wrong results. Thus, an effective mechanism of fault-tolerance is vital for ensuring the robustness of the application.

In previous chapter, we have presented the decentralized version of P2PDC that includes features aimed at making P2PDC more scalable and robust. In this chapter, we present the fault-tolerance mechanisms in P2PDC to cope with peer volatility in peer-to-peer networks. The fault-tolerance mechanisms can adapt itself according to peer role and computational scheme. Experiments on Grid'5000 platform show that the fault-tolerance mechanisms in P2PDC have small overhead and fast recovery.

Moreover, the impact of procedures that ensure robustness on computational time is small.

This chapter is organized as follows. Next section presents existing fault-tolerance techniques for parallel and distributed systems. Section 6.3 deals with the choice of fault-tolerance mechanisms in P2PDC. Section 6.4 aims at describing precisely the fault-tolerance mechanisms for worker failure, while the one for coordinator failure is detailed in the section 6.5. In section 6.6, experimental results for the obstacle problem on Grid'5000 platform are displayed and analyzed in the case of peer failure. Finally, a summary of fault-tolerance mechanisms in P2PDC is presented.

6.2 State of the art in fault-tolerance techniques

In the literature, many fault-tolerance techniques have been proposed for parallel and distributed systems. One can classify them into two main classes: replication and rollback-recovery [Treaster 2005, Sathya 2010, Arlat 2006]. While replication techniques use resource redundancy for masking the failure, rollback-recovery techniques consist in restoring the process of a failed node on another node. In the sequel, we shall detail these techniques and study their features and limitations.

6.2.1 Replication techniques

In replication techniques [Treaster 2005, Felber 1999, Arlat 2006], each process is replicated on two or more processors. A replicated process is called a replica. Replicas of a process must be coordinated in the way they give the illusion of a single logical process. If some of replicas fail, then the others replicas continue to process application. There are generally three replication strategies: *passive*, *active* and *semi-active* replication.

In *passive replication*, only a primary replica processes application, i.e. handles all incoming messages, updates its internal state and sends output messages. Others replicas are backup of the primary replica (see Figure 6.1). The primary replica regularly creates a checkpoint of its internal state. The checkpoint is either stored on a stable memory accessible by backup replicas, which are in idle state as the the primary replica is working (cold passive replication) or sent to backup replicas, which update their internal state from received checkpoint (warm passive replication). When the primary replica fails, a backup replica is elected to take its place. Since state of new primary replica is created from a checkpoint of the failed primary replica, the new primary replica may have to re-execute some operations that the failed primary replica had already done.

In *active replication*, all replicas process application, i.e. each process handles all incoming messages, updates its internal state independently, and generates output messages (see Figure 6.2). The effective output messages are selected using a decision function which depends on the assumption on the process failure. For

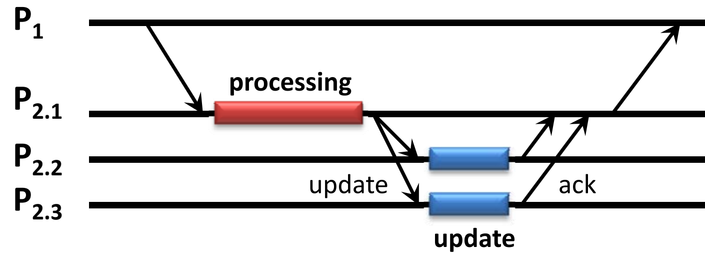


Figure 6.1: Passive replication.

simple cases, the decision function may be to select the first message available. Active replication can also overcome the arbitrary failures using a decision function by majority vote. During fault-free execution, active replication has more overhead than passive replication because active replication has to carry out a vote algorithm between replicas each time a decision is needed, e.g. choosing the effective output message. However, if some of replicas fail, then the recovery in active replication is faster than in passive replication.

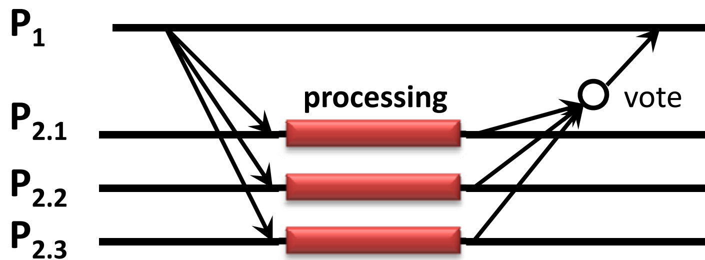


Figure 6.2: Active replication.

Semi-active replication is proposed to take advantages of both passive replication and active replication. Semi-active replication is similar to active replication in the sense that all replicas receive input messages and can treat them. However, as in passive replication, a privileged replica is responsible for certain decisions, e.g. message acceptance or refusal. The privileged replica can impose its decisions on other replicas without resorting to a vote. Optionally, the privileged replica can have also the responsibility of sending the output messages (see Figure 6.3).

Replication techniques are used in many systems like SETI@HOME [set], Condor [Litzkow 1988] or P2P-MPI [Genaud 2009].

6.2.2 Rollback-recovery techniques

Rollback-recovery techniques [Elnozahy 2002, Arlat 2006] assumes that application processes have access to some kind of stable storage that always survives even if some processes have failed. During the execution, application processes save to this stable storage a snapshot of their state, called *checkpoint*. Upon a process failure, the

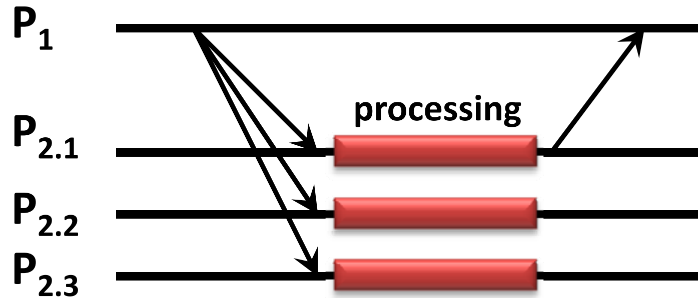


Figure 6.3: Semi-active replication.

failed process uses the checkpoint on the stable storage to restart the computation from an immediate state. Hence, the amount of lost computation is reduced. We can classify rollback-recovery techniques into two categories: *checkpoint-based* and *log-based*.

6.2.2.1 Checkpoint-based rollback-recovery

The *checkpoint-based rollback-recovery* consists in taking a snapshot of the entire system state regularly. Upon a failure, the system is restored to the most recent snapshot. The checkpoint-based rollback-recovery can be classified into three subcategories: uncoordinated checkpointing, coordinated checkpointing and communication-induced checkpointing.

- *Uncoordinated checkpointing* allows processes to take checkpoints independently. Each process may take a checkpoint when it is most convenient, thereby avoiding the synchronization complexity. However, this approach has several drawbacks in the cases where consistent global state is needed. Firstly, a processes may take useless checkpoints that are not a part of a consistent global state. Secondly, uncoordinated checkpointing may result in a potentially significant additional costs for seeking a consistent recovery line in an eventual recovery. Thirdly, uncoordinated checkpointing may lead to *domino effect*, where processes rollback indefinitely through the computation history in order to reach a consistent recovery line, resulting in the loss of large amounts of computation.

- *Coordinated checkpointing* ensures that whenever processes take checkpoints, a consistent global checkpoint is created. This requires the synchronization between processes, thereby increasing the overhead of checkpointing. But in exchange, the recovery is simplified and is not susceptible to domino effect. This is due to the fact that upon a failure, every processes rollback to their most recent checkpoint, which is always a part of the most recent consistent global checkpoint.

- *Communication-induced checkpointing* is a compromise between the two approaches. Each process can independently take checkpoints as in uncoordinated checkpointing. However, in order to avoid the domino effect, processes are forced to take checkpoints that generate a global checkpoint. Messages exchanged between processes contain extra information that allows the recipient to determine whether it should take a forced checkpoint.

6.2.2.2 Log-based rollback-recovery

In *log-based rollback-recovery*, in addition to process checkpointing, all messages received by processes are logged in a stable storage. Upon a failure, only failed process restores to precedent checkpoint and uses messages logged in the stable storage in order to perform the same computation as in initial execution. Thus, the failed process can recover to the state before the failure occurred. An orphan process is a process whose state depends on a message that was not logged to stable storage; thus this process cannot be reproduced during recovery. Log-based rollback-recovery protocols need to ensure that upon recovery of all failed processes, the system does not contain any orphan process. There are three classes of log-based rollback-recovery protocols:

- *Pessimistic logging* protocols log a given message received by a process to the stable storage before it affects the computation. Pessimistic logging protocols ensure that orphan processes are never created upon a failure. Thus the recovery upon a failure is simplified, processes that do not fail do not need to take any special actions. Moreover, garbage collection is simple, i.e. checkpoints and messages that are older than the most recent checkpoint can be discarded because they will never be used for recovery.

- *Optimistic logging* protocols log received messages to a volatile storage which is periodically flushed to stable storage. Optimistic logging protocols reduce the overhead during fault-free execution because applications are not required to be blocked while waiting for messages to be written to disk. However, since messages logged in the volatile storage will be lost when a failure occurs, some processes may become orphan processes. Thus, recovery upon a failure in optimistic logging is more complicated than in pessimistic logging because orphan processes have to rollback to state that does not depend on any lost messages.

- *Causal logging* protocols combine the advantage of both optimistic and pessimistic approaches. Like optimistic logging, causal logging protocols avoid synchronous access to stable storage except during output commit. Like pessimistic logging, causal logging protocols allow each process to commit output independently and never creates orphans, thereby isolating each process from the effects of failures that occur in other processes. However, these protocols require more complex recovery protocol.

One can find rollback-recovery techniques in many systems like BOINC [Anderson 2004], XtremWeb [xtr] or Vishwa [Reddy 2006].

6.3 Choices of fault-tolerance mechanisms

In P2PDC, peers can have different roles: coordinator or worker. Moreover, computations can be done via different computational schemes: synchronous, asynchronous. Therefore, fault-tolerance mechanism has to adapt to all peer roles and computational schemes. In the sequel, we will detail our choices of fault-tolerance strategies for each peer role and computational scheme.

In our opinion, replication strategy is not appropriate to workers for HPC applications because the number of peers involved in the computation enlarges but the computational capacity does not increase; furthermore, when communication between peers is frequent like with iterative methods, a protocol ensuring coherence between replicas will have great overhead. Log-based rollback-recovery seems also not appropriate for iterative algorithms with frequent communications between peers since communication logging will use a great volume of storage. Thus, we have chosen to deploy the checkpoint-based rollback-recovery mechanism in order to cope with worker failure. This mechanism can self-adapt to different computational schemes. A synchronous scheme needs the synchronization of all workers after each iteration, i.e a global state of computation must be reached before computation can continue. Hence, coordinated checkpointing is appropriate to this case. While in asynchronous schemes, each worker can work at its own pace. Moreover, asynchronous schemes allow message lost. Thus, uncoordinated checkpointing is appropriate to asynchronous schemes. So far, we have implemented the customized checkpointing where programmers define what data should be placed into checkpoint and how to recover from a checkpoint. Since storing checkpoints in a reliable storage may become a bottleneck, it is better that checkpoints are distributed on several locations on the network. Thus, we have modified the coordinator so that when users choose to deploy fault tolerant functionality, the coordinator does not calculate any subtask but stores checkpoints of peers in its group.

In order to cope with coordinator failure, we have chosen a replication strategy because the number of coordinators is small as compared with the number of workers and coordinators do not compute any subtask in our approach.

In the following sections, we shall present in detail our adaptive fault-tolerant mechanism.

6.4 Worker failure

In a group, workers periodically send heartbeat messages to their coordinator to inform that they are still alive. If a coordinator does not receive the heartbeat message from a worker within a time T, then the coordinator considers that this worker has failed.

In order to enable fault-tolerant functionality of workers, programmers have to call *P2P_checkpoint* function in the code. All application data that need to be placed into the checkpoint should be set as parameters of the function. For instance, in the solution of a numerical simulation problem solved via distributed iterative methods, values of the iterate vectors need to be placed into the checkpoint. In addition, when the user starts the submitter, he has to add fault-tolerance option to command line (see Appendix B); otherwise, *P2P_checkpoint* function will take no effect. When fault-tolerance option is added, all peers participating to the computation prepare specific data for checkpointing/recovery process. Coordinators store a copy of each received subtask so that if a subtask crashes before the first checkpoint is taken, then the coordinator will recover crashed subtask from the initial state. In the sequel, we will present in detail checkpoint-based rollback-recovery process for different computational schemes.

6.4.1 Coordinated checkpointing rollback-recovery for synchronous iterative schemes

Figure 6.4 shows steps of coordinated checkpointing process for synchronous iterative scheme.

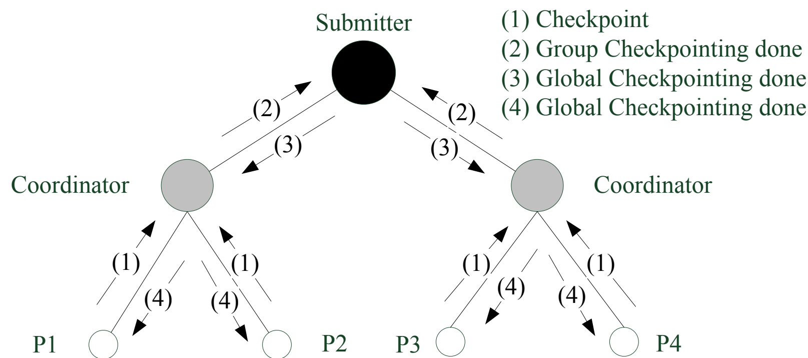


Figure 6.4: Coordinated checkpointing process for synchronous iterative schemes.

- (1) When *P2P_checkpoint* function is called at a worker, the worker creates a checkpoint and sends the checkpoint to its coordinator. We note that in the case of iterative algorithms, all peers execute the same code. Moreover, in the case of synchronous schemes, synchronization between peers is established via blocking operations of communication; thus the *P2P_checkpoint* function is called almost at the same time on all workers. After sending the checkpoint, the worker does not continue the computation immediately; it has to wait for the consistent global checkpoint of application to be generated.

- (2) When a coordinator receives a checkpoint from a worker, it verifies if it has received checkpoints of all workers in its group. When checkpoints of all

workers have been received, the coordinator notifies the submitter that the group checkpointing process is done (see Figure 6.4).

- (3) When the submitter receives notifications of all groups, then the consistent global checkpoint of application is generated. The submitter notifies all coordinators about the global checkpoint.
- (4) Coordinators transfer the global checkpoint notification to workers in its group; then, the coordinator replaces old checkpoints in local memory by new checkpoints. When a worker receives the global checkpoint notification, it replaces the old checkpoint in local memory by the new checkpoint; then it continues the computation.

When a worker fails, its coordinator detects the failure. Since the communication between peer is synchronous, others workers must wait for updates from failed worker; thus, others workers are blocked at communication operations. In the Figure 6.5, we have shown the case where worker $P4$ fails. The process of rollback and recovery to last consistent global checkpoint is the following:

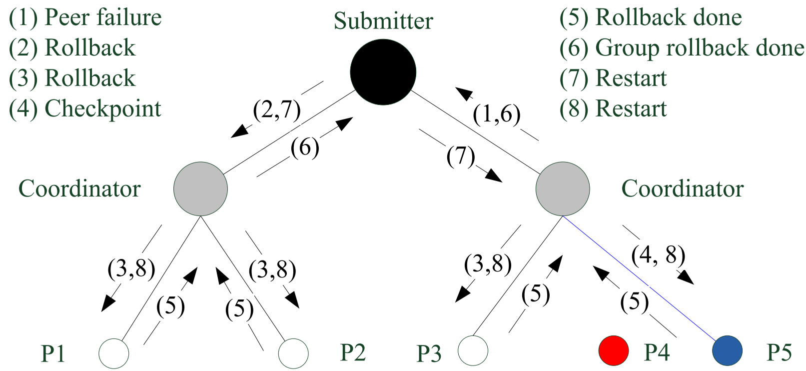


Figure 6.5: Recovery process upon a worker failure for synchronous iterative schemes.

- (1) The coordinator of failed worker P_4 notifies the submitter about peer failure.
- (2) When the submitter receives a peer failure notification, it sends the *rollback* command to coordinators.
- (3) Coordinators transfer the *rollback* command to their workers.
- (4) The coordinator of failed peer finds a free peer in the network, i.e peer P_5 in the Figure 6.5, and sends the last checkpoint of failed worker in local memory to new peer. We note that the peer collection algorithm used to find a free peer here is similar to the one used by the submitter at the beginning of the computation (see Subsection 5.2.8). Moreover, requirements about

peer's characteristics are sent from submitter to coordinators at task allocation phase. Thus, the coordinator finds a free peer for failure recovery that also has to match these requirements. If there is no free peer in the network, then the coordinator of failed peer sends cancellation messages to others peers to terminate the computation.

- (5) Workers receiving *rollback* command stop their computation, load the state from their last checkpoint in local memory; then they send rollback done message to coordinators. On the other side, the new worker $P5$ loads the state from received checkpoint and sends *rollback done* message to coordinator.

- (6) When a coordinator receives *rollback done* message from all peers in the group, it sends *group rollback done* message to submitter. In particular, when the coordinator of failed peer receives *rollback done* message from new peer, e.g. peer P_5, the coordinator sends the address of the new peer-to-peers that have exchanged updates with failed peer so that these peers can exchange updates with the new peer. Coordinators manages peers that exchange updates with peers in their group according a subscribe-publish model as follows. For each peer P_i in a group, the coordinator of P_i maintains a list L_i containing peers that exchange updates with this peer. If a peer P_j exchanges update with peer P_i, then peer P_j sends a *subscribe* message to the coordinator of peer P_i. Upon receiving subscribe message from peer P_j, the coordinator of peer P_i adds peer P_j to the list L_i. Hence, if the peer P_i fails and its state is restored at a given peer P_k, then the coordinator of peer P_i publishes the address of peer P_k to all peers in the list L_i, including peer P_j. For instance, in the Figure 6.5, if peers P_2 and P_3 have exchanged updates with peer P_4, then the coordinator of peer P_4 publishes the address of peer P_5 to peers P_2 and P_3.

- (7) When the submitter received *group rollback done* message from all coordinators, it sends *restart* command to all coordinators.

- (8) Coordinators transfer *restart* message to workers; then workers restart the computation from recovered state.

6.4.2 Uncoordinated checkpointing rollback-recovery for asynchronous iterative schemes

Figure 6.6 shows the uncoordinated checkpointing process for asynchronous iterative schemes. Since no coordination is needed in this case, the checkpoint process is very simple. When *P2P_checkpoint* function is called at a worker, the worker creates a checkpoint and sends the checkpoint to its coordinator. Then the worker continues the computation immediately; moreover the worker does not store the checkpoint in local memory. When coordinators receive checkpoints from workers, then they replace old checkpoints in their local memory by new checkpoints.

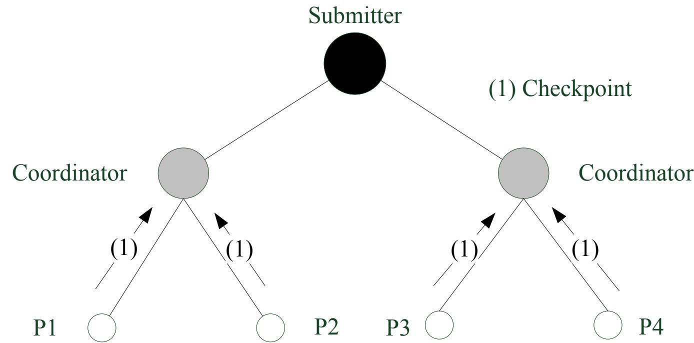


Figure 6.6: Uncoordinated checkpointing process for asynchronous iterative schemes.

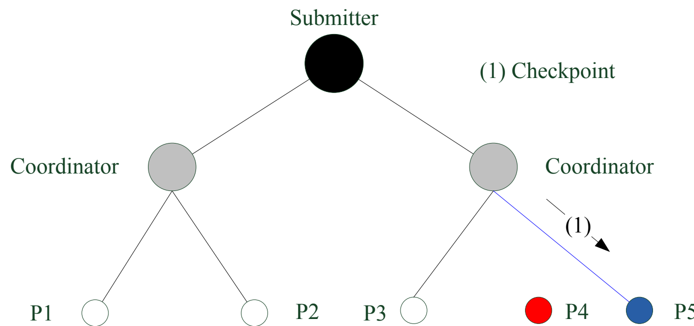


Figure 6.7: Recovery process upon a worker failure for asynchronous iterative schemes.

Recovery process upon a worker failure for asynchronous iterative schemes is also very simple, as shown in the Figure 6.7. When the worker P_4 fails, others workers continue the computation without the failed worker. The coordinator of the failed worker finds a free peer in the network, i.e the peer P_5, and sends the last checkpoint of the worker P_4 to the peer P_5. The peer P_5 loads the state from received checkpoint and starts the computation from this state. The coordinator of peer P_5 sends the address of the new peer P_5 to peers that have exchanged updates with failed peer in order that these peers can exchange updates with the new peer. If there is no free peer in the network, then the coordinator of failed peer will send cancellation messages to others peers to terminate the computation.

6.5 Coordinator failure

When the fault-tolerance properties are activated, coordinators do not execute any subtask. Moreover, coordinator is replicated on several peers in order to achieve fault-tolerance. Thus, at the beginning of a computation, the submitter has to collect more than W peers where W is the number of peers executing subtasks. The level of replication r, i.e. the number of replicas for each coordinator, can be set by

users via command line (see Appendix B); the default value of r is 3. If C is the number of groups, then the submitter has to collect $W + C \times r$ peers.

We note that the implementation of active or semi-active replication would make the fault-tolerance mechanism for worker failure more complicated; furthermore it has more overhead on workers because workers have to send their checkpoint to several replicated coordinators. Thus, in order to cope with coordinator failures, we have implemented the passive replication for coordinator as shown in the Figure 6.8, i.e only the primary coordinator communicates with the submitter and workers, others replicated coordinators are backup of the primary coordinator and store the state of the primary coordinator.

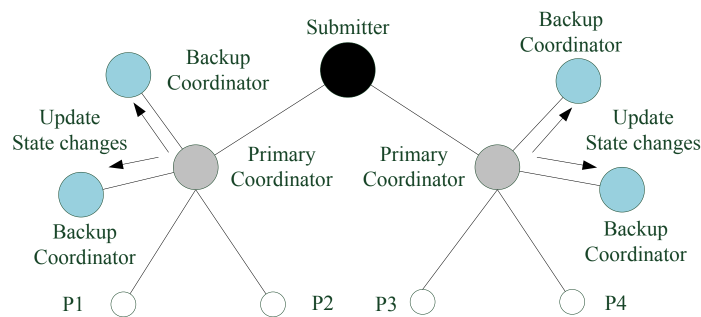


Figure 6.8: Replication of coordinators.

At task allocation phase (see Section 5.3), when the submitter divides collected peers into groups, it chooses in each group a primary coordinator and $r - 1$ backup coordinators. Afterwards, the submitter sends subtasks allocated to peers in a given group to the primary coordinator of this group. After sending subtasks to peers in the group, the primary coordinator creates a checkpoint of its state and sends the checkpoint to its backup coordinators. The checkpoint of the primary coordinator consists of: list of peers in the group, list of backup coordinators, subtasks allocated to peers in the group, checkpoints of workers and so on. Backup coordinators establish the state of the primary coordinator from received checkpoint. Upon a state change on the primary coordinator, e.g. new worker checkpoint or worker failure, the primary coordinator updates this state change to backup coordinators.

Backup coordinators periodically send heartbeat messages to the primary coordinator to inform that they are still alive. When the primary coordinator receives a heartbeat message from a backup coordinator, it sends an acknowledgement message to this backup coordinator. If the primary coordinator does not receive the heartbeat message from a backup coordinator within a time T, then it considers that this backup coordinator has failed. The primary coordinator finds a free peer in the network and send a checkpoint of its state to this peer. The new peer establishes the state of the primary coordinator from received checkpoint and then becomes a backup coordinator. On the other hand, if backup coordinators do not receive acknowledgement message from primary coordinator within a time T, then they consider that the primary coordinator has failed. Then, backup coordinators

communicate between them in order to find the least charged backup coordinator to become new primary coordinator. After that, the new primary coordinator connects to submitter and workers in the group and starts to manage the group; in addition, it finds a free peers in the network to become its backup coordinator.

6.6 Computational experiments

We consider the 3D obstacle problem with size $256 \times 256 \times 256$ (see Chapter 5) in the case where there are peer failures and the fault-tolerant functionality is implemented. Peer failures are simulated by injecting faults on peers at some given time.

6.6.1 Coordinator replication overhead

We have run the 3D obstacle problem on 64 workers in the cases where the level of coordinator replication r is set to 2, 3, 4 or 5. We have found that synchronization between coordinator replicas and coordinator failure have negligible influence on computation. This can be explained by the fact that coordinators does not execute any subtask when the fault-tolerant functionality is chosen; moreover, state changes on primary coordinator are sent to backup coordinators by an independent thread in order to minimize the influence to group management process.

6.6.2 Worker checkpointing and recovery overhead

We have run the 3D obstacle problem on 4, 8, 16, 32 and 64 workers. The machines of the cluster Sagittaire at Lyon have been used in the case of 4, 8, 16 and 32 workers. In the case where the number of workers is 64, we have used 32 machines of the cluster Sagittaire at Lyon and 32 machines of the cluster gdx at Orsay. In each case, we have injected randomly some faults at given workers. Table 6.1 shows the checkpointing time and recovery time for several cases. We note that a checkpoint of a given worker contains only current values of components of sub-blocks assigned to this workers. For instance, in the case of 4 workers, each worker is assigned a sub-block of size $128 \times 128 \times 256$ (see Subsections 5.7.1.1). Then size of a checkpoint of a worker is $128 \times 128 \times 256 \times 8 = 33554432\ bytes = 32\ Mbytes$. Checkpoint time is time to execute the *P2P_Checkpoint* function at a worker. Recovery time is the interval from the failure of a given worker to the start of computation on a new worker where the state of failed worker is restored from its latest checkpoint. Recovery time does not include time to recover to the state before the failure occurred

The checkpointing time in synchronous case is greater than in asynchronous case. This is due to the fact that in synchronous case, in checkpointing process, after sending a checkpoint to coordinators, workers are blocked until the global checkpoint is generated. Moreover, in synchronous case, all workers in a group send their checkpoints to the coordinator nearly at the same time which may result in a bottleneck at the coordinator; whereas in asynchronous case, workers in a group send

Table 6.1: Worker checkpointing and recovery overhead

Workers	Checkpoint size	Checkpointing time		Recovery time	
		Sync	Async	Sync	Async
4	32 Mb	1307 ms	372 ms	1251 ms	1257 ms
8	16 Mb	1349 ms	201 ms	628 ms	654 ms
16	8 Mb	1494 ms	101 ms	320 ms	329 ms
32	4 Mb	1631 ms	51 ms	170 ms	174 ms
64	2 Mb	919 ms	27 ms	105 ms	97 ms

their checkpoints to the coordinator at their own pace; thus sending checkpoints to coordinators in asynchronous case takes less time than in synchronous case.

When the number of workers increases, the checkpoint size decreases; while the checkpointing time in asynchronous case decreases and the checkpointing time in synchronous case increases. This is due to the fact that in synchronous case, the co-ordination overhead increases when the number of workers increases; moreover, the total checkpoint size that a coordinator has to receive from workers in checkpointing process does not change. However, in synchronous case, when the number of workers increases from 32 to 64, the checkpointing time decreases. This can be explained as follows: when 64 workers are used, workers are divided into two groups with two coordinators; then workers send checkpoints to two coordinators, each coordinator receives a half number of checkpoints.

The recovery time of a worker failure in asynchronous case is a bit greater than in synchronous case though in the synchronous case, all workers have to rollback to last checkpoint. This is due to the fact that in the synchronous case, all workers rollback to last checkpoints in local memory in parallel. Moreover, in the asynchronous case, the coordinator of the failed worker still has to receive checkpoints from others workers and others workers still send updates to each others while the recovery of worker failure is processing; whereas in synchronous case, only messages for recovery are sent while the recovery of worker failure is processing. Thus, sending checkpoint of failed workers from the coordinator to the new worker in asynchronous case takes more time than in synchronous case. However, when the number of workers is 64, the recovery time in synchronous case is greater than in asynchronous case. This is mainly due to the enlargement of coordination overhead when machines of two sites Lyon and Orsay are used.

When the number of workers increases, the recovery time in both synchronous and asynchronous cases decrease since the checkpoint size decreases.

6.6.3 Influence of worker failures on computational time

In order to study the influence of worker failure on computational time, we have run the 3D obstacle problem on 64 workers using machines on two sites Lyon and

Orsay. Checkpoints are taken every 1000 relaxations and some worker failures are generated randomly. The Figure 6.9 shows the computational time in several cases where the number of worker failures varies from 0 up to 10.

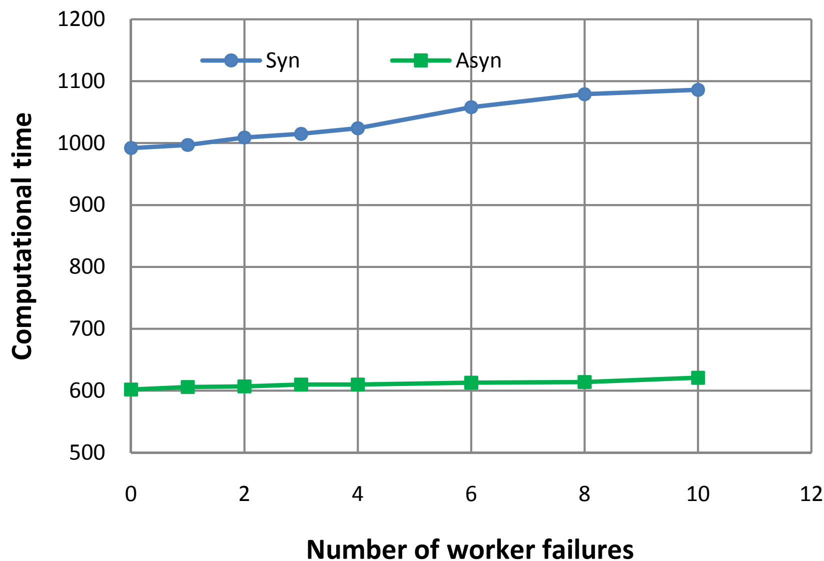


Figure 6.9: Computational time for number of worker failures from 0 up to 10.

In the Figure 6.9, we can remark that when the number of worker failures increases, the computation time increases faster for synchronous iterative algorithm than for asynchronous iterative algorithm. This is mainly due to the fact that in the synchronous case, when a worker fails, all workers have to rollback to the last checkpoints. Whereas, in asynchronous case, only the state of the failed worker is rollbacked to last checkpoint, others workers continue computing with current state. In the case where the number of failures is equal to 10, the computational time increases about 10% in synchronous case and about only 4% in asynchronous case.

6.7 Chapter summary

In this chapter, we have presented the fault-tolerance mechanisms in P2PDC to cope with peer volatility. The fault-tolerance mechanisms can adapt themselves to peer roles and computational schemes. For worker failure, the rollback recovery techniques have been chosen: while the coordinated checkpointing strategy is implemented in synchronous case, the uncoordinated checkpointing strategy is implemented in asynchronous case. For coordinator failure, the replication technique has been chosen.

Experiments on Grid'5000 with fault injection for the obstacle problem showed that the fault-tolerance mechanisms in P2PDC have small impact on the computation even with a great amount of failures. Synchronization between coordina-

tor replicas and coordinator failure have negligible influence on computation. The checkpointing and recovery processes are really fast. In the case of 64 workers and 10 worker failures, the computational time increases about 10% for synchronous iterative algorithms and about only 4% for asynchronous iterative algorithms.

Chapter 7

Contribution to a web portal for P2PDC application deployment

Contents

7.1 Introduction **107**
7.2 Background **107**
7.2.1 OML 107
7.2.2 OMF and its Portal 109
7.3 Motivation **110**
7.4 A new measurement channel for P2PDC **112**
7.4.1 Hierarchical measurements collection 112
7.4.2 Application to task deployment 114
7.5 Chapter summary **117**

7.1 Introduction

In this chapter, we present the principle of an original solution related to a web portal for P2PDC application deployment. Most of the ideas presented in this chapter are developed in collaboration with NICTA, Sydney Australia. This Portal is the combination of P2PDC with tools developed at NICTA, i.e. OML, OMF and OMF Portal in order to facilitate the deployment, management of P2PDC applications as well as the retrieval and analysis of results. The Portal is under development. Thus, in this chapter, we present only the first ideas on the web portal and introduce a new measurement channel for P2PDC on OML.

7.2 Background

In this section, we present briefly tools developed at NICTA, i.e. OML, OMF and OMF Portal.

7.2.1 OML

OML [White 2010] is a multithreaded instrumentation and measurement library, which was first developed as the companion measurement library of the cOntrol and

Measurement Framework OMF [Rakotoarivelo 2010]. This library is now a standalone open source software allowing the collection of any type of measurements from any type of distributed applications and their storage in a unified format. The OML measurement reporting can be added alongside original reporting mechanisms or as their replacement. One of the main benefits of OML reporting resides in an effortless correlation of data from different distributed sources to investigate network anomalies, or test research hypotheses or developed prototypes.

OML is composed of three main components allowing an automatic generation and collection of measurements. First, a user needs to define Measurement Points (MP) within their applications or services. An MP is an abstraction for a tuple of related metrics which are reported ("injected") by the application at the same instant during the run-time of the application. This injection can be configured in order to generate Measurement Stream (MS) composed of the entire set of tuples or just a subset. If the user selects only a subset of tuples, then the unused ones are just discarded. Furthermore, prior to sending of these streams to the locate or remote repository, these MSs can be further processed. This processing is accomplished through OML's filters. An experimenter can indeed implement a function, called hereafter filter, within the OML API to be applied on some or all of the fields of an MS to format the data or compute more specific metrics based on either a sole injection or a window of injection. For example, in the case of an application reporting the size of each packet it receives inside an MP, a filter may be configured to sum up these samples over a 1 second period in order to provide an estimate of the immediate throughput.

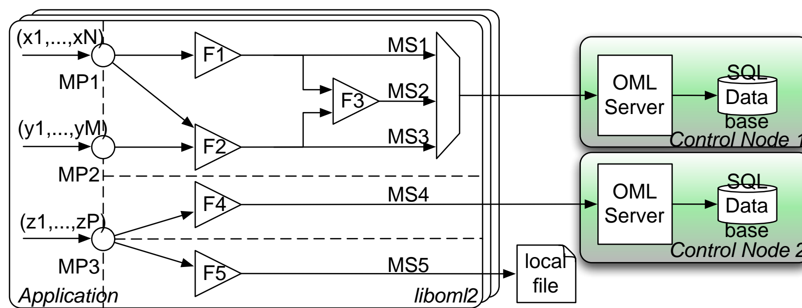


Figure 7.1: OML - the OMF Measurement Library

Figure 7.1 shows an example of OML data path. An application injects measurements into three MPs. At run-time, the tuples generated by injections in the MPs are combined in order to form five MSs. These newly created streams are then filtered, and the results are directed to one of two different collection servers or a local file. The right part of Figure 7.1 represents the server side where the OML server serves as a front-end to a database.

This library has been recently evaluated in term of its impact on the resources and the measurements themselves in [Mehani 2011]. The authors of [Mehani 2011]

have found that this library allows the experimenters to easily develop measurement applications while improving the overall performance of the measurement process when compared to the non-threaded version of an application. Furthermore, the authors have shown that OML does not impact the footprint of any tool whether it concerns the CPU or memory usage.

7.2.2 OMF and its Portal

In order to evaluate new networking technologies, researchers have developed and deployed large facilities (testbeds) complementarily to preliminary simulated results. These testbeds aim at providing real conditions for testing research works while proposing repeatability in a semi-closed environment. Nevertheless, offering and performing repeatability requires the development of management frameworks. During the last decade, the cOntrol and Management Framework [Rakotoarivelo 2010] has been developed to tackle this difficult challenge. This framework offers a suite of management, control and measurement services for networking testbeds.

From an operator perspective, OMF provides several services to manage, allocate and configure heterogeneous resources within a testbed. From an experimenter's point of view, it provides a high level domain-specific language to systematically describe an experiment (i.e. its used resources, required measurements and tasks to perform) and a set of software tools to automatically deploy and orchestrate this experiment on a given testbed.

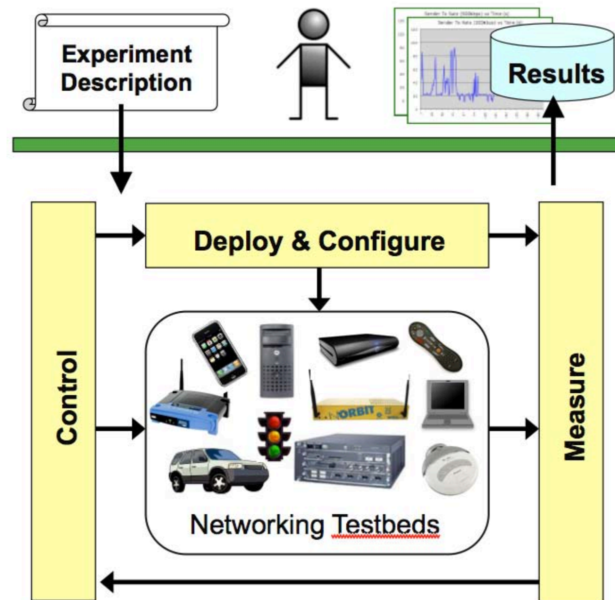


Figure 7.2: Overview of OMF architecture from the user's point of view

Figure 7.2 represents a simple view of OMF architecture from a user's experience. In this figure, we can note that every experiment starts with the definition of an

Experiment Description (see Appendix A). This script is later passed to the OMF system which in turn performs all the mandatory operations to deploy, configure and execute the different elements of the experiment. During the experiment, if the user have configured their application with OML, then measurement streams are created and automatically available.

This management framework is used widely around the world and it has been integrated within other research and educational tools. In particular, in [Jourjon 2011] Jourjon *et al.* present a portal which allows researchers to closely follow the hypothetico-deductive method. This work has been made possible thanks to the development of a remotely accessible lab book and the enhancement of the wiki aspect of a previously introduced e-learning platform called IREEL [Jourjon 2010] in order to create the LabWiki.

Through the modularity of OMF, this LabWiki could be used in order to facilitate researchers collaboration and peer verification of the finals result. Indeed, this portal offers the possibility to make public and migrate content to a public space and it offers users the possibility to create numerous projects where they can add collaborators. Furthermore, this portal integrates a graphical interface to analyze the resulting collected data. This interface is the other major contribution of LabWiki. It allows the researchers to edit or load R scripts [r] describing statistical computations to be performed on the collected data. LabWiki will run these scripts into a R interpreter which has access to the experiment data, and will present the resulting outputs (e.g. graphs, tables,...) to the researchers.

7.3 Motivation

A main advantage of peer-to-peer high performance computing is that any user can submit its own application. However, it also leads to some drawbacks related to the deployment of P2PDC applications on peer-to-peer networks. First, submitter machine has to initiate the computation, i.e. decompose the dataset, send data subset as well as application code to workers and receive results from workers either directly or via coordinators. If the submitter machine is not performant with low network bandwidth, then the submitter may become a bottleneck that leads to parallel algorithm efficiency reduction. Second, although tasks are distributed to be computed at several peers, the duration of the computation may still be long. The submitter has then to stay connected until the completion of the computation. If the submitter disconnects, then the computation terminates immediately. Third, there are more free peers during some intervals of time of the day than during others. For example, there are more free peers during the night than during the day. But some users can not connect and start their application during the night. The last drawback of the current P2PDC system is that the received results are in raw format so that users have to make further treatment to obtain more sophisticated representations like graphs.

In order to overcome these drawbacks and to facilitate the deployment, man-

agement of P2PDC applications as well as the retrieval and analysis of results, we have proposed a solution that is based on the combination of P2PDC with tools developed at NICTA, i.e. OML, OMF and OMF Portal. The first ideas is displayed in Figure 7.3.

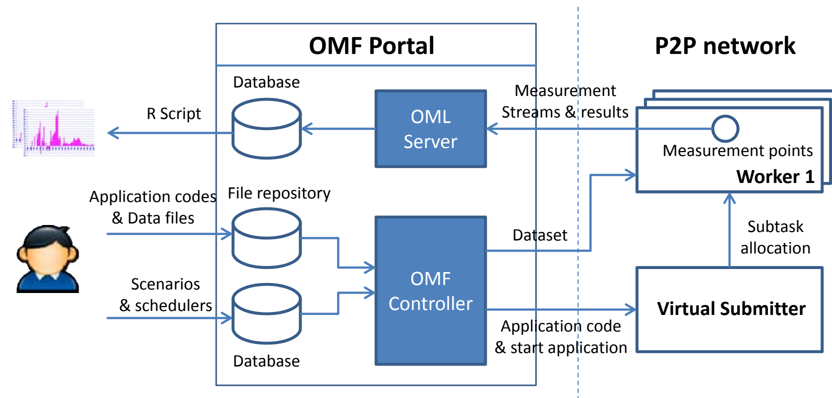


Figure 7.3: A web portal for P2PDC application deployment

In Figure 7.3, users can upload their application codes as well as datasets in file format to the Portal via the web interface. With the help of the web interface, they can also customize their application according to different scenarios, e.g. change the dataset and the number of workers. Moreover, users can schedule to start the application at the desired moment. Based on the scheduling information on the database, the OMF Controller on the Portal selects a given machine on the network to start the application. This machine is called Virtual Submitter. Virtual Submitters can be dedicated machines managed by Portal administrators or peers with attractive characteristics in the network. Once the application is launched on a given Virtual Submitter, dataset is sent directly from the Portal to workers. Results can be sent as an OML measurement stream to the OML Server either from Virtual Submitter or directly from workers. In the former case, results are sent normally from workers to Virtual Submitter via P2PDC environment. Then, Virtual Submitter makes the result aggregation and sends the final result to the Portal. In the latter case, workers send directly results to the Portal. The OML Server on the Portal stores results measurement streams into database or in a file. Users can retrieve results from the Portal. Furthermore, they can write some R script so as the Portal can be able to create graphs or tables representation of the results. We note that there may be several Portals on the network. Any organization or even any individual user can install its own Portal. With the presence of the Portal, users do not need to stay connected when the computation is running. They can reconnect later on and retrieve the result from the Portal.

7.4 A new measurement channel for P2PDC

In this section, we introduce a new measurement channel for P2PDC on OML that reduces the volume of collected measurements and thus limit the impact of the measurements on the computation. Afterward, we present the application of this measurement channel to task deployment. In particular, this part permits one to give technical details related to task deployment in connection with the previous section.

7.4.1 Hierarchical measurements collection

Current OML architecture provides users with filters enabling to perform some preprocessing on a specific measurement stream at the resource that produces it. However, in many experiments, users do not need measurements from every nodes but integrated metrics over these measurements streams. For example, in the solution of a numerical simulation problem on peer-to-peer network, users want to collect periodically the computational error of overall computation which is the maximum computational error on all nodes in order to trace the evolution of the solution. The measurement architecture of this experiment with current OML is depicted in Figure 7.4.

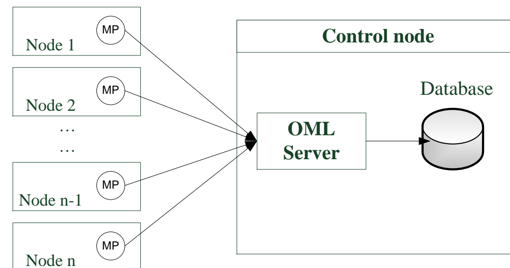


Figure 7.4: Current measurement architecture

In Figure 7.4, users create a measurement point at each node that injects periodically the computational error at this node to OML server. In turn, OML server stores those measurement streams to a database. Once the experiment has finished, users can query the measurement database with basic SQL queries in order to extract the maximum computational error on all nodes at each time steps from database. We can note that not only unnecessary data are stored in the database but also further manipulations need to be made in order to extract necessary information.

Figure 7.5 displays the maximum error in function of the time for the obstacle problem with 2 peers at NICTA and 2 peers on PlanetLab [Ott 2010]. This results have been obtained with OML and P2PDC.

Hence, we have proposed to provide users with a new type of filter that allows users to perform some preprocessing on several measurement streams from different resources. Such a preprocessing can dramatically reduce the volume of collected

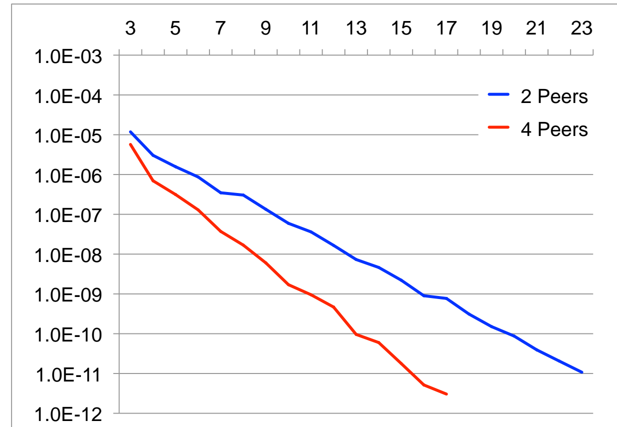


Figure 7.5: Maximum error measurement for the obstacle problem with 2 peers at NICTA and 2 peers on PlanetLab

measurements and thus limit the impact of the measurements on the computation. In order to even more minimize impact to current architecture, the new type of filter is implemented on an OML proxy-server [White 2010]. The proxy-server can be placed in the same machine as OML Server or in a separate machine. The measurement architecture is displayed in Figure 7.6.

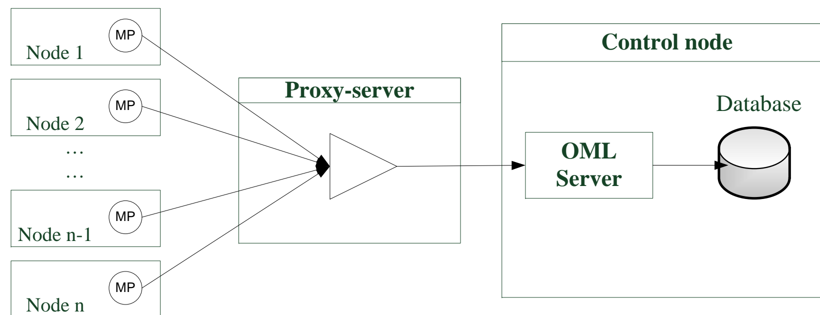


Figure 7.6: Hierarchical measurement architecture

In Figure 7.6, node does not inject measurement stream directly to OML Server but to an OML proxy-server. A *max(.)* filter is implemented on the proxy-server that calculates the maximum computational error from n entering streams (where n is the number of nodes) at each time step and forwards this value to OML Server. Hence, the volume of collected measurements stored in database at OML Server is reduced n times. Moreover, users do not need to make any further manipulation on collected measurements.

In large scale experiments, where the number of nodes involved is large and nodes spread over network, if only an OML server (or a proxy-server) collects all measure-

ment streams from all nodes, then the OML server (or the proxy-server) may become a bottleneck that leads to efficiency reduction of measurement collection. With the presence of filters on proxy-server, we can deploy a hierarchical measurement architecture that not only avoids the bottleneck at OML Server (or proxy-server) but also reduces the volume of measurement data sent over long-distance link. The measurement architecture can be summarized as in Figure 7.7.

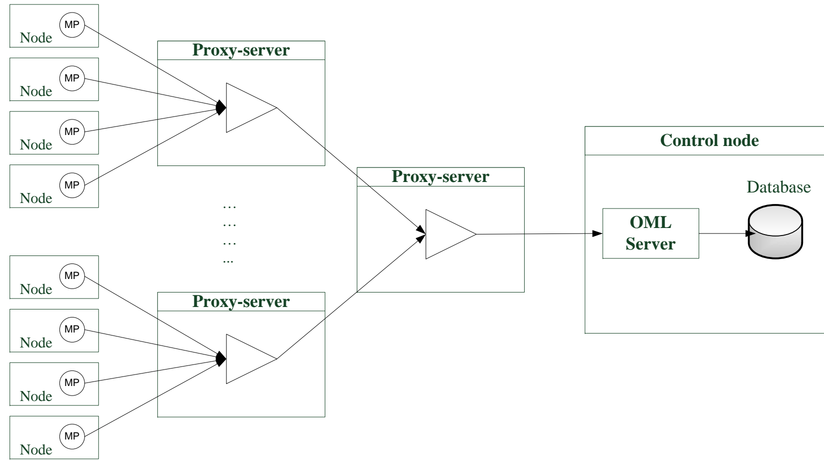


Figure 7.7: Multi-level hierarchical measurement architecture

Then, we can put inside a group of nearby nodes a proxy-server implementing a filter that pre-processes measurement streams injected by peers in this group. Afterward, a top-level proxy-server integrates measurement streams injected by group's proxy-servers and forwards integrated metrics to OML Server. We can remark that measurement streams of a group of nearby nodes are pre-processed locally inside this group and only one measurement stream is sent from a given group to top-level proxy-server.

7.4.2 Application to task deployment

At the beginning of the solution of a problem via a parallel iterative algorithm, the initial dataset is decomposed into n parts and each part needs to be sent to corresponding peers. In the P2PDC architecture, when a programmer defines a task, he needs to read the dataset from a binary file, he decomposes it into subsets and integrates data subsets to subtasks as parameters; then, data subsets are sent along with subtasks to peers. With the integration of P2PDC and OMF/OML, task submission is done through OMF Portal. The data file of a task is uploaded to a File Repository on the OMF Portal and needs to be distributed to peers when the computation begins. In this subsection, we present an efficient method that makes use of the measurement library OML in order to distribute the dataset to peers.

We recall that the measurement library OML allows researchers to define measurement points inside their program and then create automatically measurement

streams to store either locally or in a remote server. In our case, we want to use this library in a reverse manner whereby we will inject data to distribute to several clients instead of having several clients injecting measurements that would be collected by a server. The figure 7.8 gives the general idea on how to use OML for task deployment.

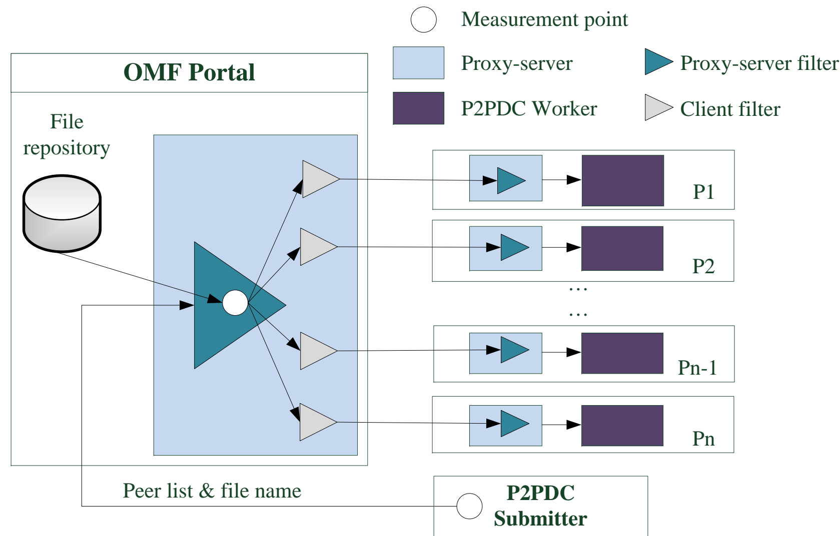


Figure 7.8: Task deployment via OML

In the case of a general problem that can be solved via parallel iterative algorithms, the dataset is often in the form of a matrix with d dimensions with $d = 1, 2, 3, \ldots$ and the data type can be int, float, double, etc. For example, in a 3D obstacle problem of size 128, the dataset is a three dimension matrix $128 \times 128 \times 128$. In the solution of this problem via 4 peers, the dataset is decomposed according to pillar decomposition into 4 sub-matrices: $[0-63][0-63][0-127]$, $[0-63][64-127][0-127]$, $[64-127][0-63][0-127]$, $[64-127][64-127][0-127]$; then, each sub-matrix is sent to a peer. In our method, users need to write an *xlm* file that defines the dataset decomposition. The *xml* file related to the above example is displayed in the listing 7.1.

Listing 7.1: XML configuration file

```
1 <P2PDC_data dim='3' size='128'>
2   <peer rank='0' segment='[0-63][0-63][0-127]'/>
3   <peer rank='1' segment='[0-63][64-127][0-127]'/>
4   <peer rank='2' segment='[64-127][0-63][0-127]'/>
5   <peer rank='3' segment='[64-127][64-127][0-127]'/>
6 </P2PDC_data>
```

The *xml* file needs to be uploaded to the File Repository on the OMF Portal along with data files.

A proxy-server is placed on OMF Portal in order to distribute automatically dataset to peers. When an experiment starts, the P2PDC Submitter injects a measurement point that contains the list of peers and name of data file as well as *xml* file to the proxy-server of OMF Portal. The proxy-server of OMF Portal transfers this measurement to a specific filter, the so-called *Init_Portal_Proxy* filter. This specific filter does not write any data to output but creates n OML client filters of type *Init_Portal_Client* (n is the number of peers) and sets parameters to each filter based on information in the *xml* file; it is done via the creation an *xml* file for OML Client. The created *xml* file for OML Client in the above example is displayed in the listing 7.2.

Listing 7.2: XML configuration file for filters

```
1  <omlc id='P2P_Initialiser' exp_id='1298606048'>
2    <collect url='tcp:163.117.253.22:3003'>
3      <mp name='mp_init_data' samples='2097152'>
4        <f fname='Init_Portal_Client' pname='value' sname='P2PDC_init'>
5          <fp name='dim' type='int'>3</fp>
6          <fp name='size' type='int'>128</fp>
7          <fp name='segment' type='string'>[0-63][0-63][0-127]</fp>
8        </f>
9      </mp>
10   </collect>
11   <collect url='tcp:163.117.253.23:3003'>
12     <mp name='mp_init_data' samples='2097152'>
13       <f fname='Init_Portal_Client' pname='value' sname='P2PDC_init'>
14         <fp name='dim' type='int'>3</fp>
15         <fp name='size' type='int'>128</fp>
16         <fp name='segment' type='string'>[0-63][64-127][0-127]</fp>
17       </f>
18     </mp>
19   </collect>
20   <collect url='tcp:193.136.124.226:3003'>
21     <mp name='mp_init_data' samples='2097152'>
22       <f fname='Init_Portal_Client' pname='value' sname='P2PDC_init'>
23         <fp name='dim' type='int'>3</fp>
24         <fp name='size' type='int'>128</fp>
25         <fp name='segment' type='string'>[64-127][0-63][0-127]</fp>
26       </f>
27     </mp>
28   </collect>
29   <collect url='tcp:193.136.124.228:3003'>
30     <mp name='mp_init_data' samples='2097152'>
31       <f fname='Init_Portal_Client' pname='value' sname='P2PDC_init'>
32         <fp name='dim' type='int'>3</fp>
33         <fp name='size' type='int'>128</fp>
34         <fp name='segment' type='string'>[64-127][64-127][0-127]</fp>
35       </f>
36     </mp>
37   </collect>
38 </omlc>
```

Then the *Init_Portal_Proxy* filter reads the data file and injects sequentially data values to all *Init_Portal_Client* filters. When an *Init_Portal_Client* filter receives a data value, it knows if it must treat this data value based on filter parameters. When data is injected, *Init_Portal_Client* filters send data to peers. On each peer, a proxy-server will receive measurement stream from *Init_Portal_Client* filter on the OMF Portal and transfers this measurement stream to a so-called *Init_Peer* filter. Like *Init_Portal_Proxy* filter, *Init_Peer* filter does not write any data to output but sends data to P2PDC worker. The communication between *Init_Peer* filter and P2PDC Worker is made via local socket.

We present now a first series of computational results obtained with OMF and P2PDC on the PlanetLab testbed. We note that in these experiments we have used only the new measurement channel for task deployment; P2PDC is not yet combined with OMF Portal.

PlanetLab is a global research network that supports the development of new network services. Since the beginning of 2003, more than 1,000 researchers at top academic institutions and industrial research labs have used PlanetLab to develop new technologies for communication protocols, distributed storage, network mapping, peer-to-peer systems, distributed hash tables, and query processing. PlanetLab currently consists of 1109 nodes at 512 sites.

We have collected 24 machines from 12 sites (2 machines on each site): 4 sites in US and 8 sites in Europe. Latency between machines at a same site is about 0.1 ms while latency between machines of different sites varies from 30 ms to 330ms. Machines are heterogeneous; processor's frequency varies from 2.4 to 3.0 GHz.

We have considered a 3-Dimensional obstacle problem with size 192 x 192 x 192. Experiments have been carried out on 1, 2, 4, 8, 16 and 24 machines. Computational time in the sequential case, i.e. with one machine, varies from 3158 s to 6555 s according to the features of the machine. The synchronous schemes are not suited to this type of networks, since latency is much greater than the duration of a single relaxation. Hence, we have considered only the asynchronous scheme. Moreover, PlanetLab limits the bandwidth used in 24 hours, thus we have reduced update's frequency in order to respect PlanetLab user's charter: a node sends updates to its neighbors every 10 relaxations. Through experiments, we found that the reduction of update's frequency increases computational time from 5% to 10%.

Computational results are presented in Figure 7.9. We note that the sequential computational time of the fastest machine is used in order to calculate speedup and efficiency.

7.5 Chapter summary

In this chapter, we have presented the contribution to a web portal for P2PDC application deployment. This Portal is the combination of P2PDC with tools developed at NICTA, i.e. OML, OMF and OMF Portal. We have given the first ideas related to the Portal architecture and explained how this Portal can facilitate the

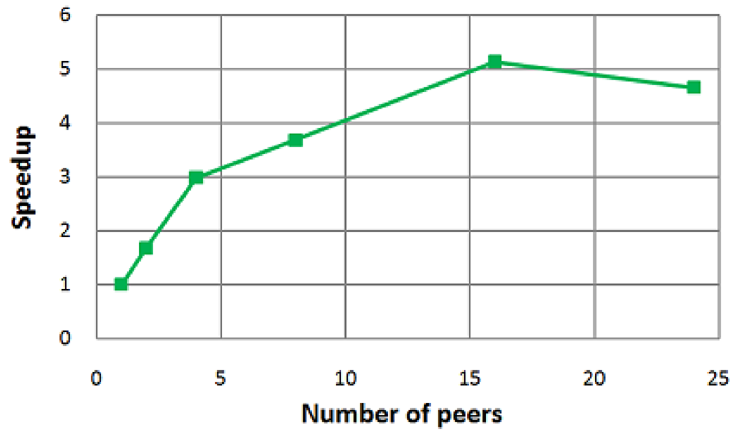


Figure 7.9: Computational results on PlanetLab

deployment, management of P2PDC applications as well as the retrieval and analysis of results. We have also introduced a new measurement channel for P2PDC on OML that reduces the volume of collected measurements and thus limit the impact of the measurements on the computation.

Chapter 8

Conclusions and perspectives

In this manuscript, we have presented our contributions to peer-to-peer high performance computing. In particular, we have shown how we have designed and implemented P2PSAP, a self-adaptive communication protocol dedicated to P2P HPC applications. P2PSAP protocol is designed in order to allow rapid update exchange between peers in the solution of numerical simulation problems via distributed iterative algorithms. The protocol can configure itself automatically and dynamically in function of application requirements like choice of scheme of computation and elements of context like topology by choosing the most appropriate communication mode between peers. We note that this approach is different from existing communication libraries for high performance computing like MPICH/-Madeleine [Aumage 2001] in allowing the modification of internal transport protocol mechanism in addition to switch between networks. P2PSAP protocol has been implemented on a small network for the solution of nonlinear optimization problems, i.e. network flow problems. A first set of computational experiments shows that the protocol permits one to obtain good efficiency particularly when using asynchronous communications or a combination of synchronous and asynchronous communications.

In chapter 4, we have presented the first version of P2PDC, an environment for peer-to-peer high performance computing. We have described the general architecture of P2PDC along with its main functionalities. We have proposed a programming model for P2PDC that facilitates the work of programmer. Indeed, in order to develop an application, programmers have to write code for only three functions; all others support activities are carried out automatically by the environment. In particular, the communication operation set is reduced, programmers do not have to care about the choice of communication mode, they just care or not about the choice of a given iterative scheme of computation, e.g. synchronous, asynchronous. The development of an application with P2PDC takes less programmer effort than with MPI and PVM. The first implementation of P2PDC with centralized and simplified functionalities has also been studied. Finally, we have displayed and analyzed computational results on the NICTA platform with up to 24 machines for a numerical simulation problem, i.e. the obstacle problem. Computational results have shown that the combination of P2PSAP with P2PDC allows to solve efficiently numerical simulation problems via distributed iterative methods, in particular when using asynchronous or hybrid schemes of computation.

In chapter 5, we have presented the decentralized version of P2PDC that includes new features aimed at making P2PDC more scalable and efficient. Indeed,

the resources manager is based on a hybrid topology that is simple but efficient and which facilitates peers collection for computation. The hierarchical task allocation mechanism accelerates task allocation to peers and avoids connection bottleneck at submitter. Furthermore, a file transfer functionality has been implemented that allows to transfer efficiently files between peers. Moreover, the communication operation set has been extended in order to facilitate the implementation of some asynchronous algorithms and their convergence detection and termination, with application to evolution problems in particular [Garcia 2011]. Experiments for the obstacle problem have been carried out on GRID'5000 platform with up to 256 peers. A *pillar decomposition* has been proposed that reduces the total size of messages sent by workers after each relaxation as compared with *slice decomposition* presented in chapter 4. A convergence detection and termination method designed by Bertsekas [Bertsekas 1991] has been implemented for asynchronous iterative schemes that detects exactly the termination and reduces unnecessary relaxations. Computational results show that the *pillar decomposition* improves significantly the efficiency of computations. Moreover, we have obtained a good efficiency (0.78) for asynchronous iterations in the case where upto 256 machines distributed over 8 clusters at 5 sites are used. This shows the interest of combining asynchronous schemes of computation with the decentralized environment P2PDC.

In chapter 6, we have presented the fault-tolerance mechanisms in P2PDC to cope with peer volatility. The fault-tolerance mechanisms can adapt themselves to peer roles and computational schemes. For worker failure, the rollback recovery techniques have been chosen: while the coordinated checkpointing strategy is implemented in synchronous case, the uncoordinated checkpointing strategy is implemented in asynchronous case. For coordinator failure, the replication technique has been chosen. Experiments on Grid'5000 with fault injection for the obstacle problem showed that the fault-tolerance mechanisms in P2PDC have small impact on the computation even with a great amount of failures. Synchronization between coordinator replicas and coordinator failure appears to have negligible impact on computation.

Finally, in chapter 7, we have presented the first ideas related to the use of OML, OMF and its Web portal in order to facilitate the deployment of P2PDC applications on peer-to-peer networks. Some aspects related to measurements in P2P applications have also been presented.

It is noted that the P2PDC environment has been used with success by several teams in France and Australia. The team MIS has implemented efficiently several parallel algorithms for 2D cutting stock problems [Hifi 2011]. The team at IRIT-ENSEEIT has also implemented efficiently electrophoresis problems and evolution Black-Scholes equations [Chau 2011, Garcia 2011]. The team at NICTA Sydney Australia has made some implementation of distributed iterative method for numerical simulation problem on PlanetLab [Ott 2010]. Moreover, the team at LIFC has integrated P2PDC into the simulation tool P2PPerf so as to make prediction of performance for several scenarios [Cornea 2011].

In future work, we note that it is needed to improve the communication protocol,

the application code and in particular decomposition schemes as well as the decentralized environment so as to obtain better efficiencies in massively parallel context. As a matter of fact, the need for scalable architectures is particularly important in peer-to-peer computing.

Hybrid methods that combine synchronous and asynchronous iterative schemes and that have been introduced in this thesis need further investigation, in particular in the case of high bandwidth network like Myrinet and Infiniband. We believe this new type of parallel and distributed iterative algorithms to be very efficient in this context.

It is also important to design an efficient way to deploy computations on peer-to-peer networks. The approach combining the decentralized P2PDC environment with OML, OMF and its Portal must be investigated further on in order to facilitate the deployment and management of P2P HPC applications. The use of a web portal will surely draw more P2PDC users. We note also that using OML measurements in combination with P2PDC can permit one to carry out steering of iterative methods. In particular, one can encompass to use OML measurements in the solution of some nonlinear optimization problems so as to switch from a gradient method to Newton method when the iterate vector is close to the solution. This will permit one to improve the convergence rate of the implemented method.

Other applications have to be considered in order to validate our approach. In particular, several logistic applications related to the solution of complex problems like traveling salesman or multi-dimensional knapsack problems have to be considered as well as others numerical simulation applications.

The combination of peer-to-peer computing with a new approach like GPU computing deserves also to be investigated.

Appendix A

OMF's Experiment Description Language

A.1 OMF's Experiment Description Language (OEDL)

OMF [Rakotoarivelo 2009, omf] defines and uses a Domain-specific Language to describe an experiment. This language is named OEDL, standing for *OMF Experiment Description Language*.

OEDL which is based on the Ruby language [rub] provides a set of specific OMF commands and statements. A new user does not need to know Ruby to write experiment description with OEDL. User can get started with only some basic OEDL commands and syntax. However, user will need to have some general entry-level programming knowledge.

An OMF Experiment Description (ED) is composed of two parts in the following order:

- **Resource Requirements and Configuration:** this part enumerates the different resources that are required by the experiment, and describes the different configurations that need to be applied to them.
- **Task Description:** this part is essentially a state-machine, which enumerates the different tasks to perform with the required resources in order to realize the experiment.

The OEDL commands can be grouped into the following categories:

- *Top-level commands:* can be used anywhere within the ED, i.e. in any of the two parts mentioned above. These commands allow to set experiment properties and to manage logging messages. For example,

  defProperty('rate', 300, 'Bits per second sent from sender')

  defines the property *rate* with the initial value 300 in order to present the number of bit par second sent from sender.

- *Topology-specific commands:* are used in the Resource Requirements and Configuration section of the ED. They allow the definition of the topology involving specific resources, and some potential related constraints. For example,

  defTopology('test:topo:origin', [1-4])

defines a topology that contains four specific nodes.

- *Group-specific commands:* are used in the Resource Requirements and Configuration section of the ED. They allow the definition of a given group of resources, the description the specific resources that should be placed in that group, and the configuration to apply to them if needed. For example,

  defGroup('receivers', [1-2])

  defines the group *receivers* that includes two specific nodes

- *Prototype-specific commands:* are used in the Resource Requirements and Configuration section of the ED. These commands allow definition of an OMF prototype. This group is composed of a main command *defPrototype* to define a new prototype and a list of sub-commands to specify the prototype like *proto.name*, *proto.description*.

- *Application-specific commands:* are used in the Resource Requirements and Configuration section of the ED. They allow the definition of a OMF application. This group is composed of a main command *defApplication* and a list of sub-commands to specify the application like *app.shortDescription*, *app.path*.

- *Execution-specific commands:* are used in the Task Description section of the ED. They allow the definition of the different tasks to execute when the experiment reaches a specific state. For example,

  group('receivers').startApplications

  starts application at all nodes of *receivers* group.

- *Resource Paths:* are used in any section of the ED. A resource path allows the access and the value assignment of a specific configuration parameter to a resource. For example,

  node.net.eth0.ip = '192.168.1.1'

  assigns the IP address *192.168.1.1* to network card *eth0* at a given node.

- *Testbed-specific commands:* are only available for specific testbed deployments, i.e. they act on particular types of resources that are only available on some specific testbeds. For example, *antenna* command injects noise into the testbed through the available antennas.

Some of these commands also provide a list of sub-commands. These sub-commands will only be usable when associated with the parent command.

A.2 Examples of experiment description

Listing A.1 presents the experiment description in the case where the size of the obstacle problem is $96 \times 96 \times 96$, the computational scheme is synchronous and 4 workers inside a same cluster are used.

Listing A.1: Examples of experiment description files

```
1 #
2 # Define the P2PDC application for submitter
3 #
4 defApplication('P2PDCAppSubmitter', 'P2PDCAppSubmitter') do |app|
5   app.shortDescription = "P2PDC wrapper application for submitter"
6   app.path = "/P2PDC/Peer/P2PDC eth0 obstacle 96 1 4"
7   app.defMeasurement('mp_submitter') do |m|
8     m.defMetric('NbrIters', :long)
9     m.defMetric('Time', :float)
10   end
11 end
12
13 #
14 # Define submitter's group
15 #
16 defGroup('submitterGroup', 'omf.nicta.node9') do |node|
17   node.addApplication('P2PDCAppSubmitter') do |app|
18     app.measure('mp_submitter', :samples => 1)
19   end
20 end
21
22 #
23 # Define the P2PDC application for workers
24 #
25 defApplication('P2PDCAppWorker', 'P2PDCAppWorker') do |app|
26   app.shortDescription = "P2PDC wrapper application for workers"
27   app.path = "/P2PDC/Peer/P2PDC eth0"
28   app.defMeasurement('mp_worker_result') do |m|
29     m.defMetric('rank', :int)
30     m.defMetric('NbrIters', :int)
31   end
32   app.defMeasurement('mp_worker_diff') do |m|
33     m.defMetric('rank', :int)
34     m.defMetric('Iters', :int)
35     m.defMetric('diff', :float)
36   end
37 end
38
39 #
40 # Define worker's group
41 #
42 defGroup('workerGroup', 'omf.nicta.node10,omf.nicta.node12,omf.nicta.
      node3,omf.nicta.node2') do |node|
43   node.addApplication('P2PDCAppWorker') do |app|
44     app.measure('mp_worker_result', :samples => 1)
45     app.measure('mp_worker_diff', :samples => 1)
```

```
46   end
47 end
48
49 onEvent(:ALL_UP_AND_INSTALLED) do |event|
50
51   group('workerGroup').startApplications
52   wait 5
53
54   group('submitterGroup').startApplications
55
56   # Wait for application execution
57   wait 800
58
59   # Stop the expiriment
60   Experiment.done
61 end
```

Appendix B

How to write and run P2PDC applications

B.1 How to write a P2PDC application

A P2PDC application must include header file **P2PDC.h** and implements the following functions:

- int TaskDefinition(P2PTask* pTask)
- int Calculate(P2PSubtask* pSubtask)
- int ResultsAggregation(P2PTask* pTask)

In *TaskDefinition* function, one can analyze parameters that user inputs at startup and set task, subtasks parameters. For the *Task*, one must set:

- *pTask->scheme*: choice of computation scheme (*SCHEME_SYN*, *SCHEME_ASYN*, *SCHEME_HYBRID*).
- *pTask->cSubtasks* (number of subtasks) and *pTask->cPeers* (number of peers). For the moment, those two parameters must have same value, i.e. one peer executes only one subtask.
- *pTask->pSubtasks*: pointer points to an array of subtasks.

For each subtask, one can set subtask owner parameters in *params* field and the size of this *params* field in the *params_size* field.

Each subtask will be assigned automatically a rank that is equal to its index in the subtask array $(0, \ldots, cSubtasks - 1)$. P2PDC environment will collect peers and send subtasks to peers automatically.

In *Calculate* function, one writes the code to compute a subtask. One can retrieve subtask rank (*iRank* field) and subtask parameters (in *params* field). One can use communication operations described in section 5.6 to communicate between peers. In the end of this function, one must set the result of subtask in *result* field and size of result in *result_size* field. Subtask result will be sent automatically to task submitter peer.

In *ResultAggregation* function, one obtains results of all subtasks. One can manipulate results, i.e. write to an output (console, file).

B.2 Compile and run a P2PDC application

B.2.1 Compile a P2PDC application

One must compile an application as a shared library (.so in linux) and place it in Problems folder. Name of shared library is the name of the application.

B.2.2 Run a P2PDC application

Attention: One must add P2PComm and Problems folder path to $LD_LIBRARY_PATH environment variable before running P2PDC environment.

- Run the resources manager server in the Server folder:

  ./Server

- Modify IP address (or domain name) of server in Tracker/db/Server and P2PDC/data/Server files.

- Run a Tracker in Traker folder:

  ./Tracker

- Start worker in P2PDC folder:

  ./P2PDC [netif_name]

  where *netif_name* is the network interface used to communicating with others workers, e.g. eth0 or eth1.

  Remark: On NICTA and PlanetLab testbeds, workers are started automatically by OMF framework. On Grid'5000 testbed, in order to avoid starting manually a large number of workers, one can create a customized image of environment where the worker program is configured as a startup program. At the beginning of experiments, one deploys this image on machines so that worker program is started automatically on machines.

- Start submitter in P2PDC folder:

  ./P2PDC [netif_name] [-ft] [problem_name] [parameters]

  where

  - *netif_name* is the network interface used to communicating with others workers, e.g. eth0 or eth1;
  - *-ft* is the fault-tolerance option.
  - *problem_name* is the name of the problem.
  - *parameters* are parameters of the application that will be passed to *TaskDefinition* function.

Appendix C

List of publications

Papers in international conferences and workshops

1. Didier El Baz, The Tung Nguyen, **A self-adaptive communication protocol with application to high performance peer-to-peer distributed computing**, *in Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing* , p.p 327–333, Pisa, Italy, February 2010.

2. The Tung Nguyen, Didier El Baz, Pierre Spiteri, Guillaume Jourjon, Ming Chau , **High Performance Peer-to-Peer Distributed Computing with Application to Obstacle Problem**, *HOTP2P 2010 in conjunction with IEEE IPDPS 2010*, Atlanta, USA, April 2010.

3. Thierry Garcia, Ming Chau, The Tung Nguyen, Didier El-Baz, Pierre Spiteri, **Asynchronous peer-to-peer distributed computing for financial applications**, *PDSEC 2011 in conjunction with IEEE IPDPS 2011*, Anchorage, USA, May 2011.

4. Bogdan Florin Cornea, Julien Bourgeois, The Tung Nguyen, Didier El Baz, **Performance Prediction in a Decentralized Environment for Peer-to-Peer Computing**, *HOTP2P 2011 in conjunction with IEEE IPDPS 2011*, Anchorage, USA, May 2011.

Posters

1. Didier El Baz, The Tung Nguyen, Julien Bourgeois, Bogdan Florin Cornea, Pierre Spiteri, Mhand Hifi, Toufik Saadi, Nawel Haddadou, **Projet ANR-CIP : Calcul intensif pair à pair**, *Poster Colloque Ter@tec 2009*, Supélec, July 2009.

2. The Tung Nguyen, Didier El Baz, **Un protocole de communication auto-adaptatif pour le calcul intensif pair à pair**, *Poster Rempar 19*, Toulouse, September 2009.

3. Didier El Baz, The Tung Nguyen, Julien Bourgeois, Bogdan Florin Cornea, Pierre Spiteri, Thierry Garcia, Mhand Hifi, Toufik Saadi, Nawel Haddadou, **ANR 07 CIS : Calcul Intensif Pair à Pair (CIP)**, *Poster Grand Colloque STIC*, Paris, January 2010.

4. Max Ott, Guillaume Jourjon, The Tung Nguyen, Didier El Baz, **Demo of the Federation of OMF Control Framework with PlanetLab: Peer-to-peer resolution of an obstacle problem using the P2PDC framework**, *Poster 7th GENI Engineering Conference*, Durham, Georgia, USA, March 2010.

Bibliography

[Al-Dmour 2004] N.A. Al-Dmour and W.J. Teahan. *ParCop: a decentralized peer-to-peer computing system.* In Parallel and Distributed Computing, 2004. Third International Symposium on/Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2004. Third International Workshop on, pages 162 – 168, July 2004. (Cited on page 15.)

[Anderson 2004] David P. Anderson. *Boinc: A system for public-resource computing and storage.* In 5th IEEE/ACM International Workshop on Grid Computing, pages 4–10, 2004. (Cited on pages 14 and 96.)

[Andrade 2003] Nazareno Andrade, Walfredo Cirne, Francisco Brasileiro and Paulo Roisenberg. *OurGrid: An approach to easily assemble grids with equitable resource sharing.* In Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing, pages 61–68, June 2003. (Cited on page 15.)

[anr] *ANR-07-CIS7-011 web site.* http://spiderman-2.laas.fr/CIS-CIP/. (Cited on page 3.)

[Arlat 2006] Jean Arlat, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, Jean-Claude Laprie and David Powell. *Tolérance aux fautes.* In Encyclopédie de l'informatique et des systèmes d'information. Vuibert, Paris, France, 2006. (Cited on pages 92 and 93.)

[Aumage 2001] Oliver Aumage and Guillaume Mercier. *MPICH/Madeleine: a True Multi-Protocol MPI for High Performance Networks.* In 15th International Parallel and Distributed Processing Symposium (IPDPS'01), 2001. (Cited on pages 28, 51 and 119.)

[Baudet 1978] G. M. Baudet. *Asynchronous iterative methods for multiprocessors.* J. Assoc. Comput., no. 2, pages 226–244, 1978. (Cited on page 21.)

[Beaumont 2011] Olivier Beaumont, Nicolas Bonichon, Philippe Duchon and Hubert Larchevêque. *Use of Internet Embedding Tools for Heterogeneous Resources Aggregation.* In Heterogeneity in Computing Workshop (HCW) – IPDPS 2011, Anchorage, Alaska, USA, 2011. (Cited on page 38.)

[Bertsekas 1983] Dimitri P. Bertsekas. *Distributed asynchronous computation of fixed points.* Methematical Programming, pages 107–120, 1983. (Cited on page 21.)

[Bertsekas 1987] Dimitri Bertsekas and Didier El Baz. *Distributed Asynchronous Relaxation Methods for Convex Network Flow Problems.* SIAM Journal on Control and Optimization, vol. 25, no. 1, pages 74–85, 1987. (Cited on pages 21 and 62.)

[Bertsekas 1989] Dimitri P. Bertsekas and John N. Tsitsiklis. Parallel and distributed computation: numerical methods. Prentice-Hall, Inc. (republished in 1997 by Athena Scientific), Upper Saddle River, NJ, USA, 1989. (Cited on pages 17, 21, 23, 62 and 82.)

[Bertsekas 1991] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and distributed iterative algorithms: a selective survey*. Automatica, vol. 25, pages 3—21, 1991. (Cited on pages 23, 24, 82 and 120.)

[Bertsekas 1998] Dimitri P. Bertsekas. Network optimization: Continuous and discrete models. Athena Scientific, 1998. (Cited on page 17.)

[Bhaya 1991] Amit Bhaya and E. Kaszkurewicz. *Asynchronous block iterative methods for almost linear equations*. Linear algebra and its applications, vol. 154, pages 478–508, 1991. (Cited on page 22.)

[bit] *BitTorrent*. http://bitconjurer.org/BitTorrent/. (Cited on page 10.)

[Bo 2003] Chonggang Wang Bo and Bo Li. Peer-to-Peer overlay networks: A survey. Technical report, 2003. (Cited on page 10.)

[Chajakis 1991] Emmanuel D. Chajakis and Stavros A. Zenios. *Synchronous and asynchronous implementations of relaxation algorithms for nonlinear network optimization*. Parallel Comput., vol. 17, no. 8, page 873–894, October 1991. (Cited on page 23.)

[Chandy 1985] K. Mani Chandy. *Distributed Snapshots: Determining Global States of Distributed Systems*. ACM Transactions on Computer Systems, vol. 3, page 63–75, 1985. (Cited on page 24.)

[Chau 2007] Ming Chau, Didier El Baz, Ronan Guivarch and Pierre Spiteri. *MPI implementation of parallel subdomain methods for linear and nonlinear convection–diffusion problems*. Journal of Parallel and Distributed Computing, vol. 67, no. 5, page 581–591, May 2007. (Cited on page 20.)

[Chau 2011] Ming Chau, Thierry Garcia and Pierre Spiteri. *Proteins separation in distributed environment computation*. In Proceedings of the 2011 international conference on Computational science and its applications - Volume Part II, ICCSA'11, page 648–663, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on page 120.)

[Chazan 1969] D. Chazan and W. Miranker. *Chaotic relaxation*. Lenear Algebra Appl., pages 199—222, 1969. (Cited on page 21.)

[Cornea 2011] Bogdan Florin Cornea, Julien Bourgeois, The Tung Nguyen and Didier El Baz. *Performance Prediction in a Decentralized Environment for Peer-to-Peer Computing*. In IPDPS Workshops - HotP2P'11: International Workshop on Hot Topics in Peer-to-Peer Systems, Anchorage, Alaska, USA, 2011. IEEE Computer Society Press. (Cited on pages 71 and 120.)

[Dean 2004] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. OSDI, page 13, 2004. (Cited on page 16.)

[Dejan 2003] S. Milojicic Dejan, Kalogeraki Vana, Lukose Rajan, Nagaraja Kiran, Pruyne Jim, Richard Bruno, Rollins Sami and Xu Zhichen. *Peer-to-Peer Computing*. Rapport technique, 2003. (Cited on page 8.)

[Dijkstra 1980] E. Dijkstra. *Termination detection for diffusing computations*. Information Processing Letters, vol. 11, no. 1, page 1–4, August 1980. (Cited on pages 23 and 24.)

[El Baz 1990] Didier El Baz. *M-functions and Parallel Asynchronous Algorithms*. SIAM Journal on Numerical Analysis, vol. 27, no. 1, pages 136–140, 1990. (Cited on pages 21 and 62.)

[El Baz 1994] Didier El Baz. *Nonlinear systems of equations and parallel asynchronous iterative algorithms*. Advances in Parallel Computing, vol. 9, pages 89–96, 1994. (Cited on pages 21 and 62.)

[El Baz 1996a] Didier El Baz. *A method of terminating asynchronous iterative algorithms on message passing systems*. Parallel Algorithms and Applications, vol. 9, pages 153–159, 1996. (Cited on page 24.)

[El Baz 1996b] Didier El Baz, Pierre Spiteri, Jean Claude Miellou and Didier Gazen. *Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems*. Journal of Parallel and Distributed Computing, vol. 38, pages 1—15, October 1996. (Cited on pages 20, 22, 47, 48, 49 and 50.)

[El Baz 1998] Didier El Baz. *Contribution à l'algorithmique parallèle. Le concept d'asynchronisme : étude théorique, mise en œuvre et application*. Habilitation à diriger des recherches, 1998. (Cited on pages 2, 18, 22, 24, 62 and 82.)

[El Baz 2005] Didier El Baz, Andreas Frommer and Pierre Spiteri. *Asynchronous iterations with flexible communication: contracting operators*. Journal of Computational and Applied Mathematics, vol. 176, no. 1, page 91–103, April 2005. (Cited on page 20.)

[El Baz 2010] Didier El Baz and The Tung Nguyen. *A self-adaptive communication protocol with application to high performance peer to peer distributed computing*. In Proc. of the 18th Euromicro conference on Parallel, Distributed and Network-Base Processing, 2010. (Cited on page 27.)

[El Tarazi 1981] M.N. El Tarazi. *Contraction et ordre partiel pour l'étude d'algorithmes synchrones et asynchrones en analyse numérique*. Thèse d'Etat faculté des sciences et techniques de l'Université de Franche-Comté besançon, 1981. (Cited on page 21.)

[El Tarazi 1982] M.N. El Tarazi. *Some convergence results for asynchronous algorithms*. Num. Math., pages 325–340, 1982. (Cited on page 21.)

[Elnozahy 2002] E. N. Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang and David B. Johnson. *A survey of rollback-recovery protocols in message-passing systems.* ACM Computing Surveys, vol. 34, pages 375—408, September 2002. (Cited on page 93.)

[Ernst-Desmulier 2005] Jean-Baptiste Ernst-Desmulier, Julien Bourgeois, Francois Spies and Jerome Verbeke. *Adding New Features In A Peer-to-Peer Distributed Computing Framework.* In Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing, page 34–41, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 15.)

[Exposito 2003] Ernesto Exposito, Patrick Senac and Michel Diaz. *FPTP: the XQoS aware and fully programmable transport protocol.* In 11th IEEE International Conference on Networks (ICON'2003), Sydney, Australia, 2003. (Cited on page 30.)

[Felber 1999] Pascal Felber, Xavier Defago, Patrick Th. Eugster and André Schiper. *Replicating Corba Objects: A Marriage Between Active And Passive Replication.* In Proceedings of the IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems II, pages 375—388, Deventer, The Netherlands, The Netherlands, 1999. (Cited on page 92.)

[Floyd a] Sally Floyd and E. Kohler. *Profile for DCCP Congestion Control ID 2: TCP-like Congestion Control.* RFC 4341. (Cited on page 43.)

[Floyd b] Sally Floyd and E. Kohler. *Profile for DCCP Congestion Control ID 4: TCP-Friendly Rate Control for Small Packets (TFRC-SP).* FRC 4828. (Cited on page 43.)

[Floyd c] Sally Floyd, E. Kohler and J Padhye. *Profile for DCCP Congestion Control ID 3: TCP-Friendly Rate Control (TFRC).* RFC 4342. (Cited on page 43.)

[Floyd 1999] Sally Floyd and T. Henderson. *The NewReno Modification to TCP's Fast Recovery Algorithm.* RFC 2582, 1999. (Cited on pages 38 and 40.)

[fol] *Folding@home.* http://folding.stanford.edu/. (Cited on page 13.)

[Foster 1996] Ian Foster and Carl Kesselman. *Globus: A Metacomputing Infrastructure Toolkit.* International Journal of Supercomputer Applications, vol. 11, page 115–128, 1996. (Cited on page 13.)

[fre] *The FreeNet Network Project.* http://freenet.sourceforge.net. (Cited on pages 1 and 7.)

[Frommer 1997] Andreas Frommer and Hartmut Schwandt. *A Unified Representation and Theory of Algebraic Additive Schwarz and Multisplitting Methods.* SIAM J. Matrix Anal. Appl., vol. 18, no. 4, page 893–912, October 1997. (Cited on page 21.)

[Garcia 2011] Thierry Garcia, Ming Chau, The Tung Nguyen, Didier El Baz and Pierre Spiteri. *Asynchronous peer-to-peer distributed computing for financial applications*. In The 12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC'11), Anchorage, Alaska, USA, 2011. (Cited on page 120.)

[gen] *Genome@home*. http://genomeathome.stanford.edu. (Cited on page 13.)

[Genaud 2009] Stephane Genaud and Choopan Rattanapoka. *A Peer-to-Peer Framework for Message Passing Parallel Programs*. volume 17 of *Advances in Parallel Computing*, pages 118–147. IOS Press, June 2009. (Cited on pages 16 and 93.)

[Giraud 1991] Luc Giraud and Pierre Spitéri. *Résolution parallèle de problèmes aux limites non linéaires*. Modélisation mathématique et analyse numérique, vol. 25, no. 5, pages 579–606, 1991. (Cited on page 62.)

[gnu] *Gnutella Protocol Development*. http://rfc.gnutella.sourceforge.net. (Cited on pages 1, 7, 8 and 11.)

[gri] *Grid5000 platform*. http://www.grid5000.fr. (Cited on pages 80 and 85.)

[Hifi 2011] Mhand Hifi, Toufik Saadi and Nawel Haddadou. *High Performance Peer-to-Peer Distributed Computing with Application to Constrained Two-Dimensional Guillotine Cutting Problem*. In Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP '11, page 552–559, Washington, DC, USA, 2011. IEEE Computer Society. (Cited on page 120.)

[Hiltunen 2000] Matti A. Hiltunen and Richard D. Schlichting. *The Cactus Approach to Building Configurable Middleware Services*. In Proceedings of the Workshop on Dependable System Middleware and Group Communication, Nuremberg, Germany, October 2000. (Cited on pages 3, 28, 30 and 31.)

[Huffaker 2002] Bradley Huffaker, Marina Fomenkov, Daniel J. Plummer, David Moore and K Claffy. *Distance Metrics in the Internet*. In in IEEE International Telecommunications Symposium, pages 200–202, 2002. (Cited on page 74.)

[Hutchison 1991] Norm Hutchison and Larry L. Peterson. *The x-kernel: An architecture for implementing network protocols*. In IEEE Transactions on Software Engineering, volume 17, pages 64–76, 1991. (Cited on pages 29 and 31.)

[Jourjon 2005] Guillaume Jourjon and Didier El Baz. *Some solutions for Peer to Peer Global Computing*. In Proc. of the 13th Euromicro conference on Parallel, Distributed and Network-Base Processing, pages 49–58, 2005. (Cited on page 9.)

[Jourjon 2010] G. Jourjon, T. Rakotoarivelo and M. Ott. *From Learning to Researching, Ease the Shift through Testbeds*. In Proc. of TridentCom 2010, volume 46 of *LNICST*, pages 496–505, Berlin Heidelberg, May 2010. Springer-Verlag. (Cited on page 110.)

[Jourjon 2011] Guillaume Jourjon, Thierry Rakotoarivelo and Max Ott. *A Portal to Support Rigorous Experimental Methodology in Networking Research*. In TridentCom 2011, April 2011. (Cited on pages 6 and 110.)

[jxt] *JXTA project*. http://java.net/projects/jxta. (Cited on page 14.)

[Kaszkurewicz 1990] E. Kaszkurewicz and A. Bhaya. *On the convergence of parallel asynchronous block-iterative computations*. Linear Algebra and its Application, vol. 131, pages 139–160, 1990. (Cited on page 22.)

[kaz] *KaZaA*. http://www.kazaa.com. (Cited on page 12.)

[Kohler 1999] E. Kohler, M. Handley and Sally Floyd. *Datagram Congestion Control Protocol (DCCP)*. RFC 2582, 1999. (Cited on pages 2, 4, 39 and 43.)

[Lee 2011] Kyungyong Lee, Tae Woong Choi, Arijit Ganguly, David I. Wolinsky, P. Oscar Boykin and Renato Figueiredo. *Parallel Processing Framework on a P2P System Using Map and Reduce Primitives*. In Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, pages 1602 –1609, May 2011. (Cited on page 16.)

[Lions 2002] Jacques-Louis Lions. Quelques méthodes de résolution des problèmes aux limites non linéaires. Dunod, 2002. (Cited on page 60.)

[Litzkow 1988] M. J Litzkow, M. Livny and M. W Mutka. *Condor-a hunter of idle workstations*. In Distributed Computing Systems, 1988., 8th International Conference on, pages 104 –111, June 1988. (Cited on pages 13 and 93.)

[Lua 2005] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma and Steven Lim. *A survey and comparison of peer-to-peer overlay network schemes*. IEEE Communications Surveys and Tutorials, vol. 7, no. 1-4, page 72–93, 2005. (Cited on page 10.)

[Lubachevsky 1986] Boris Lubachevsky and Debasis Mitra. *A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius*. J. ACM, vol. 33, no. 1, page 130–150, January 1986. (Cited on page 21.)

[Luenberger 1973] David G. Luenberger. Introduction to linear and nonlinear programming. Addison Wesley Publishing Company, 1973. (Cited on page 17.)

[Magoulès 2009] Frédéric Magoulès, Jie Pan, Kiat-An Tan and Abhinit Kumar. Introduction to grid computing, volume 10. CRC Press, 2009. (Cited on page 13.)

[Mehani 2011] Olivier Mehani, Guillaume Jourjon, Jolyon White, Thierry Rakotoarivelo, Roksana Boreli and Thierry Ernst. *Characterisation of the Effect of a Measurement Library on the Performance of Instrumented Tools.* Rapport technique, NICTA, Sydney, Australia, May 2011. (Cited on page 108.)

[Miellou 1975] Jean Claude Miellou. *Algorithmes de relaxation chaotique à retards.* R.A.I.R.O, no. 1, pages 55–82, 1975. (Cited on pages 21 and 25.)

[Miellou 1985a] Jean Claude Miellou and Pierre Spitéri. *Two criteria for the convergence of asynchronous iterations.* In Computers and computing, pages 91–95, 1985. (Cited on pages 60 and 62.)

[Miellou 1985b] Jean Claude Miellou and Pierre Spitéri. *Un critère de convergence pour des methodes generales de point fixe.* Modélisation mathématique et analyse numérique, vol. 19, no. 4, pages 645–669, 1985. (Cited on pages 21 and 62.)

[Miellou 1989] Jean Claude Miellou and D. J. Evans. *Stopping criteria for parallel asynchronous algorithms.* Computer Studies 503, Loughborough University of Technology, 1989. (Cited on page 23.)

[Miellou 1990] Jean Claude Miellou, Ph. Cortey-Dumont and M. Boulbrachêne. *Perturbation of fixed-point iterative methods.* Advances in Parallel Computing, vol. 1, pages 81–122, 1990. (Cited on page 25.)

[Miellou 1998] J. C. Miellou, D. El Baz and P. Spiteri. *A new class of asynchronous iterative algorithms with order intervals.* Mathematics of Computation, vol. 67, page 237–255, 1998. (Cited on pages 20 and 22.)

[Miranda 2001] Hugo Miranda, Alexandre Pinto and Luis Rodrigues. *Appia, a Flexible Protocol Kernel Supporting Multiple Coordinated Channels.* In Proc. 21st International conference on Distributed Computing Systems, pages 707—710, 2001. (Cited on page 29.)

[Mocito 2005] Jose Mocito, Liliana Rosa, Nuno Almeida, Hugo Miranda, Luis Rodrigues and Antonia Lopes. *Context Adaptation of the Communication Stack.* In Proceedings of the Third International Workshop on Mobile Distributed Computing, 2005. (Cited on page 29.)

[nap] *Napster.* http://www.napster.com. (Cited on pages 8 and 10.)

[Nguyen 2010] The Tung Nguyen, Didier El Baz, Pierre Spiteri, Guillaume Jourjon and Minh Chau. *High Performance Peer-to-Peer Distributed Computing with Application to Obstacle Problem.* In Proceedings of HOTP2P/IPDPS2010, 2010. (Cited on page 53.)

[nic] *NICTA testbed.* http://www.nicta.com.au. (Cited on page 65.)

[omf] *OMF Web Page.* http://omf.mytestbed.net/. (Cited on pages 65 and 123.)

[Oram 2001] Andy Oram. Peer-to-Peer: harnessing the power of disruptive technologies. O'Reilly Media, February 2001. (Cited on page 8.)

[Ortega 1970] James M. Ortega and Werner C. Rheinboldt. Iterative solution of nonlinear equations in several variables. SIAM, Philadelphia, PA, USA, 1970. (Cited on pages 17 and 18.)

[Ott 2010] Max Ott, Guillaume Jourjon, The Tung Nguyen and Didier El Baz. *Demo of the Federation of OMF Control Framework with PlanetLab: Peer-to-peer resolution of an obstacle problem using the P2PDC framework*. In Poster 7th GENI Engineering Conference, Durham, Georgia, USA, March 2010. (Cited on pages 112 and 120.)

[Owezarski 2008] Philippe Owezarski, Pascal Berthou, Yann Labit and David Gauchard. *LaasNetExp: a generic polymorphic platform for network emulation and experiments*. In Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities, pages 24:1—24:9, Innsbruck, Austria, 2008. (Cited on page 49.)

[Paweł 2004] Wojciechowski Paweł, Rütti Olivier and Schiper André. *SAMOA: Framework for Synchronisation Augmented Microprotocol Approach*. In Proceedings of the 18th International Parallel and Distributed Processing Symposium, Santa Fe, New Mexico, 2004. (Cited on page 30.)

[r] *The R Project for Statistical Computing*. at: www.r-project.org/. (Cited on page 110.)

[Rakotoarivelo 2009] Thierry Rakotoarivelo, Maximilian Ott, Guillaume Jourjon and Ivan Seskar. *OMF: a control and management framework for networking testbeds*. In SOSP Workshop on Real Overlays and Distributed Systems, pages 54—59, New York, NY, USA, 2009. (Cited on pages 65 and 123.)

[Rakotoarivelo 2010] Thierry Rakotoarivelo, Maximilian Ott, Guillaume Jourjon and Ivan Seskar. *OMF: A Control and Management Framework for Networking Nestbeds*. SIGOPS Operating Systems Review, vol. 43, no. 4, January 2010. (Cited on pages 6, 108 and 109.)

[Ratnasamy 2001] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp and Scott Shenker. *A scalable content-addressable network*. In Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, pages 161—172, New York, NY, USA, 2001. (Cited on pages 12 and 72.)

[Reddy 2006] M. Venkateswara Reddy, A. Vijay Srinivas, Tarun Gopinath and D. Janakiram. *Vishwa: A reconfigurable P2P middleware for Grid Computations*. In International Conference on Parallel Processing, page 381–390, 2006. (Cited on pages 16 and 96.)

[Robert 1969] François Robert. *Blocs-H-matrices et convergence des methodes iteratives classiques par blocs.* Linear Algebra and Application, pages 223–265, 1969. (Cited on page 25.)

[Robert 1975] François Robert, Michel Charnay and François Musy. *Itérations chaotiques série-parallèle pour des équations non-linéaires de point fixe.* Applications of Mathematics, vol. 20, no. 1, pages 1–38, 1975. (Cited on page 25.)

[Rowstron 2001] Antony I. T. Rowstron and Robert Druschel. *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems.* In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, pages 329—350, 2001. (Cited on pages 72 and 77.)

[rub] *Ruby official site.* http://www.ruby-lang.org. (Cited on page 123.)

[Sathya 2010] S. Siva Sathya and K. Syam Babu. *Survey of fault tolerant techniques for grid.* Computer Science Review, vol. 4, pages 101—120, 2010. (Cited on page 92.)

[Savari 1996] S. A. Savari and D. P. Bertsekas. *Finite termination of asynchronous iterative algorithms.* Parallel Comput., vol. 22, no. 1, page 39–56, January 1996. (Cited on pages 24 and 25.)

[SCT 2000] *Stream Control Transmission Protocol.* RFC 2690, 2000. (Cited on page 2.)

[set] *Seti@home.* http://setiathome.berkeley.edu/. (Cited on pages 13 and 93.)

[Spitéri 2002] Pierre Spitéri and Ming Chau. *Parallel Asynchronous Richardson Method for the Solution of Obstacle Problem.* In Proc. of the 16th Annual International Symposium on High Performance Computing Systems and Applications, pages 133–138, Moncton, Canada, 2002. (Cited on pages 5, 60, 61, 62 and 63.)

[Stoica 2003] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek and Hari Balakrishnan. *Chord: a scalable peer-to-peer lookup protocol for internet applications.* Journal IEEE/ACM Transactions on Networking, vol. 11, no. 1, pages 17—32, February 2003. (Cited on pages 12, 72 and 77.)

[Szyld 1998] Daniel B. Szyld. *Different Models Of Parallel Asynchronous Iterations With Overlapping Blocks.* Computational and applied mathematics, vol. 17, page 101–115, 1998. (Cited on page 21.)

[TCP 1981] *Transmission Control Protocol.* RFC 793, 1981. (Cited on pages 2 and 28.)

[top] *TOP500.* http://www.top500.org/. (Cited on page 1.)

[Treaster 2005] Michael Treaster. *A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems.* ACM Computing Research Repository, vol. 501002, pages 1—11, 2005. (Cited on page 92.)

[Tsitsiklis 1987] J. N. Tsitsiklis. *On the Stability of Asynchronous Iterative Processes.* Mathematical Systems Theory, vol. 20, page 137–153, 1987. (Cited on page 22.)

[UDP 1980] *User Datagram Protoco.* RFC 768, 1980. (Cited on pages 2 and 28.)

[Van Wambeke 2008] Nicolas Van Wambeke, François Armando, Christophe Chassot and Ernesto Expósito. *A model-based approach for self-adaptive Transport protocols.* Journal Computer Communications, vol. 31, no. 11, page 2699–2705, July 2008. (Cited on page 28.)

[Varga 1962] Richard S Varga. Matrix iterative analysis. Prentice Hall, 1962. (Cited on page 61.)

[Venet 2010] Cédric Venet. *Numerical methods for acoustic simulation of large-scale problems.* PhD thesis, Ecole centrale de Paris, 2010. (Cited on page 21.)

[Verbeke 2002] Jerome Verbeke, Neelakanth Nadgir, Greg Ruetsch and Ilya Sharapov. *Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment.* In Proceedings of the 3rd International Workshop on Grid Computing, page 1–12. Springer-Verlag, 2002. (Cited on page 14.)

[White 2010] Jolyon White, Guillaume Jourjon, Thierry Rakotoarivelo and Max Ott. *Measurement Architectures for Network Experiments with Disconnected Mobile Nodes.* In TridentCom 2010, May 2010. (Cited on pages 6, 107 and 113.)

[wik] *Wikipedia.* http://www.wikipedia.org/. (Cited on page 8.)

[Wong 2001] Gary T. Wong, Matti A. Hiltunen and Richard D. Schlichting. *A Configurable and Extensible Transport Protocol.* In Proceedings of IEEE INFOCOM, pages 319–328, 2001. (Cited on pages 28, 32 and 33.)

[xtr] *XtremWeb.* http://www.xtremweb.net. (Cited on pages 14 and 96.)

[Zhao 2006] Jia Zhao and Jian-De Lu. *Solving Overlay Mismatching of Unstructured P2P Networks using Physical Locality Information.* In Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing, pages 75–76, Washington, DC, USA, 2006. (Cited on page 74.)