Faculté des sciences appliquées
Département d'électricité, électronique et informatique
Institut Montefiore

Design and Implementation of a Distributed Lattice Boltzmann-based Fluid Flow Simulation Tool

Thèse présentée par
Gérard Dethier
en vue de l'obtention du grade de
Docteur en Sciences (orientation Informatique)

Octobre 2010

Université
de Liège

Faculté des sciences appliquées
Département d'électricité, électronique et informatique
Institut Montefiore

Design and Implementation of a Distributed Lattice Boltzmann-based Fluid Flow Simulation Tool

Thèse présentée par
Gérard Dethier
en vue de l'obtention du grade de
Docteur en Sciences (orientation Informatique)

Octobre 2010

Résumé

Les simulations basées sur les méthodes de Lattice Boltzmann sont bien adaptées aux simulations d'écoulements de fluides à l'intérieur de structures complexes rencontrées en génie chimique, telles que les milieux poreux ou les empilements structurés utilisés dans des colonnes de distillation et de distillation réactive. Elles requièrent toutefois de grandes quantités de mémoire (environ 10 gigaoctets). Par ailleurs, leur exécution sur un seul ordinateur de bureau puissant nécessiterait un temps très long (environ deux ans).

Il est possible de réduire à la fois le temps d'exécution et la quantité de mémoire requise par ordinateur en exécutant les simulations LB de manière distribuée, par exemple en utilisant un cluster. Un Cluster Hétérogène Dynamique (CHD) est une classe de clusters impliquant des ordinateurs qui sont interconnectés au moyen d'un réseau local, qui ne sont pas nécessairement fiables et qui ne partagent pas la même architecture, le même système d'exploitation, la même puissance de calcul, etc. En revanche, les CHD sont faciles à installer, à étendre et peu coûteux.

Concevoir et développer un logiciel capable de gérer des CHD à grande échelle de façon efficace, extensible et robuste et capable d'effectuer des simulations LB à très grande échelle constitue un défi. L'hétérogénéité de la puissance de calcul doit être prise en compte afin d'éviter que certains ordinateurs soient débordés et ralentissent le temps global d'exécution. En outre, une panne d'un ou de plusieurs ordinateurs pendant l'exécution d'une simulation ne devrait pas empêcher son achèvement.

Dans le contexte de cette thèse, un outil de simulation appelé LaBoGrid a été conçu. LaBoGrid utilise des outils existants de répartition statique de la charge et implémente une méthode originale de répartition dynamique de la charge, ce qui lui permet de distribuer une simulation LB de manière à minimiser son temps d'exécution. De plus, un mécanisme distribué et extensible de tolérance aux pannes, fondé sur une sauvegarde régulière de l'état de simulation, est proposé. Enfin, LaBoGrid se base sur un modèle distribué de type « maître-esclaves » qui est robuste et potentiellement extensible.

Abstract

Lattice Boltzmann-based (LB) simulations are well suited to the simulation of fluid flows in complex structures encountered in chemical engineering like porous media or structured packing used in distillation and reactive distillation columns. These simulations require large amounts of memory (around 10 gigabytes) and would require very long execution times (around 2 years) if executed on a single powerful desktop computer.

The execution of LB simulations in a distributed way (for example, using cluster computing) can decrease the execution time and reduces the memory requirements for each computer. Dynamic Heterogeneous Clusters (DHC) is a class of clusters involving computers inter-connected by a local area network; these computers are potentially unreliable and do not share the same architecture, operating system, computational power, etc. However, DHCs are easy to setup and extend, and are made of affordable computers.

The design and development of a software system which organizes large scale DHCs in an efficient, scalable and robust way for implementing very large scale LB simulations is challenging. In order to avoid that some computers are overloaded and slow down the overall execution, the heterogeneity of computational power should be taken into account. In addition, the failure of one or several computers during the execution of a simulation should not prevent its completion.

In the context of this thesis, a simulation tool called LaBoGrid was designed. It uses existing static load balancing tools and implements an original dynamic load balancing method in order to distribute the simulation in a way that minimizes its execution time. In addition, a distributed and scalable fault-tolerance mechanism based on the regular saving of simulation's state is proposed. Finally, LaBoGrid is based on a distributed master-slave model that is robust and potentially scalable.

Remerciements

Je voudrais remercier :

Mon promoteur, P.A. de Marneffe, pour sa confiance et ses conseils avisés tout au long de ces années de doctorat.

P. Marchot, M. Crine et D. Toye, de m'avoir accepté dans leur équipe pour ce projet multi-disciplinaire mêlant chimie, physique et informatique.

Les membres de mon jury, B. Boigelot, P. Gribomont, P. Manneback, J. H. Piater et M. Pirotton, d'avoir pris le temps de lire et de critiquer ce travail.

Mes collègues chimistes, Saïd Aferka, Sébastien Calvo et Djomice Beugre, pour notre travail commun sur ce projet (et autres activités ludiques associées).

Cyril Briquet, pour ses coups de pouce décisifs au début et tout au long du doctorat, pour le travail de recherche commun et pour tous les bons conseils prodigués sans retenue.

Mes collègues algorithmiciens, Xavier Dalem, Thomas Leuther, Hajar Siar, Sébastien Jodogne, Cédric Thiernesse et Marie-Thérèse Ratz, pour l'excellente ambiance de travail qu'ils ont contribué à créer au sein du service et tous les bons moments, pas nécessairement professionnels, passés ensemble.

Mes amis musiciens et non-musiciens, d'avoir toléré ma mauvaise humeur quasi chronique en cette période de fin de rédaction.

Gisela Henn, pour sa relecture attentive et ses conseils en langue anglaise (désolé pour les maux de tête).

Mes parents, grands-parents, parrain, marraine, frère, soeurs, oncles, tantes, cousins, cousines, etc. pour leur soutien et les moments de détente que nous avons passés ensemble et dont j'ai bien eu besoin tout au long de ces cinq dernières années. Je propose que l'on fête ça.

Enfin, je voudrais remercier Pascale, ma compagne, pour son soutien de tous les jours, voire de toutes les heures, le bien-être et la stabilité qu'elle m'a apportés lors de ces derniers mois. Les choses auraient probablement été plus difficiles sans toi, merci !

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Computational Fluid Dynamics (CFD) is a branch of fluid mechanics that uses numerical methods to study the motion of fluid flows by solving Navier-Stokes equations. Lattice Boltzmann (LB) methods is a class of CFD methods well suited to the simulation of flows in complex structures like porous media or packed beds in distillation columns. LB simulations (i.e. simulations based on LB methods) imply the update of real values stored in a multidimensional array during a given number of time steps.

The multidimensional array used in the context of LB simulations generally contains several millions of real values. An LB simulation can therefore require around 10 gigabytes of memory. In addition, it is not unusual to run a simulation for more than 10000 time steps. In this context, if we suppose that a single computer with a Pentium IV 3GHz processor is equipped with enough memory, it may take around 2 years to complete the simulation's execution.

The execution of LB simulations in a distributed way (i.e. by using several computers) can decrease the execution time. In addition, the amount of memory required on each computer is smaller than for a single computer. However, the computers of the distributed execution environment (i.e. that contribute to the execution of a simulation) may neither be reliable nor share the same architecture, computational power (i.e. the maximum number of instructions that can be executed by time unit), Operating System (OS), etc. In addition, the probability of failure increases with the number of computers. Finally, when the number of computers executing the simulation is large (more than 100 computers), scalability problems may arise.

1.1 Thesis Statement

In this thesis, we show that it is possible to design and develop a software system which organizes large scale clusters of inherently unreliable computers in an efficient, scalable and robust way for implementing very large scale LB simulations.

1.2 Overview

Simulations based on LB methods can require large amounts of memory (several gigabytes). In addition, if executed using a single processor, a simulation can last for months on a powerful desktop computer. The parallelization of LB simulations can accelerate the execution because several simulation processes are then executed by several processors (one per processor) at the same time.

In order to execute large parallel simulations, a supercomputer with many identical processors and a large amount of memory could be used. However, this kind of system is sometimes out of reach from financial and/or technical point of views.

An alternative can be found in clusters made of several affordable desktop computers inter-connected by a simple Local Area Network (LAN). The memory and the computational power of such systems can easily be increased by adding computers to the cluster.

However, the availability of the computers of the cluster can be limited. Following scenario [1] illustrates this property: 50 desktop computers are used to run LB simulations. However, 2 days a week, 20 of these computers are reserved for student works and cannot run simulations at the same time. As a result, from simulation point of view at least, the cluster is composed of 50 computers 5 days a week and 30 computers the two remaining days.

In addition, the cluster can be made of computers that do not have the same processor architecture, OS or computational power. In addition, the reliability of used computers can be limited because of faulty hardware and/or faulty software. The execution of simulation processes can therefore be interrupted.

We call this kind of systems *dynamic heterogeneous clusters*: “dynamic” because the computers availability changes over time (even during the computing

[1] This scenario is inspired from a real cluster of computers managed by the algorithmics and software engineering laboratory of University of Liège (ULg) and shared between students for their practical works and researchers of the EECS department of the ULg.

of one simulation) and "heterogeneous" because of the variety of architectures, OSes, etc.

The efficient execution of distributed LB simulations on dynamic heterogeneous clusters is challenging. The following paragraphs describe the problems that need to be solved.

The heterogeneous computational power of available computers must be taken into account in order to reduce the execution time. If it is not the case (in particular with an homogeneous distribution of the work among computers), some computers take more time than others to execute a time step of the simulation (this impairs on the efficient use of the system by introducing waiting synchronization delays on the faster computers). By moving work from slower computers to faster ones, the global execution time can be reduced. *Load balancing* methods should therefore be used to compute the distribution of work among the cluster's computers. The problem of computing the work distribution before the application is executed is called *static load balancing*.

Another problem is that the limited availability and reliability of the computers causes the number of executed simulation processes to vary over time. The interruption of a simulation process prevents the completion of the simulation. In order to avoid this, a *fault-tolerance* mechanism must be implemented to make distributed LB simulations *fault-tolerant*.

When a simulation process is interrupted, the work that was assigned to it must be redistributed to the remaining processes. Similarly, when a new process is available, work should be redistributed in order to use the newly available resource. Work redistribution can therefore occur several times during application's execution. This problem is called *dynamic load balancing* and arises additional constraints regarding static load balancing:

- work distribution computation must be fast and have a modest memory usage,
- the amount of work (and associated data) moved between processes must be minimized.

First constraint stems from the fact that work distribution is executed in parallel of the application. Second constraint minimizes the work migration time and, therefore, the total execution time of the distributed application.

We implemented distributed LB simulations using a master-slave model. The slaves processes execute the simulation and are managed by a master process that:

- distributes the work among the slaves (possibly several times during the simulation's execution),
- detects and handles slaves interruptions and/or arrivals,
- provides required data for simulations,
- retrieves the results.

This model has two drawbacks:

- it is not robust because the master process is a single point of failure,
- it is not scalable because the master process may act like a bottleneck when the number of slaves is large.

To solve these drawbacks, or at least reduce their importance, another model based on a robust distributed implementation of the master process is suggested. This model is based on a self-healing tree organization of the processes. It can be seen as a generalization of the master-slave model where several masters are organized into a tree and manage each a set of slaves.

In the context of this thesis, we designed a simulation tool called LaBoGrid. LaBoGrid implements a fault-tolerance mechanism as well as static and dynamic load balancing methods. It is based on an original framework facilitating the development of master-slave-based distributed applications. This framework provides tools to implement the master process in a robust, scalable and distributed way. LaBoGrid is therefore able to execute LB simulations in an efficient, potentially scalable and robust way in the context of dynamic heterogeneous clusters.

In addition, LaBoGrid has following properties:

- extensible: LaBoGrid is used in an experimental context, new simulation methods or algorithms may have to be integrated regularly.
- easy to use and extend: the typical user of LaBoGrid may not have high programming skills.

1.3 Outline

Chapter 2 describes LB methods and a simple sequential implementation of LB simulations. A first parallel implementation of LB simulations is given and its parallel efficiency is evaluated.

Chapter 3 suggests optimizations for the sequential implementation. Attempts to improve the efficiency of the parallel implementation are also presented.

Chapter 4 presents the architecture of LaBoGrid, in particular the generic framework facilitating the development of master-slave-based distributed applications.

Chapter 5 shows how existing static load balancing tools can be used to reduce the execution time of LB simulations executed by computers with heterogeneous computational powers.

Chapter 6 explains how to implement fault-tolerant distributed LB simulations.

Chapter 7 introduces an original dynamic load balancing method.

Chapter 8 addresses the robust, scalable and distributed implementation of LaBoGrid's master.

Finally, Chapter 9 concludes this document.

1.4 Context

This thesis has been conducted in the frame of a project[2] involving the laboratory of chemical engineering (LGC[3]) and the laboratory of algorithmics and software engineering (AIL[4]) of the University of Liège (ULg).

The LGC owns X-ray tomography systems giving access to complex geometries found, for example, in porous media like metallic foams or packed beds in distillation columns. In particular, an original system able to scan columns of up to 3 meters height and 0.5 meter diameter with a resolution of 0.4 millimeter is used. Another tomography system of the LGC is able to scan objects holding in a cube of 5 centimeters with a resolution of 5 micrometers.

The goal of the project including this thesis is to create new experimental and simulation-based tools to better characterize fluid flows in this kind of structures. Produced tools must take into account structures' complexity and the fact that flows may be turbulent.

Four theses were produced in the context of the project. Saïd Aferka [12] and Sébastien

[2] Action de Recherche Concertée (ARC) n° 05/10-334 : Techniques expérimentales avancées et modélisation par automates cellulaire des systèmes multiphasiques : Application aux colonnes de distillation et de distillation réactive.

[3] Laboratoire de Génie Chimique.

[4] Service d'Algorithmique et d'Ingénierie du Logiciel.

Calvo [23] mainly addressed the experimental aspects. The thesis of Djomice Beugre [18] details the LB model used to describe fluid flows in complex structures. Finally, this thesis describes the design and the implementation of a simulation tool based on the flow model defined by Djomice Beugre.

The work of Cyril Briquet [21] on grid computing (a form of distributed computing based on "coordinated resource sharing and problem solving in dynamic, multi-institutional collaborations" [59]) and particularly on peer-to-peer (P2P) grid computing [21, 16, 29] (a recent subdomain of grid computing based on the fully decentralized organization of resource sharing based on the "exchange" of computing time) suggested the potential of this kind of system to gather large amounts of computational power by giving access to many unreliable computers.

The obtained execution environment is similar to dynamic heterogeneous clusters except that the computers are potentially connected through the Internet. In Chapter 6, we describe how LaBoGrid can be executed by CanoPeer [1], the P2P grid computing middleware designed by Cyril Briquet, and we show therefore the potential of using a grid computing middleware to execute distributed LB simulations.

Chapter 2

Lattice Boltzmann Methods And Their Implementation

2.1 Introduction

A viscous fluid flow is described by Navier-Stokes (N-S) equations. A solution to these equations is called a *velocity field* and describes the velocity of the fluid at points in space and time.

Computational Fluid Dynamics (CFD) is a branch of fluid mechanics that uses numerical methods to study fluid flows by solving N-S equations. For this purpose, fluids (liquid or gas) flowing through solid structures are simulated. Computers can only handle a finite amount of values with a finite precision. A bounded spatial domain is therefore discretized into connected cells forming a mesh or grid.

Most CFD methods solve the motion equations at the points defined by the mesh. Lattice Boltzmann (LB) methods use an alternative approach: fluid is described by fictitious particles moving on a regular grid. These particles then collide with each other or against solid obstacles at the cells of the mesh.

LB methods are particularly interesting for simulating fluid flows in complex boundaries encountered in chemical engineering like porous media or packed beds in distillation columns [18]. However, LB simulations may require large amounts of memory (10 gigabytes) and run for months if executed on a single powerful desktop computer. The distributed implementation of LB simulations can decrease the execution time and reduce the memory requirements for each computer.

A possible execution environment are heterogeneous clusters (see Chapter 1). However, the execution of distributed applications in this kind of environment arises a potential portability issue: the implementation is executed on computers that potentially do not share same architecture, OS, etc. An implementation of LB simulations executed in the context of a heterogeneous cluster should therefore solve this problem.

2.1.1 Chapter Outline

LB methods are briefly introduced in Section 2.2. A complete and precise presentation of LB methods is out of the scope of this thesis. The content of this section directly depends on the work of Djomice Beugre [18].

The portability issue and the choice of the programming language used for our implementation is discussed in Section 2.3.

A basic sequential implementation of LB simulations is presented in Section 2.4.

A parallel implementation of LB simulations is described in Section 2.5.

The parallel efficiency of the parallel implementation presented in Section 2.5 is evaluated and discussed in Section 2.6.

Finally, Section 2.7 concludes this chapter.

2.2 Lattice Boltzmann Methods

2.2.1 Definitions

In LB methods, the spatial domain ($\mathbb{R}^3$) is discretized into a regular grid called *lattice*. Each node of the lattice is called a *site* and is associated to a point in space. If the point associated to a site is part of a solid, the site is an *obstacle*. Time is also regularly discretized. Finally, the set of possible velocities for a particle ($\mathbb{R}^3$) is reduced to a set of q 3 dimensions velocity vectors $\mathbf{v}_i$ with $i = 0, 1, \ldots, q-1$. These vectors are defined such as a particle at point $\mathbf{p}$ in space at time t and moving according to a velocity $\mathbf{v}_i$ is at point $\mathbf{p} + \mathbf{v}_i$ at time $t + \Delta t$ where Δt is the time sampling period.

A site has a *position* $\mathbf{x}$ in the lattice with $\mathbf{x} \in \mathbb{Z}^3$. Function $s : \mathbb{Z}^3 \rightarrow \{true, false\}$ is used to define the nature of a site: if $s(\mathbf{x})$ is *true* then the site at position $\mathbf{x}$ is an obstacle. Otherwise, the site is not an obstacle.

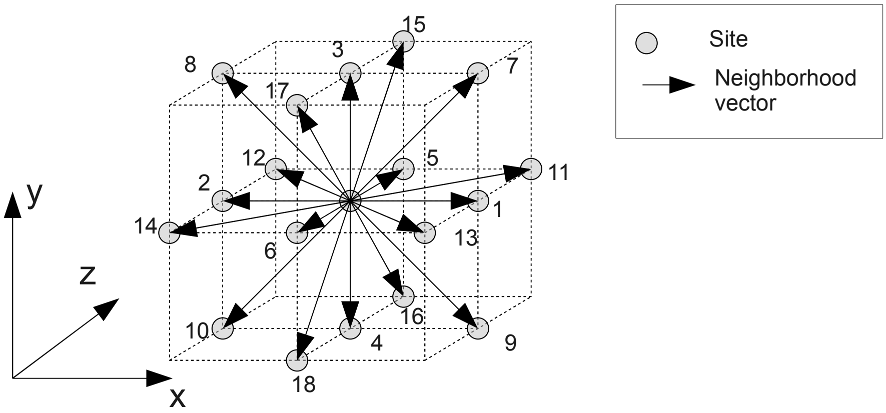

Figure 2.1: A site (center of the cube) of a D3Q19 lattice and its neighborhood. The numbers are identifiers of the neighborhood vectors (i is the identifier of vector $\mathbf{n}_i$).

A set of q *neighborhood vectors* $\mathbf{n}_i$ with $i = 0, 1, \ldots, q-1$ and $\mathbf{n}_i \in \mathbb{Z}^3$ is defined such as $\mathbf{x} + \mathbf{n}_i$ is the position of another site of the lattice. $\mathbf{n}_i$ has the same direction as $\mathbf{v}_i$ and is defined such as a particle at the point associated to position $\mathbf{x}$ at discrete time t and moving according to velocity $\mathbf{v}_i$ is at the point associated to position $\mathbf{x} + \mathbf{n}_i$ at discrete time $t + 1$.

The *neighborhood* of a site at position $\mathbf{x}$ is the set of q sites at positions $\mathbf{x} + \mathbf{n}_i$ with $i = 0, 1, \ldots, q-1$. The neighborhood of the site may contain the site itself if one of the neighborhood vectors is equal to the zero vector (noted $\mathbf{0}$).

Lattices used in the context of LB methods are generally classified using the notation DdQq where d is the number of dimensions and q the number of velocity vectors. Note that for all lattices used in LB methods, for any velocity $\mathbf{v}_i$, $\mathbf{v}_j$ exists such as $\mathbf{v}_i + \mathbf{v}_j = \mathbf{0}$. Similarly, for any neighborhood vector $\mathbf{n}_i$, $\mathbf{n}_j$ exists such as $\mathbf{n}_i + \mathbf{n}_j = \mathbf{0}$.

Figure 2.1 shows a site of a D3Q19 lattice and its neighbors. On the figure, only 18 neighborhood vectors are represented. The last neighborhood vector being zero vector: the site is also a neighbor of itself. Table 2.1 shows the 19 neighborhood vectors of a D3Q19 lattice.

LB methods are based on the following equation:

$$f_i(\mathbf{x} + \mathbf{n}_i, t + 1) = f_i(\mathbf{x}, t) + \Omega_i \quad i = 0, 1, \ldots, q-1 \tag{2.1}$$

where $\mathbf{x}$ is the position of a site, $\mathbf{n}_i$ a neighborhood vector and t discrete time. $f_i : \mathbb{Z}^3 \times \mathbb{Z} \to \mathbb{R}$ is called the *particle distribution function*.

i	n_i	i	n_i
0	$(0,0,0)$	9	$(1,-1,0)$
1	$(1,0,0)$	10	$(-1,-1,0)$
2	$(-1,0,0)$	11	$(1,0,1)$
3	$(0,1,0)$	12	$(-1,0,1)$
4	$(0,-1,0)$	13	$(1,0,-1)$
5	$(0,0,1)$	14	$(-1,0,-1)$
6	$(0,0,-1)$	15	$(0,1,1)$
7	$(1,1,0)$	16	$(0,-1,1)$
8	$(-1,1,0)$	17	$(0,1,-1)$
		18	$(0,-1,-1)$

Table 2.1: The neighborhood vectors of D3Q19 lattices given in the form (x,y,z).

$f_i(\mathbf{x},t)$ is a real value from the interval [0..1], called *density*, that can be interpreted [18] as the probability of having particles at position $\mathbf{x}$ moving along velocity vector $\mathbf{v}_i$ at time t. A velocity vector equal to zero vector therefore allows to represent particles at rest.

The term Ω_i is called *collision operator*. It describes the interaction of particles located at a point at a given time.

The discretized density field ρ and the discretized velocity field $\mathbf{u}$ can be computed using the particle distribution function:

$$\rho(\mathbf{x},t) = \sum_{i=0}^{q-1} f_i(\mathbf{x},t) \tag{2.2}$$

$$\mathbf{u}(\mathbf{x},t) = \frac{1}{\rho(\mathbf{x},t)} \sum_{i=0}^{q-1} f_i(\mathbf{x},t)\mathbf{v}_i \tag{2.3}$$

$\rho(\mathbf{x},t)$ is a real value (density of the fluid at position $\mathbf{x}$ and time t) and $\mathbf{u}(\mathbf{x},t)$ is a vector from $\mathbb{R}^3$ (velocity of fluid at position $\mathbf{x}$ and time t).

The computation of the densities at time $t+1$ in function of densities at time t generally involves two main phases: *propagation* (sometimes called streaming) and *collision*. Propagation is the operation of "moving" $f_i(\mathbf{x},t)$ values to $\mathbf{x}$'s neighbors $\mathbf{x}+\mathbf{n}_i$ with $i = 0,1,\ldots,q-1$. Collision is the computation of the new particle distribution functions values $f_i(\mathbf{x},t+1)$ given the interaction of particles (and, therefore, based on propagated values). These two operations can be highlighted by decomposing Equation 2.1 into two equations, first representing propagation and second collision:

$$\hat{f}_i(\mathbf{x}+\mathbf{n}_i,t) = f_i(\mathbf{x},t) \tag{2.4}$$

$$f_i(\mathbf{x}, t+1) = \hat{f}_i(\mathbf{x}, t) + \Omega_i \tag{2.5}$$

2.2.2 Collision Operator

One pillar of LB methods is the collision operator Ω_i and particularly the way it is calculated. One way is the Bhatnagar-Gross-Krook (BGK) operator [20]:

$$\Omega_i^{BGK}(\mathbf{x}, t) = \frac{1}{\tau}(f_i^{eq}(\mathbf{x}, t) - \hat{f}_i(\mathbf{x}, t)) \quad i = 0, 1, \ldots, q-1 \tag{2.6}$$

where τ is a relaxation time related to fluid viscosity and f_i^{eq} the local equilibrium distribution (the notation f_i^{eq} stands for $f_i^{eq}(\mathbf{x}, t)$, this shortcut will frequently be used hereafter to refer to functions of position and time like ρ, $\mathbf{u}$, f_i, etc.). The f_i^{eq} distribution can be computed in order to recover Navier-Stokes macroscopic equations with BGK model [65, 26].

Velocity vectors can be organized in the following sets:

$$\begin{array}{rcl}
X^+ & = & \{1, 7, 9, 11, 13\} \\
X^- & = & \{2, 8, 10, 12, 14\} \\
Y^+ & = & \{3, 7, 8, 15, 17\} \\
Y^- & = & \{4, 9, 10, 16, 18\} \\
Z^+ & = & \{5, 11, 12, 15, 16\} \\
Z^- & = & \{6, 13, 14, 17, 18\}
\end{array}$$

Each set contains identifiers of velocity vectors having a positive or negative component for each axis. For example, set X^+ contains the identifiers of velocity vectors having x component greater than zero and X^- the identifiers of velocity vectors having x component lower than zero.

The following values are needed to present the computation of f_i^{eq} values in the case of D3Q19 lattices:

$$\begin{array}{rcl}
u_x & = & \frac{1}{\rho}\left(\sum_{k \in X^+} \hat{f}_k - \sum_{k \in X^-} \hat{f}_k\right) \\
u_y & = & \frac{1}{\rho}\left(\sum_{k \in Y^+} \hat{f}_k - \sum_{k \in Y^-} \hat{f}_k\right) \\
u_z & = & \frac{1}{\rho}\left(\sum_{k \in Z^+} \hat{f}_k - \sum_{k \in Z^-} \hat{f}_k\right) \\
u_c & = & \left(1 - \frac{3}{2}(u_x + u_y + u_z)^2\right)
\end{array}$$

Local equilibrium distribution is then calculated as follows [18]:

$$
\begin{array}{lcl@{\quad}lcl}
f_0^{eq} & = & \frac{\rho}{3}u_c & f_1^{eq} & = & \frac{\rho}{18}(u_c + 3u_x + \frac{9u_x^2}{2}) \\
f_2^{eq} & = & \frac{\rho}{18}(u_c - 3u_x + \frac{9u_x^2}{2}) & f_3^{eq} & = & \frac{\rho}{18}(u_c + 3u_y + \frac{9u_y^2}{2}) \\
f_4^{eq} & = & \frac{\rho}{18}(u_c - 3u_y + \frac{9u_y^2}{2}) & f_5^{eq} & = & \frac{\rho}{18}(u_c + 3u_z + \frac{9u_z^2}{2}) \\
f_6^{eq} & = & \frac{\rho}{18}(u_c - 3u_z + \frac{9u_z^2}{2}) & f_7^{eq} & = & \frac{\rho}{36}(u_c + 3(u_x + u_y) + \frac{9(u_x+u_y)^2}{2}) \\
f_8^{eq} & = & \frac{\rho}{36}(u_c + 3(-u_x + u_y) + \frac{9(-u_x+u_y)^2}{2}) & f_9^{eq} & = & \frac{\rho}{36}(u_c + 3(u_x - u_y) + \frac{9(u_x-u_y)^2}{2}) \\
f_{10}^{eq} & = & \frac{\rho}{36}(u_c + 3(-u_x - u_y) + \frac{9(-u_x-u_y)^2}{2}) & f_{11}^{eq} & = & \frac{\rho}{36}(u_c + 3(u_x + u_z) + \frac{9(u_x+u_z)^2}{2}) \\
f_{12}^{eq} & = & \frac{\rho}{36}(u_c + 3(-u_x + u_z) + \frac{9(-u_x+u_z)^2}{2}) & f_{13}^{eq} & = & \frac{\rho}{36}(u_c + 3(u_x - u_z) + \frac{9(u_x-u_z)^2}{2}) \\
f_{14}^{eq} & = & \frac{\rho}{36}(u_c + 3(-u_x - u_z) + \frac{9(-u_x-u_z)^2}{2}) & f_{15}^{eq} & = & \frac{\rho}{36}(u_c + 3(u_y + u_z) + \frac{9(u_y+u_z)^2}{2}) \\
f_{16}^{eq} & = & \frac{\rho}{36}(u_c + 3(-u_y + u_z) + \frac{9(-u_y+u_z)^2}{2}) & f_{17}^{eq} & = & \frac{\rho}{36}(u_c + 3(u_y - u_z) + \frac{9(u_y-u_z)^2}{2}) \\
f_{18}^{eq} & = & \frac{\rho}{36}(u_c + 3(-u_y - u_z) + \frac{9(-u_y-u_z)^2}{2}) & & &
\end{array}
$$

The BGK operator uses a single relaxation time and is sometimes called Single Relaxation Time (SRT) collision operator. However, it is numerically unstable when simulating turbulent flows.

More recently, it has been shown that using multiple relaxation times improves numerical stability in particular when simulating turbulent flows [31]. The Multiple Relaxation Times (MRT) collision operator can be written as follows:

$$\Omega_i^{MRT}(\mathbf{x},t) = \left\{C[\mathbf{f}^{eq}(\mathbf{x},t) - \hat{\mathbf{f}}(\mathbf{x},t)]\right\}_i \tag{2.7}$$

where C is a $q \times q$ collision matrix, $\mathbf{f}^{eq}(\mathbf{x},t)$ a vector defined as

$$(f_0^{eq}(\mathbf{x},t), f_1^{eq}(\mathbf{x},t), \ldots, f_{q-1}^{eq}(\mathbf{x},t))^T$$

and $\hat{\mathbf{f}}(\mathbf{x},t)$ a vector defined as

$$(\hat{f}_0(\mathbf{x},t), \hat{f}_1(\mathbf{x},t), \ldots, \hat{f}_{q-1}(\mathbf{x},t))^T.$$

When a single relaxation time is used, matrix C is equal to $\frac{1}{\tau}I$ where I is the identity matrix. In this case, $\Omega_i^{MRT} = \Omega_i^{BGK}$.

Matrix C can be expressed in function of two other matrices: $C = M^{-1}SM$. Matrix M is a transformation matrix and S contains the relaxation times. These matrices are shown hereafter for D3Q19 lattices [31]:

$$M = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-30 & -11 & -11 & -11 & -11 & -11 & -11 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\
12 & -4 & -4 & -4 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & -4 & 4 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & -4 & 4 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & -4 & 4 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
0 & 2 & 2 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -2 & -2 & -2 & -2 \\
0 & -4 & -4 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -2 & -2 & -2 & -2 \\
0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -2 & -2 & 2 & 2 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1
\end{pmatrix}$$

M is orthogonal and therefore $M^{-1} = M^T$.

$$S = diag(0, 1.19, 1.4, 0, 1.2, 0, 1.2, 0, 1.2, \frac{1}{\tau}, 1.4, \frac{1}{\tau}, 1.4, \frac{1}{\tau}, \frac{1}{\tau}, \frac{1}{\tau}, 1.98, 1.98, 1.98)$$

where τ is a relaxation time that allows to parametrize viscosity. Other values of S are fixed to enhance numerical stability [31].

Equation 2.7 can therefore be rewritten as follows:

$$\Omega_i^{MRT}(\mathbf{x},t) = \left\{M^{-1}SM\mathbf{f}^{eq}(\mathbf{x},t) - M^{-1}SM\mathbf{f}(\mathbf{x},t)]\right\}_i \tag{2.8}$$

In pratice, matrices $M^{-1}S$ and $M^{-1}SM$ are computed once initially and vector $M\mathbf{f}^{eq}(\mathbf{x},t) = \mathbf{m}^{eq}(\mathbf{x},t)$ is directly computed at each time step as follows (instead of computing vector $\mathbf{f}^{eq}$):

$$\begin{array}{lllllll}
m_0^{eq} & = & \rho & & m_1^{eq} & = & -11\rho + 9 \parallel \mathbf{j} \parallel^2 \\
m_2^{eq} & = & 3\rho - \frac{11}{2} \parallel \mathbf{j} \parallel^2 & & m_3^{eq} & = & j_x \\
m_4^{eq} & = & -\frac{2}{3} j_x & & m_5^{eq} & = & j_y \\
m_6^{eq} & = & -\frac{2}{3} j_y & & m_7^{eq} & = & j_z \\
m_8^{eq} & = & -\frac{2}{3} j_z & & m_9^{eq} & = & 2j_x^2 - j_y^2 - j_z^2 \\
m_{10}^{eq} & = & 0 & & m_{11}^{eq} & = & j_y^2 - j_z^2 \\
m_{12}^{eq} & = & 0 & & m_{13}^{eq} & = & j_x j_y \\
m_{14}^{eq} & = & j_y j_z & & m_{15}^{eq} & = & j_x j_z \\
m_{16}^{eq} & = & 0 & & m_{17}^{eq} & = & 0 \\
m_{18}^{eq} & = & 0 & & & &
\end{array}$$

where ρ is the local density and $j_\alpha = \rho u_\alpha$ ($\alpha = x, y, z$).

Large Eddy Simulation (LES) techniques, used to describe turbulent flows, can be applied to LB methods by combining the turbulent viscosity model of

Smagorinsky [72] with SRT [43] and MRT [50] collision models. The basic idea of the method is to locally (i.e. independently for each site) increase relaxation time: new relaxation time τ_{tot} is equal to $\tau + \tau_t$ where τ is the initial relaxation time and τ_t the effect of smaller scales (turbulences contribution).

Value τ_t is calculated using following equation:

$$\tau_t = \frac{1}{2}\left(\sqrt{\tau^2 + 18\sqrt{2}C_{smg}^2\sqrt{Q}} - \tau\right)$$

where C_{smg} is the Smagorinsky subgrid constant (generally, $0 \leq C_{smg} < 0.5$) and Q a value defined as follows:

$$Q = \sum_{\alpha \in x,y,z} \sum_{\beta \in x,y,z} \sum_{k=0}^{18} \left[\hat{f}_k - f_k^{eq}\right] n_{k\alpha} n_{k\beta}$$

where $n_{k\alpha}$ is the component along axis α of neighborhood vector $\mathbf{n}_k$.

The use of Smagorinsky's turbulent viscosity model introduces an execution time overhead because of the computation of τ_t at each time step. This overhead is even more important when the turbulent model is combined with MRT collision model: matrices $M^{-1}S$ and $M^{-1}SM$ need to be recalculated at each time step instead of being calculated only once at the beginning of the simulation.

2.2.3 Boundary Conditions

As stated previously, a fluid simulated with numerical methods must be defined on a bounded spatial domain. Boundary conditions are used to describe the fluid dynamics on its boundaries composed of the lattice borders and fluid-solid interfaces.

A general definition for the *border* of a lattice is the set of sites that lack at least one neighbor: let $\mathbf{x}$ be the position of one of these sites, it exists a neighborhood vector $\mathbf{n}_i$ such as $\mathbf{x} + \mathbf{n}_i$ is not the position of a site of the lattice (the position is out of the lattice).

A 3 dimensions finite lattice (D3Qq family) is a cube or a cuboid (see Figure 2.2). The components of position $\mathbf{x} = (x, y, z)$ have ranges defined in the following way: $0 \leq x < xSize$, $0 \leq y < ySize$ and $0 \leq z < zSize$ where $xSize$, $ySize$ and $zSize$ are the size of each dimension. The *size* of a 3D lattice is defined by the vector $(xSize, ySize, zSize)$ and the number of sites of the lattice is given by $xSize \times ySize \times zSize$.

The border sites of a 3D lattice have a position with at least one of the following properties:

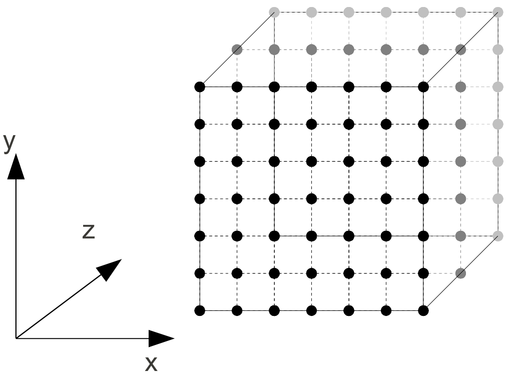

Figure 2.2: 3D lattices are cubes or cuboids.

- $x = 0$
- $x = xSize - 1$
- $y = 0$
- $y = ySize - 1$
- $z = 0$
- $z = zSize - 1$

As of propagation equation (Equation 2.4), some densities are undefined after propagation for the border sites of the lattice: density $\hat{f}_i(\mathbf{x} + \mathbf{n}_i, t)$ is not defined for a border site at position $\mathbf{x}$ because it has no neighbor at position $\mathbf{x} - \mathbf{n}_i$.

These undefined densities are called *incoming densities*. On Figure 2.3, the arrows represent the incoming densities in three situations: case (a) corresponds to a site with $y = ySize - 1$, case (b) to a site with $x = xSize - 1$ and $y = ySize - 1$, and case (c) to the site with $x = xSize - 1$, $y = ySize - 1$ and $z = zSize - 1$. All other situations are similar to one of these 3 cases. Boundary conditions are used to set these values.

When a border site has no neighbor at position $\mathbf{x} + \mathbf{n}_i$, propagation causes densities to go out of the lattice. These densities are called *outgoing densities*. Examples of outgoing densities can be seen in Figure 2.4: the three cases are similar to the cases of Figure 2.3.

Due to the stochastic nature of LB methods, boundary conditions are not easy to derive and proposed solutions are generally ad-hoc. To handle fluid behavior at the interface with a solid, a method called bounce-back is commonly used. In order to settle the flow in simple structures, a body force can be applied to the fluid. In this situation, periodic boundary conditions are used: outgoing densities

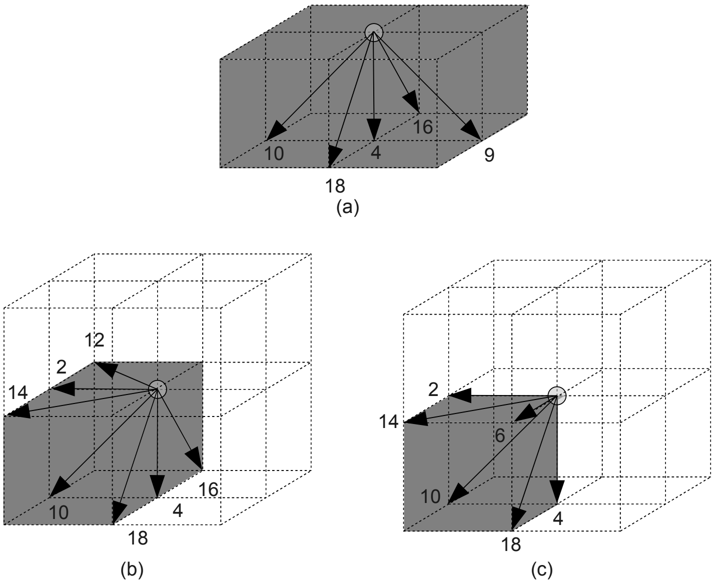

Figure 2.3: Incoming densities for a site on (a) a plane, (b) an edge and (c) a corner. The arrows represent the incoming densities.

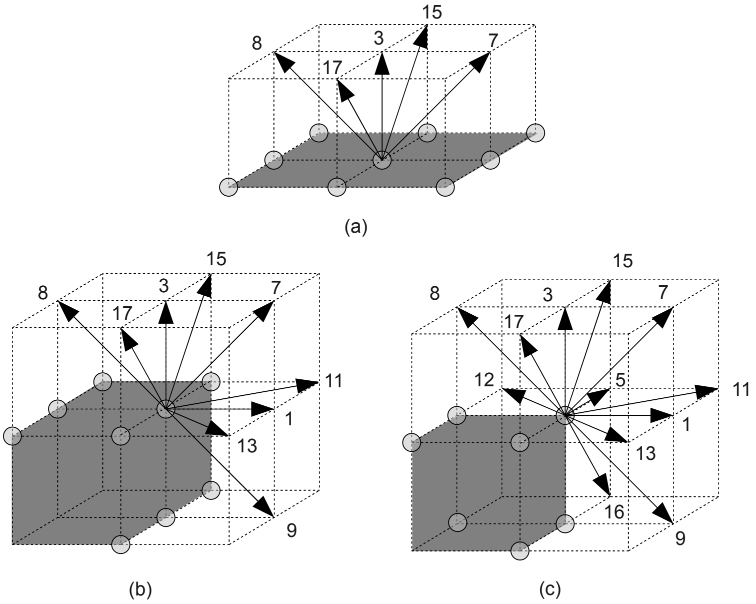

Figure 2.4: Outgoing densities for a site on (a) a plane, (b) an edge and (c) a corner. The arrows represent the outgoing densities.

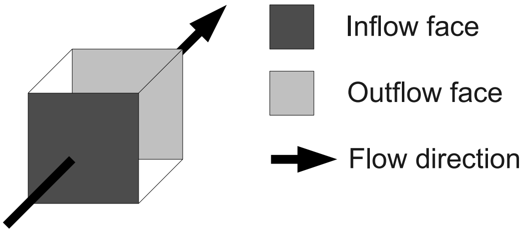

Figure 2.5: Inflow and outflow planes of a 3D lattice. These planes are normal to the flow direction.

are used to set incoming densities (what goes out at one side comes in at the opposite side). In more complex structures with potentially turbulent flows, pressure conditions should be specified.

In the case of 3D lattices, the *inflow* sites (sites where fluid enters the lattice) and *outflow* sites (sites where fluid leaves the lattice) are part of opposite faces of a cube or cuboid. These faces are normal to flow direction (see Figure 2.5). Periodic boundary conditions are applied to the faces that are parallel to flow direction.

Bounce-Back

The main idea of the bounce-back [78] method (initially used with cellular automata) is that a particle following a given direction and that collides with an obstacle "bounces" in the opposite direction.

After the propagation step, the collision operator is applied on sites that are not obstacles. If a site is an obstacle, the bounce-back method is applied instead and the following equation is used instead of Equation 2.5:

$$f_i(\mathbf{x}, t+1) = \hat{f}_j(\mathbf{x}, t) \tag{2.9}$$

where i and j are such as $\mathbf{n}_i = -\mathbf{n}_j$.

For instance, if we suppose that $s(\mathbf{x})$ is *false* and $s(\mathbf{x}+\mathbf{n}_i)$ is *true*. Then, if $\mathbf{n}_i = -\mathbf{n}_j$, the following equalities are derived from Equation 2.4 (first and third equality) and Equation 2.9 (second equality):

$$\hat{f}_j(\mathbf{x}, t+1) = f_j(\mathbf{x}+\mathbf{n}_i, t+1) = \hat{f}_i(\mathbf{x}+\mathbf{n}_i, t) = f_i(\mathbf{x}, t)$$

This illustrates the fact that particles moving in direction $\mathbf{n}_i$ and colliding against an obstacle come back in the opposite direction at time $t+1$.

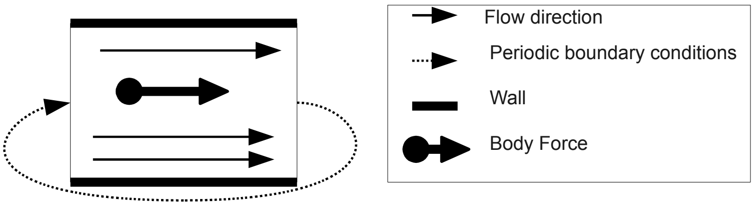

Figure 2.6: Periodic boundary conditions for a flow in a pipe. The longitudinal section of a pipe element is represented.

Body Force and Periodic Boundary Conditions

In simple structures like pipes, a flow can be settled by applying a body force **G** on the fluid. In this case, Equation 2.1 can be rewritten as follows:

$$f_i(\mathbf{x}+\mathbf{n}_i, t+1) = f_i(\mathbf{x}, t) + \Omega_i + g_i \tag{2.10}$$

where g_i is a forcing term computed in function of real body force **G**. Inflow and outflow planes are considered as contiguous and the propagation equation (Equation 2.4) is then used to set incoming densities from inflow plane with outgoing densities of outflow plane.

This method can be used to simulate a flow in an infinite pipe in the flow direction where walls parallel to flow direction contain the fluid. The lattice then represents a section of the pipe. Periodic boundary conditions are applied to simulate the infinite pipe (see Figure 2.6): outgoing densities of the outflow plane are used to set corresponding incoming densities of the inflow plane.

Periodic boundary conditions are *full* when periodic boundary conditions are applied for the whole border of the lattice (instead of only inflow and outflow planes). In this case, outgoing densities of any face or edge of the lattice are used to set corresponding incoming densities of opposite face or edge. Full periodic boundary conditions can be seen as a way to make a lattice infinite in all directions.

Full periodic boundary conditions are applied when using the following equation instead of propagation equation (Equation 2.4):

$$\hat{f}_i((\mathbf{x}+\mathbf{n}_i) \bmod \mathbf{s}, t) = f_i(\mathbf{x}, t) \tag{2.11}$$

where **s** is the size of the lattice and $((\mathbf{x}+\mathbf{n}_i) \bmod \mathbf{s})_j = (x_j + n_{ij}) \bmod s_j$.

In this case, the neighborhood of a site must be redefined: the neighborhood of a site at position **x** is the set of q sites at positions $(\mathbf{x}+\mathbf{n}_i) \bmod \mathbf{s}$ with $i = 0, 1, \ldots, q-1$. In this case, a site has always q neighbors.

Pressure conditions

With pressure conditions, incoming densities of a site are extrapolated using other propagated densities. In LB methods, pressure conditions are obtained by imposing density conditions. A flow can be obtained by forcing a higher density for inflow sites than for outflow sites.

A method has been proposed [81] to compute incoming densities for inflow and outflow sites given an input density ρ_{in} and an output density ρ_{out}. This method has been adapted to D3Q19 lattices [18] to obtain a flow in x-axis's direction and is summarized below.

The set of density identifiers can be partitionned into 3 subsets:

$$\begin{cases} X^0 & = & \{0,3,4,5,6,15,16,17,18\} \\ X^+ & = & \{1,7,9,11,13\} \\ X^- & = & \{2,8,10,12,14\} \end{cases}$$

where X^0 is the set of identifiers of densities with velocity vectors normal to x-axis (and therefore to flow direction), X^+ the set of identifiers of densities with velocity vectors having a positive x component and X^- the set of identifiers of densities with velocity vectors having a negative x component (see Table 2.1).

The following value is needed to compute input pressure conditions:

$$c_{in} = \rho_{in} - \left[\sum_{k \in X^0} f_k + 2 \sum_{k \in X^-} f_k \right]$$

Incoming densities (f_i with $i \in X^+$) for inflow sites can then be computed as follows:

$$\begin{aligned} f_1 &= f_2 + \frac{1}{3} c_{in} \\ f_7 &= f_{10} + \frac{1}{6} c_{in} \\ f_9 &= f_8 + \frac{1}{6} c_{in} \\ f_{11} &= f_{14} + \frac{1}{6} c_{in} \\ f_{13} &= f_{12} + \frac{1}{6} c_{in} \end{aligned}$$

The following value is needed to compute output pressure conditions:

$$c_{out} = \rho_{out} - \left[\sum_{k \in X^0} f_k + 2 \sum_{k \in X^+} f_k \right]$$

Incoming densities (f_i with $i \in X^-$) for outflow sites can then be computed as follows:

$$\begin{aligned}
f_2 &= f_1 + \frac{1}{3} c_{out} \\
f_{10} &= f_7 + \frac{1}{6} c_{out} \\
f_8 &= f_9 + \frac{1}{6} c_{out} \\
f_{14} &= f_{11} + \frac{1}{6} c_{out} \\
f_{12} &= f_{13} + \frac{1}{6} c_{out}
\end{aligned}$$

2.2.4 Initial Conditions

LB methods base equation (Equation 2.1) describes the evolution of particle distribution functions. However, initial $f_i(\mathbf{x}, 0)$ values are needed. A common approach is to initialize the fluid at rest (no flow) [18]:

$$\begin{aligned}
f_0 &= \tfrac{1}{3} \\
f_i &= \tfrac{1}{18} \\
f_j &= \tfrac{1}{36}
\end{aligned}$$

where $i \in \{1, 2, 3, 4, 5, 6\}$ and $j \in \{7, 8, \ldots, 18\}$.

The values produced by a previous simulation can be used as initial conditions in order to continue its execution.

2.3 Language Choice

The distributed LB simulation tool that is presented in this thesis is an experimental software. Its maintenance and evolution capabilities must therefore be strong. In addition, the software should be adapted to heterogeneous clusters.

C, C++ or Fortran are generally chosen to write numerically intensive programs. In particular, Fortran is appreciated by scientists because of its native support for complex numbers and multi-dimensional arrays.

However, Fortran is not well suited to nonscientific programming involving complex data structures and provides few tools to organize and maintain the source code of complex applications composed of many components requiring several thousands of lines of code each. In addition, the binary file produced by the compilation of a Fortran program is bound to a particular architecture and OS and is therefore not directly portable.

C++ provides the Object Oriented Programming (OOP) paradigm to organize the source code. However, the same portability issue as Fortran remains.

When executing a distributed application in an environment where computers have several architectures, different operating systems and/or different libraries installed, the lack of portability is an important issue.

The Java programming language [4] mostly solves the portability issue: the compilation of a Java program produces byte-code files that can be executed by any Java Virtual Machine (JVM). Implementations of the JVM are available for most popular OSes and architectures. Moreover, Java provides the OOP paradigm as well as convenient tools (like garbage collector, standard libraries, etc.) that ease and accelerate software development. However, Java has drawbacks regarding scientific programming :

- No support for complex numbers (like C/C++).
- No direct support for multi-dimensional arrays (like C/C++).
- Potential performance problems: the additional complexity layer introduced by the execution of the byte-code by a JVM introduces an overhead in execution time when compared to a native implementation.

First drawback can be ignored in the context of LB simulations because no complex numbers support is needed. Second drawback will be discussed in Chapter 3.

Regarding last drawback, a recent publication [15] has shown that the performance of Java implementations for computationally intensive tasks executed on distributed systems tends to be similar to native implementations using Fortran and MPI (Message Passing Interface, an Application Programming Interface (API) specification for inter-process communications). This is mostly because JVMs implement a mechanism called Just-In-Time compilation: parts of the byte-code are converted at run time into native code and directly executed by the CPU.

With tasks involving the exchange of many small messages (less than 50 bytes), Java implementation is 2 to 6 times slower than native implementation. LB simulations executed in a distributed way involve the exchange of many messages. However, these messages are generally represented on more than 3 kilobytes.

Java is a reasonable compromise between efficiency and portability. In addition, it is adapted to the organization and maintenance of complex projects involving several components and requiring many lines of code. It was therefore chosen to implement our LB simulation tool.

2.4 Sequential Implementation

Previously, LB methods were presented from a mathematical and physical point of view. In this section, a sequential implementation of LB simulations is given.

LB methods are based on the calculation of particle distribution functions values using Equation 2.1. The notation $f_i(\mathbf{x},t)$ suggests the use of a 4D array to represent f: 3 dimensions for the position $\mathbf{x}$ and one for the velocity i.

Another array of same size can be used to represent $\hat{f}$ after propagation phase (see Equation 2.4). The result of the collision operator being applied on $\hat{f}$ values is stored back into the array representing f (see Equation 2.5). This process can be applied again until the right number of time steps has been reached. The solid function s is represented by a 3D array of booleans.

These arrays can be declared as follows:

```
f :  array[0..xSize-1, 0..ySize-1, 0..zSize-1, 0..18] of real;
fHat :  array[0..xSize-1, 0..ySize-1, 0..zSize-1, 0..18] of real;
s :  array[0..xSize-1, 0..ySize-1, 0..zSize-1] of boolean;
```

where `xSize`, `ySize`, `zSize` represent the size of the lattice (in number of sites) for respectively x, y and z dimensions.

The Algorithm 2.1 illustrates a typical LB simulation code on a D3Q19 lattice. `timeSteps` is the number of time steps of the simulation.

`"Fill s"` fills `s` in a way depending on the solid shape. Generally, the solid shape is read from a file. `"Initialize f"` fills `f` using the theory described in Section 2.2.4.

```
"Fill s";
"Initialize f";
t := 0;
do t < timeSteps →
   "Propagate values";
   "Apply boundary conditions";
   "Apply collision";
   t := t + 1
od
```

Algorithm 2.1: Simple LB simulation code.

Algorithm 2.2 describes the propagation step. Values are propagated in function of the neighborhood vectors defined previously in Table 2.1. The use of the modulo operator to compute the destination position in `fHat` implements the periodic boundary conditions described in Section 2.2.3.

```
"Initialize x, y and z for the scan of f";
do "scan of f is not finished" →
   fHat[x, y, z, 0] = f[x, y, z, 0];
   fHat[(x+1) mod xSize, y, z, 1] = f[x, y, z, 1];
   fHat[(x-1) mod xSize, y, z, 2] = f[x, y, z, 2];
   ⋮
   fHat[x, (y-1) mod ySize, (z-1) mod zSize, 18] = f[x, y, z, 18];
   "Go to next site"
od
```

Algorithm 2.2: `"Propagate values"`.

Algorithm 2.3 describes command `"Apply boundary conditions"` and implements pressure boundary conditions. For the sake of readability, only the particular case of a flow in *x* direction is presented. `"Set incoming densities of fHat[0, y, z, .]"` and `"Set incoming densities of fHat[xSize - 1, y, z, .]"` commands are based on the theory described in Section 2.2.3.

```
y, z := 0, 0
do y < ySize →
   {Set values for inflow plane}
   if not s[0, y, z] →
      "Set incoming densities of fHat[0, y, z, .]"
   □ s[0, y, z] → skip
   fi;
   {Set values for outflow plane}
   if not s[xSize - 1, y, z] →
      "Set incoming densities of fHat[xSize - 1, y, z, .]"
   □ s[xSize - 1, y, z] → skip
   fi;
   if z < zSize - 1 → z := z + 1
   □ z = zSize - 1 → z := 0; y := y + 1
   fi
od
```

Algorithm 2.3: "Apply boundary conditions".

Algorithm 2.4 describes command "Apply collision" and implements collision, bounce-back and body force (see Sections 2.2.2 and 2.2.3). "Apply collision operator on fHat[x, y, z, .]" effectively applies collision and body force. The result of this operation i.e. the 19 densities associated to position (x, y, z) is stored in the f array at the same position.

2.4.1 In-place Propagation

An in-place version of propagation (Algorithm 2.2) can be written in order to eliminate the fHat array. The memory usage of LB simulations is therefore decreased.

Propagation can be seen as the application of a translation to all values associated to a given velocity. For example (see Figure 2.1):

- A translation vector $(1, 0, 0)$ is applied on each value associated to velocity 1 and f[x + 1, y, z, 1] := f[x, y, z, 1] (with 0 $\leq$ x $<$ xSize-1).
- A translation vector $(-1, 1, 0)$ is applied on each value associated to velocity 8 and f[x - 1, y + 1, z, 8] := f[x, y, z, 8] (with 0 $<$ x $<$

```
"Initialize x, y and z for scan of f";
do "scan of f is not finished" →
   if s[x, y, z] → {Bounce-back}
      f[x, y, z, 0] := fHat[x, y, z, 0];
      f[x, y, z, 1] := fHat[x, y, z, 2];
      ⋮
      f[x, y, z, 18] := fHat[x, y, z, 15];
   □ not s[x, y, z] →
      "Apply collision operator on fHat[x, y, z, .]"
   fi;
   "Go to next site"
od
```

Algorithm 2.4: `"Apply collision"`.

`xSize` and `0 ≤ y < ySize - 1`).

This translation can be applied directly to values of `f` array by using three nested loops, one on each position component (`x`, `y` and `z`) and moving the value at current position to the new position (see below).

However, the scan direction must be chosen in function of the translation vector. In the case of velocity 8 for example, Algorithm 2.5 is not correct. It leads to a situation where `f[x-1, 1, z, 8]` = `f[x-1, 2, z, 8]` = ... = `f[x-1, ySize-1, z, 8]` for fixed `x` and `z`.

Changing the scan direction for `y` index solves the problem. A correct in-place propagation of velocity 8 values is described by Algorithm 2.6.

The loop invariant P is given by the following expression and defines the area of array `f` containing propagated values after each iteration:

$$\begin{gathered}\{P\colon\ (0 < x \le xSize) \wedge (0 \le y < ySize - 1) \wedge (0 \le z < zSize) \wedge \\ \text{“Densities associated to velocity 8 have been propagated for sites at positions} \\ (i,j,k)\text{”} \wedge ((0 \le i < x-1) \wedge (1 \le j < ySize) \wedge (0 \le k < zSize)) \vee \\ ((i = x-1) \wedge (y+1 < j < ySize) \wedge (0 \le k < zSize)) \vee \\ ((i = x-1) \wedge (j = y+1) \wedge (0 \le k < z))\}\end{gathered}$$

After the loop terminated its execution, the following expression is true:

$$x = xSize \wedge y = ySize - 2 \wedge z = 0$$

```
x, y, z := 1, 0, 0;
do x < xSize →
   f[x-1, y+1, z, 8] := f[x, y, z, 8];
   if z < zSize - 1 → z := z + 1
   □ z = zSize - 1 →
      z := 0;
      if y < ySize - 1 → y := y + 1
      □ y = ySize - 1 →
         y := 0; x := x + 1
      fi
   fi
od
```

Algorithm 2.5: Incorrect in-place propagation of velocity 8 values.

```
x, y, z := 1, ySize - 2, 0; {P}
do x < xSize → {P}
   f[x-1, y+1, z, 8] := f[x, y, z, 8];
   if z < zSize - 1 → z := z + 1
   □ z = zSize - 1 →
      z := 0;
      if y > 0 → y := y - 1
      □ y = 0 →
         y := ySize - 2; x := x + 1
      fi
   fi {P}
od {P}
```

Algorithm 2.6: Correct in-place propagation of velocity 8 values.

The densities associated to velocity 8 of sites at positions (i, j, k) have then been propagated with i, j, k defined by the following expression:

$$((0 \leq i < xSize - 1) \wedge (1 \leq j < ySize) \wedge (0 \leq k < zSize))$$

This shows that the sites at positions (i, j, k) with i, j, k defined as follows:

$$((i = xSize - 1 \vee j = 0) \wedge 0 \leq k < zSize)$$

have an obsolete value of density 8. These values are the incoming densities for velocity 8.

In order to simplify next algorithms description, a notation is introduced to describe a scan. Let α be an index variable, a scan type of on an array using this variable can be described using following notation:

- α^0: α is incremented from 0 to maximum value (α dimension size minus one).
- α^+: α is incremented from 1 to maximum value. The scan is *positive*.
- α^-: α is decremented from maximum value minus one to 0. The scan is *negative*.

The composed notation $\alpha^i\beta^j\gamma^k$ represents a scan on a 3D array using variables α, β and γ.

For example, Algorithm 2.6 uses a $x^+y^-z^0$ scan and can be rewritten as follows:

```
"Initialize $x^+y^-z^0$ scan";
do "$x^+y^-z^0$ scan is not finished" $\rightarrow$
   f[x-1, y+1, z, 8] := f[x, y, z, 8];
   "Go to $x^+y^-z^0$ next"
od
```

The translation vector that must be applied on values of array `f` for a velocity q can be written $\mathbf{t}^q = (t_x^q, t_y^q, t_z^q)$ where t_α^q is equal to 0, 1 or -1 ($\alpha = x, y, z$). For a given velocity q, the scan direction is determined using following rules:

- if $t_\alpha^q = 1$, an α^- scan is used.

- if $t^q_\alpha = -1$, an α^+ scan is used.
- if $t^q_\alpha = 0$, an α^0 scan is used.

Note that Algorithm 2.6 respects these rules.

Algorithm 2.7 describes an in-place propagation. This algorithm does not implement periodic boundary conditions: outgoing densities are not extracted and incoming densities are not set.

```
{Velocity 0 :  Nothing to do}
{Velocity 1 :  translation (1,0,0)}
"Initialize x, y, z for $x^-y^0z^0$ scan";
do "$x^-y^0z^0$ scan is not finished" →
   f[x+1, y, z, 1] := f[x, y, z, 1];
   "Go to $x^-y^0z^0$ next"
od;

{Velocity 2 :  translation (-1,0,0)}
"Initialize x, y, z for $x^+y^0z^0$ scan";
do "$x^+y^0z^0$ scan is not finished" →
   f[x-1, y, z, 2] := f[x, y, z, 2];
   "Go to $x^+y^0z^0$ next"
od;

"Apply propagation for velocities 3..17";

{Velocity 18 :  translation (0,-1,-1)}
"Initialize x, y, z for $x^0y^+z^+$ scan";
do "$x^0y^+z^+$ scan is not finished" →
   f[x, y-1, z-1, 1] := f[x, y, z, 1];
   "Go to $x^0y^+z^+$ next"
od
```

Algorithm 2.7: In-place propagation.

With in-place propagation, boundary conditions (Algorithm 2.3) and collision (Algorithm 2.4) algorithms must be adapted to directly modify `f`. In order to include periodic boundary conditions, LB simulation algorithm 2.1 should be rewritten as shown in Algorithm 2.8.

```
"Initialize f";
"Fill s";
t := 0;
do t < timeSteps →
   "Copy outgoing densities to buffers";
   "Propagate values";
   "Use buffers' content to set incoming densities";
   "Apply boundary conditions";
   "Apply collision";
   t := t + 1
od
```

Algorithm 2.8: Simulation code with in-place propagation.

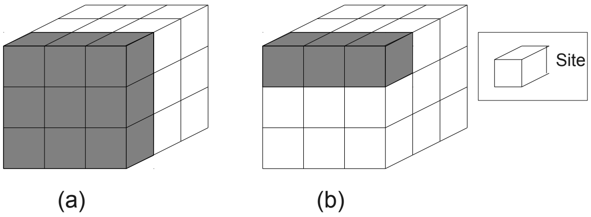

Figure 2.7: Example of a lattice face (a) and a lattice edge (b).

Periodic Boundary Conditions with In-place Propagation

Before commands `"Copy outgoing densities to buffers"` and `"Use buffers' content to set incoming densities"` from Algorithm 2.8 can be presented, some definitions are needed.

A site of a lattice with size (s_x, s_y, s_z) is part of a *face* of the lattice if at least one component α of its position in the lattice is equal to 0 or $s_\alpha - 1$ ($\alpha \in \{x, y, z\}$).

A site of a lattice with size (s_x, s_y, s_z) is part of an *edge* of the lattice if two components α and β of its position in the lattice are respectively equal to 0 or $s_\alpha - 1$ and to 0 or $s_\beta - 1$ ($\alpha, \beta \in \{x, y, z\}, \alpha \neq \beta$).

Figure 2.7 illustrates these two definitions. Note that a site can be part of a face and an edge at the same time.

A neighborhood vector (see Section 2.2.1) can be associated to each face or edge of a lattice. The situation can be illustrated by Figure 2.1 where the central

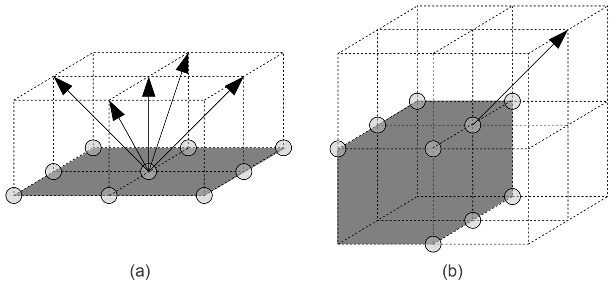

Figure 2.8: Number of outgoing densities for a face (a) and an edge (b) for a D3Q19 lattice. The arrows represent the outgoing densities.

site is replaced by a lattice. A number i can be associated to each face or edge of a lattice: it is the number of the associated neighborhood vector.

In the context of D3Q19 lattices, there are 5 outgoing densities per site of a face and 1 outgoing density per site of an edge (see Figure 2.8). All the densities coming out of a face of a lattice can therefore be stored into a 3D array, one dimension for the 5 densities and the two others for the position in the face. All densities coming out of an edge of a lattice can be stored into a 1D array.

A D3Qq lattice has 6 faces and 12 edges. The outgoing densities of a D3Q19 lattice can therefore be stored into 6 3D arrays and 12 1D arrays. These 18 arrays are called *output buffers*.

Each output buffer is identified by the number of its associated face or edge. For example, outgoing densities of face $x = \mathit{xSize} - 1$ are put into output buffer 1 and outgoing densities for edge $x = \mathit{xSize} - 1, z = \mathit{zSize} - 1$ are put into output buffer 11. Output buffers can be declared as follows:

```
var
   out1 :  array[0..ySize-1, 0..zSize-1, 0..4] of real;
   "Declaration of other face output buffers";
   out11 :  array[0..ySize-1] of real;
   "Declaration of other edge output buffers"
```

The outgoing densities of a face or an edge are used to set the incoming densities from the opposite face or edge. Figure 2.9 illustrates the situation where the outgoing densities of the face associated to neighbor vector $\mathbf{n}_1$ are used to set the incoming densities of the face associated to neighbor vector $\mathbf{n}_2$. Note that the two

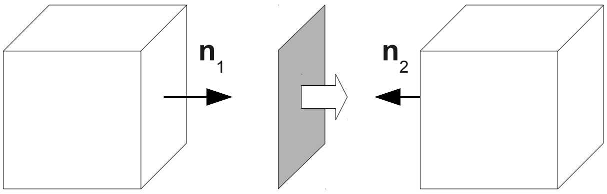

Figure 2.9: Outgoing densities of the face associated to neighbor vector $\mathbf{n}_1$ are used as incoming densities for the face associated to neighbor vector $\mathbf{n}_2$. The two cubes represent the same lattice.

cubes represent the same lattice if periodic boundary conditions are used.

In Algorithm 2.8, command `"Copy outgoing densities to buffers"` fills the output buffers with all outgoing densities of the lattice. It is described in Algorithm 2.9.

The loop invariant *P* of Algorithm 2.9 is given by the following expression:

$$\{P\colon (0 \le y \le ySize) \wedge (0 \le z < zSize) \wedge \text{“}\textit{Value at position (i,j,k) of output buffer } \texttt{out1} \textit{ is set”} \wedge ((0 \le i < y) \wedge (0 \le j < zSize) \wedge (0 \le k < 5)) \vee ((i = y) \wedge (0 \le j < z) \wedge (0 \le k < 5))\}$$

P describes the area of `out1` that has already been filled. When the loop terminates, following expression is true:

$$y = ySize \wedge z = 0$$

This implies that `out1` is filled at positions (i, j, k) with:

$$(0 \le i < ySize) \wedge (0 \le j < zSize) \wedge (0 \le k < 5)$$

which represents the whole array.

Command `"Use buffers' content to set incoming densities"` is described by Algorithm 2.10.

The loop invariant *P* of Algorithm 2.10 describes the area of array `f` that contains valid incoming densities. *P* is given by following expression:

$$\{P\colon (0 \le y \le ySize) \wedge (0 \le z < zSize) \wedge \text{“}\textit{Incoming densities of } \texttt{f} \textit{ are valid at positions } (0,i_1,j_1,1), (0,i_2,j_2,7), (0,i_3,j_3,9), (0,i_4,j_4,11) \textit{ and } (0,i_5,j_5,13)\text{”} \wedge Q_1 \wedge Q_2 \wedge Q_3 \wedge Q_4 \wedge Q_5\}$$

```
{Fill output buffer 1}
"Initialize y, z for $y^0z^0$ scan"; {$P$}
do "$y^0z^0$ scan is not finished" $\rightarrow$ {$P$}
   out1[y, z, 0] := f[xSize-1, y, z, 1];
   out1[y, z, 1] := f[xSize-1, y, z, 7];
   out1[y, z, 2] := f[xSize-1, y, z, 9];
   out1[y, z, 3] := f[xSize-1, y, z, 11];
   out1[y, z, 4] := f[xSize-1, y, z, 13];
   "Go to $y^0z^0$ next" {$P$}
od; {$P$}
"Fill other face output buffers";
{Fill output buffer 11}
y := 0;
do y < ySize $\rightarrow$
   out11[y] := f[xSize-1, y, zSize-1, 11];
   y := y + 1
od;
"Fill other edge output buffers"
```

Algorithm 2.9: "Copy outgoing densities to buffers".

```
{Use output buffer 1 to set incoming densities}
"Initialize y, z for $y^0z^0$ scan"; {$P$}
do "$y^0z^0$ scan is not finished" → {$P$}
    f[0, y, z, 1] := out1[y, z, 0];
    if y + 1 < ySize → f[0, y+1, z, 7] := out1[y, z, 1];
    □ y + 1 = ySize → skip
    fi;
    if y - 1 >= 0 → f[0, y-1, z, 9] := out1[y, z, 2];
    □ y - 1 < 0 → skip
    fi;
    if z + 1 < zSize → f[0, y, z+1, 11] := out1[y, z, 3];
    □ z + 1 = zSize → skip
    fi;
    if z - 1 >= 0 → f[0, y, z-1, 13] := out1[y, z, 4];
    □ z - 1 < 0 → skip
    fi;
    "Go to $y^0z^0$ next" {$P$}
od; {$P$}
"Use other face output buffers to set incoming densities";
{Use output buffer 11 to set incoming densities}
y := 0;
do y < ySize →
    f[0, y, 0, 11] := out11[y];
    y := y + 1
od;
"Use other edge output buffers to set incoming densities"
```

Algorithm 2.10: "Use buffers' content to set incoming densities".

where Q_k is the definition of i_k and j_k with $k = 1,2,3,4,5$. Q_k expressions are given below.

$$Q_1\text{: } ((0 \le i_1 < y) \wedge (0 \le j_1 < zSize)) \vee ((i_1 = y < ySize) \wedge (0 \le j_1 < z))$$

$$\begin{gathered} Q_2\text{: } ((1 \le i_2 \le y < ySize) \wedge (0 \le j_2 < zSize)) \vee \\ ((i_2 = y+1 < ySize) \wedge (0 \le j_2 < z)) \vee \\ ((y = ySize) \wedge (1 \le i_2 < ySize) \wedge (0 \le j_2 < zSize)) \end{gathered}$$

$$Q_3\text{: } ((0 \le i_3 < y-1) \wedge (0 \le j_3 < zSize)) \vee ((0 \le i_3 = y-1) \wedge (0 \le j_3 < z))$$

$$Q_4\text{: } ((0 \le i_4 < y) \wedge (1 \le j_4 < zSize)) \vee ((i_4 = y < ySize) \wedge (1 \le j_4 \le z))$$

$$Q_5\text{: } ((0 \le i_5 < y) \wedge (0 \le j_5 < zSize-1)) \vee ((i_5 = y < ySize) \wedge (0 \le j_5 < z))$$

After loop's execution, at least the following expression E is true:

$$y = ySize$$

The following incoming densities are therefore valid after the execution of the loop associated to invariant P.

- $(0, i_1, j_1, 1)$ with $(0 \le i_1 < ySize) \wedge (0 \le j_1 < zSize)$,
- $(0, i_2, j_2, 7)$ with $(1 \le i_2 < ySize) \wedge (0 \le j_2 < zSize)$,
- $(0, i_3, j_3, 9)$ with $(0 \le i_3 < ySize-1) \wedge (0 \le j_3 < zSize)$,
- $(0, i_4, j_4, 11)$ with $(0 \le i_4 < ySize) \wedge (1 \le j_4 < zSize)$,
- $(0, i_5, j_5, 13)$ with $(0 \le i_5 < ySize) \wedge (0 \le j_5 < zSize-1)$.

Listed ranges are implied by $E \wedge Q_k$ with $k = 1,2,3,4,5$. The following incoming densities are therefore still not valid after the execution of the loop:

- $(0, i_2, j_2, 7)$ with $(i_2 = 0) \wedge (0 \le j_2 < zSize)$,
- $(0, i_3, j_3, 9)$ with $(i_3 = ySize-1) \wedge (0 \le j_3 < zSize)$,
- $(0, i_4, j_4, 11)$ with $(0 \le i_4 < ySize) \wedge (j_4 = 0)$,
- $(0, i_5, j_5, 13)$ with $(0 \le i_5 < ySize) \wedge (j_5 = zSize-1)$.

Valid incoming densities are set at these positions when copying the content of output buffers 7, 9, 11 and 13 into array `f`. For example, the copy of output buffer 11 into array `f` (see Algorithm 2.10) sets incoming densities for the area $(0, i_4, j_4, 11)$ listed above.

2.5 Parallel and Distributed Implementation

With the current trend of multi-core processors and distributed computing, parallel and distributed implementations are necessary to take advantage of all available processing power of several or even only one computer. The simulation code presented in previous section should therefore be parallelized (i.e. decomposed into several processes or threads in order to be simultaneously executed by several processors). This section describes the parallelization of LB simulations.

2.5.1 Problem Subdivision

A way to parallelize a program is to subdivide the problem it solves into smaller subproblems. Each subproblem is then associated to a different process. The subproblems are solved in parallel, potentially with communications between processes. Finally the global solution is obtained by aggregating the solutions of the subproblems.

An obvious way to subdivide LB simulations is to partition the lattice and associate each partition (called *sublattice*) to a different process. Sublattices are, in fact, lattices. The name is introduced only to make a clear difference between a lattice and its partitions. Several definitions (size, border, face, edge, outgoing densities, incoming densities) and properties associated to lattices are also applicable to sublattices.

In the same way a lattice is partitioned into sublattices, the solid is partitioned accordingly into *subsolids*. A subsolid is also a solid in the same way a sublattice is a lattice.

When a site from the border of a sublattice has no neighbor, it is because its neighbor either is in another sublattice, or it is not in the lattice (and the site is then also part of the border of the lattice). The former situation introduces a new kind of boundary. These boundaries have no real physical meaning, they only exist because of the partitioning. However, they imply additional incoming densities and outgoing densities regarding the subdivided lattice: outgoing (respectively incoming) densities of a sublattice are not necessarily outgoing (respectively incoming) densities of the lattice.

Two sublattices are *neighbors* if their borders contain sites that are neighbors in the global lattice. The incoming densities of sites from the border of a sublattice are set with outgoing densities of sites from the border of a neighbor sublattice. Two processes are neighbors if they execute the simulation code on neighboring sublattices.

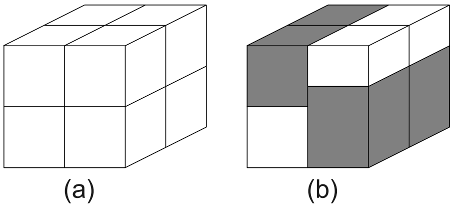

Figure 2.10: Constrained (a) and unconstrained (b) partitioning of a lattice. Constrained partitioning imposes that all sublatties have one and only one contiguous sublattice by face or edge.

If periodic boundary conditions (see Section 2.2.3) are considered, the definition of the neighborhood of a site implies that a sublattice is potentially its neighbor.

To simplify the operations of extraction of outgoing densities and setting of incoming densities, following constraint is enforced on lattice partitioning: all sublattices have one and only one neighboring sublattice by face or edge. Figure 2.10 illustrates a lattice partitioning applying this constraint and a partitioning that does not. The greyed sublattices of situation (b) have more than one neighboring sublattice for one of their faces.

Each sublattice has a unique identifier represented by an integer. The neighborhood of sublattices can be represented by a 3D array of sublattice identifiers called *sublattices neighborhood array* and noted `SN`. Each identifier is present exactly one time in the array. The *position* (x,y,z) of a sublattice with identifier i in the `SN` implies that `SN[x, y, z] = i`.

As stated in Section 2.4.1, the faces and edges of a lattice (and therefore a sublattice) can be associated to a neighborhood vector. Let A be a sublattice at position $\mathbf{p}$, $\mathbf{n}_j$ be a neighborhood vector, $\mathbf{r}$ be the size of `SN` and B be a sublattice at position $(\mathbf{p}+\mathbf{n}_j)$ *mod* $\mathbf{r}$ (the modulus implies full periodic boundary conditions). The outgoing densities of sublattice A from the face or edge associated to vector $\mathbf{n}_j$ are used by sublattice B as incoming densities for the face or edge associated to the opposite vector of $\mathbf{n}_j$.

Figure 2.11 illustrates the situation where the outgoing densities of the face of sublattice A associated to neighborhood vector $\mathbf{n}_1$ are used to set the incoming densities of the face of sublattice B associated to vector $\mathbf{n}_2$. Note that Figure 2.11 is very similar to Figure 2.9, the only difference is that, in Figure 2.11, the two cubes represent two distinct sublattices instead of the same lattice.

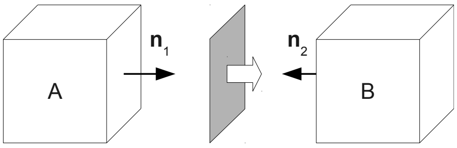

Figure 2.11: Outgoing densities of the face associated to neighbor vector $\mathbf{n}_1$ of sublattice A are used as incoming densities for the face associated to neighbor vector $\mathbf{n}_2$ of sublattice B.

2.5.2 Lattice Partitioning

Let (s_x, s_y, s_z) be the size of the lattice to partition and let N be the number of sublattices to obtain (chosen according to the expected number of processes). The partitioning problem can be reduced to finding the *partitioning factors* q_x, q_y and q_z such as $N = q_x q_y q_z$ and $q_\alpha < s_\alpha$ with $\alpha = x, y, z$. The partitioning factors give the size of the sublattices neighborhood array defined in previous section. Each dimension α is then divided into q_α equal parts if s_α is divisible by q_α with $\alpha = x, y, z$. The N sublattices then have size $(s_x/q_x, s_y/q_y, s_z/q_z)$.

If s_α is not divisible by q_α, sublattices at positions $\mathbf{p}$ with $0 \leq p_\alpha < (s_\alpha \bmod q_\alpha)$ have size $\mathbf{z}$ with $z_\alpha = s_\alpha/q_\alpha + 1$. Sublattices at positions $\mathbf{p}'$ with $(s_\alpha \bmod q_\alpha) \leq p'_\alpha < q_\alpha$ have size $\mathbf{z}'$ with $z'_\alpha = s_\alpha/q_\alpha$.

A simple method to partition the lattice is to slice it: for example, q_x and q_y are equal to 1 and $q_z = N$. A drawback of this method is that it cannot be applied if $N > s_z$. In addition, it has been shown [39] that the number of border sites of a sublattice for a fixed total number of sites is minimized when a sublattice is a cube. The more the shape of the sublattice is far from a cube, the larger is the number of border sites.

Thus, the partitioning problem can be described as a constrained optimization problem: find q_x, q_y and q_z such as

1. $q_x, q_y, q_z \in \mathbb{N}^\star$,
2. $N = q_x q_y q_z$,
3. $(s_x/q_x)^2 + (s_y/q_y)^2 + (s_z/q_z)^2$ is minimized.

We solved this problem algorithmically: a prime factorization is first applied on N, the obtained factors are then multiplied until three factors remain. These three factors are the partitioning factors. Algorithm 2.11 generates the list of prime factors of N. The result of this algorithm is a list sorted in descending order containing at least one prime factor of N (N itself if it is a prime number).

```
var
   x, d :  integer;
   e :  "list of integers";
begin
x, d, e := N, 2, "empty list";
do x > 1 →
   if x mod d = 0 →
      "Add d at the beginning of e";
      x := x div d
   □ x mod d ≠ 0 → d := d + 1
   fi;
   d := d + 1
od
end
```

Algorithm 2.11: Decomposition of N into a list of prime numbers.

Partitioning factors are then calculated using this list with Algorithm 2.12. Partitioning factors (initially equal to one) are modified only by multiplying themselves with a prime factor of N that was not already used. This ensures that $N = q_x q_y q_z$ after all factors have been used. At each iteration, the partitioning factor associated to the largest dimension is increased. Since the list of prime factors is sorted in descending order, the largest factors are applied to the largest dimensions. This should prevent the production of degenerated sublattices (i.e. far from cubes) if possible.

2.5.3 Parallel Simulation

As stated previously, in order to execute an LB simulation in parallel, a lattice can be subdivided into sublattices. The simulation code can then be executed on each sublattice by a separate process.

A master process initially partitions the lattice using the method described in previous section and associates a sublattice to each worker process (a process executing the simulation code). Following information is provided by the master to the workers:

- a sublattice and the associated subsolid,

```
var
   x, d :  integer;
   L : "list of dividers of N";
begin
sx, sy, sz := xSize, ySize, zSize;
qx, qy, qz := 1, 1, 1;
do "next integer of L exists" →
   d := "get next integer of L";
   m := max(sx, sy, sz);
   if sx = m →
      qx := qx * d; sx := sx div d
   □ sy = m →
      qy := qy * d; sy := sy div d
   □ sz = m →
      qz := qz * d; sz := sz div d
   fi
od
end
```

Algorithm 2.12: Computation of q_x, q_y and q_z using the list of dividers of N.

- the size of array `SN` and the position **p** of this sublattice in `SN`,
- the neighborhood of the process.

The LB simulation code executed by each worker processor is described by Algorithm 2.13. It is a distributed version of Algorithm 2.8.

```
"Wait information from master";
"Initialize sublattice";
"Fill subsolid";
t := 0;
do t < timeSteps →
   "Copy outgoing densities to buffers";
   "Send sublattice's outgoing densities to neighbors";
   "Propagate sublattice values";
   "Receive outgoing densities from neighbors";
   "Use received outgoing densities to set incoming densities";
   "Apply sublattice boundary conditions";
   "Apply sublattice collision";
   "Wait all outgoing densities have been sent";
   t := t + 1
od
```

Algorithm 2.13: Parallel LB simulation algorithm.

`"Wait information from master"` blocks until all required information is received. `"Copy outgoing densities to buffers"` and `"Propagate sublattice values"` commands were previously described by algorithms 2.9 and 2.7 respectively. `"Wait all outgoing densities have been sent"` ensures that command `"Copy outgoing densities to buffers"` does not overwrite data before they are completely transmitted.

Sublattice Incoming and Outgoing Densities

Let S be the set of all border sites of all sublattices and B the set of global lattice border sites. If there are several sublattices, B is a subset of S. `"Apply sublattice boundary conditions"` sets the incoming densities of the sublattice's border sites that are also in B. Thus, it is described by a slightly modified version of Algorithm 2.3: let $\mathbf{p}$ be the position of the sublattice (see Section 2.5.1), the incoming densities of the inflow plane are set only if sublattice's inflow plane is included in the lattice's inflow plane; if flow direction is the same as α-axis (with $\alpha = x, y, z$), this is true if $p_\alpha = 0$.

Similarly, the incoming densities of the outflow plane are set only if sublattice's outflow plane is included in lattice's outflow plane. If flow direction is the same as α-axis (with $\alpha = x, y, z$), this is true if $p_\alpha = q_\alpha - 1$ where q_α is the partitioning factor associated to dimension α.

In Algorithm 2.8 (sequential implementation), the outgoing densities of the lattice are copied into the output buffers. The content of these buffers is then used in order to set lattice's incoming densities. With sublattices, the content of output buffers cannot be used anymore to set incoming densities because the outgoing densities to use potentially come from another sublattice.

To store the outgoing densities coming from other sublattices, *input buffers* are defined. In the same way than output buffers, each input buffer is associated to a face or an edge of the sublattice and thus to a neighborhood vector. They can be represented by 3D arrays for faces or 1D arrays for edges. Input buffers can be declared as follows:

```
var
   in1 :  array[0..ySize-1, 0..zSize-1, 0..4] of real;
   "Declare other face input buffers";
   in11 :  array[0..ySize-1] of real;
   "Declare other edge input buffers"
```

`"Send sublattice's outgoing densities to neighbors"` sends the content of output buffers to the neighboring processes. `"Receive outgoing densities from neighbors"` receives all outgoing densities from neighboring processes and stores them into the input buffers such as the content of output buffer i is put into input buffer j with $\mathbf{n}_i = -\mathbf{n}_j$. Incoming densities are set only when all outgoing densities have been received from neighbors. This operation is therefore blocking.

`"Use received outgoing densities to set incoming densities"` is described by Algorithm 2.14.

```
{Use input buffer 2 to set incoming densities}
"Initialize y, z for $y^0z^0$ scan";
do "$y^0z^0$ scan is not finished" →
   f[0, y, z, 1] := in2[y, z, 0];
   if y + 1 < ySize → f[0, y+1, z, 7] := in2[y, z, 1];
   □ y + 1 = ySize → skip
   fi;
   if y - 1 >= 0 → f[0, y-1, z, 9] := in2[y, z, 2];
   □ y - 1 < 0 → skip
   fi;
   if z + 1 < zSize → f[0, y, z+1, 11] := in2[y, z, 3];
   □ z + 1 = zSize → skip
   fi;
   if z - 1 >= 0 → f[0, y, z-1, 13] := in2[y, z, 4];
   □ z - 1 < 0 → skip
   fi;
   "Go to $y^0z^0$ next"
od;

"Use other face input buffers to set incoming densities";

{Use input buffer 14 to set incoming densities}
y := 0;
do y < ySize-1 →
   f[0, y, 0, 11] := in14[y];
   y := y + 1
od;

"Use other edge input buffers to set incoming densities"
```

Algorithm 2.14: Set sublattice incoming densities.

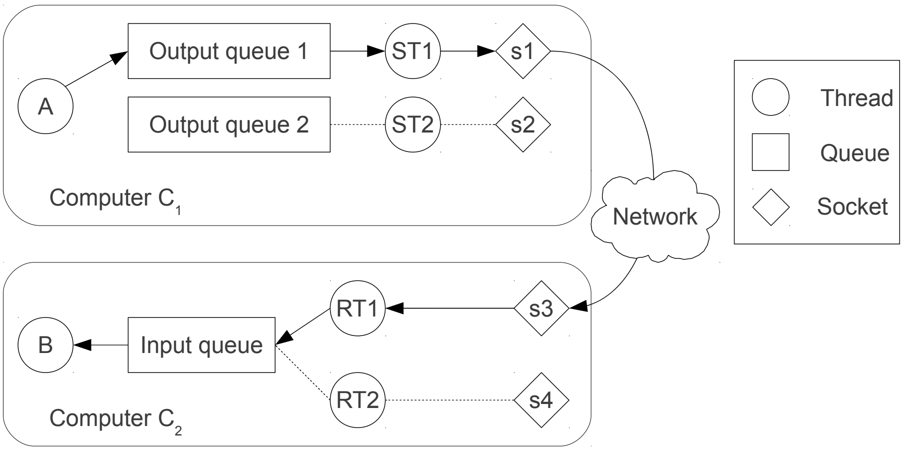

Figure 2.12: Transmission queues system.

2.5.4 Data Transmission

Our implementation of LB simulations is written using the Java programming language. To take advantage of several CPUs of a computer, several threads can be instantiated.

Transmission queues can be used to exchange messages between threads on a shared memory architecture. For example, a thread A sends a message to thread B by putting it into a queue. Thread B can then retrieve the message from the queue. Another queue is used by B to send messages to A. There must be two queues by pair of neighboring threads to obtain full-duplex communication.

In the context of a distributed implementation of LB simulations, communications occur between threads potentially distributed among different computers. Data are then transferred through a network using sockets (endpoints of a bidirectional communication) to send and receive bytes.

The above system based on transmission queues can easily be combined with socket communications. For bidirectional communications between two computers, one socket is needed on each computer but two threads are associated to each socket: a sending thread (extracting messages from a queue and sending them through a socket) and a receiving thread (reading messages from a socket and putting them into a queue).

Such a communication system is illustrated in figure 2.12. Let A and $ST1$ be threads running on computer C_1 and B and $RT1$ be threads running on computer C_2. Socket s_1 connects C_1 to C_2 on C_1 and socket s_3 connects C_2 to C_1 on C_2. Note that s_1 and s_3 are the endpoints of the same communication channel. If thread A

wants to send a message to thread B, it puts a message in output queue 1. Thread $ST1$ extracts the message from output queue 1 and sends it to computer C_2 using socket s_1 (the message is converted into a bytes stream). Thread $RT1$ running on C_2 then reads the message from socket s_3 (the message is reconstructed from a byte stream) and puts it in the input queue. Thread B can then extract the message from the input queue. Thread $ST2$ sends messages to another computer than C_2 and thread $RT2$ receives messages from another computer than C_1.

The messages a worker thread sends during an LB simulation to a neighbor thread can be declared in the following way:

```
type
   Message = record
      outId :  integer;
      data :  "output buffer content";
   end
```

where `outId` is the identifier of an output buffer and `data` a pointer to the content of the output buffer identified by `outId`. `"Send sublattice outgoing densities to neighbors"` command introduced in previous section puts such messages in an output queue. To select the output queue, the destination computer must be known. This information is given by master process when providing thread's neighborhood.

`"Receive sublattice incoming densities"` retrieves messages such as defined above from input queue and fills input buffers with their content (the `data` pointer does not point to an output buffer but to a copy of it on destination computer). Field `outId` is used to choose the input buffer to fill (if `outId` = i, input buffer j is chosen such as $\mathbf{n}_i + \mathbf{n}_j = \mathbf{0}$). This input buffer is then filled with the content pointed by `data`.

In the transmission system described previously, a message has to be converted into a byte stream to be sent using a socket. Also, a byte stream read from a socket will have to be converted into a message. Java provides an obvious solution to this problem: object serialization. Object serialization is therefore used by sending and receiving threads.

2.6 Distributed LB Simulation Speedup

Speedup quantifies how much a parallel program is faster than its corresponding sequential version. It is noted S_p where p is the number of processors used and is defined as follows:

$$S_p = \frac{T_1}{T_p}$$

where T_1 is the execution time of the sequential version and T_p the execution time using p processors. The ideal speedup is obtained when $S_p = p$.

The *efficiency* of a parallel program is defined as follows:

$$E_p = \frac{S_p}{p}$$

It varies between 0 and 1 with programs featuring and ideal speedup having an efficiency of 1.

To evaluate the speedup and the efficiency of our distributed LB simulation implementation, a simulation on a lattice of $(64,64,64)$ sites using the SRT collision operator has been executed during 200 time steps on a cluster of Pentium Celeron 2.4 Ghz processors connected by a switched-ethernet 100 Mbits network.

The obtained execution times are shown in Figure 2.13. We can see that the execution time clearly decreases when the number of processors grows. The execution time of the simulation is divided by almost 10 when executed on 32 computers.

However, Figure 2.14 shows that the speedup is far from ideal.

Figure 2.15 shows the efficiency of the LB simulation distributed implementation. Starting with 0.82 with 2 processors, the efficiency is only 0.31 with 32 processors.

The observed poor efficiency is explained by the fact that the percentage of total execution time spent in synchronization and data transmission increases with the number of processors, computers are inter-connected with a “slow” (in comparison to high-bandwidth technologies like Infiniband generally used in the context of High Performance Computing) network and, finally, additional operations are required regarding a sequential simulation to copy outgoing densities from a sublattice to output buffers and copy the content of input buffers into a sublattice.

The efficiency of a parallel or distributed implementation depends on the ratio of computation time to transmission time. A high value of this ratio leads to a good efficiency. In the context of distributed LB simulations, the value of this ratio

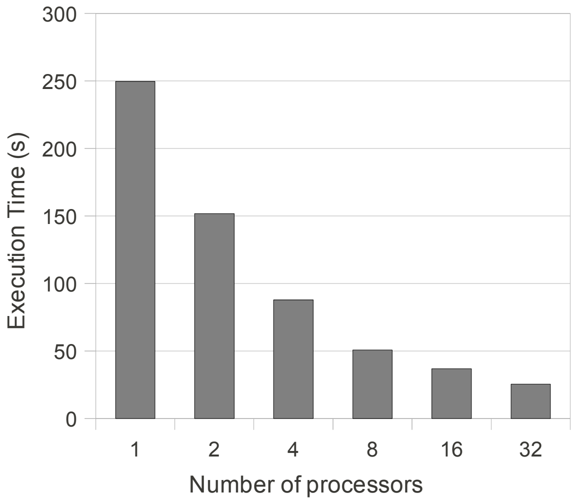

Figure 2.13: Execution time of distributed LB simulation.

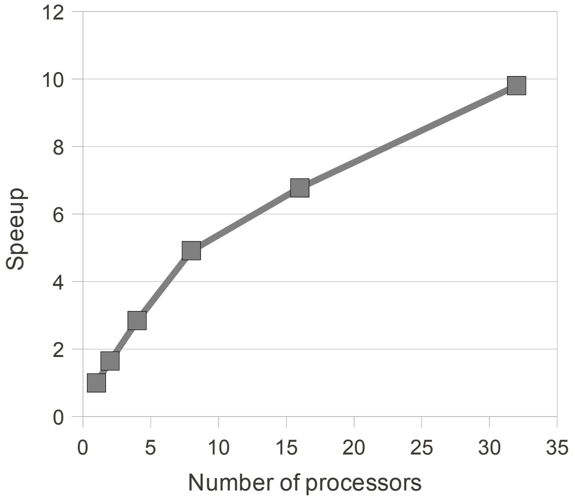

Figure 2.14: Parallel LB simulation speedup.

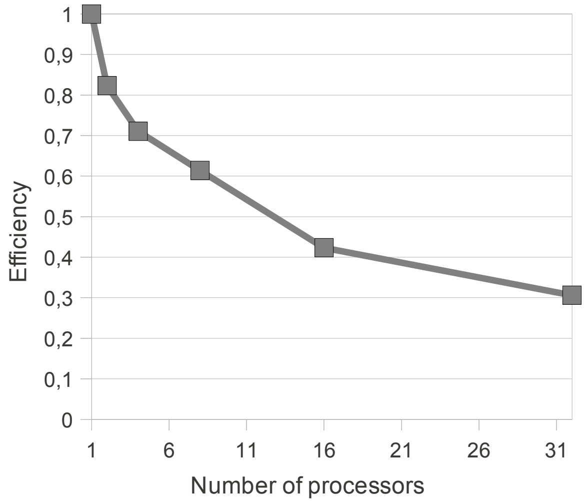

Figure 2.15: Parallel LB simulation efficiency.

depends on the size of the sublattice and the computational complexity of the simulation code. In particular, the SRT collision operator has a lower computational complexity than the MRT operator.

Figure 2.16 shows the efficiency of previous simulation (64-SRT) compared to a simulation on a lattice of $(32,32,32)$ sites (32-SRT) and on the same lattice of $(64,64,64)$ sites but using the MRT collision operator (64-MRT). As expected, a higher efficiency is obtained when using a more complex simulation codes (MRT instead of SRT) and a lower efficiency is obtained with a smaller lattice (and therefore smaller sublattices).

2.7 Conclusion

In this chapter, we presented a class of CFD methods called Lattice Boltzmann (LB) methods. These are interesting for simulating fluid flows in complex boundaries and are well suited to parallel and distributed computing.

Distributed computing introduces a potential portability problem that is solved by implementing distributed LB simulations in Java. It has been shown [15] that Java is an acceptable compromise between efficiency and portability.

Simple sequential and parallel implementations have been presented. The parallel implementation has been evaluated by executing distributed LB simulations on up to 32 computers inter-connected by a switched-ethernet 100 Mbits network.

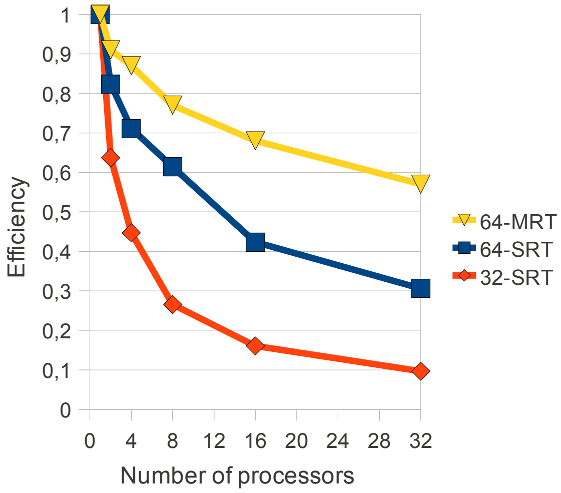

Figure 2.16: Comparison of efficiency when using different simulation parameters.

We evaluated the parallel efficiency of our implementation of distributed LB simulations and observed that it mainly depends on sublattices size (which depends on the number of processors/computers and the size of the lattice) and the computational complexity of the collision operator. For example, we observed a “good” efficiency of 0.91 for a simulation on a $(64,64,64)$ lattice using an MRT collision operator executed on 2 computers but a “bad” efficiency of 0.10 for a simulation on a $(32,32,32)$ lattice using an SRT collision operator executed on 32 computers.

A poor parallel efficiency does not mean that the execution time was not reduced by using more computers. In the context of the simulation on the $(64,64,64)$ lattice using an MRT collision operator, the execution time was divided by a factor 18.27 when using 32 computers instead of a single one and by 2.64 with a $(32,32,32)$ lattice and using the SRT collision operator. It is therefore generally more interesting to execute a simulation in parallel than sequentially.

Chapter 3

Optimized Implementation of Lattice Boltzmann Simulations

3.1 Introduction

In Chapter 2, LB methods have been presented and a simple sequential implementation was proposed. A parallel version of this implementation was also presented.

LB simulations imply massive memory accesses. The sequential implementation can therefore be optimized by ensuring a better data locality (i.e. data associated to addresses that are close in memory address space should be accessed at close moments in time).

A first way to improve data locality is to change the representation of multi-dimensional arrays: Java provides an arrays-of-arrays representation that does not necessarily ensure a good data locality. An alternative storage scheme is to represent a multi-dimensional array in a one-dimensional array and explicitly (i.e. at application level) compute the position of an element in the 1D array given its position in the multi-dimensional array.

This new representation allows to apply an optimization method suggested by Murphy [58] which reduces the complexity of propagation (see Section 2.4). However, the representation required by Murphy's method does not ensure the data locality principle during collision pahse (see Section 2.4). To solve this, the collision implementation has been adapted.

Registers
Cache Memory
Main Memory

Figure 3.1: Hierarchical organization of memory.

3.1.1 Chapter Outline

In Section 3.2, the data locality principle is defined and the representation of multidimensional arrays in Java is discussed.

Section 3.3 describes how propagation can be improved with a method inspired from the work of Murphy [58].

Section 3.4 presents how collision's implementation can be adapted to the new storage scheme to further improve implementation's efficiency.

The simple implementation presented in Chapter 2 is compared to the optimized implementation in Section 3.5.

Finally, Section 3.6 concludes this chapter.

3.2 Data Locality and Representation of Multi-dimensional Arrays

In today's computers, memory has a hierarchical organization which can be visualized as a layered pyramid [39] (see Figure 3.1). Layers at the bottom are large (they can contain many bytes) but have high access times and layers at the top are small but have low access times. Data used as operands of an instruction are loaded into the registers from cache memory (which has generally also several levels having same properties as the layers of the "memory pyramid"). If data are not available in cache memory, a *cache miss* occurs and the program's execution is interrupted until data are transferred from main memory to cache memory. The minimum amount of data that can be loaded into cache memory is given by a value called *cache line size*. When a cache miss occurs, a portion of main memory is copied into cache memory.

Typical code optimization includes the minimization of the number of cache misses by using an optimal data organisation in memory and an optimal data ac-

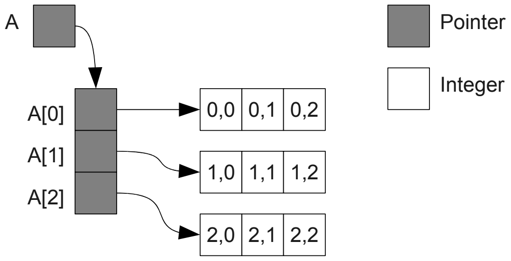

Figure 3.2: 2D array A of 3×3 integers represented using arrays of arrays.

cess “scheduling”. The *data locality* (data brought into cache are used at least once before being flushed out of it) principle is then ensured.

In the context of LB simulations on a D3Q19 lattice, 4D arrays of double precision floating point values are used. Some languages (C/C++ with static allocation, Fortran) support directly multi-dimensional arrays. Other languages (Java, C/C++ with dynamic allocation) support multi-dimensional arrays indirectly by using arrays of arrays (see Figure 3.2 for an example of a 2D array).

Arrays of arrays allow more flexibility (e.g. in a 2D, lines can have different sizes or even not be allocated) but have drawbacks: data locality is not necessarily optimized (if accessing elements line by line in a 2D array, two subsequent lines are not necessarily contiguous) and many pointers have to be stored in arrays in addition to data.

In general, for a d-dimensional array of size $(s_1, s_2, \ldots, s_d)$, the number of additional pointers that have to be stored is given by:

$$\sum_{i=1}^{d-1} \prod_{j=1}^{i} s_j \tag{3.1}$$

If arrays of arrays are used to represent the 4-dimensional array used for a lattice of size (s, s, s) (see Section 2.4), the total amount of memory (expressed in bytes) needed by the lattice is given by $T(s) = D(s) + P(s)$ where D is the memory needed for the values and P the memory needed for the pointers. The values $D(s)$ and $P(s)$ can be calculated as follows:

$$D(s) = 19 s_d s^3 \tag{3.2}$$

$$P(s) = s_p(s + s^2 + s^3) \tag{3.3}$$

where s_d is the size of a double precision floating point value in bytes and s_p the size of a pointer, also expressed in bytes. The value $R^{val}(s) = D(s)/(D(s) + P(s))$

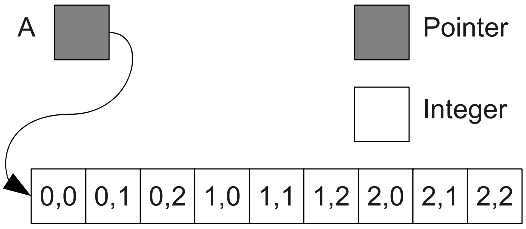

Figure 3.3: 2D array of 3 × 3 integers represented using a 1D array.

gives the percentage of total memory allocated for an array actually used to store values. In the context of the 4D array used to store densities in LB simulations, $R^{val}(s)$ can be rewritten as follows:

$$\begin{aligned} R^{val}(s) &= \frac{19 s_d s^3}{19 s_d s^3 + s_p(s + s^2 + s^3)} \\ &= \frac{19 s_d}{19 s_d + s_p(s^{-2} + s^{-1} + s^0)} \end{aligned}$$

In the minimal case ($s = 1$),

$$R^{val}_{min} = R^{val}(1) = \frac{19 s_d}{19 s_d + 3 s_p} \tag{3.4}$$

and when s tends to infinity,

$$R^{val}_{max} = \lim_{s \to \infty} R^{val}(s) = \frac{19 s_d}{19 s_d + s_p} \tag{3.5}$$

A double precision floating point value is represented by 8 bytes (64 bits) so $s_d = 8$. On a 64 bits architecture, a pointer is also represented on 8 bytes ($s_p = 8$). In this situation, $R^{val}_{min} \approx 86\%$ and $R^{val}_{max} = 95\%$. This means that the memory "wasted" by pointers when using arrays of arrays is at most of about 14% but at least 5% of total used memory. For example, when $s = 100$, $D(100) = 152 \times 10^6$ ($\approx$ 145 Mbytes), $P(100) = 8080800$ ($\approx$ 8 MBytes) and $R^{val}(100) \approx 94\%$.

In order to represent lattices (and solids), the flexibility of arrays of arrays is not needed. The drawbacks of this technique can be avoided by representing multi-dimensional arrays in a 1D array (as done internally in languages supporting directly multi-dimensional arrays). Figure 3.3 shows the representation of array A from Figure 3.2 in a 1D array. No additional pointers are used and consecutive lines are contiguous in memory.

In general, for a dD array of size $(s_1, s_2, \ldots, s_d)$, a 1D array of size $\prod_{i=1}^{d} s_i$ is needed. An *index function* $\delta : \mathbb{N}^d \to \mathbb{N}$ that, from the position $(i_1, i_2, \ldots, i_d)$ (with $0 \le i_j < s_i$ for $i = 1, 2, \ldots, d$) of an element of the dD array, gives the position in the 1D array must also be defined. Following index function can be used:

$$\delta(i_1, i_2, \ldots, i_d) = \sum_{j=1}^{d} i_j \prod_{k=j+1}^{d} s_k \tag{3.6}$$

In the example given in Figure 3.3, $\delta(i_1, i_2) = i_1 \times 3 + i_2$ with $0 \le i_1, i_2 < 3$.

Let vf be the 1D representation of a D3Q19 lattice of size $(xSize, ySize, zSize)$. The size of vf (in number of floating point values) is given by $xSize \times ySize \times zSize \times 19$. To have the same data organization as used in simple implementation (see Chapter 2), the following index function must be used:

$$\begin{aligned}
\delta(x, y, z, q) \quad = \quad & x \times (ySize \times zSize \times 19) + \\
& y \times (zSize \times 19) + \\
& z \times 19 + \\
& q
\end{aligned}$$

With this representation, in order to ensure data locality, lattice elements should be accessed in the following way:

```
i := 0;
do i < xSize*ySize*zSize*19 →
   "Access vf[i]";
   i := i + 1
od
```

Following loop on x, y, z and q accesses elements in the same order than previous loop (for scan notation, see Section 2.4.1) with function `delta(x,y,z,q)` returning $\delta(x, y, z, q)$:

```
"Initialize $x^0y^0z^0q^0$ scan";
do "$x^0y^0z^0q^0$ is not finished" →
   "Access vf[delta(x, y, z, q)]";
   "Go to $x^0y^0z^0q^0$ next"
od
```

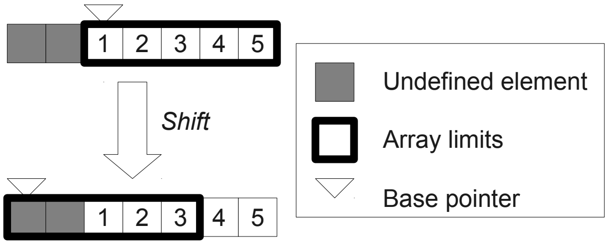

Figure 3.4: Array shift with base pointer move.

The collision operator's implementation presented in Section 2.4 ensures data locality but propagation's implementation does not.

3.3 Optimizing Propagation Step

An optimized propagation method based on an alternative data organization has been proposed by Murphy [58]. This method significantly reduces the complexity of propagation (without the copy of outgoing and incoming densities, see Section 2.4.1): for a cubic lattice of size (L,L,L), complexity becomes $O(1)$ instead of $O(L^3)$). We slightly modified the method to reduce its memory overhead and to adapt it to the Java language which does not allow pointer arithmetic required by Murphy's technique.

3.3.1 Array Shift

Murphy's method is based on the constant time shift of elements of an array. The shift operation is implemented by moving the base pointer (pointer to the first element of the array) of the array to the "right" (base pointer is increased) or to the "left" (base pointer is decreased).

For example, Figure 3.4 illustrates the shift an array containing five elements two positions to the right. The base pointer of the array is moved two positions to the left. As shown in the figure, an element at position i in the array before shift has position $i+2$ after shift.

With a shift of n positions to the right, n elements are undefined at the beginning of the array. With a shift of n positions to the left, n elements are undefined at the end of the array.

The shift of the elements of an array is defined by an integer called the *offset*: with an offset equal to n, after the shift, the element at position i is at position $i+n$

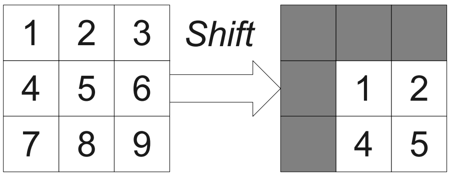

Figure 3.5: 2D array shift using offset vector $(1,1)$.

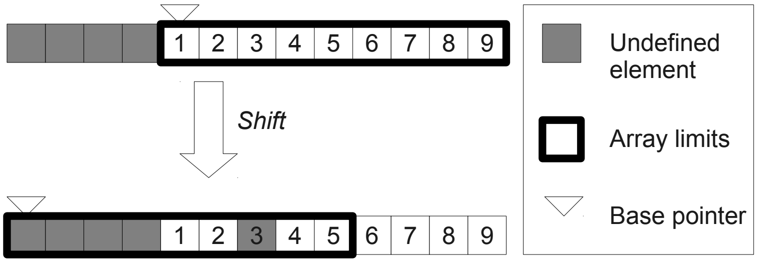

Figure 3.6: Shift of the vector representation of a (3×3) array.

if it is in the array boundaries. If n is positive, elements are shifted to the right. If n is negative, elements are shifted to the left.

The shift of the elements of a dD array is defined by a vector called *offset vector*: with an offset vector v an element at position $\mathbf{p}$ has position $\mathbf{p}+\mathbf{v}$ after the shift if it is in the array boundaries. Figure 3.5 illustrates the shift of the elements of a 2D array of size $(3,3)$ with an offset vector $(1,1)$.

If a dD array is represented using a 1D array, the shift of its elements using vector $\mathbf{v}$ is obtained by shifting the 1D array elements using offset $\delta(\mathbf{v})$. Figure 3.6 illustrates the shift of the elements of the simple array backing the 2D array of Figure 3.5 with offset $\delta(1,1) = 1 \times 3 + 1 = 4$. In the figure, the element with label "3" has been marked as undefined. Its value is, in fact, known but does not make sense in the context of the multi-dimensional array shift.

Note that in order to use the base array pointer move method to shift elements of an array without potentially overwriting required data in memory, enough memory needs to be reserved to the left and/or to the right of the array.

3.3.2 Propagation Based on Array Shifting

In-place propagation presented in Section 2.4.1 (without extraction of outgoing densities and setting of incoming densities) can be implemented using multi-dimensional array shifting if lattice values are reorganized: the values associ-

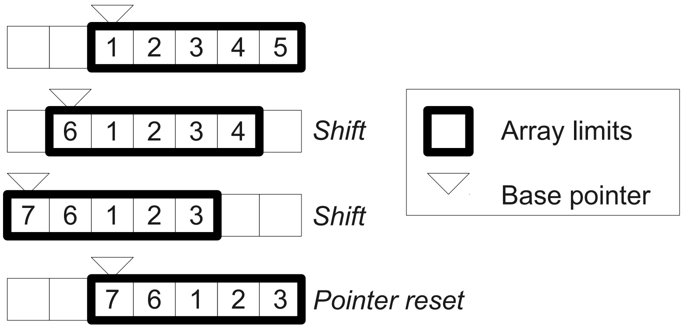

Figure 3.7: Two left shifts and a pointer reset.

ated to each velocity should be grouped in separate 3D arrays (represented by 1D arrays). Instead of a 4D array, a D3Q19 lattice is then represented by 19 one-dimensional arrays. The index function $\delta(x,y,z)$ for these arrays is given by:

$$\begin{aligned}\delta(x,y,z) &= x\times(ySize\times zSize)+\\ &\quad y\times(zSize)+\\ &\quad z\end{aligned}$$

As stated in Section 2.4.1, propagation is the application of a translation vector to all values associated to each velocity. Propagation can therefore be implemented by shifting each of the 19 arrays of new representation by a fixed offset.

The offset d_i to apply to the array associated to velocity i is given by $\delta(\mathbf{n}_i)$ where $\mathbf{n}_i$ is the neighborhood vector associated to velocity i (see Chapter 2).

As stated previously, memory must be reserved around the array in order to move the array base pointer without overwriting other used data. In its implementation, Murphy allocates a supplementary space of $|d_i|\times k$ values. This way, k shifts can occur before no more space is available to continue to shift array base pointers. At this point, array elements are copied back to their initial positions and the base pointer is reset. k new shifts can then occur again. Figure 3.7 shows an array of five elements with 2 additional positions reserved at its left. Two right shifts with an offset of 1 can then be performed before pointer and data are reset.

3.3.3 Circular Array Shift

To avoid the memory overhead and pointer reset operation of Murphy's method, we use a *circular array*: the base pointer never moves but an offset pointing to the actual first position of the circular array can be displaced (see Figure 3.8).

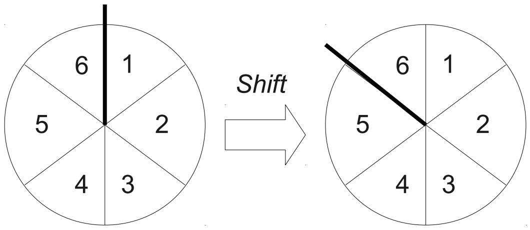

Figure 3.8: Circular array and one position right shift.

A circular array represented using a simple array can be declared as follows:

```
var
   v :  array[0..N-1] of "type";
   off :  integer
```

where v contains the data of the circular array, N is the size of the circular array and off the index of the first element of the array. The element at position *i* in the circular array is at position $(off+i)$ mod N in v.

To apply an offset *a* to the elements of v, the following instructions are executed:

```
off := (off - a) mod N
```

In an implementation of propagation using circular array shift, a D3Q19 lattice can be represented as follows:

```
var
   f :  array[0..18, 0..M-1] of real;
   offs :  array[0..18] of integer
```

where f contains the density distribution function values, M is equal to *xSize* × *ySize* × *zSize* and offs contains the offsets associated to each 3D array representing the values associated to each velocity. Initially, offs is filled with zeros.

Propagation can then be implemented as described by Algorithm 3.1. This algorithm can replace "Propagate values" in Algorithm 2.8 describing an LB simulation with in-place propagation. It can also be used in the parallel LB simulation's implementation.

```
{Velocity 0 :  Nothing to do}

{Velocity 1 :  translation (1,0,0)}
offs[1] := (offs[1] - ySize * zSize) mod M;

{Velocity 2 :  translation (-1,0,0)}
offs[2] := (offs[2] + ySize * zSize) mod M;

"Apply propagation for velocities 3..17";

{Velocity 18 :  translation (0,-1,-1)}
offs[18] := (offs[18] + zSize + 1) mod M
```

Algorithm 3.1: Circular array shift-based propagation.

3.3.4 Evaluation of New Propagation Implementation

Figure 3.9 shows the execution time of propagation implemented using circular array shifts ("off" in the legend) and the method described in Section 2.4.1 ("simple" in the legend). The x-axis gives the value L for a (L,L,L) lattice. The execution time (y-axis) is the time to execute one propagation on a lattice of given size.

We observe that the optimized method is almost 26 times faster than simple method with a $(80,80,80)$ lattice. This result is not surprising as the complexity of propagation is reduced from $O(L^3)$ to $O(L^2)$ (the quadratic term comes from the copy of densities into output buffers and from input buffers, see Section 2.5.3).

Propagation based on circular array shifts is more efficient than Murphy's method [58] because:

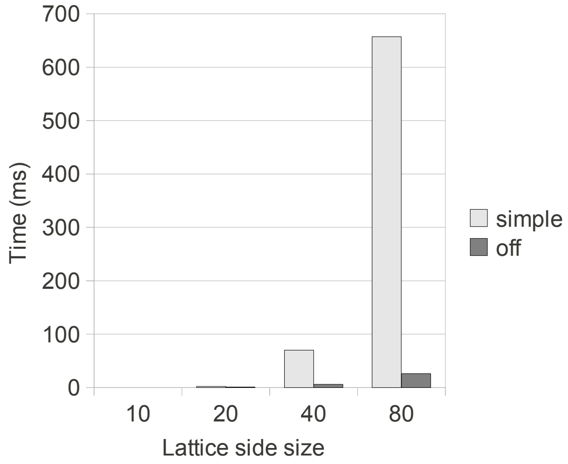

Figure 3.9: Comparison of execution time of propagation

- there is no substantial memory overhead,
- no “pointer reset” operations are needed.

However, the cost of accessing elements of the array is increased because of the additional modulus. This cost is discussed in next section.

3.4 Adapting Collision to New Data Organization

The new data organization dramatically improves the efficiency of propagation operation (see Figure 3.9). However, due to data locality problems and a much more complex element access (access to one element of a circular array implies several additions, multiplications and a modulus), collision operation requires nearly twice as much time as with simple organization (see Figure 3.10).

To reduce the number of cache misses and complex element accesses, the values associated to a part of lattice sites can be copied into a small 2D array (called a *block*) with B lines (B is the maximum number of sites the block can contain) and 19 columns (each line represents the densities of a site). The copy operation (called lattice-block copy) must be written in order to minimize the number of cache misses. Collision is then applied on the block sites and the lattice updated with block data. As for lattice-to-block copy, block-to-lattice copy must minimize the number of cache misses.

A block can be declared as follows:

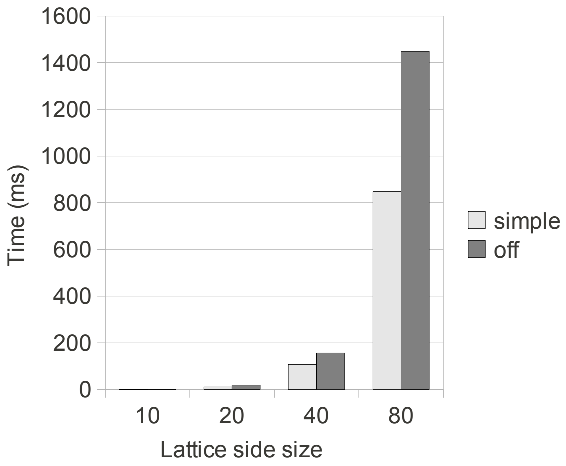

Figure 3.10: Comparison of execution time of collision when using initial and new data organizations.

```
type
   Block = record
      size :  integer;
      data :  array[B, 0..18] of real;
      xPos :  array[B] of integer;
      yPos :  array[B] of integer;
      zPos :  array[B] of integer
   end
```

where `B` is the maximum size of the block, `size` its actual size, `data` the array containing densities and `xPos`, `yPos` and `zPos` arrays containing respectively the *x*, *y* and *z* components of the extracted sites in the lattice.

In order to improve data locality and avoid the complex access to sites, contiguous values in the lattice are copied velocity by velocity into the block. Because circular arrays are used to store densities associated to a given velocity, the area that must be copied into the block can be separated in two: one at the end of the array, one at the beginning. In this case, copy is done in two passes. In Java, array copies have a native implementation. Block copies are therefore efficient operations.

In Algorithm 3.2, lattice-block copy is implemented by procedure `extract-`

`Block`. This procedure is correct if `B < XSIZE * YSIZE * ZSIZE` where `XSIZE`, `YSIZE` and `ZSIZE` are the components of the size of a D3Q19 lattice. The copy of contiguous values of the lattice into a block is described in Algorithm 3.3.

```
procedure extractBlock(start : integer; bl : Block);
var
   yzSize, xyzSize : integer;
begin
   yzSize := YSIZE * ZSIZE;
   xyzSize := XSIZE * yzSize;
   bl.size := min(xyzSize - start, B);

   if bl.size = 0 → skip
   □ bl.size > 0 →
      setXYZPos(start, bl);
      q := 0;
      do q < 19 →
         from := (offs[q] + start) mod xyzSize;
         to := (offs[q] + start + bl.size) mod xyzSize;
         "Copy of densities from lattice to block";
         q := q + 1
      od
   fi
end
```

Algorithm 3.2: `extractBlock` procedure. This procedure implements the lattice-block copy.

Procedure `setXYZPos`, described by Algorithm 3.4, sets the *x*, *y* and *z* components of each site of a block `bl`. After `setXYZPos` is executed, `bl.xPos[i]`, `bl.yPos[i]` and `bl.zPos[i]` are respectively the *x*, *y* and *z* component of `bl`'s `i`th site with $0 \leq$ `i` <`bl.size`.

The principle behind block-lattice copy is similar to lattice-block copy and is implemented by procedure `updateLattice` described by Algorithm 3.6.

The collision implementation can be rewritten using these procedures to modify lattice densities through blocks. Lattice's sites are extracted block by block using procedure `extractBlock`. Collision operator or bounce-back is then applied on an extracted block' sites. The data locality principle is ensured because the

```
if from < to →
   pos := q;
   i := from;
   do i < to →
      bl.data[pos] := f[q, i];
      i, pos := i + 1, pos + 19
   od
□ from > to →
   pos := q;
   i := from;
   do i < xyzSize →
      bl.data[pos] := f[q, i];
      i, pos := i + 1, pos + 19
   od;
   i := 0;
   do i < to →
      bl.data[pos] := f[q, i];
      i, pos := i + 1, pos + 19
   od
fi
```

Algorithm 3.3: "Copy of densities from lattice to block".

```
procedure setXYZPos(start : integer; bl : Block);
var
   yzSize, x, y, z : integer
begin
   yzSize = YSIZE * ZSIZE;

   {Position of the first site of block bl}
   x = (start / yzSize);
   y = (start mod yzSize) / ZSIZE;
   z = start mod ZSIZE;

   site = 0;
   do z < ZSIZE and site < bl.size →
      xPos[site] = x;
      yPos[site] = y;
      zPos[site] = z;
      site, z := site + 1, z + 1
   od;
   if z < ZSIZE → skip
   □ z = ZSIZE →
      z := 0; "Update x, y and z"
   fi;

   do site < bl.size →
      xPos[site] = x;
      yPos[site] = y;
      zPos[site] = z;
      site, z := site + 1, z + 1;
      "Update x, y and z"
   od
end
```

Algorithm 3.4: `setXYZPos` procedure. This procedure sets the x, y, z components of extracted sites in a given block.

```
if z < ZSIZE → skip
□ z = ZSIZE →
   z := 0; y := y + 1;
   if y < YSIZE → skip
   □ y = YSIZE →
      y := 0; x := x + 1
   fi
fi
```

Algorithm 3.5: "Update x, y and z"

```
procedure updateLattice(start : integer; bl : Block);
var
   yzSize, xyzSize : integer;
begin
   if bl.size = 0 → skip
   □ bl.size > 0 → skip
      yzSize := YSIZE * ZSIZE;
      xyzSize := XSIZE * yzSize;
      q := 0;
      do q < 19 →
         from := (offs[q] + start) mod xyzSize;
         to := (offs[q] + start + bl.size) mod xyzSize;
         "Copy of densities from block to lattice";
         q := q + 1
   fi
end
```

Algorithm 3.6: updateLattice procedure. This procedure updates a D3Q19 lattice with a given block's data.

```
if from < to →
   pos := q;
   i := from;
   do i < to →
      f[q, i] := bl.data[pos];
      i, pos := i + 1, pos + 19
   od
□ from > to →
   pos := q;
   i := from;
   do i < xyzSize →
      f[q, i] := bl.data[pos];
      i, pos := i + 1, pos + 19
   od;
   i := 0;
   do i < to →
      f[q, i] := bl.data[pos];
      i, pos := i + 1, pos + 19
      od
   fi
od
```

Algorithm 3.7: `"Copy of densities from block to lattice"`.

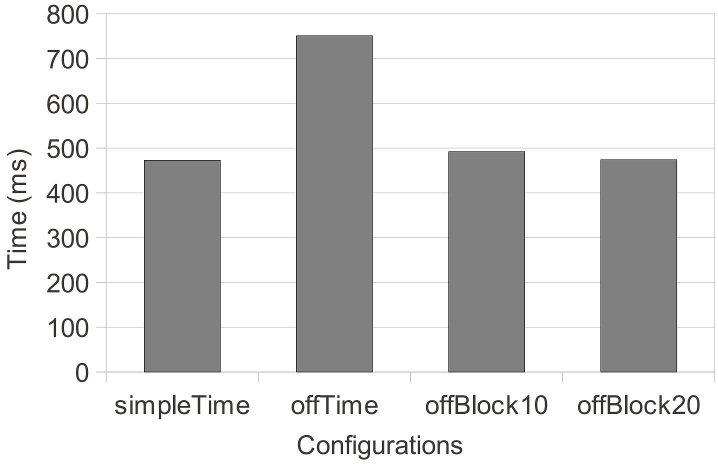

Figure 3.11: Comparison of execution time of collision when using four combinations of data representations and block access parameters.

densities of a site are contiguous in a block. Finally, the updated block is copied back into lattice using `updateLattice` procedure. Algorithm 3.8 describes the new implementation of the collision.

3.4.1 Evaluation of New Collision Implementation

Figure 3.11 shows the execution times of collision using different lattices representations and access methods. Given time (expressed in milliseconds) is the time to execute one collision on all sites of a $(64,64,64)$ D3Q19 lattice. These execution times have been measured on a Pentium Celeron 2.4 Ghz computer of the cluster used in Section 2.6. The labels of the figure's legend have the following meaning:

- simple: Simple data organization.
- off: Propagation optimized data organization, no block access for collision.
- offBlock10: Propagation optimized data organization, block access (maximum block size B=10).
- offBlock20: Propagation optimized data organization, block access (maximum block size B=20).

Unsurprisingly, "off" configuration is the slowest because the data locality principle is not ensured by the data organization introduced in previous section.

```
var
   block : Block;
   k, pos, site, x, y, z : integer
begin
   k := 0;
   do k < XSIZE*YSIZE*ZSIZE →
      {Copy next block's content from the lattice}
      extractBlock(k, block);
      pos, site := 0, 0;
      {Apply collision operator or bounce-back to block's
      sites}
      do site < block.size →
         x := block.xPos[site];
         y := block.yPos[site];
         z := block.zPos[site];
         if s[x, y, z] →
            {Bounce-back}
            block.data[pos] := block.data[pos];
            block.data[pos + 1] := block.data[pos + 2];
            ⋮
            block.data[pos + 18] := block.data[pos + 15];
         □ not s[x, y, z] →
            "Apply collision operator on block[pos..pos + 18]"
         fi;
         site, pos := site + 1, pos + 19
      od;
      {Update lattice with block's content}
      updateLattice(k, block);
      k := k + blockSize
   od
end
```

Algorithm 3.8: Collision using block access method.

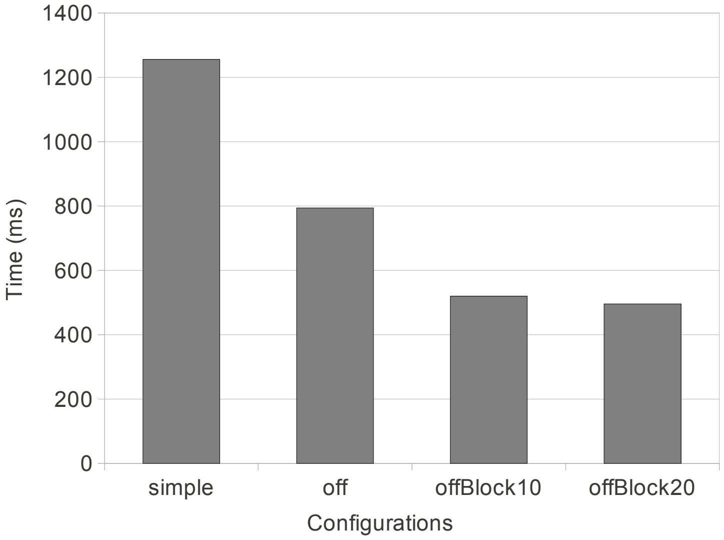

Figure 3.12: Comparison of execution time for complete simulation when using four combinations of data representations and block access parameters.

However, block access method brings collision execution time with optimized data organization at the same level as collision with simple data organization.

3.5 Comparison of Simple and Optimized Implementations

Figure 3.12 shows the execution times for one time step of a complete simulation (propagation, boundary conditions and collision) using the configurations presented above on a $(64,64,64)$ D3Q19 lattice. As can be seen on the figure, optimized implementation is more than twice faster than simple implementation.

The execution time gain of optimized implementation can be computed as follows:

$$1-\frac{t_o}{t_s}$$

where t_o is the execution time of optimized implementation and t_s the execution time of simple implementation. Table 3.1 shows the execution time gain of optimized implementation (using optimized storage scheme and block access with $B=20$) for several lattice sizes.

Figure 3.13 shows the comparison of execution time of a parallel LB simula-

lattice size	Gain (%)
16^3	47
32^3	48
64^3	61

Table 3.1: Execution time gain of optimized implementation regarding simple implementation.

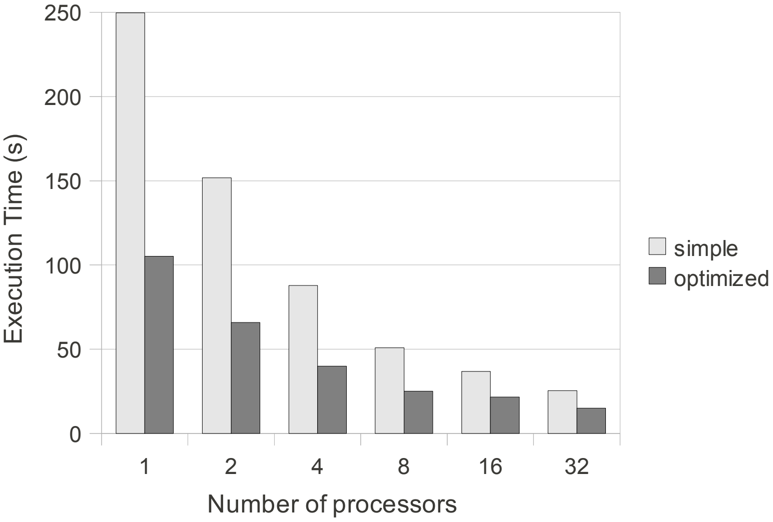

Figure 3.13: Comparison of execution time of simple and optimized simulations' implementation.

tion using simple and optimized implementations. Same setup as in Section 2.6 has been used (200 time steps on a $(64,64,64)$ D3Q19 lattice executed on a Pentium Celeron computers cluster). Figure 3.14 compares the speedup of simple and optimized implementations. Optimized implementation is clearly faster (from 2.38 times with 1 processor to 1.69 times with 32 processors). However, simple implementation features a better speedup. This is due to the fact that simple implementation does not ensure data locality, the subdivision of the lattice into smaller sublattices therefore artificially improves data locality.

On Figure 3.14, a "bump" is observed for the speedup of optimized implementation when using 8 processors. There are two reasons for this:

- With 8 processors, sublattices are all cubes and the number of border sites is minimized (see Section 2.5.2).

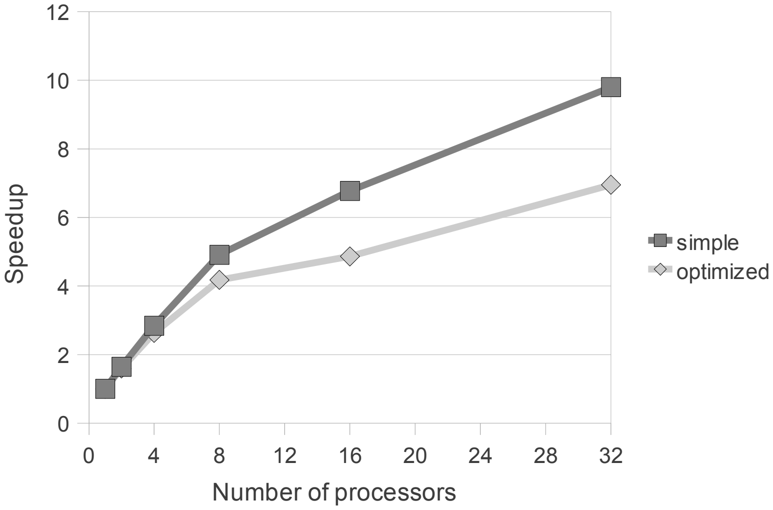

Figure 3.14: Comparison of speedup of simple and optimized simulation

- Block access method described previously has an efficiency that depends on the shape of the lattice (and therefore sublattice). We have observed that a cubic shape for sublattices produces the best results.

3.6 Conclusion

The simple implementation presented in Chapter 2 has been optimized using a method described by Murphy [58] that reduces the execution time of propagation. Murphy's method was adapted in order to take into account the properties of Java. We also improved the method in order to reduce its memory overhead.

In addition, the implementation of collision has been modified in order to take into account the new data representation implied by Murphy's method.

The execution time of LB simulations is more or less reduced by half for both sequential and parallel implementations when proposed optimizations are used instead of the simple implementation presented in Chapter 2. We also observe that the speedup of distributed LB simulations becomes slightly smaller with the optimized implementation. This is because the distribution of the implementation from Chapter 2 artificially improves data locality.

Chapter 4

Architecture of the Simulation Tool

4.1 Introduction

The implementation of distributed LB simulations is presented in chapters 2 and 3. In addition to be efficient, the implementation should be flexible as it may have to be extended or modified regularly. In addition, these extensions and modifications should be accessible to users that may not have high programming skills.

LaBoGrid, the software developed in the context of this thesis, is both a library to ease the writing of any type of LB simulation program (like Palabos [8], Sailfish [10] and El'Beem [3]) and a simulation tool based on LB methods (like PowerFlow® [9]) able to run a sequence of simulations in a distributed way. Thanks to techniques described in chapters 5, 6, 7 and 8, LaBoGrid is adapted to dynamic heterogeneous clusters, which is not the case with above tools.

LaBoGrid is based on an original generic framework presented in this chapter. This framework was developed to ease the writing of software components communicating in an asynchronous way possibly through the network when the components are executed by different computers. We expect this generic framework to introduce a simulation execution time overhead when compared to the specific implementation of chapter 3. However, we will observe that this overhead remains acceptable.

4.1.1 Chapter Outline

In order to explain some architectural choices, several LaBoGrid use cases are presented in Section 4.2.

The Lattice Boltzmann Simulations Library (LBSL), consisting of a collection of generic classes, and a way to describe any LB simulation using this library is given in Section 4.3.

LaBoGrid's implementation and the generic framework it is based on are described in Section 4.4.

Section 4.5 briefly explains how the user can configure LaBoGrid.

The execution times of distributed LB simulations using a specific implementation or LaBoGrid are compared in Section 4.6.

Finally, Section 4.7 concludes this chapter.

4.2 LaBoGrid Use Cases

The Laboratory of Chemical Engineering of the Univeristy of Liège currently uses LaBoGrid to obtain the velocity field of fluids flowing in complex structures like porous media or packed beds. These structures are obtained by X-ray tomography which produces, after numerical reconstruction of the 3D images, large matrices of voxels (containing several millions of voxels). These matrices are used as solids of LB simulations and impose the size of the lattice to use. A solid matrix is stored in a file that LaBoGrid takes as input.

The *result* of an LB simulation is the state of its lattice (i.e. the densities at all sites of the lattice).

Here are some use cases for LaBoGrid and what they imply:

1. *The user wants to run a simulation for a given number of time steps using a given solid file. At the end of the simulation, he wants to get the state of the lattice for further analysis.*

   To run a simulation, the user has to provide at least a solid file and a number of time steps. Also, the state of the lattice should be written to disk at the end of the simulation so the user can run further analysis on it later.

2. *The user wants to run a simulation using a given solid file; the simulation code has to regularly log the speed of the fluid at some given sites of the lattice. The user does not need the state of the lattice at the end of the simulation.*

   The user should have access to specific information about the simulation during its execution. Also, the final lattice state is not required in this case.

The writing of lattice's state to disk at the end of the simulation should therefore be an optional operation. Logging is an interesting feature for two reasons: it can give a hint on the lattice state before the simulation ends and it generally represents far less data than the complete state of the lattice (which can require several gigabytes to be represented).

3. *The user wants to run a simulation using a given solid file; the simulation code has to regularly log the speed of the fluid at some given sites of the lattice before and after the application of collision operator. The user does not need the state of the lattice at the end of the simulation.*

   The user should have a fine control on when the logging occurs (for example, before and after the application of the collision operator, not only at the end of a simulation iteration).

4. *The user wants to compare a new collision operator implementation to the implementation he used before. Therefore, he wants to run two simulations during the same number of time steps and using the same solid starting from a fluid at rest and finally, get the obtained lattice states in order to compare them.*

   The user should be able to easily change some elements of a simulation like lattice type (number of dimensions and velocities), collision operator or boundary conditions and not only their parameters.

5. *The user wants to run a simulation using as initial conditions the result of another simulation.*

   The result of a previous simulation that has been saved to disk (see item 1) can be given to LaBoGrid to set the initial conditions of a simulation.

6. *The user wants to run two subsequent simulations, the second one using the result of first one as starting point. The user only wants to get the result of the second simulation.*

   The result of a simulation that will be used as initial conditions of next simulation in a sequence does not need to be written to a file provided to the user.

More complex use cases can be generated by combining several of the use cases presented above.

4.3 Lattice Boltzmann Simulations Library

As stated in Section 2.3, Java was chosen as implementation language. The Lattice Boltzmann Simulations Library (LBSL), included in LaBoGrid, is therefore a set of classes representing the generic concepts of an LB simulation:

- the lattice,
- the solid,
- the collision operator,
- the boundary conditions.

In the context of parallel and distributed LB simulations, lattice and solid need to be partitioned into sublattices and subsolids (see Section 2.5). Different partitioning methods can be used. These methods depend on the number of dimensions of the lattice and the solid. However, the result is always the same: a collection of partitions. So, two additional concepts can be added for parallel and distributed simulations:

- the partition,
- the partitions collection generator.

4.3.1 Lattice

The `Lattice` class is the base class of all lattices. A lattice has a size represented by an integer vector. The length of this vector is the number of dimensions of the lattice and each component of this vector is the size of corresponding dimension.

A descriptor is associated to lattices: it defines the velocity vectors of the lattice. Lattice representation is the responsibility of the subclasses.

Figure 4.1 shows the class diagram of `Lattice` and `LatticeDescriptor` classes.

The behaviour of `Lattice` class includes:

- The Input/Output of lattice state (densities associated to each site of the lattice) from/to a file (`read` and `write` methods).

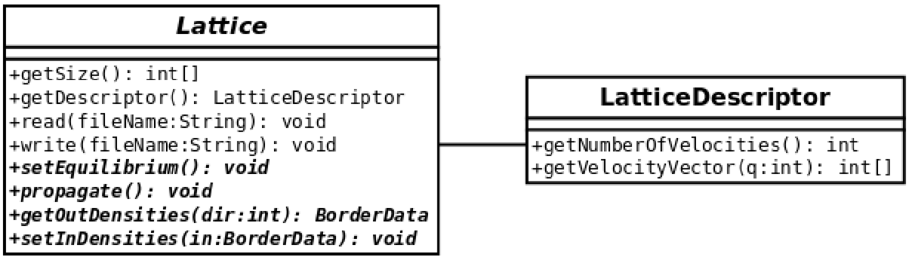
Lattice
+getSize(): int[]
+getDescriptor(): LatticeDescriptor
+read(fileName:String): void
+write(fileName:String): void
+setEquilibrium(): void
+propagate(): void
+getOutDensities(dir:int): BorderData
+setInDensities(in:BorderData): void
LatticeDescriptor
+getNumberOfVelocities(): int
+getVelocityVector(q:int): int[]

Figure 4.1: Class diagrams for `Lattice` and `LatticeDescriptor`.

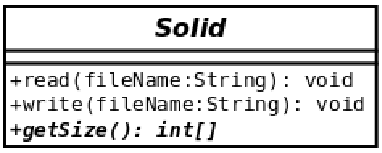

Figure 4.2: Class diagram for `Solid`.

- The initialization of densities leading to a fluid at rest (densities are initialized with equilibrium distribution; `setEquilibrium` method).

- The in-place propagation of densities and output buffers filling (see Section 2.4.1; `propagate` method).

- The extraction of outgoing densities given an output direction (`getOutDensities` method).

- The setting of incoming densities (`setInDensities` method).

The `BorderData` class wraps the outgoing densities associated to an output direction.

4.3.2 Solid

The `Solid` class is the base class of all solids. Like lattices, solids have a size represented by an integer vector and can be read/written from/to a file. A `Solid` instance implements the solid function s defined in Section 2.2.1. In particular, a possible implementation of s is to use a boolean array (see Section 2.4).

Figure 4.2 shows the class diagram of `Solid` class.

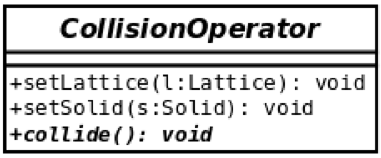

Figure 4.3: Class diagram for `CollisionOperator`.

4.3.3 Collision Operator

The `CollisionOperator` class is the base class of all implementations of the collision operator (like SRT and MRT operators presented in Section 2.2.2). Subclasses of `CollisionOperator` generally also implement bounce-back conditions for fluid-solid interface sites.

Figure 4.3 shows `CollisionOperator` class diagram. The behavior of the class includes:

- the association of a collision operator instance to given lattice and solid (`setLattice` and `setSolid` methods),
- the application of collision operator to all sites of the lattice (`collide` method).

4.3.4 Boundary Conditions

The `BoundaryConditions` class is the base class for all boundary conditions except solid-fluid interface boundary conditions (generally implemented by a subclass of `CollisionOperator`) and periodic boundary conditions (can be implemented directly using `getOutDensities` and `setInDensities` methods from `Lattice` class).

Figure 4.4 shows the class diagram of `BoundaryConditions` class. The behavior of the class includes:

- the association of a boundary conditions instance to given lattice and solid (`setLattice` and `setSolid` methods),
- the application of boundary conditions to some border sites of the lattice (`apply` method).

4.3.5 Generic Sequential Simulation Code Using LBSL

Given the generic classes defined previously, all instances of a sequential LB simulation are captured by Program 4.1. Of course, the specific instantiated classes must be compatible: a collision operator defined for a 2D lattice cannot be used with a 3D lattice, solid and lattice must have same size, etc.

```
int maxT = ...; // The number of time steps
Lattice latt = ...;
Solid solid = ...;
CollisionOperator col = ...;
BoundaryConditions bound = ...;

initLatticeAndSolid(latt, solid);
col.setLattice(latt);
col.setSolid(solid);
bound.setLattice(latt);
bound.setSolid(solid);
LatticeDescriptor desc = fluid.getLatticeDescriptor();
int velNum = desc.getNumberOfVelocities();
for(int i = 0; i < maxT; ++i) {
   latt.propagate();
   for(int q = 0; q < velNum; ++q)
      latt.setInDensities(latt.getOutDensities(q));
   bound.apply();
   col.collide();
}
```

Program 4.1: Generic LB simulation code using LBSL.

The call to function `initLatticeAndSolid` represents the initialization of the lattice and the solid. The instructions of this function depend on the chosen initial

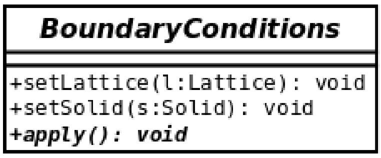

Figure 4.4: Class diagram for `BoundaryConditions`.

conditions.

For example, to start a simulation from a fluid at rest and with a solid read from a file named “solidData”, function `initLatticeAndSolid` would have the following body:

```
latt.setEquilibrium();
solid.read("solidData");
```

Another example of initial conditions is to reuse the result of a previous simulation as starting point. The result of a simulation is the state of the lattice at the end of the simulation. If file “stateData” contains the result of a previous simulation, to reuse this state, function `initLatticeAndSolid` would have following body:

```
latt.read("stateData");
solid.read("solidData");
```

Of course, these initial conditions have no meaning if previous simulation was run using a solid that is not the same as the one from file “solidData”.

The innermost loop on `q` implements periodic boundary conditions. In a parallel simulation, this loop should be replaced by instructions that send `BorderData` instances extracted using `getOutDensities` method to neighbors, and receive `BorderData` instances from neighbors used to set incoming densities with method `setInDensities` (see Program 4.2).

4.3.6 Logging and Operators Chain

According to LaBoGrid use cases, the simulation code should be able to regularly log information about the fluid or other simulation-related information. For example, the speed of the fluid at a given point should be logged every 10 time steps. Another example is the logging of current time step every 5 time steps (to see simulation progression).

```
// Wait information from master process
...
// Next variables are initialized in function of
// received configuration
int maxT = ...; // The number of time steps
Lattice latt = ...;
Solid solid = ...;
CollisionOperator col = ...;
BoundaryConditions bound = ...;

initLatticeAndSolid(latt, solid);
col.setLattice(latt);
col.setSolid(solid);
bound.setLattice(latt);
bound.setSolid(solid);
LatticeDescriptor desc = fluid.getLatticeDescriptor();
int velNum = desc.getNumberOfVelocities();
for(int i = 0; i < maxT; ++i) {
   latt.propagate();
   for(int q = 0; q < velNum; ++q) {
      BorderData bd = latt.getOutDensities(q);
      // send bd to a neighboring process
      ...
   }
   for(int q = 0; q < velNum; ++q) {
      // receive bd from a neighboring process
      BorderData bd = ...;
      latt.setInDensities(bd);
   }
   bound.apply();
   col.collide();
}
```

Program 4.2: Generic parallel LB simulation code using LBSL.

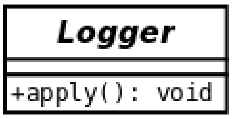

Figure 4.5: Class diagram for `Logger`.

A *logger* is a software component that is activated at each simulation iteration. The *refresh rate* defines when the logger actually logs information about the simulation: let *i* be the current iteration and *r* the refresh rate, the logger logs information when *i* mod *r* is equal to zero. Several *output types* can be used: printing in a file, a message sent to the controller that prints its content to a file, etc. When configuring the logger (see Section 4.5), the wanted output type must be provided. Finally, a logger has a unique identifier. A log can therefore be identified using this information and, for example, be redirected to a particular file.

The `Logger` class is the base class for all loggers. Figure 4.5 shows the class diagram of `Logger`. The `apply` method logs data when required (in function of current iteration and refresh rate). Program 4.3 (based on Program 4.1) gives an example of logging happening after collision.

```
...
Logger log = "a specific Logger instance";

...
for(int i = 0; i < T; ++i) {
   latt.propagate();
   for(int q = 0; q < velNum; ++q)
      latt.setInDensities(latt.getOutDensities(q));
   bound.apply();
   col.collide();
   log.apply();
}
```

Program 4.3: Example of a generic LB simulation code with logging.

In Program 4.3, if the user wants to change the place where the logging occurs (for example, between boundary conditions and collision), he must modify the simulation code.

LaBoGrid should be easy to use, even for users who do not necessarily have high programming skills. Therefore, the choice of specific classes for lattice, solid, collision operator and logger, as well as the selection of the logging place, should require as little code modifications by the user as possible.

A first step towards this goal is the introduction of the concept of *processing chain*: it is a representation of the operations that are applied on the lattice at each simulation iteration. For example, the processing chain of Program 4.3 contains following operations: propagation, periodic boundaries, boundary conditions, collision, logging.

A processing chain is a list of *processing elements*. There are two types of processing elements: *operators* and *loggers*. An operator potentially modifies the state of the lattice or sublattice (propagation, collision, etc.). The interface of a processing element is similar to the interface of `Logger` class: one `apply` method returning no value and taking no argument. A generic simulation code using a processing chain is given by Program 4.4. We suppose that `ProcessingChain` class implements the `java.lang.Iterable` interface. Notation `Iterator<T> it` means that a call to method `it.next()` returns the next element of the processing chain (if it exists, otherwise an exception is thrown) and that this element is an instance of class `T`.

```
Lattice latt = ...;
Solid solid = ...;
ProcessingChain pc = ...;

initLatticeAndSolid(latt, solid);
for(int i = 0; i < T; ++i) {
   Iterator<ProcessingElement> it = pc.iterator();
   while(it.hasNext()) {
      ProcessingElement pe = it.next();
      pe.apply();
   }
}
```

Program 4.4: Generic LB simulation code using a processing chain.

Section 4.5 introduces a description for LB simulation codes like the one given by Program 4.4. The user will therefore be able to describe a simulation without having to explicitly write the associated Java code (excepting for specific classes that may be needed for the requested simulation).

4.3.7 Partitions

The `Partition` class attributes are defined by concepts and constraints introduced in sections 2.4.1 and 2.5.1: output direction (one per velocity vector except rest vector), partition neighborhood (one partition per output direction) and sublattices neighborhood array (array containing sublattices identifiers and in which adjacent identifiers mean adjacent sublattices).

The `Partition` class represents the portion of discrete space associated to a sublattice and a subsolid. There is a one-to-one relation between a partition and a sublattice and a partition and a subsolid. The attributes of `Partition` class include:

- the partition identifier (an integer) which is the associated sublattice's identifier,
- the partition size (an integer array),
- the list of neighboring partitions identifiers,
- a map of all output directions to a partition identifier,
- the position of the origin of the partition in complete space,
- the position of the partition in the sublattices neighborhood array,
- a boolean array indicating if the plane or edge associated to a given velocity vector is an outflow,
- a boolean array indicating if the plane or edge associated to a given velocity vector is an inflow.

The behavior of `Partition` class is only composed of accessors to the attributes.

4.3.8 Partitions Generator

The partitions generator produces a collection of partitions given the global lattice size, the number of partitions to generate and the lattice descriptor. Figure 4.6 shows the class diagram of the base class of all partitions generators.

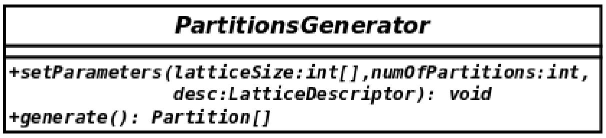
PartitionsGenerator
+setParameters(latticeSize:int[],numOfPartitions:int,
desc:LatticeDescriptor): void
+generate(): Partition[]

Figure 4.6: Class diagram for `PartitionsGenerator`.

For example, the `generate` method of a subclass of `PartitionsGenerator` that partitions 3D lattices can implement the algorithm described in Section 2.5.2.

4.4 LaBoGrid Components

LaBoGrid is composed of several software components generally distributed on several computers. The components hosted by different computers interact by sending messages through the network. In addition, the components hosted by the same computer mostly have to run in parallel and their interactions easily lead to race conditions and deadlocks. These elements suggested a generic framework based on agents communicating in an asynchronous way called *asynchronous agents*.

4.4.1 Asynchronous Agents

Each *agent* runs in a separate thread called *agent thread*. All interactions between agents are based on *events*. An agent can submit an event to another agent which handles it. Each agent handles at most one event at a time (to ensure mutual exclusion on agent's state). An event is represented by an object. An event submitted to an agent is inserted in its *events queue*. The program executed by an agent thread is essentially described by Algorithm 4.1.

The loop of Algorithm 4.1 is called *event handling loop* and, during its execution, the agent is in a state called *event handling state*. Before the event handling loop is executed, some initialization operations can take place, the agent is then in *initialization state*. If an error occurs during initialization, the loop is not executed.

The event handling loop terminates its execution if an error occurred during an event handling or when a stop event is taken from queue. Finally, clean-up operations can be executed after message handling phase, the agent is then in *closing state*.

An *error handler* can be associated to an agent. The error handler is generally

```
"Initialize agent";
if "An error occured during initializing" → skip
□ "No error occured during initializing" →
   do "Agent is running" →
      "Extract an event from queue";
      "Handle extracted event";
      if "No error occured during event handling" → skip
      □ "An error occured during event handling" →
         "Stop handling loop"
      fi
   od
fi;
"Close agent"
```

Algorithm 4.1: Program executed by agent thread.

an agent as well. In case an error occurred during initialization or event handling (i.e. a `Throwable` was thrown), it is signaled to the error handler. Note that if an error is signaled to the error handler (by submitting an event if error handler is an agent). The source of the error is always about to terminate its execution (in case of error, the agent always enters closing phase). This implies that an agent has not to be explicitly stopped when it produced an error.

A more detailed description of the classes implementing agents and error handlers is given in Appendix A.

4.4.2 Distributed Components and Deployment

The parallel implementation of an LB simulation presented in Section 2.5.3 requires a master that sends initial information to workers. In the context of LaBoGrid, the master is part of a component called the *Controller* and a worker is part of a component called *Distributed Agent* (DA). Typically, the Controller runs on a reliable computer and DAs are executed on cluster computers. A computer executes at most one DA. Each DA has a unique identifier (given by the Controller) and knows the address of the Controller.

The concepts represented by the Controller and the DA are common to all distributed applications based on a master-slave model. These agents can therefore

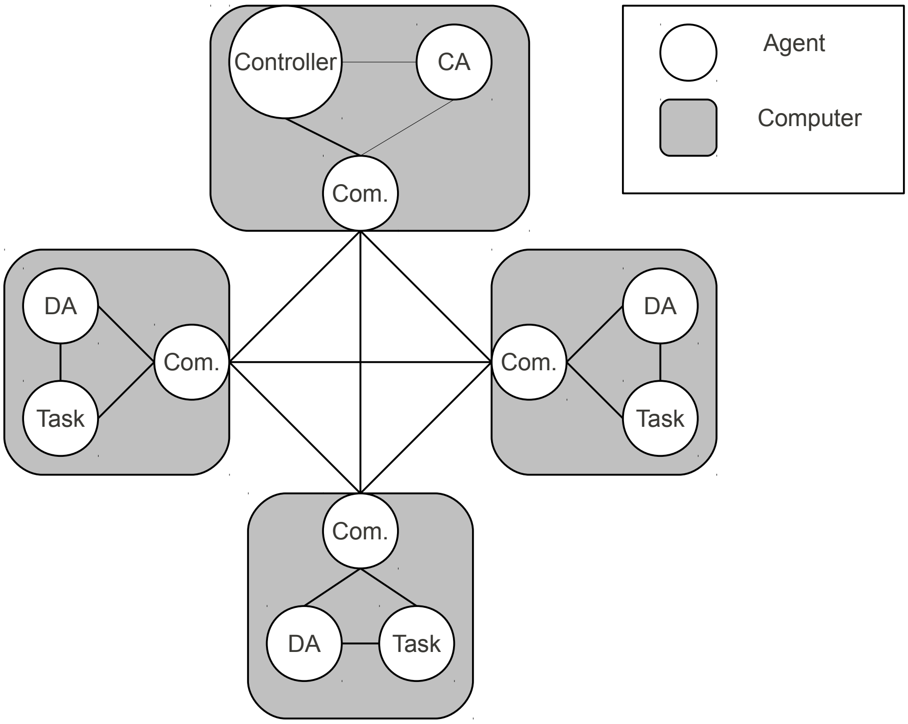

Figure 4.7: Deployment of LaBoGrid.

be generic. Specific agents attached to them actually implement a particular application. A specific *Controller Agent* (CA) is associated to the Controller and a specific *Task* is associated to each DA. In the context of LaBoGrid, the CA implements simulation's master process and a Task a simulation's worker process.

Agents may have to exchange messages through the network. An additional agent called *Communicator* is executed on each computer running the Controller or a DA. The Communicator handles all network related operations (establishing connections, receiving/sending messages, etc.).

Figure 4.7 shows the deployment of presented agents and the link between them. A link between two agents means they interact directly by inserting events in their respective queues. Communicators interact by sending messages through the network.

Controller and Controller Agent

The Controller maintains the list of all available DAs currently running. To fill this list, all DAs first connect to the Controller. The Controller then associates a unique identifier to each DA.

It also instantiates the Controller Agent whose behavior is specific to the application. In the context of LaBoGrid, the CA keeps track of the simulation that is currently running. At the beginning of each simulation, it also distributes required information to all DAs.

According to LaBoGrid use cases, the user should be able to retrieve the result of a simulation and store it to disk in order to, for example, restore it later and use it as initial conditions for another simulation. The result of a simulation is the state of the lattice at the end of the simulation. However, at the end of a distributed simulation, lattice's state is partitioned.

A solution is to generate lattice's global state in memory using the state of all sublattices and then write this global state to disk in one file. However, this method potentially requires a lot of available memory on the computer executing the CA. Another solution is to store the state of each sublattice in a separate file (called *state file*). The associated subsolid is written in the same file. The collection of partitions generated for the simulation is also written to a file. The global lattice state can then be reconstructed using the collection of partitions and the state files.

There are 3 possible situations when starting a simulation, leading to different simulation initializations:

- Fluid is initially at rest and solid is read from a file:

    1. Space is partitioned by a `PartitionsGenerator` instance.
    2. A partition (and therefore a sublattice and a subsolid) is associated to each Task. Addresses of neighboring Tasks are sent to each Task (two Tasks are neighbors if they run a simulation on neighboring sublattices).
    3. The solid is read from a given file. Subsolids are extracted from solid according to generated partitions and sent to workers.
    4. In case the user requested the result to be stored to disk, generated partitions collection is written to a file called *partitions file*.
    5. When all Tasks are ready (sublattices are all initialized with a fluid at rest), the CA signals to all Tasks the simulation can start.

- The result of a previous simulation is given and must be used as initial conditions:

    1. Partitions collection is read from a given partitions file.

2. A partition (and therefore a sublattice and a subsolid) is associated to each Task. Addresses of neighboring Tasks are sent to each Task.
3. Subsolids and sublattices are read from given files and sent to workers.
4. When all Tasks are ready (sublattice and subsolid have successfully been received), the CA signals to all Tasks the simulation can start.

- The result of previous simulation in the sequence is used as initial conditions: in this case, Tasks already have all required information to start the simulation.

Distributed Agent and Task

The Distributed Agent (DA) knows the address of the Controller. It first connects to it in order to receive a unique identifier. Once it is identified, the DA is able to instantiate its associated Communicator and, finally, the specific Task agent that implements a particular application.

LaBoGrid Task waits for initial data and/or a signal from CA to start a simulation. Once a Task has finished its simulation, it signals it to the CA.

If simulation's result is required by the user, the CA downloads it from each finished Task. Each DA then waits for data and/or a signal for the next simulation or stops its execution if there is no more simulation to be executed.

Communicator

The Communicator accepts connections from other remote Communicators and sends and receives messages. Agents called *Message Output Streams* (MOS) are instantiated by the Communicator to send messages to remote Communicators. There is one MOS per remote Communicator. *Message Input Streams* (MIS) are components receiving messages from a remote Communicators. There is one MIS per remote Communicator.

MOS and MIS agents implement respectively the sender and the receiver threads from communication model described in Section 2.5.4.

Messages received by MISs are routed by the Communicator to associated DA or Controller. Figure 4.8 illustrates the path followed by a message sent by a Communicator to another.

There are 2 types of connections accepted by the Communicator:

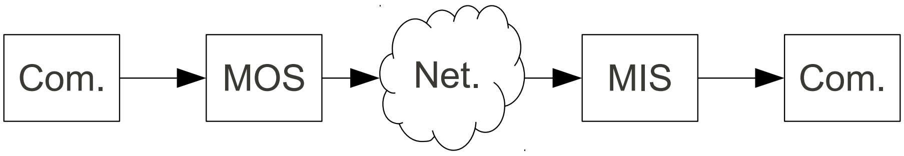

Figure 4.8: Message path.

- a connection from a new DA (only Controller's Communicator accepts this kind of connection),
- a connection from a MOS.

The first kind of connection is handled by the Controller which assigns an identifier to the new DA. The second kind of connection leads to the instantiation of a new MIS to be paired with connecting MOS.

The messages the Communicator sends are instances of `Message` class. Its members include the address of sender Communicator and the address of the destination. A destination can also be specified using the identifier of the DA associated to destination Communicator. In this case, before the message can be sent or a connection established, the Communicator must query the address associated to the given identifier to the Controller. A special identifier is used to represent the Controller and, as the address of the Controller is known by every DA (and Controller), it does not need to be queried.

To limit the number of address queries to the Controller, addresses are cached. Also, if a MOS connects to a Communicator, it first sends the address of its Communicator and this address is stored by destination Communicator as there is a high probability that messages will be sent to connecting Communicator.

Several connections to the same computer can be requested to the Communicator. In this case, only one connection is established and all components use this same connection. The MOS avoids concurrent access to the socket by sending the messages it extracts from its message queue. As writing a message to the MOS means inserting a message in its queue, MOS default write operation is asynchronous. When a MOS is closed, it sends all messages inserted in its message queue before closing was requested, it stops extracting messages from its message queue (general behavior of an agent) and it finally closes its underlying socket.

Communicator features two message sending modes:

- a datagram mode,
- a connected mode.

In datagram mode, the destination field of the message must be set and simply submitted to the Communicator. In case the MOS for given destination already exists, a message is written to this MOS. Otherwise, a new MOS is instantiated, the message is written to this new MOS and, finally, the MOS closing is scheduled. The MOS is not closed immediately in case other messages for same destination are submitted a short time later.

In connected mode, an indirect access to a MOS is given to write messages: a proxy object called *Message Output Stream Accessor* (simply called accessor hereafter) is used. The accessor forwards messages to send to MOS. When an accessor is closed, it notifies the Communicator. If no more accessors are associated to a MOS, its closing is scheduled. The accessor also implements synchronous message sending: it writes the message to MOS and waits for a notification from the MOS signaling that the message was successfully sent or an error occurred.

4.5 LaBoGrid Configuration

An XML configuration file written by the user is used to describe the sequence of simulations LaBoGrid must execute. This XML file is given as input to LaBoGrid's CA and Tasks.

In Java, a class can be instantiated given its name. The only constraint is that the class must have a constructor without arguments. This implies that a specific class name given in the XML configuration file can be used to instantiate the class in LaBoGrid provided that the class was loaded by the JVM executing LaBoGrid. Additional JAR files can therefore be provided, these files contain the implementation of subclasses implementing base classes from LBSL referenced in the configuration file.

The content of the XML configuration file is only briefly described in this section. For a more detailed presentation, see Appendix B.

The XML file is composed of three parts:

1. LB configurations,
2. Processing chains description,
3. Simulations description.

These parts are described below.

4.5.1 LB Configurations

This part is composed of a set of LB configurations. An LB configuration has a unique identifier and contains the required information to instantiate a lattice and a solid for a simulation and generate the collection of partitions needed to decompose the lattice and the solid into sublattices and subsolids (see Section 4.3.8).

4.5.2 Processing Chains Description

This part is composed of a set of processing chain descriptions. A processing chain description has a unique identifier and is composed of a sequence of loggers and operators descriptions. It contains the required information to instantiate the processing chain to apply to the lattice at each time step (see Section 4.3.6).

4.5.3 Description of Simulations

The set of simulations to be executed is called an *experiment*. An experiment is composed of a sequence of simulations sequences. A *simulation sequence* can contain one or more simulations. In a simulation sequence, the result of intermediate simulations is not necessarily retrieved but each simulation of the sequence (except the former) uses the state of previous simulation as initial conditions.

A simulation is described by an LB configuration, a processing chain and the number of time steps to execute. In addition, a *simulation input* and a *simulation output* can be defined. A simulation input is a component that can be used to get a file from a specific medium (local file, remote file, FTP server, etc.). A simulation output is a component that can be used to put a file to a specific medium. The files obtained/put from/to an input/output are solid files, state files and partitions files (see Section 4.4.2).

Simulation output definition is always optional: when defined, simulation's result is written to it. Simulation input definition is mandatory for the first simulation of a simulations sequence in order to read the solid file or the result of a previous simulation to use as initial conditions.

4.6 Results

In order to observe the impact of using the general framework presented in this chapter for distributed LB simulations' implementation instead of the specialized

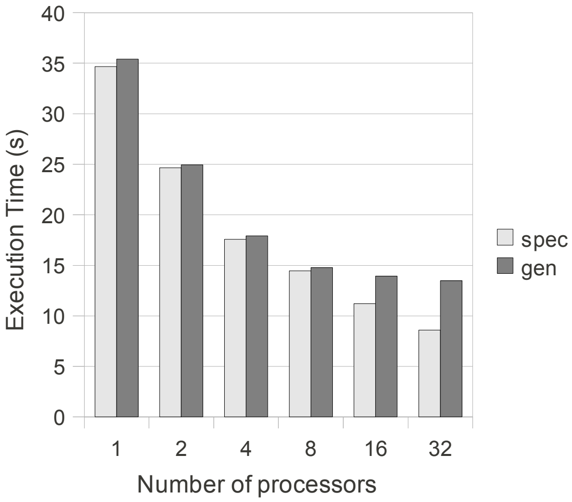

Figure 4.9: Comparison of the execution times obtained using a specialized (spec) and a generic (gen) implementation of a distributed LB simulation on a $(64,64,64)$ D3Q19 lattice with an SRT collision operator.

code used to produce the results of Chapter 3, we compared the execution times obtained in the two cases.

Figure 4.9 shows the execution times of a distributed LB simulation on a $(64,64,64)$ D3Q19 lattice using the SRT collision operator obtained when using the two implementations on an increasing number of processors. Unsurprisingly, the generic implementation is always slower than the specific one with a difference factor between the execution times of the two implementations varying from 1.02 when using 1 processor to 1.58 when using 32 processors.

The main cause of generic implementation's slowness comes from communication layer because data read from a socket must be routed to the destination thread. This operation is more complex in the generic implementation. This explains the fact that the difference between the two implementations becomes larger when the number of used processors increases: the number of messages exchanged through the network becomes more important and highlights the more complex routing algorithm of generic implementation.

The impact of using the generic implementation is less important for distributed simulations implying bigger sublattices. Figure 4.10 shows the execution time of the same simulation but on a $(128,128,128)$ lattice. The factor between the execution times obtained with the two implementations distributed on 32 processors is then reduced to 1.17. The reduction of the overhead factor is caused by

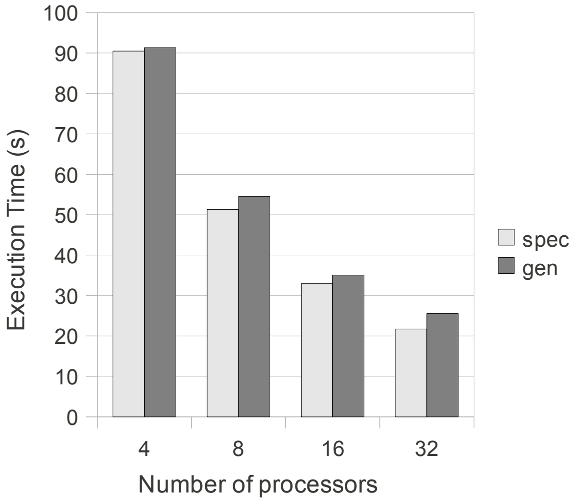

Figure 4.10: Comparison of the execution times obtained using a specialized (spec) and a generic (gen) implementation of a distributed LB simulation on a $(128, 128, 128)$ lattice with an SRT collision operator.

the fact that the computation time to communication time ratio is larger.

4.7 Conclusion

In this chapter, we presented the architecture of LaBoGrid, a powerful tool allowing the execution of structured sequences of distributed LB simulations. The sequence of simulations and the parameters of these simulations are described by the user in an XML file given as input to LaBoGrid.

LaBoGrid can also be used as a library to easily write sequential simulation codes based on LB methods. Though this is not the main purpose of LaBoGrid, it allows to easily try alternative implementations of some important components of LB simulations (lattice, solid, collision operator, etc.).

Finally, LaBoGrid is written using a generic framework based on asynchronous agents. This generic framework provides a message passing service (the Communicator) and generic classes that can be used to describe any distributed application based on a master-slave model (CA and Task).

We observed that the generic framework used by LaBoGrid introduces an overhead in distributed simulations execution time when compared to the specific implementation used to produce the results of Chapter 3. However, this overhead remains reasonably low. For example, there is a factor 1.17 between the execution times of the two implementations for a distributed LB simulation on a $(128,128,128)$ lattice using an SRT collision operator and executed on 32 processors. This factor decreases when a more complex simulation code and/or larger sublattice are used, i.e. when the computation time to communication time ratio increases.

The generic framework presented in Section 4.4 allows LaBoGrid's implementation extension with additional agents implementing the techniques described in next chapters. Therefore, the generic implementation of LB simulations is considered as a reasonable trade-off between efficiency and extensibility.

Chapter 5

Static Load Balancing

5.1 Introduction

In Chapter 4, a simulation tool based on LB methods called LaBoGrid is described. In this system, a Controller and several Distributed Agents (DAs) are executed in parallel: each DA executes a Task running the simulation code on a sublattice (a part of the global lattice, see Chapter 2) and the Controller steers the whole process.

LaBoGrid is able to set the initial conditions of a simulation by using the result of a previous one: at the end of a simulation, the state of each sublattice is written to a state file (one per sublattice). This state file can then be read later to set the initial state of a sublattice.

There are two problems with the presented architecture:

- If a simulation with X sublattices was run on X computers (one sublattice per computer), the result of the simulation cannot directly be reused to set the initial conditions of a simulation that must be run on Y computers with $X > Y$. Also, if $X < Y$, some computers will have no sublattice associated to them and will therefore not be used for the simulation.
- All computers receive the same amount of work: they will all execute the same number of instructions because sublattices are equally sized and simulation code executed by each computer is mostly the same. In this situation, fast computers will wait for slow computers at each time step.

The first problem can be solved by associating one or several sublattices (instead of only one sublattice) to each Task and therefore computer. The sublattices can therefore be distributed on any number of computers as long as this number is less or equal to the total number of sublattices.

The execution time of a distributed application, and in particular LB simulations, is essentially equal to the addition of *processing time* and *communication time*. Processing time is the time to process all application data in parallel. Communication time is the time to transfer data between processes during the execution of the application.

The sublattices should be distributed among computers in a way the computational power of the computers (the maximum number of instructions they can execute per time unit) is considered. It is the case when more sublattices are associated to faster computers and less to slower ones. The processing time can then be minimized.

In addition to taking computers' power into account, the amount of data exchanged between computers during the execution of an LB simulation should be minimized: the data exchanged between two adjacent sublattices can be transmitted through memory instead of the network if these sublattices are on the same computer. The communication time can then be minimized by taking sublattices adjacency into account.

In the context of distributed LB simulations, the problem of finding the optimal sublattices distribution that minimizes a distributed simulation's execution time is called *load balancing*. If this problem must be solved only once before the simulation is executed, it is called *static load balancing*.

A distributed application can be represented by a graph called *application graph*. The cluster that will execute this application can also be represented by a graph called *resource graph*. These representations allow the use of a class of tools called *mappers* (JOSTLE [76], METIS [47], SCOTCH [62], PaGrid [45], etc.). These tools solve the load balancing problem for given application and resource graphs.

When the application graph and the resource graph are known in advance and do not change during the application's execution, the static load balancing problem can be solved using static mappers.

5.1.1 Chapter Outline

Section 5.2 introduces the graph representation of a distributed application and the cluster that will execute it. These graph representations allow the use of static mappers to solve the static load balancing problem.

Section 5.3 presents the application graph associated to distributed LB simulations. The distributed representation of the application graph is also discussed.

Section 5.4 explains how the performance of computers that will execute a distributed LB simulation is evaluated to be taken into account during load balancing process.

Section 5.5 describes how the resource graph can be built and when this process occurs in LaBoGrid's execution.

Section 5.6 presents the execution time of distributed LB simulations using different mappers.

Section 5.7 concludes this chapter.

5.2 Graph Representation

Some parallel/distributed applications can be represented by an undirected graph called *application graph*. The nodes of the application graph represent data that can be processed in parallel. An edge between two nodes of the application graph means that data are exchanged between the processes handling the data associated to these nodes. Nodes and edges can potentially be weighed: the weight of a node is proportional to the amount of instructions needed to process its associated data and the weight of an edge is proportional to the amount of data exchanged between the two associated nodes during the application execution.

A cluster, a computational grid or a supercomputer can also be represented by an undirected graph called *resource graph*. The nodes of the resource graph represent a computer or a processor. An edge between two nodes means they are interconnected by a network link or they can use shared memory to communicate. This graph can also be weighed: the weight of a node is proportional to the computational power of the associated computer or processor (the maximum number of instructions it can execute per time unit) and the weight of an edge is proportional to its bandwidth.

A graph G can be represented by a set of nodes N and a set of edges E and noted $G = (N,E)$. Let $G_1 = (N_1,E_1)$ and $G_2 = (N_2,E_2)$ be two graphs, the mapping of G_1 onto G_2 is composed of a partitioning of N_1 and a relation associating each partition to a node of N_2. More formally, a mapping is:

- a partitioning $Q = \{P_1,P_2,\ldots,P_n\}$ with $n = |N_2|$ and $\cup_{i=1}^{n} P_i = N_1$,

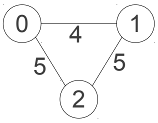

Figure 5.1: Example of undirected graph with weighted edges.

- a bijection $f : Q \to N_2$ such as each partition of Q is associated to one and only one node of N_2.

The graph mapping problem is common in scientific simulations [33] where mesh based computations are represented using a graph. Many graph partitioning tools already exist (JOSTLE [76], METIS [47], SCOTCH [62], PaGrid [45], etc.). Among the listed tools, only PaGrid tries to directly minimize the execution time of the application (other tools like MiniMax [52] and PART [27] also directly minimize execution time but the former does not take into account application's granularity[1] and the latter features a substantial execution time). SCOTCH does not directly minimize execution time but takes into account the fact that the resource graph can be weighted.

In our context, LB simulations are run on a cluster with computers being part of the same network (any computer is linked to any computer and all links have the same bandwidth) but having potentially different computational powers. The edges of related resource graph thus have all the same weight but nodes have potentially different weights.

5.2.1 Graph Data Structures

Let $G = (V, E)$ be a graph where V is the set of vertices and E the set of edges. Let $n = |V|$ be the number of vertices in the graph. We suppose that $V = [0..n-1]$. An edge $e \in E$ can therefore be represented by a pair (i, j) with $0 \leq i, j < n$ and $i \neq j$.

Edges can be weighted. In this case, E contains triples (i, j, w) where $0 \leq i, j < n$, $i \neq j$ and $w > 0$. w is the weight of the edge.

An example of undirected graph with weighted edges is given in Figure 5.1.

G can be represented using an *adjacency matrix* $\mathbf{m}$. $\mathbf{m}$ is a $n \times n$ matrix defined as follows:

[1] Granularity is the ratio of the amount of computation to the amount of communication; this ratio is low for fine-grained applications and high for coarse-grained applications.

- $m_{ii} = 0$ with $0 \leq i < n$,
- $m_{ij} = 1$ if $(i, j) \in E$ with $0 \leq i, j < n$ and $i \neq j$,
- $m_{ij} = 0$ if $(i, j) \notin E$ with $0 \leq i, j < n$ and $i \neq j$.

The values of matrix **m** can also be used to represent the weight of an edge. If $m_{ij} = 0$, there is no edge going from i to j. If $m_{ij} > 0$, m_{ij} is the weight of the edge going from i to j.

If G is undirected, **m** is symmetric.

The adjacency matrix **m** for the graph of Figure 5.1 is defined as follows:

$$\mathbf{m} = \begin{pmatrix} 0 & 4 & 5 \\ 4 & 0 & 5 \\ 5 & 5 & 0 \end{pmatrix}$$

Another data structure to represent G is the *adjacency list*. The i^{th} element of the list is the list of vertices adjacent to i. j is in the i^{th} list of vertices if $(i, j) \in E$.

If a weight is associated to the edges, the elements adjacency list's i^{th} element are pairs of integers (j, w) where j is the adjacent vertex and w the weight associated to the edge. (j, w) is in the i^{th} list of vertices if $(i, j, w) \in E$.

The adjacency list for the graph of Figure 5.1 is defined as follows:

$$\{\{(1,4),(2,5)\},\{(0,4),(2,5)\},\{(0,5),(1,5)\}\}$$

The adjacency list and the adjacency matrix contain the same information. However, they facilitate different operations:

- The adjacency list efficiently provides all adjacent vertices to a given vertex.
- The adjacency matrix allows to efficiently test if two given vertices are adjacent.

The choice of the representation therefore depends on the kind of required operation.

5.3 Distributed LB Simulations Application Graph

Distributed LB simulations can be represented by an application graph. One possibility is to associate a lattice site to each node of the application graph (there is a bijection between the set of all lattice sites and the set of application graph's nodes). However, the partitions produced by the tools presented in previous section do not necessarily imply an organization in a regular grid of the sites associated to a partition. This implies that arrays are potentially not directly usable to represent the organization of the sites associated to a given partition. Adapting simulation algorithms to use more general data structures than arrays to represent sites' organization would lead to an important execution time overhead.

The solution we chose is to use sublattices as nodes of the application graph. The advantage of this choice is that the simulation code can remain unchanged. Because sublattices have all the same size, the nodes of the LB simulation application graph have all the same weight. Edges do not because the number of outgoing densities can be different in function of output direction (see Section 2.4.1).

In Section 2.5.1, we introduced the concept of Sublattices Neighborhood Array (SNA), a multi-dimensional array of sublattice identifiers. A sublattice identifier is associated to each position in this array and the identifier of each sublattice appears exactly one time in the array.

The SNA represents the neighborhood of each sublattice: let A be a sublattice at position $\mathbf{p}$ in the array, if sublattice B is the neighbor of A associated to neighborhood vector $\mathbf{n}_j$, position of B is $(\mathbf{p}+\mathbf{n}_j)\ mod\ \mathbf{r}$ where $\mathbf{r}$ is the size of the SNA. This means that identifiers of neighboring sublattices are adjacent in sublattices neighborhood array.

The *sublattices graph* is the graph representation of the SNA: each vertex is associated to a sublattice and if two sublattices are adjacent in the SNA, an edge connects their associated vertices in the sublattices graph. The edges of the sublattices graph is weighted by the amount of bytes exchanged between the sublattices of given edge every simulation iteration.

The sublattices graph is represented using an adjacency list: to each sublattice is associated the list of adjacent sublattices. The adjacency list is represented by an array indexed using sublattices identifiers. Each element of the array is an array of records, each record containing the weight of the associated edge and the identifier of adjacent sublattice. Figure 5.2 illustrates this data structure. The chosen data structure allows to efficiently have access, given a sublattice identifier, to its adjacent sublattices. However, each edge is represented two times.

The partitioning of the sublattices graph means a partition is associated to each

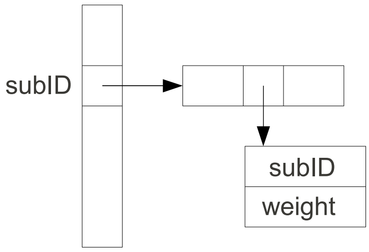

Figure 5.2: Representation of Sublattices graph.

sublattice. In the context of LaBoGrid, a partition is associated to a particular DA and the DA identifier can be used as partition identifier. The result of a partitioning can therefore be represented by an array of DA identifiers (actually integers) indexed using sublattice identifiers. This array is called *mapping table*.

Another interesting structure that can be built from the mapping table is the *partitions table* that gives the set of sublattices associated to a DA given its identifier.

Mapping and partitions tables contain the same information but are efficient for different operations:

- mapping table efficiently provides the identifier of the partition (DA identifier) for a given sublattice,
- partitions table efficiently provides the sublattices associated to a given partition.

5.3.1 Sublattices Graph Distribution

Before the execution of an LB simulation, a partition of the sublattices graph is associated to each DA. Before the simulation starts, the Controller provides following structures to each DA:

- a part of the sublattices graph called *partial sublattices graph*,
- a part of the mapping table called *partial mapping table*.

The partial sublattices graph is represented using an array of pointers to records. Each record contains the identifier of a sublattice and a pointer to an array. This array is the list of the edges to neighboring sublattices: each entry is a pointer to

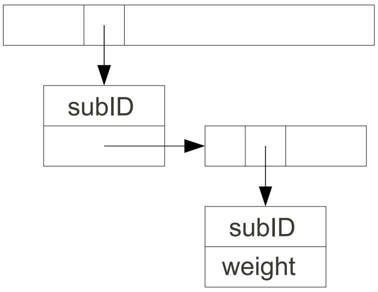

Figure 5.3: Representation of partial sublattices graph.

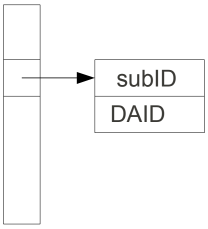

Figure 5.4: Representation of partial mapping table.

a record giving the identifier of the neighbor and the weight of associated link. Figure 5.3 illustrates this structure.

The partial mapping table is represented by an array of pointers to records. These records contain a sublattice identifier and a DA identifier. The entries of this array correspond to sublattices that are the neighbor of a sublattice from the partial sublattices graph but are located on another DA (given by the second entry of the record). Figure 5.4 illustrates this structure.

The partial sublattices graph and partial mapping table arrays can be sorted regarding the identifier of the sublattice associated to an entry in order to enable the efficient access to the information associated to a sublattice given its identifier.

The partial sublattices graph is used to connect simulation threads on a particular DA. The partial mapping table provides, given a sublattice identifier, the DA a simulation thread must connect to in order to send a part of its outgoing densities.

5.4 Evaluation of Computer Performance

To set the weights of resource graph's nodes, several methods can be considered:

- Simply use the computational power of computers: weight is proportional to the maximum number of instructions executed by computer per time unit. Memory bandwidth and cache memory size are then ignored. However, the content of Chapter 3 suggests that these parameters have a significant impact on performance.

- Define a model that takes computational power, memory bandwidth, cache memory size, etc. as input parameter and outputs a weight. The definition of this kind of model is not always trivial and a new model is needed each time the simulation instructions change. This is problematic as we stated in Chapter 4 that the user can easily change these instructions.
- Estimate the number of sites a computer can handle per time unit when no communication occurs. This value is called *Contextual Computational Power* (CCP). The CCP can be obtained by executing a benchmark code (for example, a small simulation). The main advantage of this approach is that it encompasses all parameters impacting the execution time of LB simulations (computational power, memory bandwidth, etc.).

Last method was chosen and is described in this section.

To estimate the CCP of a computer, the execution time t_{exec} in seconds of a simulation on a lattice with s sites during N time steps is measured on a particular computer. The CCP is then calculated using following relation:

$$CCP = \frac{s \times N}{t_{exec}}.$$

It is expressed in sites per second.

When a computer has several processors or cores, the simulation is independently executed in parallel by each processor or core. Let p be the number of processors of the computer, the CCP is then calculated using following relation:

$$CCP = \frac{p \times s \times N}{t_{exec}}.$$

The CCP of a computer depends on its configuration (OS, CPU speed, memory bandwidth, etc.) but also on the parameters of the simulation used during benchmark. Following simulation parameters can impact the CCP:

- Lattice type (dimensions, number of neighbors per site) and size,
- Solid structure (because bounce-back is less complex to compute than a collision operator, see Section 2.2.3),

- Processing chain without loggers (defines the instructions to execute at each time step, see Section 4.3.6).

Therefore, a CCP value is uniquely identified by a computer identifier, a lattice type, a lattice size and the content of a processing chain when loggers are ignored. The notation $CCP(n, p)$ represents the CCP for the computer associated to a node n of the resource graph evaluated using simulation parameters p.

In the following, we analyse the parameters that, in addition to computer performances, influence the value of the CCP for a given computer. Following observations highlight the fact that the CCP must be evaluated for every combination of computer and simulation parameters.

In practice, we chose to ignore solid structure during the evaluation of a computer's CCP because most simulations we execute imply a porosity (i.e. the ratio of the volume of voids to the total volume) that is high (more than 90%). The portion of time for the execution time of bounce-back is therefore negligible regarding the portion of time for the execution of the collision operator. In addition, we consider that porosity is homogeneously distributed among subsolids (i.e. all subsolids have the same porosity).

5.4.1 Processing Chain Content

Benchmarks consisting of a sequential LB simulation on a lattice of size $(20, 20, 20)$ with 200 time steps are executed using different collision operators (see Section 2.2.2):

- the SRT operator (SRT on the legend of Figure 5.5),
- the SRT operator with Smagorinsky viscosity model (SRT + smago),
- the MRT operator (MRT),
- the MRT operator with Smagorinsky viscosity model (MRT + smago).

A D3Q19 lattice with the optimized representation presented in Chapter 3 is used.

Benchmarks are run on two types A and B of computers. Computer A has the following configuration:

- CPU: Pentium IV 3.06 Ghz with hyper-threading support (processor has two separated pipelines)

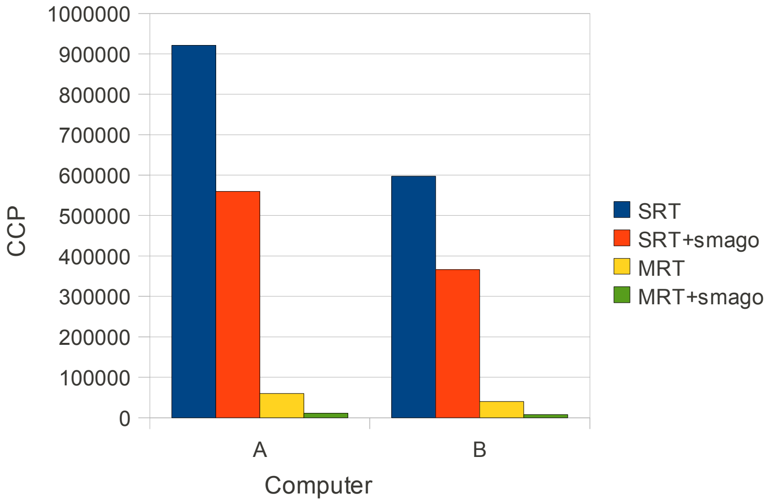

Figure 5.5: CCP in function of hardware configuration and different collision operators.

	A	B
SRT	921	597
SRT+smago	560	366
MRT	60	40
MRT+smago	11	8

Table 5.1: CCP in function of hardware configuration (columns A and B) and different collision operators (SRT, SRT+smago, MRT, MRT+smago). Given values result from the division of the actual CCP by 1000.

- Cache: 1 MB

Computer B has the following configuration:

- CPU: Pentium Celeron 2.40 Ghz
- Cache: 128 KB

Figure 5.5 shows the CCP for the four collision operators presented above evaluated on the two given computer types A and B. This information is also given by Table 5.1. For SRT collision operator, computer A is 1.54 times as fast as computer B. When using more complex collision operators requiring more instructions, the factor between the CCP estimated for the two computer types

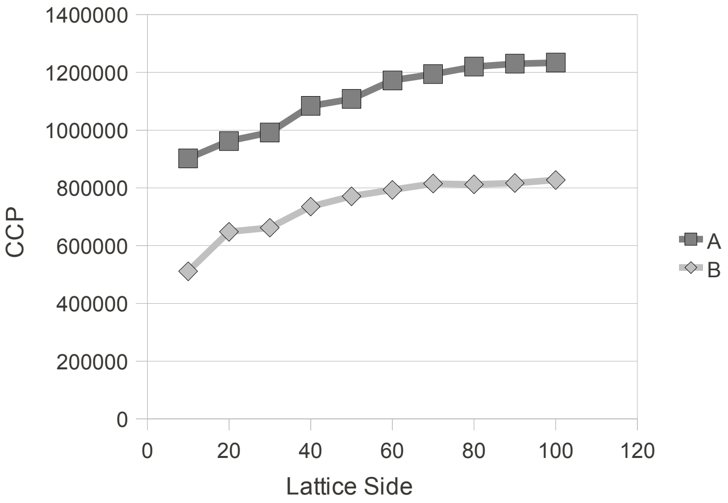

Figure 5.6: CCP in function of hardware configuration and lattice size.

becomes smaller (from 1.52 for SRT+smago down to 1.44 for MRT+smago). This is probably because the portion of execution time used to access data becomes smaller when collision operator becomes more complex: the disadvantage of small cache memory of computer type B becomes less important.

SRT operator is far less complex to compute than MRT operator: the factor between CCP with SRT and MRT operators is of approximately 15 on both computer types.

Finally, the use of Smagorinsky viscosity model also decreases the CCP. On computer type A, there is a factor 1.64 with SRT operator and 5.43 with MRT operator. On computer type B, similar factors are obtained: 1.63 with SRT operator and 5.26 with MRT operator. The use of the Smagorinsky model has a substantial cost in the context of the MRT operator because it implies several small matrices multiplications.

5.4.2 Lattice Size

Figure 5.6 shows the CCP plotted for increasing sizes of lattice. The x-axis gives the side of the cubic lattice and y-axis the measured CCP. An LB simulation with the SRT collision operator was used.

The CCP increases with the size of the lattice, this is related to the fact that the optimized implementation presented in Chapter 3 gives better performances as the size of the lattice increases.

5.5 Resource Graph Generation

In the context of LaBoGrid, several distributed LB simulations are executed in sequence. Therefore, several potentially different application graphs need to be mapped onto the resource graph during LaBoGrid's execution: a load balancing process is triggered before the execution of each simulation.

The topology of the resource graph does not change during LaBoGrid's execution. However, $CCP(n, p)$ values need to be reevaluated when p changes and this implies that the weights of the nodes of the resource graph potentially change before every simulation.

Algorithm 5.1 describes when the load balancing process is triggered during a typical execution of LaBoGrid. Computers are benchmarked in parallel, the part "Benchmark computers" essentially consists in the Controller sending benchmark parameters to all DAs and then waiting benchmark for the results. CCP values calculated during a previous execution of LaBoGrid can be reused if simulation implementation has not changed meanwhile. They are then given as input parameters to LaBoGrid.

```
do "There are simulations to execute" →
   p := "Parameters of current simulation";
   if "CCP values available for p" →
      skip
   □ "CCP values not available for p" →
      "Benchmark computers"
   fi;
   "Generate application and resource graphs";
   "Map application graph onto resource graph";
   "Distribute sublattices on computers";
   "Execute current simulation";
   "Go to next simulation"
od
```

Algorithm 5.1: Static load balancing during LaBoGrid's execution.

5.6 Results

In this section, we analyze the execution time of distributed LB simulations when using two mappers: SCOTCH [62] and PaGrid [45]. SCOTCH is used with two configurations:

1. the resource graph is weighted; this configuration is called *heterogeneous SCOTCH*.
2. the resource graph is not weighted; this configuration is called *homogeneous SCOTCH*.

Homogeneous SCOTCH corresponds to the situation where all computers are considered as having the same CPU power.

Heterogeneous SCOTCH [62] is able to map a fully weighted application graph on a fully weighted resource graph. PaGrid [45] is able to map an application graph with weighted edges onto a fully weighted resource graph. The absence of weights on the application graph's nodes is not important as sublattices would all have the same weight (they represent the same amount of instructions). PaGrid tries to directly minimize the application's execution time.

In the figures of this section, following labels are used:

- PaGrid is self-explanatory,
- HeScotch stands for heterogeneous SCOTCH,
- HoScotch stands for homogeneous SCOTCH.

In addition to computer types A and B presented in Section 5.4, a third type C is introduced with following specifications:

- CPU: 2 Intel Xeon 5130 2GHz (two cores per CPU, four cores in total)
- Cache: 4096KB

Table 5.2 compares types A, B and C regarding their CCP evaluated using an LB simulation on a $(30,30,30)$ D3Q19 lattice using MRT collision operator. Type C computer is 6 times faster than type A computer which is 1.5 times faster than type B computer.

Figure 5.7 shows the execution time of one iteration of a distributed LB simulation on a lattice of size $(176,176,176)$ and using MRT collision operator in

A	B	C
60	40	385

Table 5.2: CCP of computers of type A, B and C evaluated using an LB simulation on a $(30,30,30)$ D3Q19 lattice using MRT collision operator.

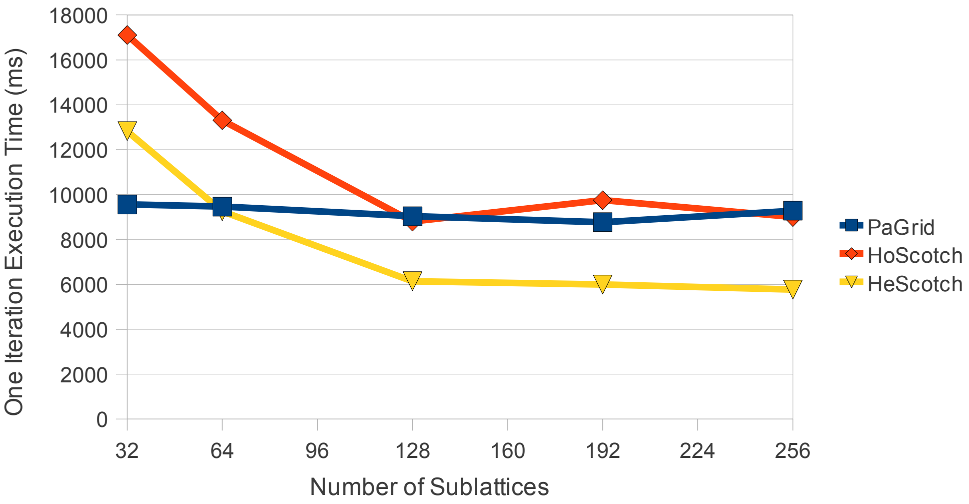

Figure 5.7: Execution time of one LB simulation iteration with a $(176,176,176)$ lattice in function of the number of sublattices.

function of the total number of sublattices distributed among cluster's computers. The simulation is run on a cluster composed of 8 A computers, 8 B computers and 1 C computer.

PaGrid gives the best results when mapping 32 sublattices. When mapping at least 64 sublattices, heterogeneous SCOTCH leads to the lowest execution times.

We observe that increasing the total number of sublattices leads to lower execution times. This is due to the fact that the mappers can produce finer mappings when the number of sublattices is high. For example, the execution time of the simulation is divived by 2.22 when mapping 256 sublattices instead of 32 sublattices with heterogeneous SCOTCH mapper.

Finally, the same simulation was executed when associating exactly one sublattice per computer (there are 17 sublattices distributed among 17 computers). This case represents the best homogeneous distribution that can be achieved with cuboid sublattices because all sublattices have almost the same size and there is exactly one sublattice per computer. The measured time for one iteration is 7770 milliseconds.

With more than 128 sublattices, heterogeneous SCOTCH provides distributions that lead to lower execution times than the best homogeneous case which shows that load balancing has an interest in the context of distributed LB simulations: when mapping 256 sublattices extracted from a $(176, 176, 176)$ lattice using heterogeneous SCOTCH, the execution of a simulation using an MRT collision operator is 1.34 times faster than with best homogeneous distribution.

A final observation is that the execution time obtained by using the mappings produced by the homogeneous SCOTCH mapper is at least greater by a factor 1.13 than the execution time obtained with the best homogeneous distribution. This difference has two reasons:

- the homogeneous SCOTCH mapper allows small imbalances between the partitions it produces. This means that the sublattices distribution is generally not perfectly homogeneous.

- there is an execution overhead when the sites associated to a computer are organized into several sublattices instead of a single one. The overhead is caused by the fact that more outgoing and incoming densities have to be handled when the number of sublattices increases for a fixed number of sites.

5.7 Conclusion

In this chapter, a graph representation of distributed LB simulations called application graph was introduced. This representation is interesting because it allows to use existing static load balancing tools based on graph mapping called static mappers. These tools require a graph representation of the cluster that will execute the distributed application called resource graph.

In our context, the edges of the application graph and the nodes of the resource graph are weighted. In order to weight the resource graph nodes, a score called CCP (Contextual Computational Power) was introduced. This score gives the number of lattice sites a given computer can execute per time unit. The CCP can directly be used to weight the resource graph nodes. The CCP is obtained using benchmarks involving the execution of short LB simulations.

Two static mappers were evaluated by comparing the execution time of distributed LB simulations using the produced mappings: PaGrid and SCOTCH. Two configurations of SCOTCH were used: heterogeneous SCOTCH (resource graph is weighted) and homogeneous SCOTCH (resource graph is not weighted). The observed execution times show that taking the heterogeneous CPU power of cluster's computers into account is an interesting approach that reduces the overall execution time of LB simulations when compared to the execution time obtained without considering heterogeneity. For example, the execution time of a distributed LB simulation on a $(176, 176, 176)$ lattice using the MRT collision operator was divided by 1.34 when distributing 256 sublattices on 17 computers of a heterogeneous cluster instead of assigning one of 17 sublattices to each computer.

We also observed that, for mapping tools to be efficient, the application graph should represent a fine-grained application. In our case, this amounts to decrease the size of the sublattices by increasing their number. However, increasing the number of sublattices per computer implies an execution time overhead. The grain of distributed LB simulations should therefore be increased only when a gain is expected (i.e. when computers have heterogeneous computational powers).

Chapter 6

Fault-Tolerance

6.1 Introduction

In previous chapters, several aspects of an efficient implementation of distributed LB simulations were described. However, even with an efficient implementation, some LB simulations can run for several days. In this context, it is not unusual that a process does not complete its execution because, for example, of a software failure (operating system freeze, program interruption, etc.) or a hardware failure (temporary network failure, memory failure, etc.). These problems are called *failures* in this chapter.

A distributed application is *fault-tolerant* if it is able to continue its execution in case of failure. The lack of fault-tolerance can be a limiting factor in the scalability of a distributed application because the probability of failure grows with the number of computers/processors.

In case of failure, a distributed LB simulation cannot be finished because the part of lattice's state associated to the interrupted process is unavailable. In addition, because simulation processes are waiting for data coming from their neighbors, a deadlock may appear because some processes will never receive the data they are waiting for.

To avoid the problems described above, the complete state of the simulation should be available even in case of failure. A common mechanism called checkpoint/restart [68, 49, 35, 36, 63] can then be used: the state of the application is regularly saved and, in case of failure, the latest saved state is reloaded to restart the execution from this point.

In order to be retrieved in case of failure, the state should be saved into a reliable storage. In the context of LaBoGrid, the computer executing the Controller could be used but this creates a bottleneck when saving the state and when loading it. Another solution is to use a distributed storage: the computers executing the simulation processes are used to store the simulation's state. Each computer saves the state of the process it is running. However, this does not prevent the loosing of a part of simulation's state in case of failure. Data replication must be used to obtain a reliable distributed storage: the state of each process is replicated on several computers, therefore if one of these computers becomes unexpectedly unavailable, the states it hosted are probably still available on other computers.

A failure detection mechanism is required to be able to trigger simulation restart in case of failure. It has been shown that, in asynchronous systems, failure detectors are not reliable [25]: a failure can stay unnoticed for some time (false negative) and a failure can be erroneously detected because of network congestion, slow computers, etc. (false positive). However, hypothesis on the execution environment (see Section 6.3) reduce the probability of false positives. Moreover, false negatives are acceptable, in our context, as long as failures are eventually detected.

A failure can be temporary. In this case, when the affected computer is ready again, it should be reused to execute a process of the distributed application. In addition, if many failures occur, the number of processes of the application decreases and can finally reach zero, in which case the application is not executed anymore. Therefore, if the number of failures during an execution can be large, a mechanism that searches for available computers (computers able to execute a process of the application) should be implemented. The process of finding ready computers is called *resource discovery*.

A part of the research presented in this chapter (sections 6.2, 6.4 and 6.5) is the result of a cooperative work with Cyril Briquet who designed CanoPeer [1], a Peer-to-Peer (P2P) grid computing middleware for distributed applications composed of a set of independent tasks potentially processing large input data files and producing output data files. CanoPeer integrates a mechanism to find new resources to execute submitted tasks. It can therefore be used by LaBoGrid as a resource discovery service.

The task scheduling policy of CanoPeer can lead to the simultaneous interruption of the execution of several tasks. In the context of independent tasks, this only postpones the termination of the job because the interrupted tasks are executed and completed later. With a distributed LB simulation, the simulation is restarted, possibly from a previously saved state if it is available. The state's availability probability should be maximized even in the case of simultaneous failures caused by CanoPeer's scheduling policy.

6.1.1 Chapter Outline

Section 6.2 introduces CanoPeer, a P2P grid computing middleware designed by Cyril Briquet. This section also describes how LaBoGrid and CanoPeer interact. Finally, CanoPeer is presented as a resource discovery service for LaBoGrid.

Section 6.3 presents the failure detection mechanism implemented in LaBoGrid.

Section 6.4 describes the checkpoint/restart mechanism adapted to distributed LB simulations. This mechanism is based on the regular replication of the state of the simulation on several computers: each computer saves the part of the state it hosts on a set of other computers called *replication neighborhood*.

Section 6.5 describes the construction of the replication neighborhoods the distributed checkpoint/restart mechanism presented in Section 6.4 depends on. The replication neighborhoods are built in a way that maximizes the probability of state's availability even in the case of multiple simultaneous failures caused by CanoPeer.

The distributed checkpoint/restart mechanism is disk-based and implies the management of files distributed on several computers. Section 6.6 introduces a distributed file system-oriented approach of this task.

Section 6.7 presents execution times of fault-tolerant distributed LB simulations. The impact of replication and the choice of its parameters on the total execution time is discussed.

Section 6.8 concludes this chapter.

6.2 CanoPeer

Grid computing is a recent form of distributed processing that can be defined by *"coordinated resource sharing and problem solving in dynamic, multi-institutional collaborations"* [60]. Shared resources are mainly CPU time, memory and/or disk space. These resources are used by distributed applications and the set of available resources can change over time and during the execution of the distributed application.

Peer-to-Peer (P2P) grid computing [21, 16, 29] is a very recent subdomain of grid computing. In these systems, the organization of resource sharing is fully decentralized.

CanoPeer [1] is a P2P grid computing middleware designed by Cyril Briquet in the context of his PhD thesis [21]. Essentially, CanoPeer's purpose it to schedule independent tasks grouped in jobs on computers from potentially several institutions.

The main software components of CanoPeer are:

- the resource,
- the peer,
- the user agent.

The *resource* is executed on a worker computer and is able to execute one task at a time. It also downloads the required input files to be processed by the task.

A *peer* controls a set of resources. Peers are connected and form the peer network. A peer shares its resources with its neighbors in the peer network. Peers try to complete submitted tasks as fast as possible. Therefore, if a peer is not able to directly schedule all submitted tasks on the resources it controls, it submits tasks to its neighbors in the peer network: if these have idle resources, they will schedule the tasks on them.

The *user agent* is the software component a human user can interact with in order to submit a job (a set of tasks) to a peer.

6.2.1 Task Scheduling

The tasks of a job submitted by a user agent to a given peer are scheduled on the resources it controls and/or on resources controlled by neighbor peers in the peers network. When a job is completed (all its tasks have been successfully executed), the results of the tasks are forwarded to the user agent and made accessible to the human user.

From the point of view of a particular peer p, a task scheduled by peer p on a resource it controls is called a *local* task; a task forwarded to peer p by another peer and scheduled by peer p on a resource it controls is called a *remote* task. In this case, peer p has “shared” one of its resources.

When a peer schedules tasks, local tasks can have priority over remote tasks (it depends on the scheduling policy implemented by the peer [21]). This means that, if there is no idle resource controlled by a peer to execute submitted tasks, the peer will interrupt the execution of remote tasks on resources it controls in order to execute local tasks instead. If there are no resources running remote tasks, tasks are queued and forwarded to other peers to be potentially scheduled as remote tasks.

A task that does not complete its execution because of the crash of a resource or because it was interrupted in order to free a resource is queued by the peer that submitted it initially to be scheduled again. This implies that a submitted task is eventually executed.

The described scheduling policy implies that a task's execution is interrupted not only in case of failure: CanoPeer becomes an additional source of interruptions and increases the probability that a task is interrupted before completion. Another important consequence of this policy is that grouped task interruptions are not uncommon events: if a peer has scheduled several remote tasks, it can interrupt all of them to run newly submitted local tasks.

In the context of CanoPeer, the tasks of a particular job are generally independent. The interruption of a particular task does therefore not interrupt the overall job's execution but only postpones its completion.

6.2.2 LaBoGrid Integration

LaBoGrid's main components are the Controller and the Distributed Agents (DAs). The DAs execute the distributed LB simulation code. The Controller essentially steers the whole process. DAs can be executed by CanoPeer resources (one DA per resource).

A *LaBoGrid task* (LB task) is a CanoPeer task that runs a DA when executed by a resource. A *LaBoGrid job* (LB job) is a CanoPeer job composed of several LB tasks. The submission to CanoPeer of an LB job composed of n LB tasks therefore eventually leads to the execution of n DAs.

The Controller is executed by an "out-of-the-Grid" computer and uses a CanoPeer user agent to submit LB jobs. The LaBoGrid configuration file (see Section 4.5) is associated to the LB job as input file of LB tasks.

LB tasks are not independent, they must be executed simultaneously. A CanoPeer peer always tries to minimize the completion time of a job by running simultaneously as many tasks of the job as possible (possibly by submitting some tasks to other peers). The LB tasks dependence constraint is therefore compatible with the goal of CanoPeer peers (shortest job's execution time).

A version of LaBoGrid incorporating only the features presented so far (optimized distributed LB simulations, static load balancing) could be submitted to CanoPeer as is. However, it is very likely that the execution of LaBoGrid will not finish successfully because one or several LB tasks and therefore DAs will be interrupted before the completion of the LB jobs submitted by LaBoGrid's Controller.

6.2.3 Resource Discovery

As stated in previous section, the Controller submits LB jobs to CanoPeer. An LB job submission can be seen as a request for new DAs to execute simulations. New DAs have to be requested in two situations:

- initially,
- after the detection of one or several failures (new DAs should replace the "lost" ones).

The number of LB tasks the initial LB job contains should be equal to the number of resources that are ready to execute a DA. These resources are controlled by the peer the LB job was submitted to or its neighbors in the peer network. However, this number is unknown and can vary over time (if resources crash or new resources become available).

The submission by the Controller of an LB job containing an arbitrary large number of LB tasks is problematic. As stated previously, if n LB tasks are submitted to CanoPeer, n DAs are eventually executed. However, if n is greater than the number of available resources m, m DAs are actually executed and $(n-m)$ LB tasks are queued. If the m executed DAs complete the execution of all requested LB simulations, the Controller and the m executed DAs finish their execution. The $(n-m)$ queued LB tasks are then executed, even if the Controller is already closed. An LB task executed when the Controller that submitted it is already closed is called a *useless task*.

The life-time of useless tasks is small (the connection of executed DAs to the Controller fails as Controller is not running anymore). However, the time to execute all queued tasks can become large if they are numerous (for example, if 1000 LB task are scheduled on 10 resources and executed during 1 second each before they detect that the Controller is not running anymore, at least 100 seconds are required before all useless tasks have been executed). Also, this imposes a pointless load on peers that queue useless tasks.

The maximum number of DAs needed to execute an LB simulation is known, it is the total number of sublattices. However, the total number of sublattices can be much larger than the number of available resources. Submitting an LB job with a number of LB tasks equal to the number of sublattices is therefore potentially problematic as well.

An LB task executed by a CanoPeer resource and running a DA that registered to the Controller is called a *registered task*. We suggest a method where tasks are progressively submitted: an LB job contains a limited number of LB tasks; once all submitted LB tasks become registered tasks, a new LB job can be submitted. If some LB tasks are queued because there are not enough available resources, there is no need to submit additional LB jobs. The number of LB tasks per submitted LB job represents the maximum number of useless tasks that will be executed.

The component of LaBoGrid that submits LB jobs to CanoPeer is called the *job submitter*. Two event types can trigger a DA request:

- The number of sublattices has changed. This can occur when executing a new simulation. More DAs may be requested for this new simulation.
- A new DA has connected to the Controller. If the number of registered tasks is equal to the total number of submitted LB tasks, a new LB job can be submitted to CanoPeer.

The behavior of the job submitter is described by Algorithm 6.1. `s` is the total number of submitted LB tasks, `r` the number of registered tasks and `m` the maximum number of LB tasks needed for current simulation. `s` is not updated in case of DA failure because associated tasks will be eventually executed again. `"Take an event from queue"` extracts an event from job submitter's event queue. If no event is available, the command waits until an event is available.

Algorithm 6.2 highlights the condition that must be verified to submit a new LB job: the number of registered tasks must be equal to the number of submitted LB tasks (i.e. all submitted LB tasks are running a DA) and the number of submitted LB tasks must be smaller than the maximum number of DAs needed to execute current simulation. If the condition is verified, an LB job composed of K LB tasks is submitted to CanoPeer (K is a parameter of job submitter). Note that the number of registered tasks cannot be larger than the number of submitted LB tasks.

6.3 Failure Detection

In order to recover from a failure, it must first be detected. Failure detection, in our context, consists in detecting if a DA is still running (no failure) or not (failure) during the execution of a simulation.

```
s, r, m := 0, 0, 0;
do "Job submitter has not been interrupted" →
   "Take an event from queue";
   if "Event signals new DA connection" →
      r := r + 1;
      "Request new DAs if needed"
   □ "Event signals new number of sublattices" →
      m := "New number of sublattices";
      "Request new DAs if needed"
   □ "Event signals a DA failure" →
      r := r - 1
   fi;
od
```

Algorithm 6.1: Behavior of job submitter.

```
if s = r and s < m →
   "request K new DAs";
   s := s + K
□ s ≠ r or s ≥ m → skip
fi
```

Algorithm 6.2: "Request new DAs if needed".

A common mechanism used to detect failures is the “heartbeat”: a message is regularly sent to the monitored DA; if it is not acknowledged after a given amount of time (timeout), the monitored DA is considered as “down” (“up” otherwise).

In synchronous systems, this type of detector is reliable. However, it is not the case in asynchronous systems [25]: a failure can remain unnoticed for some time (false negative) and a failure can be erroneously detected because of network congestion, slow computers, etc. (false positive).

We ignore network congestion and consider that computers slow enough to not be able to acknowledge a small heartbeat message can be considered as a real failure: a pathologically slow computer slows the whole simulation down and should better be ignored for the rest of the simulations’ execution. Finally, we consider the network (or at least the used communication protocol) as reliable (no message is lost). For all these reasons, the heartbeat mechanism is considered as a reliable enough failure detector.

In LaBoGrid, this mechanism is partially implemented by Message Output Streams (MOS, see Section 4.4.2). Each time a heartbeat message is sent, an acknowledgement (ACK) is awaited from remote Message Input Stream (MIS). If no ACK is received after a given amount of time or there was a stream error (an error detected by TCP) while sending the message, the remote DA is considered as down.

The Controller regularly sends a heartbeat message to all DAs. When a failure is detected, the Controller triggers the checkpoint/restart process on all remaining DAs. In addition, all DAs regularly send a heartbeat message to the Controller. If a failure is detected, DAs must stop their execution.

In the context of CanoPeer, a remote task executing a DA can be interrupted in order to execute a very short local task (i.e. a task that is completed in a few seconds). The DA is then interrupted for a very short period of time and is maybe executed again on the same computer. Because of the failure detection method described above, this interruption is not necessarily detected because it may happen between two heartbeats. However, even with this kind of very short interruption, the simulation cannot continue.

In order to detect this situation, a MIS receiving a message checks if the identifier in the destination field of the message is the same as hosting DA’s identifier. If it is not the case, the micro-interruption is signaled to remote MOS because the DA executed after the micro-interruption has not the same identifier as the DA executed before the micro-interruption. The MOS therefore detects the micro-interruption.

6.4 Checkpoint/restart for Distributed LB Simulations

The checkpoint/restart mechanism [68, 49, 35, 36, 63, 38] consists in saving regularly the state of a program in order to reload it in case of failure. The operation of saving the state of the program is called *checkpointing*. When the program is interrupted because of a failure, its last saved state is loaded, potentially on another computer. This operation is called *rollback*. After a rollback, the execution of the application can be restarted.

In case of failure, the processing time consumed since the last checkpoint is lost because of the rollback. Checkpointing frequency should therefore grow with the probability of failure. However, checkpointing has a cost: the execution of the program must be paused while its state is saved and state saving may take time.

Checkpointing can be transparent or controlled at the application level. Application-level checkpointing enables a finer control on the timing and on the selection of relevant data to save [63, 35]. Transparent checkpointing [38, 68] is controlled by the operating system or the middleware. The main advantage of this method is that it does not need a modification of the application code. However, it may involve an overhead in the amount of data to save. Application-level checkpointing is chosen for LaBoGrid, in particular because a fine control is required for our distributed checkpointing method (see Section 6.4.2).

The state of an application can be stored to disk (disk-based checkpointing) or directly into memory (diskless checkpointing). The latter solution avoids long access times associated to disk-based data storage [63]. However, disk-based checkpointing saves RAM and is therefore chosen.

For the sake of clarity, the checkpoint/restart mechanism will first be introduced for nondistributed LB simulations. After that, the distributed checkpoint/restart mechanism used in LaBoGrid will be presented.

6.4.1 Checkpoint/restart for Sequential LB simulations

The state of an LB simulation is composed of the densities associated to all lattice sites. Checkpointing therefore consists in storing all these values to disk in a file called *state file*.

A single state file could be used to store all states alternatively. The problem is that if the program is interrupted while it is writing data into the file, the contents of the file are corrupted and no restart is possible. Data from different checkpoints are therefore stored into a new file. In order to save disk space, each time a new state was completely written to disk, the previous state file is deleted.

Algorithm 6.3 (based on Algorithm 2.1) describes an LB simulation with a checkpoint every P iterations and restarting from iteration S. P is called *checkpoint period* and S *restart iteration*. If S is equal to 0, the lattice is initialized at equilibrium and no state file is read. Otherwise, the state file associated to iteration S is read to set the initial values of the lattice.

```
"Set lattice at state S";
t := S;
do t < timeSteps →
   "Propagate values";
   "Apply boundary conditions";
   "Apply collision";
   if t mod P = 0 →
      "Save lattice state to disk";
      "Delete previous state file"
   □ t mod P ≠ 0 → skip
   fi;
   t := t + 1
od
```

Algorithm 6.3: Sequential LB simulation with checkpointing.

6.4.2 Distributed Checkpoint/restart

The distributed LB simulation code presented in Algorithm 2.13 can be adapted in the way presented in Algorithm 6.3 to implement the distributed checkpoint/restart mechanism. However, a state file then contains the state of a sublattice and its associated subsolid. In case of failure, to be able to restart the distributed simulation, all state files associated to the restart iteration must be available, otherwise the state of the lattice is not completely available.

The file associated to the previous state should be deleted only if all current state files are completely written to disk: if a failure occurs when a DA executed by a slower computer is writing its state file, the previous state is potentially not completely available anymore because DAs executed by faster computers have already deleted the state files from it. DAs are therefore synchronized by the Controller regarding the deletion of state files.

Finally, state files generated by a DA should not be stored only on the computer hosting the DA: in case of failure of a computer hosting a DA, the state files generated by the associated DA would not be available anymore and the simulation could not be restarted. One solution is to store all state files on the computer hosting the Controller (this computer is considered as reliable). In case of failure of one or several worker computers, all state files are still available. However, this storage scheme is not efficient because the Controller acts as a bottleneck.

Another solution is to replicate state files on several worker computers. If one of these computers fails, a state file is still available on another computer. Each DA therefore sends its state files to a set of other DAs which write the received file to the disk of their associated computer. This set of DAs is called *replication neighborhood*. The construction of the replication neighborhood for all DAs is discussed later in this chapter (see Section 6.5).

The robustness of this scheme increases with the number of state file replicas available. However, the cost of replication in terms of execution time increases linearly with the number of replicas. There is therefore a robustness/efficiency trade-off regarding the number of replicas. The number of replicas to generate for a state file is called *replication degree*. The replication degree gives the size of replication neighborhoods.

Algorithm 6.4 (based on Algorithm 2.13) describes a distributed LB simulation code with a checkpoint every `P` iterations, a replication degree `D` and restarting from iteration `S`. `"Checkpoint sublattice state"` command defines the state's replication (see Algorithm 6.5).

The time to execute `"Save sublattice state to disk"` and `"Send state file to D other DAs"` is called *replication time*.

To reduce the impact of replication on simulation execution time, state files replication can be executed in parallel of the simulation. `"Checkpoint sublattice state"` can then be rewritten as shown in Algorithm 6.6.

`"Start new replication process"` starts a new thread that executes in sequence the commands `"Send state file to D other DAs"`, `"Signal state file was replicated"` and `"Wait all state files have been replicated"`.

`"Wait for end of previous replication process"` ensures that there is only one replication process in progress at a time for all DAs.

```
"Wait informations from master";
"Set sublattice at state S";
t := S;
do t < timeSteps →
    "Extract sublattice outgoing densities";
    "Send sublattice outgoing densities to neighbors";
    "Propagate sublattice values";
    "Receive sublattice incoming densities";
    "Set sublattice incoming densities";
    "Apply sublattice boundary conditions";
    "Apply sublattice collision";
    "Wait for all outgoing densities to have been sent";
    "Checkpoint sublattice state";
    t := t + 1
od
```

Algorithm 6.4: Parallel LB simulation with checkpointing.

```
if t mod P = 0 →
    "Save sublattice state to disk";
    "Send state file to D other DAs";
    "Signal state file was replicated";
    "Wait for all state files to have been replicated";
    "Delete previous state file"
□ t mod P ≠ 0 → skip
fi
```

Algorithm 6.5: "Checkpoint sublattice state".

```
if t mod P = 0 →
   "Save sublattice state to disk";
   "Wait for the end of previous replication process";
   "Delete obsolete state file"
   "Start new replication process"
□ t mod P ≠ 0 → skip
fi
```

Algorithm 6.6: "Checkpoint sublattice state" (parallel version).

Configuration	compressed size (KB)	Compression Ratio
Equilibrium	2.09	0.0017
Random	1120.25	0.93

Table 6.1: Compressed size of state file and compression ratio for a state file.

6.4.3 State Files Compression

State files could contain compressed data in order to reduce their transmission time and therefore the replication time. A state file contains all sublattice values (a vector of real values for each site) and subsolid values (a boolean for each site of the lattice). The compression ratio (ratio of compressed data size to uncompressed data size) for subsolids should be good (around 0.1%) as subsolid values are very redundant. The compression ratio for sublattices depends on the state of the simulated flow.

The compression ratio of a state file containing a D3Q19 sublattice of size $(20,20,20)$ and a subsolid containing only empty space (all values are equal to false) has been evaluated in two configurations:

- fluid at equilibrium,
- completely random fluid (this fluid has no physical meaning).

The uncompressed state file weighs 1195.88 kilobytes for both configurations.

Table 6.1 shows the compressed size and compression ratio for these two configurations. With a fluid at equilibrium, a very good compression ratio is achieved as the redundancy is very important. With a fluid initialized with random values,

#iterations	pipe	foam
0	0.0017	0.0023
1	0.0032	0.0189
10	0.0067	0.5847
100	0.0030	0.6875
1000	0.0030	0.6875

Table 6.2: Compression ratio of state files after increasing numbers of iterations in the context of two different simulations: a flow in a rectangular pipe and a flow through a metallic foam.

it is clearly not the case. However, “real-world” fluids may exhibit more redundancy.

To illustrate the evolution of lattice values redundancy, the compression ratio for state files was measured for increasing numbers of iterations in two different flow simulations:

1. a flow trough a rectangular pipe,
2. a flow trough a metallic foam.

The measured compression ratios are given in Table 6.2. Compression ratio stabilizes for both simulations after 100 iterations.

We observe that for simulation 1 (pipe), the compression ratio remains good throughout the execution. This is because the obtained flow is laminar and lattice values are therefore very redundant (lattice slices perpendicular to flow direction are identical).

For simulation 2 (foam), compression ratio stabilizes at around 69% which is better than the completely random case. This simulation is representative of the kind of simulations that are typically run with LaBoGrid.

Compression reduces the size of the data that must be replicated and should therefore reduce the transmission time of these data. However, it has a cost regarding the time to generate the state file. Table 6.3 shows the time to generate a state file and write it to disk with and without compression in the case of simulation 2 after 100 iterations. It takes about 12 times more time to generate a state file and write it to disk when compression is enabled. However, state files are transmitted multiple times through the network because of replication but written only one time to disk. Compression’s cost is therefore potentially acceptable if the replication degree is large.

Write type	Write time (ms)
Uncompressed	45
Compressed	531

Table 6.3: Time to generate a state file and write it to disk with and without compression.

Replication time with compression enabled or disabled can be estimated in function of replication degree. It is then possible to establish if compression should be used (i.e. if replication time without compression is higher than replication time with compression).

Let W_u be the time to generate and write a state file to disk without compression and W_c be the time to generate and write the same file but with compression. Let T_u be the time to transmit the uncompressed file and T_c be the time to transmit the compressed file. Finally, let d be the replication degree. d is calculated such as

$$W_u + dT_u > W_c + dT_c \tag{6.1}$$

Let B be the bandwidth of the network used to transmit state files ($B = 10240$ kilobytes/second is an estimate of the bandwidth of TCP/IP communications over a 100 megabits Ethernet network). In the case of simulation 2, after 100 iterations, the parameters of Equation 6.1 have the following values:

$$\begin{aligned}
W_u &= 0.045 \\
W_c &= 0.531 \\
T_u &= \frac{1638.4}{10240} = 0.16 \\
T_c &= \frac{1126.4}{10240} = 0.11
\end{aligned}$$

All parameters are expressed in seconds.

When solving inequality in Equation 6.1, we find that $d > 9.72$. Compression is therefore interesting when the replication degree is greater or equal to 10.

The choice of the replication degree is based on the level of robustness that is required. A replication degree equal to 10 implies that at least 9 computers can fail simultaneously without affecting the availability of full simulation's state. In environments like CanoPeer, this level of robustness can be required. It is then interesting to compress state files.

However, in more reliable environments where the simultaneous failure of several computers is an uncommon event, a lower replication degree is used ($d = 1$). State files compression then implies an overhead regarding the replication of uncompressed files.

6.5 Replication Neighborhood Construction

As stated previously, a DA that replicates its state sends its state file to a set of DAs called replication neighborhood. The replication degree gives the size of replication neighborhoods. Two important aspects have to be taken into account when building replication neighborhoods:

- Robustness: in the context of CanoPeer, grouped failures can be a common event. Replication neighborhoods must be built in a way that ensures that state files are still available in case of grouped failure (and, of course, single failures).
- Efficiency: each DA should be part of the same number of replication neighborhoods in order to uniformly share state files transmissions. Also, this ensures an efficient usage of distributed disk space.

Grouped failures have the property that interrupted DAs have mostly been executed by resources controlled by the same peer. State files should therefore be replicated to resources controlled by different peers: if a state file is replicated to three resources from three different peers and two grouped failures occur because of two of these peers, the replicated file is still available. This replication scheme is also robust to single failures.

The construction of replication neighborhoods can be expressed as a graph optimization problem. We present hereafter the graph optimization problem related to the construction of replication neighborhoods as well as a heuristic solution to this problem.

6.5.1 Replication Graphs

The *resource-level replication graph* (RRG) is a directed graph where each vertex represents a resource. An edge going from vertex p to vertex q means that the DA executed by the resource associated to q is in the replication neighborhood of the DA executed by the resource associated to p.

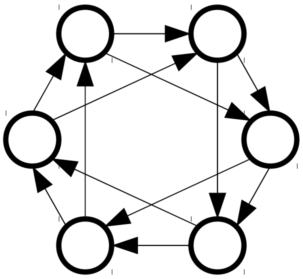

Figure 6.1: Well conditioned RRG with 6 resources and a replication degree of 2.

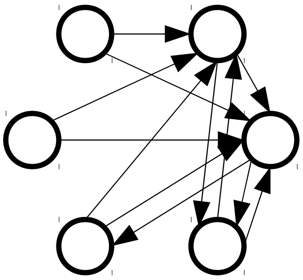

Figure 6.2: Degenerated RRG with 6 resources and a replication degree of 2.

The replication degree *d* imposes that each vertex of the RRG has exactly *d* outgoing edges. To uniformly distribute transmissions, each vertex should ideally have *d* incoming edges.

Figure 6.1 shows a well-conditioned RRG with a replication degree equal to two. Each vertex has exactly two incoming and two outgoing edges. A degenerated RRG is shown in Figure 6.2. Two resources centralize most of the replicas and thus, act as bottlenecks. Such a situation should be avoided if possible.

The *peer-level replication graph* (PRG) is a directed graph where each vertex represents a peer. An edge going from vertex *p* to vertex *q* means that a resource from the peer associated to *q* is in the replication neighborhood of a resource from the peer associated to *p*.

A vertex of the PRG is weighted with the number of resources owned by its associated peer. An edge of the PRG going from a vertex *p* to a vertex *q* is weighted by the number of resources from the peer associated to *q* being in the replication neighborhoods of resources from the peer associated to *p*.

Figure 6.3 shows the RRG from Figure 6.1 where resources are grouped by peer. The associated PRG is shown in Figure 6.4.

The RRG of Figure 6.3 does not fulfill the constraints defined above: state files

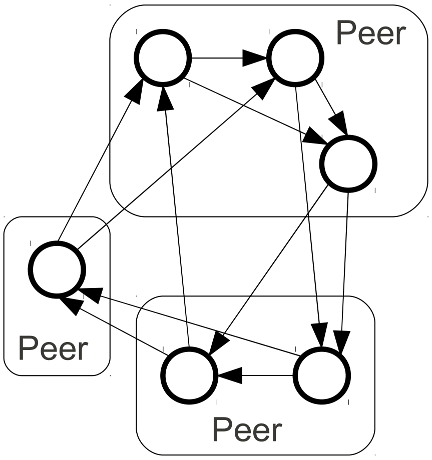

Figure 6.3: RRG from Figure 6.1 with resources organized by peer.

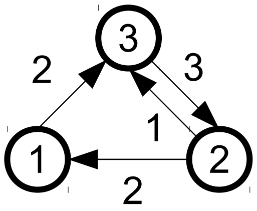

Figure 6.4: PRG based on organization presented in Figure 6.3.

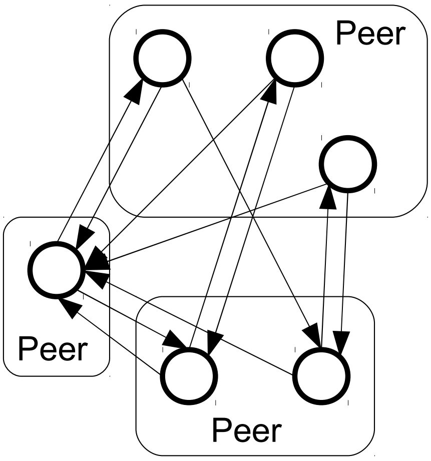

Figure 6.5: Corrected version of the RRG from Figure 6.3.

are replicated to resources that are part of the same peer. The RRG of Figure 6.5 is a corrected version of the RRG of Figure 6.3 that ensures that each resource replicates its state files on resources from different peers. We can observe that this robustness constraint potentially implies the production of a degenerated RRG.

The replication neighborhood of a particular DA can directly be extracted from an RRG like showed in Figure 6.5. In the method presented below, the PRG is built by solving an optimization problem (see sections 6.5.2 and 6.5.3). The RRG is then directly constructed in function of the PRG (see Section 6.5.4).

6.5.2 PRG Optimization Problem

Let V_{PRG} be the set of vertices of the PRG. The PRG built from a set of peers must have the following properties (with $p \in V_{PRG}$):

1. No edge connects a vertex to itself. This ensures that the resources from a peer replicate their state to resources from other peers.

2. Given replication degree d and given the number of resources r_p controlled by the peer associated to vertex p, p must have $d \times r_p$ outgoing edges.

3. Let s_p be the number of incoming edges of vertex p and L be the list of $\frac{s_p}{r_p}$ values for all vertices of the PRG. Then, $Var(L)$ the variance of the values in L is minimized. This property implies a load balancing regarding the number of state files each resource receives during state replication.

The set of vertices of the PRG is directly built from the set of peers. Building the edges set is a constrained optimization problem: the first two properties listed above are constraints and third property is the objective function.

The optimization variables are noted x_{pq} where $p,q \in V_{PRG}$. x_{pq} gives the weight of the edge going from vertex p to vertex q ($x_{pq} = 0$ means there is no edge between p and q). The optimization problem can therefore be stated as follows:

$$\min_{\forall p,q \in V_{PRG} \,:\, x_{pq}} Var(L)$$

under the following constraints:

- $\forall p \in V_{PRG} \ : \ x_{pp} = 0$
- $\forall p \in V_{PRG} \ : \ \sum_{q \in V_{PRG}} x_{pq} = d \times r_p$
- $\forall p,q \in V_{PRG} \ : \ x_{pq} \geq 0$

6.5.3 Construction of the PRG

To solve the optimization problem, we use a heuristic method relying on a common scheme: variables are initialized in a way that all constraints are met but $Var(L)$ (cost function) is not necessarily minimized. Variables are then iteratively adjusted in order to minimize $Var(L)$ without violating the constraints.

Let *out*, *in* and *inAvg* be three functions defined as follows:

$$\begin{aligned} out(p) &= \sum_{q \in V_{PRG}} x_{pq} \\ in(q) &= \sum_{p \in V_{PRG}} x_{pq} \\ inAvg(q) &= \frac{in(q)}{r_q} \end{aligned}$$

$out(p)$ gives the number of outgoing edges of vertex p, $in(q)$ the number of incoming edges for vertex q and $inAvg(q)$ the average number of incoming edges per resource of the peer associated to vertex q. The cost function $Var(L)$ can be expressed in function of *inAvg*:

$$Var(L) = Var\left(\{inAvg(p) \mid \forall p \in V_{PRG}\}\right)$$

When adjusting variables, $out(p)$ must remain equal to $d \times r_p$. This means that if a value x_{pq} is increased, another value $x_{pq'}$ must be decreased accordingly.

However, $inAvg(q)$, $inAvg(q')$ and therefore $Var(L)$ are potentially different. The adjustments should be done in order to reduce $Var(L)$ to eventually reach a local optimum.

We consider variables x_{pq} are represented by an array x of N lines and N columns where N $= |V_{PRG}|$. Algorithm 6.7 describes the construction of x. x is initialized and iteratively adjusted. The cost function is evaluated after each adjustments pass (command "Balance rows of x"). The given threshold E controls the termination of the algorithm: when the gain obtained after an adjustments pass is lower than this threshold, the loop terminates its execution. E is a strictly positive real number.

```
var
   c1, c2 :  real
begin
   "Initialize x";
   c2 := "Evaluate cost function";
   "Balance rows of x";
   c1 := "Evaluate cost function"; {P}
   do (c2 - c1) ≥ E → {P}
      "Balance rows of x";
      c2 := c1;
      c1 := "Evaluate cost function" {P}
   od {P}
end
```

Algorithm 6.7: Construction of x.

The loop invariant P from Algorithm 6.7 is defined by the following expression:

$\{P$: *"c1 is the value of cost function at time t"* $\wedge$ *"c2 is the value of cost function at time t-1"* $\wedge\ c2 \geq c1\}$

P essentially states that the algorithm reduces the cost or leaves it unchanged at each iteration but never increases it.

In order to prove the termination of this loop, we can define the sequence $S = c_1, c_2, \ldots, c_N$ of cost function's values evaluated after each adjustments pass (each time command `"Evaluate cost function"` is executed). This sequence has following properties:

1. $(c_i - c_{i+1}) > E$ for $0 \leq i < N-1$,
2. $(c_{N-1} - c_N) \leq E$.

The termination function t for the loop in Algorithm 6.7 is therefore given by the following expression:

$$t = \left\lceil \frac{c(x)}{E} \right\rceil$$

where $c(x)$ is the result of `"Evaluate cost function"` command and E the threshold. $c(x)$ must be a strictly decreasing real function when evaluated at each iteration before loop's commands. $c(x)$ is divided by E in order to guarantee that the difference between the values obtained from two subsequent evaluation of t is at least 1. Finally, ceiling function is applied to the result of the division of $c(x)$ by E in order to make t an integer function.

"Initialize x"

Algorithm 6.8 describes the initialization of `x`. It ensures that `x[p,p] = 0` with $0 \leq$ `p` $<$ `N` and that the sum of the elements of a row p of `x` is equal to $(d \times r_p)$, in particular when $(d \times r_p)$ is not divisible by `N-1`.

"Evaluate cost function" and "Balance rows of x"

The method proposed to adjust variables indirectly minimizes $Var(L)$: the distance between the smallest and the largest values of *inAvg* is used as a new cost function (and returned by command `"Evaluate cost function"`). It is iteratively reduced, eventually leading to a reduction of $Var(L)$.

However, the new cost function is not strictly equivalent to $Var(L)$ because even if the distance between the smallest and the largest values of *inAvg* cannot be reduced, the variance of the values in-between could maybe still be reduced. A solution for the new cost function is therefore potentially suboptimal regarding $Var(L)$.

Algorithm 6.9 describes the command `"Balance rows of x"`. It essentially implies an adjustment for each row of `x`.

The sum of the elements of each row must remain constant during adjustments. For a fixed row p, two columns q' and q'' are chosen such that $inAvg(q') =$

```
r := $(d \times r_p)$ mod (N-1);
v1 := $(d \times r_p)$ div (N-1);
v2 := v1 + 1;
p := 0;
do p < N →
   q := 0;
   s := 0;
   do q < N →
      if p = q → x[p,q] := 0
      □ p ≠ q and s < r → x[p,q] := v2; s := s + 1
      □ p ≠ q and s ≥ r → x[p,q] := v1
      fi;
      q := q + 1
   od;
   p := p + 1
od
```

Algorithm 6.8: "Initialize x".

```
p := 0;
do p < N →
   "Balance row p";
   p := p + 1
od
```

Algorithm 6.9: "Balance rows of x".

$\min_q inAvg(q)$, $inAvg(q'') = \max_q inAvg(q)$, $p \neq q'$ and $p \neq q''$. A value α greater or equal to zero is then computed such as following expression is minimized.

$$\left|\left(inAvg(q') + \frac{\alpha}{r_{q'}}\right) - \left(inAvg(q'') - \frac{\alpha}{r_{q''}}\right)\right| \tag{6.2}$$

The command `"Balance row p"` implies the search for `q′` and `q″` and the update of `x` as follows:

- `x[p,q′] := x[p,q′] + α`
- `x[p,q″] := x[p,q″] - α`

The core of `"Balance row p"` is the computation of α. Let $\tilde{\alpha}$ be a real value such as $\tilde{\alpha} = \alpha + \varepsilon$ with $0 \leq \varepsilon < 1$, Equation 6.2 is minimized as follows:

$$\left(inAvg(q'') - \frac{\tilde{\alpha}}{r_{q''}}\right) - \left(inAvg(q') + \frac{\tilde{\alpha}}{r_{q'}}\right) = 0$$

and

$$\tilde{\alpha} = \frac{r_{q'}r_{q''}(inAvg(q'') - inAvg(q'))}{(r_{q'} + r_{q''})}$$

This solution is greater or equal to zero because $inAvg(q'') \geq inAvg(q')$ and $r_{q'}$ and $r_{q''}$ are greater or equal to zero by definition.

The value of α is then simply equal to $\lfloor \tilde{\alpha} \rfloor$. In this case:

$$\left(inAvg(q'') - \frac{\alpha}{r_{q''}}\right) - \left(inAvg(q') + \frac{\alpha}{r_{q'}}\right) = e$$

with

$$e = \frac{\varepsilon}{r_{q''}} + \frac{\varepsilon}{r_{q'}}$$

a positive real value. This implies that:

$$inAvg(q') \leq inAvg(q') + \frac{\alpha}{r_{q'}} \leq h(q'') - \frac{\alpha}{r_{q''}} \leq inAvg(q'')$$

The distance between the smallest and the largest value of *inAvg* is therefore either smaller, either unchanged. This is true after each row has been balanced (after the execution of command `"Balance row p"`). This property is compatible with the loop invariant P of Algorithm 6.7.

6.5.4 Construction of the RRG Using the PRG

The set of vertices V_{RRG} of the RRG is built from the set of resources. The set of edges of the RRG is constructed using the array x constructed with Algorithm 6.7. x[p,q] gives the number of edges that can go from a resource controlled by the peer associated to p to a resource controlled by the peer associated to q.

For a given vertex $r \in V_{RRG}$ associated to a resource controlled by a peer p, there must be d outgoing edges to d other vertices. These d other vertices should represent resources controlled by d other peers than p. There can be an edge from vertex r to a vertex $s \in V_{RRG}$ if the resource associated to s is controller by a peer q and $x_{pq} > 0$.

A vertex $r \in V_{RRG}$ is connected to another vertex $s \in V_{RRG}$ if a directed edge goes from r to s. A vertex is unconnected if it is not connected to any other vertex. Algorithm 6.10 describes the construction of the RRG edges set by using x: A is the set of unconnected vertices, each vertex $r \in A$ is removed from A and connected to d other vertices. These vertices are chosen in function x.

```
"All vertices of $V_{RRG}$ are added to A"; $\{P_1\}$
do "A is not empty" $\rightarrow$ $\{P_1\}$
    "Remove vertex $r$ from A";
    p := "the peer controlling $r$";
    k := 0; $\{P_2\}$
    do k < $d$ $\rightarrow$ $\{P_2\}$
        "Select next peer q such as x[p,q] > 0";
        "Select next resource $s \in V_{RRG}$ of peer q";
        "Connect $r$ to $s$";
        x[p,q] := x[p,q] - 1;
        k := k + 1 $\{P_2\}$
    od $\{P_2\}$
    $\{P_1\}$
od $\{P_1\}$
```

Algorithm 6.10: Construction of the RRG edges set.

Following expression gives loop invariant P_2 from Algorithm 6.10:

$$\{P_2\text{: “}\textit{vertex r is connected to } \texttt{k} \textit{ other vertices}\text{”}\}$$

P_1 is given by the following expression:

$$\{P_1\text{: } \textit{“vertices of } (V_{RRG} \setminus A) \textit{ are connected to d other vertices”}\}$$

When A is empty (after loop's execution), P_1 implies that all vertices of V_{RRG} are connected to d other vertices. Replication neighborhoods of size d can therefore be extracted for each resource.

`"Select next peer q such as x[p,q] > 0"` and `"Select next resource` $s \in V_{RRG}$ `of peer q"` are implemented such as the elements of the sets are selected in a cyclic way: always in the same order and after the last element of the set has been selected, the next element that will be select is the first element of the set. This ensures that the number of incoming edges is uniformly distributed among the vertices associated to a given peer.

6.6 Distributed File System

State replication causes several identical files to be stored on several computers. In case of failure, a particular state file has to be made available on a particular computer to restart the simulation. If the state file is not available in the local file system, it must first be downloaded from another computer. A DA must therefore be able to derive the address of another DA given a state file's name in order to download the associated file, this file being stored in the local file system of the computer hosting contacted DA.

A fault-tolerant Distributed File System (DFS) [5, 2, 6] could be used to store and retrieve state files. However, because of the portability objective of LaBoGrid, the use of an external system should be discarded. The DFS should be an embedded component of LaBoGrid. Moreover, in LaBoGrid, the state file replication process must be easily controlled:

- files stored on a computer must be replicated at least to the computers from its replication neighborhood,
- file replication should be triggered by LaBoGrid, not by the DFS.

The DFS used by LaBoGrid should provide following capabilities:

- **Adding an existing local file to DFS**: an existing file is inserted into the DFS to be made available to other computers. After insertion, the file must be in read-only mode (no deletion or modification is allowed).

- **Retrieve a file from DFS**: a file stored in the DFS is copied (if necessary) into the local file system of a particular computer.
- **Replicating a local file to a given set of other computers**: a local file previously inserted into DFS is replicated to a given set of computers. It may be interesting to only insert a file into DFS without replicating it (for example, when a file can be shared without requiring fault-tolerance). This is why adding and replicating a file are two separate operations.
- **Deleting files from DFS**: Files that are not needed anymore by any computer can be removed from all computers hosting them. The file names of the files to delete are defined by a regular expression. This avoids to explicitly name a potentially large number of files upon deletion.

6.6.1 LaBoGrid Embedded Distributed File System

A *file unique identifier* (FUID) is associated to each file inserted into the DFS. The FUID of a file is represented by a string. This allows operations on groups of files (i.e. deletion) whose FUIDs match a given regular expression. Note that the FUID of a file is not necessarily its file name: a file having FUID "state_9_0.dat" can be stored locally in a file named "down_42.bin".

The *global file locations table* (gFLT) provides, given a FUID, the address of the computers hosting the associated file. The gFLT is maintained and hosted by the Controller. DAs maintain a partial copy of it called *partial file locations table* (pFLT) in order to reduce the number of requests sent to the Controller to get the locations of a given file. The pFLT is not necessarily consistent with the gFLT and is updated only if needed. The maintenance cost of the pFLTs is therefore minimized.

The implementation of the component hosted by the Controller and the DAs is based on the asynchronous agents framework presented in Section 4.4.1. The DFS calls are therefore asynchronous. These components are called *Distributed File System Peers* (DFSP).

A file can be partitioned into smaller parts called *chunks*. Files are transmitted chunk by chunk between DFSPs to avoid to completely load file's contents into memory. Several files can be sent/received at the same time. Each DFSP maintains its *local files table* (LFT) that provides the path to a local file given its associated FUID.

The DFS operations previously introduced are now described in more details. For the sake of simplicity, the asynchronous nature of the presented algorithms is hidden (the exchange of messages in an asynchronous way implied by these operations is not explicitly exposed).

Adding an existing local file to DFS

When adding an existing file into DFS, its FUID has to be provided by the application (i.e. it is not the DFS that chooses the FUID to associate to a given file). The file is then added to LFT, pFLT is updated and a message is sent to the Controller which updates the gFLT.

In order to guarantee the uniqueness of state files' FUID, the following naming scheme is used: the state of sublattice *s* at time step *i* is stored in the file associated to the FUID "*s_i*.state".

Retrieve a file from DFS

To retrieve a file from DFS, a FUID must be provided. The result is the path to a local file or an error if the file could not be found. Algorithms 6.11 describes the behavior of the DFSP when a file is requested. Expression `"File successfully downloaded"` is true if command `"Download file from locations provided by pFLT"` successfully downloaded requested file from one of the provided locations. It is false if the file could not be downloaded from any of the provided locations or no location is provided.

If a file is not successfully downloaded from locations coming from the pFLT, it is maybe because the pFLT was not up-to-date. This is why the download is then triggered with new locations coming from gFLT (see Algorithm 6.12).

Replicating a local file to a given set of other computers

The FUID of the file to replicate and a list of destination DAs (the replication neighborhood) must be provided. The given FUID must have an entry in the LFT. A message is sent to given DAs (actually, to the DFSP they host), which update their pFLT and download the file.

The DFSP that initiated the replication is called the *source*. The DFSPs that downloaded the file to replicate are called *destinations*. When a destination has downloaded the file to replicate, it adds it to its LFT, sends a message to source which updates its pFLT and finally (when all destinations downloaded the file) notifies the Controller which updates the gFLT.

```
if "FUID is present in LFT" →
   "Result becomes path associated to key FUID in LFT"
□ "FUID is not present in LFT but present in pFLT" →
      "Download file from locations provided by pFLT";
      if "File successfully downloaded" →
         "Downloaded file is added to LFT";
         "Result becomes path to downloaded file"
      □ not "File successfully downloaded" →
         "Get locations from gFLT and download file"
      fi
□ "FUID is not present in LFT and not present in pFLT" →
   "Get locations from gFLT and download file"
fi
```

Algorithm 6.11: Retrieval of a file from DFS.

```
"Request file locations from gFLT";
if "No locations have been received" → "ERROR"
□ "Locations have been received" →
   "Update pFLT with received file locations";
   "Download file from locations provided by pFLT";
   if "File successfully downloaded" →
      "Downloaded file is added to LFT";
      "Result becomes path to downloaded file"
   □ not "File successfully downloaded" → "ERROR"
   fi
fi
```

Algorithm 6.12: "Get locations from gFLT and download file".

At the end of the replication, the pFLT of the source and the gFLT contain at least the source and the destinations in the locations associated to the FUID. The pFLTs of the destinations contain at least the source. The LFT of the source and the destinations contains an entry for the given FUID.

Deleting files from DFS

The files to be deleted have a FUID matching a given regular expression. Entries are first removed from gFLT by the Controller. A message is then broadcasted to all DFSPs to remove their local files (delete the file from local file system and remove associated entries in pFLT and LFT).

No operation (update, retrieve, replicate) on a file being deleted should be in execution or triggered during the deletion process.

6.6.2 Distributed File System Oriented Checkpoint/restart

When the checkpoint process is triggered, a state file is stored into the local file system for each sublattice.

Once it is created, the state file is added to the DFS in order to be replicated. When all replications are finished for all state files, obsolete state files can be deleted. Algorithm 6.13 is based on Algorithm 6.5 and describes this process.

```
if t mod P = 0 →
    "Save sublattice state to disk";
    "Insert state file into DFS";
    "Request replication of state file to DFS";
    "Wait for all state files to have been replicated";
□ t mod P ≠ 0 → skip
fi
```

Algorithm 6.13: "Checkpoint sublattice state" (DFS version).

Obsolete state files are deleted by the Controller when all state files have been replicated: the deletion of files matching the regular expression “.*_*i*\.state” (any state file generated at iteration *i*) is requested where *i* is the iteration at which previous checkpoint process was triggered.

At the end of a sequence of simulations, state files are not required by the next simulation of the experience (see Section 4.5.3). They are therefore deleted from DFS using the regular expression ".*\.state".

When a simulation is restarted, state files are retrieved from DFS using associated FUIDs. If a file could not be retrieved, the complete state of the simulation is not available anymore and the simulation must be restarted from the beginning. This happens if all the computers that host a copy of the same state file fail at the same time. However, a replication degree (see Section 6.4.2) "high enough" makes this event very unlikely.

The definition of a "high enough" replication degree depends on the execution environment. In the context of a classical cluster, a replication degree equal to 1 or 2 is high enough because computers are reliable most of the time and the simultaneous failure of several computers is an unlikely event. However, when LaBoGrid's DAs are executed by resources managed by CanoPeer, the replication degree should depend on the number of different peers controlling the resources. For example, if the resources executing the DAs are controlled by 4 different peers (each resource is associated to exactly one peer) and 3 peers interrupt all the LB tasks executed by their resources at the same time, the whole state is still available if a replication degree greater or equal to 3 is used and replication neighborhoods are built using method presented in Section 6.5.

6.7 Results

In this section, the execution times of distributed LB simulations with different replication parameters are measured and compared in function of different failure probabilities and replication parameters.

6.7.1 State Replication Impact on Execution Time

A distributed simulation on a D3Q19 lattice of size $(176, 176, 176)$ with 100 iterations implying a state replication every 10 iterations was executed on a cluster of 25 type A (Pentium IV 3.06Ghz, see Section 5.4) computers. A state replication is composed of the generation and the writing of a state file to disk, the replication of this file on several replication neighbors and the deletion of obsolete state files. The state files can contain compressed or uncompressed data.

Figure 6.6 shows the execution time of the distributed LB simulations for an increasing number of replication neighbors and producing compressed or uncom-

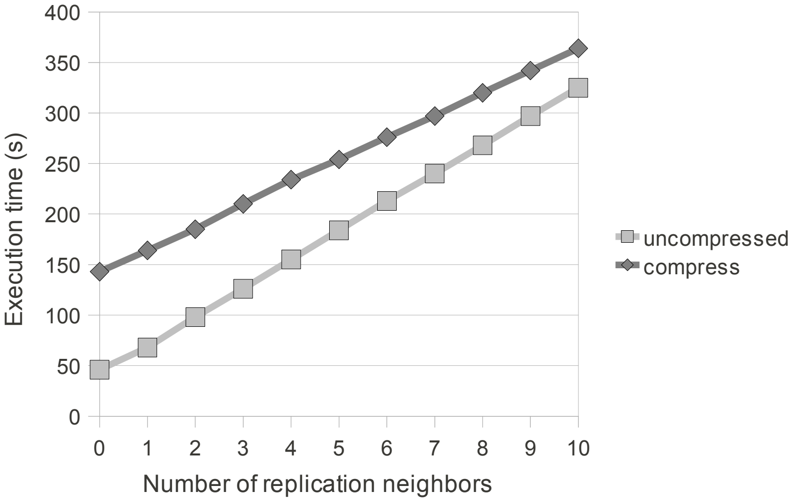

Figure 6.6: Execution time of an LB simulation on a D3Q19 lattice of size $(176, 176, 176)$ using different replication parameters.

pressed state files. The execution time with no replication neighbor is the execution time when state file is generated and written to disk but not replicated. The execution time when no state file is produced is 35 seconds.

With centralized state replication (all state files are stored on the Controller), the total execution time is of 760 seconds (more than twice the execution time when replicating state on 10 replication neighbors).

These measures illustrate the discussion of Section 6.4.3 on the interest of compressing state files: state file compression becomes interesting only if the number of replication neighbors is large enough (more than 10 neighbors in this case).

We also observe that the execution time increases linearly in function of the number of replication neighbors: with uncompressed state files, the execution time increases of approximately 28.55 seconds each time a replication neighbor is added. This additional execution time has two main components: transmission time (state files are sent to an additional computer) and deletion time (there are more files to delete when suppressing obsolete state files).

The deletion time component is negligible regarding transmission time: the complete state of the simulation is represented on 831.88 MBytes when it is not compressed ($176^3 \times 19 \times 8$ bytes for the lattice and $176^3 \times 8$ bytes for the solid). If we suppose that data are evenly distributed among computers, each computer is

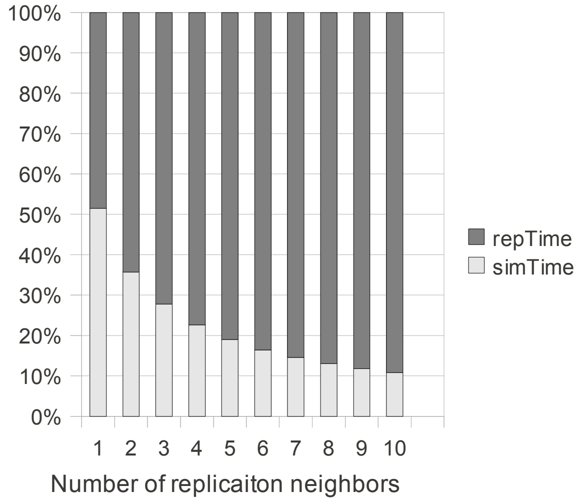

Figure 6.7: Contribution of simulation and replication times to the total execution time in the case of a simulation with 100 iterations.

responsible of 33.28 MBytes. The additional transmission time when incrementing the number of replication neighbors is therefore at least equal to 27.91 seconds (33.28 / 11.92 where 11.92 is the approximated bandwidth of a 100 megabits ethernet network) and it remains 0.64 seconds introduced by deletion time among other effects.

The execution times from Figure 6.6 show that state replication produces an important overhead: the execution time is at least doubled when enabling state replication and using only one replication neighbor. Figure 6.7 compares the contribution of simulation and replication times to total execution time and, starting from two replication neighbors, the contribution of simulation time to the total execution time is lower than 50%.

However, the number of iterations (100) of these simulations is small regarding more realistic values (1000-10000 iterations). Figure 6.8 compares the contributions of simulation and replication to the total execution time when the number of iterations of the simulation is set to 1000 and the number of replications (8) remains unchanged (it occurs every 100 iterations). In this case, the contribution of replication time to the total execution time remains under 50% even with 10 replication neighbors.

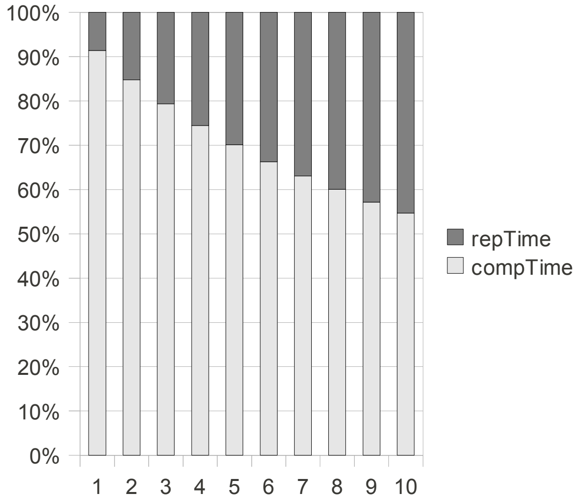

Figure 6.8: Contribution of simulation and replication times to the total execution time in the case of a simulation with 1000 iterations.

6.7.2 Execution Time in Case of Failure

When a failure occurs and state replication is not enabled, the simulation must be restarted from the initial state. Let $P(X = i)$ be the probability of the simulation being interrupted i times after the same amount of time $t_{failure}$. We suppose that $P(X \geq N)$ is equal to 0. The *mean execution time* t_{noRep} without state replication is given by

$$t_{noRep} = \sum_{i=0}^{N-1} P(X = i)(t_{exec} + i \times t_{failure})$$

where t_{exec} is the time to execute the simulation without state replication or failure.

When state replication is enabled, in case of failure, the simulation is restarted from the last saved state (we suppose it is still available after failure). The “lost” part of the simulation is therefore potentially smaller. However, as stated previously, state replication has a cost. The mean execution time t_{rep} with state replication is given by

$$t_{rep} = \sum_{i=0}^{N-1} P(X = i)(t_{exec} + t_{rep} + i \times t_{lost})$$

where t_{rep} is the replication time and t_{lost} the time elapsed between last state replication and failure. The replication time depends on the number of state replications and the number of replication neighbors. t_{lost} depends on the number of iterations between two state replications and the moment the failure occurs.

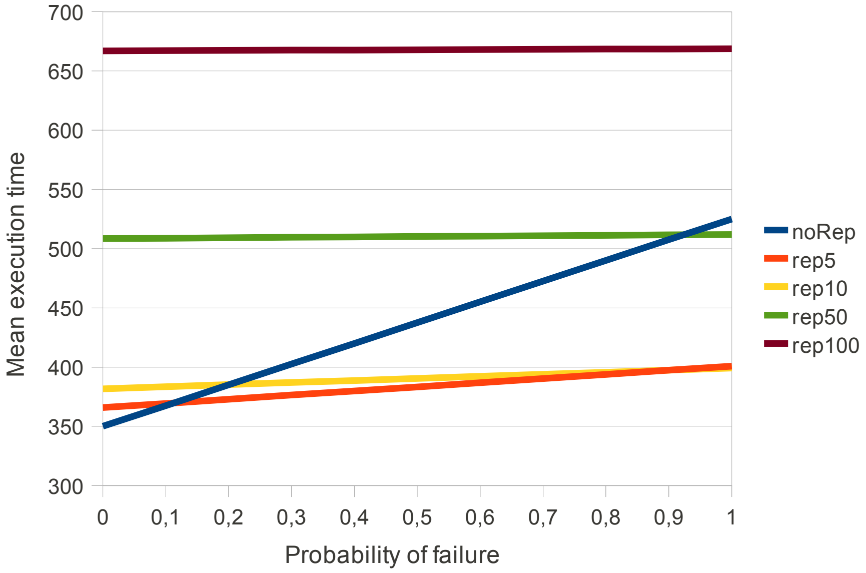

Figure 6.9: Mean execution time for different reTheseplication parameters in function of failure probability.

If $P(X)$ distribution is known, a choice can be made regarding the use or not of replication in order to minimize the mean execution time. If replication is used, replication parameters can also be selected such as the mean execution time is minimized.

In the following of this section, we suppose that P distribution is known and analyze the particular case of a simulation on a D3Q19 lattice of size $(176, 176, 176)$ executed during 1000 iterations. When state replication is used, state files are replicated on one replication neighbor.

Enable Or Disable State Replication

Let $P(X \geq 2) = 0$, $t_{failure} = t_{exec}/2$ and t_{lost} be equal to half the time to execute simulation steps between two state replications. Figure 6.9 shows the mean execution time without state replication (noRep) or with state replication triggered 5 (rep5), 10 (rep10), 50 (rep50) and 100 (rep100) times during simulation execution in function of the probability of failure $P(X = 1)$.

State replication should be used only if the mean execution time remains smaller than the mean execution time without state replication. In particular, we observe this situation three times on Figure 6.9:

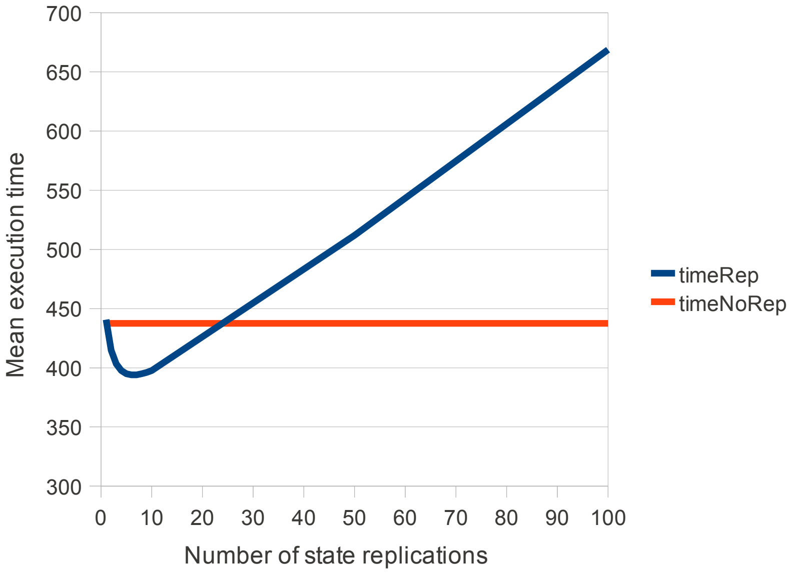

Figure 6.10: Mean execution time in function of the number of replications.

- when replicating state 50 times, with a failure probability greater than approximately 0.90,
- when replicating state 10 times, with a failure probability greater than approximately 0.20,
- when replicating state 5 times, with a failure probability greater than approximately 0.10.

Note that the $P(X \geq 2) = 0$ assumption is potentially false for long simulations. In this case, state replication becomes more interesting.

Choosing Replication Parameters

Let $P(X = 1) = 0.5$, $t_{failure} = t_{exec}/2$ and t_{lost} be equal to half the time to execute a simulation between two state replications.

Figure 6.10 shows the mean execution time in function of the number of state replications executed during the simulation. We observe that:

- with more than 25 state replications, replication is not better than the simple simulation restart;

- the mean execution time is minimized when doing 7 state replications.

This essentially shows that, if failure probabilities can be obtained, the choice of replication parameters can be optimized in order to minimize the mean execution time.

6.8 Conclusion

In this chapter, we introduced the concept of fault-tolerance: the ability of a distributed application to continue its execution if one or serveral of its processes are unexpectedly interrupted. The related problems of failure detection (i.e. the detection of the unexpected interruption of a process) and resource discovery (i.e. the search for new resources that could execute a new process) were also presented.

In order to detect the unexpected interruption of a process, a "heartbeat" based mechanism is used: the processes regularly exchange messages called heartbeats. If one of these messages could not be delivered to its destination (this event is signaled by the reception of an acknowledgement) after a given amount of time, a failure is detected.

Resource discovery is implemented by CanoPeer [1], a P2P Grid middleware designed by Cyril Briquet. LaBoGrid's DAs are submitted as tasks to CanoPeer which "finds" computers to execute them. The scheduling policies of CanoPeer may lead to the interruption of tasks whose execution is then postponed. CanoPeer is therefore an additional source of failures. In addition to that, the nature of the failures caused by CanoPeer (several tasks are interrupted simultaneously) requires adapted fault-tolerance capabilities.

Fault-tolerant distributed LB simulations are achieved through a distributed checkpoint/restart mechanism: the state of each process is regularly saved to disk and replicated to several other computers called replication neighbors. In case of failure, the last saved state is reloaded and the simulation restarted from this point. The set of replication neighbors of each computer is constructed in order to be tolerant to the failures caused by CanoPeer (i.e. the probability of failure of the computers hosting all replicas of the same state must be minimized).

The regular checkpointing of a distributed simulation's state increases the total execution time of the simulation. However, we showed that if the probability of failure is high enough, it is, on average, interesting to enable state replication for LB distributed simulations. We also showed that, if the probability of failure is known, the replication parameters can be chosen in order to minimize the average total execution time of a distributed LB simulation.

A more general analysis on the choice of replication parameters would be interesting but is not addressed in this thesis.

Chapter 7

Dynamic Load Balancing

7.1 Introduction

In Chapter 5, Static Load Balancing (SLB) applied to distributed LB simulations was presented. Before each simulation, sublattices are distributed on computers in function of their computational power in a way that minimizes execution time. The computational power of these computers is obtained by benchmarking them with small LB simulations.

In Section 5.2, a graph representation was introduced to describe a distributed application (for example, distributed LB simulations) and the system that executes it (for example, a cluster). The graph representing the application is called application graph and the graph representing the system is called resource graph.

In the context of Chapter 5, the resource graph and the application graph do not change during the application's execution. Application's distribution therefore occurs only once before application is executed. This is a static load balancing problem. However, if the application graph and/or the resource graph change during the execution of the application, the application's distribution should occur several times during the execution of the application. This is a *Dynamic Load Balancing* (DLB) problem.

In Chapter 6, we introduced methods to make LaBoGrid fault-tolerant: even in case of failure (software or hardware crash, process interruption), LaBoGrid can continue its execution and successfully finish it (i.e. all requested simulations are executed). In case of failure, the previously saved state of the running simulation is reloaded by remaining computers and simulation is restarted.

In this case, sublattices must be redistributed because the state of some of them is not available anymore. In addition, the integration of LaBoGrid and CanoPeer [1], a P2P grid computing middleware (see Section 6.2), allows to replace lost computers by new ones if possible. Sublattices should then also be redistributed to take advantage of new available resources. The sublattices' distribution process is therefore triggered potentially several times during application execution: this is a DLB problem.

In the context of LaBoGrid, the DLB problem exists if the resource graph changes during the execution of LB simulations. It is potentially the case when LaBoGrid is executed by CanoPeer but also when LaBoGrid is executed without CanoPeer on a cluster of unreliable computers.

The DLB problem has first been addressed in the context of applications with computational requirements varying over time executed by parallel or distributed systems. One typical example of this kind of application is the class of numerical simulations using adaptive mesh refinements [64] where nodes or edges are added or removed from mesh during simulation's execution in order to reduce the error of the calculation. Work must therefore migrate between processors to maintain load balance.

In our context, the computational requirements of the application (i.e. a distributed LB simulation) do not vary during a simulation. However, the set of computers executing the simulation can change: computers can be removed from it (in case of failure) or added to it (when a new computer is "discovered"). Also, the heterogeneity of computers' power (the number of processors and their individual speed) has to be taken into account.

Hendrickson and Devine [42] proposed properties that should be featured by a dynamic load balancing method:

1. Computational work should be well balanced among computers.
2. Inter-computers communications should be minimized.
3. Execution time should be small.
4. Memory usage should be modest.
5. The amount of work to migrate after a redistribution should be minimized.

Properties 1 and 2 also apply to static load balancing methods and contribute to minimize the application's execution time. Properties 3 and 4 are related to the fact that the computation of work distribution occurs in parallel of the application's execution: computational power and memory used for work distribution cannot be used by the application and vice-versa. Finally, property 5 minimizes the work migration time and, therefore, the total execution time of the distributed application.

A distributed implementation can help to fulfill the objectives of properties 3 and 4: the time to compute a work distribution should be smaller if DLB mapper is executed by several processors in parallel than by a single one. Also, the memory needs of each computer participating in the work distribution process should be smaller than if executed on a single computer.

Parallel static load balancing tools exist (JOSTLE [76], ParMetis [69], PT-Scotch [28]) but are not well suited to dynamic load balancing, mainly because of property 5. Also, they would not interface well with LaBoGrid because they are generally provided in the form of C libraries and therefore rise potential portability issues (see Section 2.3).

Methods based on local improvements [41, 77, 79] are an interesting class of DLB methods. Computers are organized into small overlapping sets called *migration sets*. Computers of a same migration set regularly exchange information about their respective work load. If a computer of a migration set is overloaded, work is migrated to the other computers of its set. As the migration sets overlap and migrations may regularly be triggered, work may spread from migration set to migration set and finally among all computers.

Dynamic load balancing algorithms based on local improvements are incremental methods that have the following advantages:

1. They scale very well if implemented in parallel (because they are based on local information only: the work load of a migration set's computers).
2. One iteration of such algorithm (involving the migration of work in function of the work load of a migration set's computers at a given time) is usually short timed and inexpensive in terms of additional memory and computational power.

However, many iterations may be required to converge towards global balance. With distributed LB simulations, global balance is important, in particular when sublattices are large. Solutions have been proposed [44] to reduce the number of iterations and the amount of migrated work but they cannot easily be implemented in parallel, in particular because they are not based anymore on local information only.

Therefore, we propose a DLB method inspired from the principles of local improvements methods (organization of computers into small migration sets) but using global load information (the load of all computers is taken into account) while remaining scalable and without slow convergence.

7.1.1 Chapter Outline

Section 7.2 presents the common organization of most local improvements methods. These methods can be described by two consecutive phases: the balancing phase and the migration phase.

Section 7.3 describes an efficient implementation of the migration phase.

The balancing phase is generally based on iterative methods that require many message exchanges between computers if implemented in a distributed way. In order to reduce the number of exchanged messages during balancing phase, we propose to use an adapted version, described in Section 7.4, of an existing algorithm called Tree Walking Algorithm (TWA).

The distributed implementation of the proposed dynamic load balancing method is addressed in Section 7.5.

Section 7.6 exposes the results of the proposed dynamic load balancing method.

Section 7.7 concludes this chapter.

7.2 Local Improvements Methods

Most local improvements methods [41, 77, 79] imply two separate phases:

1. the *balancing phase*,
2. the *migration phase*.

The balancing phase essentially consists in equalizing the work load on working computers by scheduling work migration. Work migration occurs during migration phase. The objects to migrate (sublattices in our context) are generally chosen in order to minimize communication cost during application execution. If the application is represented by a graph (see Section 5.2), this is equivalent to minimize the edge-cut of the graph partition associated to each computer.

7.2.1 Balancing Phase

In the balancing phase, the amount of work to move from one computer to another in order to balance load is decided. This phase only uses local load information: each computer is part of a migration set and is connected to the other computers of its migration set. The computers of a migration set exchange their load information. The connections induced by the migration sets can be represented by a graph called *migration graph*. This graph is the resource graph with possibly some edges removed.

Most existing methods are based on the diffusive scheme [30] where a computer work load is modeled by the following equation:

$$w_i^{t+1} = w_i^t + \sum_j \alpha_{ij}(w_j^t - w_i^t) \tag{7.1}$$

where w_i^t gives the work load of computer i at time t and α_{ij} is a constant such as:

1. $\forall i, j \ : \ \alpha_{ij} \geq 0$;
2. $\forall i \ : \ 1 - \sum_j \alpha_{ij} \geq 0$.

If computers i and j are not connected (there is no edge between i and j in the migration graph), $\alpha_{ij} = 0$.

It has been shown [30] that above method always converges to uniform distribution if the migration graph is connected and either or both of the following conditions hold:

- $\exists j \ : \ (1 - \sum_i \alpha_{ij}) > 0$,
- the migration graph is not bipartite.

At convergence, $w_i^T = w^\star$ for all computer i and global balance is reached. The amount of work each computer will send or receive from its neighbors can then be calculated.

The number T of iterations before convergence is reached depends on the number of computers, the way they are interconnected and the initial work load distribution. For example, this number varies from 50 to 1000 for random work loads generated for 64 processors connected respectively as a 6-dimensional hypercube and a ring [32].

In the context of distributed LB simulations, a diffusive scheme can be used to compute the number of sublattices n_i associated to a computer i. In Section 5.4, the CCP, giving the maximum number of sites a computer can handle per time unit, was introduced. Let c_i be the CCP of computer i, we can then write:

$$w_i^t = \frac{S \times n_i^t}{c_i} \tag{7.2}$$

where S is the number of sites per sublattice and n_i^t the number of sublattices associated to computer i at time t. In above equation, the transmission time (the time to exchange data during execution) is ignored.

Equations 7.1 and 7.2 lead to following update rule for the number of sublattices associated to a computer i at time t:

$$n_i^{t+1} = n_i^t + \sum_j \alpha_{ij} \left(\left(\frac{c_i}{c_j} \right) n_j^t - n_i^t \right) \tag{7.3}$$

where α_{ij} values have the same definition as for Equation 7.1.

In Equation 7.3, n_i^t is a real number. This implies that the number of sublattices to migrate computed in function of this value and the number of sublattices actually associated to a computer is also a real number. However, the number of sublattices to migrate should be a natural number. This issue is addressed in Section 7.4.1.

7.2.2 Migration Phase

If load balancing is solved for an application that can be represented by a graph (see Section 5.2), a partition of the application graph is associated to each computer. Work migration then consists in moving vertices of the application graph from a computer to another and thus, modifying the partitions associated to each computer.

In the migration phase, vertices to migrate are generally selected with the goal of minimizing communication costs during application's execution. Note that for a distributed application involving no communications, the selection process is trivial: any vertex can be moved as there are no edges.

A modified version of the gain criteria from the graph partitioning algorithm of Kernighan and Lin (KL) [48] is generally used to select the vertices to move.

Original KL Gain Criteria

Let $G = (V,E)$ be an undirected graph where V is the set ot vertices and E the set of edges. The cost matrix c gives the weights associated to the edges of E: let p,q

be vertices of V, if there is no edge connecting p and q then $c_{pq} = 0$, otherwise $c_{pq} = w_{pq}$ where w_{pq} is the weight of the edge connecting p and q.

Let A and B be two partitions of V ($V = A \cup B$ and $A \cap B = \emptyset$), the cost of the partitioning is given by:

$$cost = \sum_{p \in A,\ q \in B} c_{pq}$$

This value is the sum of the weights of the edges crossing partitions boundaries.

The algorithm designed by Kernighan and Lin builds partitions A and B while minimizing the partitioning cost (it therefore minimizes the edge-cut).

The core of their algorithm is the construction of two sets $X \subset A$ and $Y \subset B$ with $|X| = |Y|$ to build a new partitioning:

$$\begin{aligned} A^\star &= (A \setminus X) \cup Y \\ B^\star &= (B \setminus Y) \cup X \end{aligned}$$

such as the cost of this new partitioning is lower: there must be a gain to exchange X and Y.

Some concepts are necessary to define the gain of exchanging X and Y:

- the *external cost* E_a^B of a vertex $a \in A$: $E_a^B = \sum_{y \in B} c_{ay}$
- the *internal cost* I_a^A of a vertex $a \in A$: $I_a^A = \sum_{x \in A} c_{ax}$

The definition of E_b^A and I_b^B with $b \in B$ is similar.

Let z be the total cost due to all connections between vertices from A and B that do not imply a or b:

$$z = \sum_{\substack{p \in A,\ q \in B, \\ p \neq a,\ q \neq b}} c_{pq}$$

The cost T of partitioning can be written:

$$T = z + E_a^B + E_b^A - c_{ab}.$$

If a and b are exchanged ($A' = (A \setminus \{a\}) \cup \{b\}$ and $B' = (B \setminus \{b\}) \cup \{a\}$), the cost becomes:

$$T' = z + I_a^A + I_b^B + c_{ab}.$$

The gain of exchanging a and b is then written:

$$gain = T - T' = D_a^{B,A} + D_b^{A,B} - 2c_{ab} \tag{7.4}$$

where $D_a^{B,A} = E_a^B - I_a^A$ and $D_b^{A,B} = E_b^A - I_b^B$.

Algorithm 7.1 describes the KL refinement algorithm. It is based on the exchange of subsets of A and B until no more gain can be obtained. In this case, a local optimum has been reached. `h` represents the gain of exchanging `X` and `Y`. If `h` is lower or equal to zero, there is no gain to exchange `X` and `Y`. However, it may happen that a local optimum is never reached because the algorithm exchanges the same vertices indefinitely and `h` never becomes lower or equal to 0. The maximum number of exchanges is therefore bounded by a given number `T`.

```
var
   A, B : "set of vertex";
   X, Y : "set of vertex";
   h, i :  integer
begin
   "Build X and Y and set h";
   if h $\leq$ 0 $\rightarrow$ skip
   $\square$ h $>$ 0 $\rightarrow$
      A := (A $\setminus$ X) $\cup$ Y; B := (B $\setminus$ Y) $\cup$ X;
   fi;
   i := 0;
   do h $>$ 0 and i $<$ T $\rightarrow$
      "Build X and Y and set h";
      if h $\leq$ 0 $\rightarrow$ skip
      $\square$ h $>$ 0 $\rightarrow$
         A := (A $\setminus$ X) $\cup$ Y; B := (B $\setminus$ Y) $\cup$ X;
      fi;
      i := i + 1
   od
end
```

Algorithm 7.1: KL refinement algorithm.

Algorithm 7.2 describes the construction of sets X and Y. A sequence of N gains is generated with $N = \min(|A|, |B|)$. Each gain is produced by the exchange of a pair of vertices selected such as the gain of the exchange is maximized.

In command `"Find m ∈ V1 and n ∈ V2 that maximize the gain g"`, `g` takes following value:

$$D_m^{V2 \cup E',\, V1} + D_n^{V1 \cup F',\, V2} - 2c_{mn}$$

```
var
   V1, V2 :  "set of vertex";
   E, F : array[0..N-1] of "vertex";
   S : array[0..N-1] of integer;
   j, g :  integer;
   m, n :  "vertex"
begin
   V1, V2 := A, B;
   j := 0;
   do "V1 is not empty" and "V2 is not empty" →
      "Find m ∈ V1 and n ∈ V2 that maximize the gain g";
      "Remove m from V1"; "Remove n from V2";
      E[j] := m; F[j] := n; S[j] := g;
      j := j + 1
   od;
   "Build X and Y from E, F and S and set h"
end
```

Algorithm 7.2: `"Build X and Y and set h"`.

where E' and F' are sets containing the vertices at positions 0 to `i-1` included from arrays `E` and `F` respectively.

X and Y sets are then generated by selecting the sequence of exchanges that leads to the maximum accumulated gain (see Algorithm 7.3).

```
var
   k, sum :  integer
begin
   sum := S[0]; k := 1; h := sum;
   j := 1;
   do j < N →
      sum := sum + S[j];
      if sum ≤ h → skip
      □ sum > h →
         h := sum; k := j + 1
      fi;
      j := j + 1
   od;
   "Initialize X and Y as empty sets";
   j := 0;
   do j < k →
      "Add E[j] to X"; "Add F[j] to Y";
      j := j + 1
   od
end
```

Algorithm 7.3: `"Build X and Y from E, F and S and set h"`.

Gain Criteria With Unidirectional Vertices Migration

Unlike original KL algorithm based on the exchange of vertices, in the migration phase, a given number m of vertices must be moved from a partition to another.

We reuse the notations from previous section: let A and B be two partitions. The problem here is to find a subset $X \subset A$ such as $|X| = m$ and such that the edge-cut is minimized for partitions $(A \setminus X)$ and $(B \cup X)$.

The same reasoning as for Equation 7.4 can be used. Let z be the total cost due to all connections between vertices from A and B that do not imply a:

$$z = \sum_{\substack{p \in A,\ q \in B, \\ p \neq a}} c_{pq}$$

The cost T of partitioning can be written:

$$T = z + E_a^B.$$

If a is moved to B, the cost becomes:

$$T' = z + I_a^A.$$

The gain of moving a to B is then written:

$$gain = T - T' = D_a^{B,A} \tag{7.5}$$

where $D_a^{B,A} = E_a^B - I_a^A$.

Algorithm 7.4 describes the KL-based unidirectional vertices migration.

```
var
   A, B : "set of vertices";
   i, g :  integer
begin
   i := 0;
   do i < m →
      "Find n ∈ A that maximizes the gain g";
      "Remove n from A"; "Add n to B";
      i := i + 1
   od
end
```

Algorithm 7.4: Unidirectional KL migration.

In command `"Find n ∈ A that maximizes the gain g"`, `g` has value $D_n^{B,A}$.

7.3 Implementation of Migration Phase

In Section 7.2.2, a method to select m vertices to migrate from partition A to partition B while minimizing edge-cut has been presented. It implies the construction

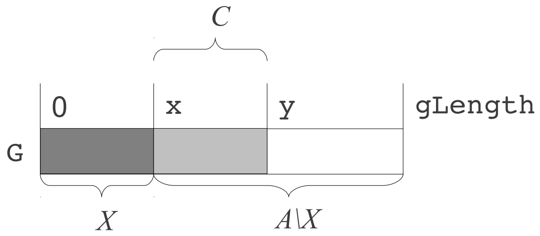

Figure 7.1: G areas illustration.

of a subset X of the vertices of A such as $|X| = m$ and the vertices of X are migrated from A to B.

The selection of the vertex to move to X must be efficiently implemented. Let C be a set defined as follows:

$$C = \{p \mid p \in (A - X) \ \wedge \ \forall q \in X \cup B : c_{pq} > 0\}$$

C is a subset of $(A \setminus X)$ and contains all vertices that are part of the *boundary* of $(A \setminus X)$. Most of the time, $|C| \ll |A \setminus X|$. Therefore, a common way to accelerate the selection of vertices to migrate is to search only in C [42]. C can also be empty if no vertex of $(A \setminus X)$ is connected to a vertex of B or X.

7.3.1 Data Structures

In our implementation, X, $(A \setminus X)$ and C are represented in the same array `G` (see Figure 7.1). An element of the array is a pointer to a record describing a particular vertex. A vertex can therefore be moved from one set to another with a swap operation.

The `Vertex` record representing a particular vertex v is described in Algorithm 7.5.

- `id` is the identifier of v and is provided by the application,
- if `index` ≥ 0, `index` is the index of v in `G` . If `index` < 0, it means v is in partition B and has no entry in `G`.
- `intC`, `extC`, `difC` respectively represent the internal cost, the external cost and the difference of the two of v.
- `edges` contains pointers to the records describing the vertices adjacent to v. `edges[i]` points to the record describing the i^{th} neighbor of v with $0 \leq i <$ `nEdges`. This neighbor is in A or B.

- weights contains the weights of the links to neighboring vertices. The weight of the link to i^{th} neighbor of v is given by weights[i] with $0 \leq i <$ nEdges.

The adjacency list of Vertex record only contains the neighbors that are in A or B. Some edges of the original graph may therefore be ignored.

In addition to the definition of Vertex, Algorithm 7.5 contains the declaration of G and the indexes x and y delimiting the areas of G.

```
type
   Vertex = record
      id, index :  integer;
      intC, extC, difC : integer;
      nEdges :  integer;
      edges :  array[0..nEdges-1] of ↑Vertex;
      weights :  array[0..nEdges-1] of integer
   end
var
   x, y :  integer;
   gLen :  integer;
   G : array[0..gLen-1] of ↑Vertex
```

Algorithm 7.5: Declarations for uni-directional vertices migration algorithm.

7.3.2 Initialization of G

The initialization of G essentially consists in copying vertices information from partition A into G.

Partitions A and B are graphs that can be represented using an adjacency list (see Section 5.2.1). The array G is initialized in function of the adjacency lists representing A and B.

Algorithm 7.6 provides the definitions and declarations for A's adjacency list (adjA) and B's adjacency list (adjB).

Record Edge describes an edge going out of a particular vertex:

- id is the vertex identifier associated to destination vertex,

- `weight` is the weight of the edge.

Record `AdjEntry` describes an entry of an adjacency list:

- `id` is the identifier of the associated vertex,
- `nEdges` is the number of adjacent vertices,
- `edges` is the list of edges to adjacent vertices.

```
type
   Edge = record
      id, weight :  integer
      end;
   AdjEntry = record
      id :  integer;
      nEdges :  integer;
      edges :  array[0..nEdges-1] of Edge
      end
var
   adjA : array[0..adjALen-1] of AdjEntry;
   adjB : array[0..adjBLen-1] of AdjEntry
```

Algorithm 7.6: Definitions and declarations for *A*'s and *B*'s adjacency lists.

`adjA` and `adjB` are sorted regarding the field `id` of their entries. A dichotomous search can therefore be used to find an entry associated to a given vertex.

The construction of `G` is done in two passes. First pass copies the vertex identifier of each entry of `adjA` into `G`. It also sets the `index` field of the entries of `G`. The second pass initializes the adjacency information of each entry of `G` (fields `nEdges`, `edges` and `weights`). The second pass depends on the first pass because all entries of `G` must be allocated before it can be applied.

First Pass

First pass is straightforward and is described by Algorithm 7.7. The definition of type `Vertex` comes from Algorithm 7.5. After the application of first pass, `G[i]` is associated to the same sublattice than `adjA[i]` with 0 ≤`i`<`adjALen`. `G` is therefore sorted regarding the field `id` of its entries.

```
var
   G : array[0..adjALen-1] of ↑Vertex;
   i :  integer
begin
   i := 0;
   do i < adjALen →
      "Allocate G[i]";
      G[i]↑.id := adjA[i].id;
      G[i]↑.index := i;
      i := i + 1
   od
end
```

Algorithm 7.7: First pass of `G`'s initialization.

Second Pass

In second pass, the edges associated to each entry of `adjA` are copied into the adjacency list of the associated `G` entry. However, some edges may be ignored because an edge going from a vertex of *A* to a vertex that is not in *A* and not in *B* is not taken into account by KL-based unidirectional vertex migration (see Section 7.2.2).

Let v be a particular vertex of *A*. Three properties can be defined to label the edges going from v to its neighbors:

- the α label is associated to an edge connecting v to another vertex of *A*,
- the β label is associated to an edge connecting v to a vertex of *B*,
- the γ label is associated to an edge connecting v to a vertex that is from another partition than *A* and *B*.

Figure 7.2 shows a vertex v of *A* and the labels associated to the edges connecting v to its neighbors.

When copying edges from an entry of `adjA` to an entry of `G`, γ edges are ignored.

Commands `"find id in adjA"` and `"find id in adjB"` search for an entry of `adjA` and `adjB` associated to the given vertex identifier `id`. A dichotomous

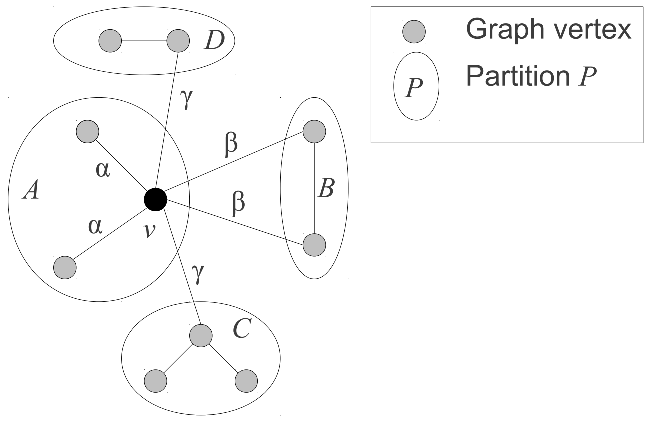

Figure 7.2: Labels associated to the edges connecting a vertex of A to its neighbors.

search regarding vertex identifier (`id` field of `AdjEntry`) is used. If the search is not successful (no entry with given sublattice identifier is found), the result of the command is a negative integer. Otherwise, it is the index of the entry associated to `id`.

If the given vertex identifier `id` represents a neighbor of a vertex v of A, the result of these two commands can be used to find out if the edge connecting v to the vertex associated to `id` is an α, β or γ edge and, therefore, if it should be added to edges list associated to v in `G` or be ignored. Let c be the result of `"find id in adjA"` and d be the result of `"find id in adjB"`. The edge connecting v to the vertex associated to `id` has following property:

- α if $(c \geq 0)$ and $(d < 0)$,
- β if $(c < 0)$ and $(d \geq 0)$,
- γ if $(c < 0)$ and $(d < 0)$.

The situation where c and d are greater or equal to zero cannot happen because a vertex cannot be in two different partitions at the same time.

The second pass of `G`'s initialization is described by algorithms 7.8, 7.9, 7.10 and 7.11. Algorithm 7.9 essentially selects the edges to copy from `adjA[i]` into `G[i]` and adds them to a list. Algorithm 7.11 copies the content of the generated list into `G[i]`↑`.edges`.

```
type
   EListEntry = record
      id, weight :  integer
   end
var
   eLst :  "list of ↑EListEntry"
begin
   i := 0;
   do i < adjALen →
      "Generate list eLst of G[i]'s edges list";
      "Initialize G[i]↑.edges using eLst";
      i := i + 1
   od
end
```

Algorithm 7.8: Second pass of `G`'s initialization.

7.3.3 Construction of X

After `G` has been initialized as described in Section 7.3.2, the set X of m vertices to migrate from partition A to partition B can be constructed.

`G` is used to represent A, X, $(A \setminus X)$ and C by associating areas of `G` to each set (see Section 7.3.1 and in particular Figure 7.1). Constant time swap operations can therefore be used to move a vertex from one set to another.

When two elements of `G` are swapped, their field `index` must be updated because `index` represents their position in `G` and this position changes in case of swap. Algorithm 7.12 defines the `swap` procedure used to swap two elements of `G`.

Algorithm 7.13 describes the construction of X in `G` array. `G`, `x` and `y` are declared in Algorithm 7.5. Fields `id`, `index`, `nEdges`, `edges` and `weights` of the entries of `G` have been initialized like shown in Section 7.3.2. The other fields (`intC`, `extC`, `difC`) are initialized by `"Compute costs for vertices of` A `and build` C`"`.

`M` vertices are then added iteratively to set X (which is initially empty) by `"Move vertex from` $(A \setminus X)$ `to` X`"`. This command increments `x`. The costs and set C must be updated each time of vertex is moved from $(A \setminus X)$ to X. This is

```
var
   ge :  ↑EListEntry;
   j, w, aInd, bInd :  integer;
   z :  Edge;
begin
   eLst := "Allocate empty list";
   j := 0;
   do j < y↑.adjL →
      z := adjA[i].edges[j];
      w := z.id;
      aInd := "find id in adjA";
      bInd := "find id in adjB";
      if (aInd < 0) and (bInd < 0) → skip {γ edge}
      □ (aInd ≥ 0) and (bInd < 0) → {α edge}
         "Initialize ge";
         ge↑.index := aInd;
         "Add ge to eLst"
      □ (aInd < 0) and (bInd ≥ 0) → {β edge}
         "Initialize ge";
         ge↑.index := -1;
         "Add ge to eLst"
      fi;
      j := j + 1
   do
end
```

Algorithm 7.9: "Generate list eLst of G[i]'s edges list".

```
"Allocate ge";
ge↑.id := w;
ge↑.weight := z.weight
```

Algorithm 7.10: "Initialize ge".

```
G[i]↑.nEdges := "length of eLst";
"Allocate G[i]↑.edges";
j := 0;
ge := "first element of eLst";
do j < G[i]↑.adjLength →
   G[i]↑.edges[j] := G[ge↑.index];
   G[i]↑.weights[j] := ge↑.weight;
   ge := "next element of eLst";
   j := j + 1
od
```

Algorithm 7.11: `"Allocate and initialize G[i]↑.edges using eLst"`.

```
procedure swap(G : array[0..gLength-1] of ↑Vertex; i : integer;
   j : integer);
var
   tmp : ↑Vertex
begin
   if i = j → skip
   □ i ≠ j →
      tmp := G[i];
      G[i] := G[j];
      G[j] := tmp;
      G[i]↑.index := i;
      G[j]↑.index := j
   fi
end
```

Algorithm 7.12: `swap` procedure.

done by command "Update costs and C" in function of the last vertex added to X.

```
x := 0;
"Compute costs for vertices of $A$ and build $C$";
"Move vertex from $(A\setminus X)$ to $X$";
"Update costs and $C$"; {$P$}
do x < M → {$P$}
   "Move vertex from $(A\setminus X)$ to $X$";
   "Update costs and $C$" {$P$}
od {$P$}
```

Algorithm 7.13: Construction of X set.

The loop invariant P from Algorithm 7.13 is given by following expression:

$$\{P\text{: } (0 \le i < x) \wedge \text{“}\textit{vertex associated to position i is part of X}\text{”} \wedge \text{“}\textit{all costs and C are up to date}\text{”} \}$$

After loop's execution, X therefore contains M vertices because x $=$ M.

Compute costs for vertices of A and build C

Initially, A covers the whole area of G and X is empty. The costs of all entries of G must be computed.

If the external cost associated to a vertex of A is greater than zero, it implies that the vertex is connected to at least one vertex from partition B. It should therefore be added to C.

Initially, C is empty and y is therefore equal to zero. To add a vertex of A associated to position i such as $i \geq$ y to C, it is simply swapped with the vertex at position y which is not part of C. y is then incremented and C contains an additional vertex.

Algorithm 7.14 describes command "Compute costs for vertices of A and build C".

Figure 7.3 illustrates the loop invariant of the loop from Algorithm 7.14. The grey area corresponds to the entries whose costs have been set. When i $=$ gLen, the costs of all entries of G are set.

```
i := 0; y := 0; {P}
do i < gLen → {P}
   n := G[i];
   "Set costs for n";
   if n↑.extC > 0 → swap(G, i, y); y := y + 1
   □ n↑.extC = 0 → skip
   fi;
   i := i + 1 {P}
od {P}
```

Algorithm 7.14: "Compute costs for vertices of *A* and build *C*".

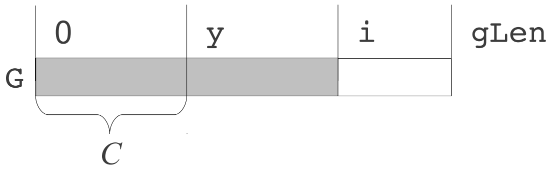

Figure 7.3: Loop invariant of loop from Algorithm 7.14.

Command "`Set costs for n`" sets the costs of the entry `n` points to. In a way similar than in Section 7.3.2, we can define properties that can be associated to the edges connected to a particular vertex v of A:

- a δ edge connects v to another vertex of $(A \setminus X)$,
- an ε edge connects v to a vertex of B or X.

The vertex associated to `n` must be in A (it is the case in Algorithm 7.14 because `n` points to an entry of `G` which is in A by definition). The edges from array `n↑.edges` can therefore be labelled with above properties.

The internal cost of `n` is equal to the sum of the weights associated to the δ edges of `n↑.edges`. The external cost of `n` is equal to the sum of the weights associated to the ε edges of `n↑.edges`.

The edge associated to `n↑.edges[j]` with $0 \leq$ `j` $<$ `n↑.nEdges` is a δ edge if `n↑.edges[j]↑.index` is greater or equal to `x`. In this case, `n↑.edges[j]↑` is an entry of `G` that is in $(A \setminus X)$.

If `n↑.edges[j]↑.index` $<$ `x`, the edge associated to `n↑.edges[j]` is an ε edge because if $0 \leq$ `n↑.edges[j]↑.index` $<$ `x`, `n↑.edges[j]↑` is an entry of `G` that is in X and if `n↑.edges[j]↑.index` < 0, `n↑.edges[j]↑` is associated to a vertex in B (see Section 7.3.2).

Algorithm 7.15 describes "`Set costs for n`".

Move Vertex from $(A \setminus X)$ to X

As stated in Section 7.2.2, the vertex v to move from $(A \setminus X)$ to X must be selected in a way that maximizes the gain of moving v. The gain obtained when moving v from $(A \setminus X)$ to X is the difference of the external and the internal costs associated to v.

In the context of this implementation, this amounts to search for the entry `G[i]` with `x` $\leq$ `i` $<$`gLen` such as `G[i]↑.difC` is maximized.

In order to accelerate the search for the vertex to move, it is searched for in C if C is not empty because $|C| \ll |A \setminus X|$ most of the time. The entry to find is then `G[i]` with `x` $\leq$ `i` $<$`y` such as `G[i]↑.difC` is maximized.

When the vertex to move is found, it is moved from $(A \setminus X)$ to X. Let `G[maxI]` be the entry associated to the vertex to move. The entries `G[x]` and `G[maxI]` are swapped.

```
intC := 0; extC := 0; j := 0;
do j < n↑.nEdges →
   m := n↑.edges[j];
   w := n↑.weights[j];
   if m↑.index < x → {ε edge}
      extC := extC + w
   □ m↑.index ≥ x → {δ edge}
      intC := intC + w
   fi;
   j := j + 1
od;
n↑.intC, n↑.extC, n↑.difC := intC, extC, extC - intC
```

Algorithm 7.15: "`Set costs for n`".

However, if `x` < `y` (*C* is not empty) and `maxI` ≥ `y` (`G[maxI]` is associated to a vertex that is not in *C*), `G[maxI]` is associated to a vertex that was erroneously removed from boundary *C* by the swap. `G[maxI]` must therefore be put back in *C* by swapping it with `G[y]` and incrementing `y`.

Algorithm 7.16 describes "`Move vertex from` $(A \setminus X)$ `to` X".

Update Costs and *C*

Only the costs of the vertices adjacent to the last vertex *v* moved to *X* may be updated (the costs associated to the vertices that are not adjacent to *v* are not affected by the move of *v*). After the execution of command "`Move vertex from` $(A \setminus X)$ `to` X", vertex *v* is associated to the entry `G[x-1]`.

The costs associated to a vertex *w* adjacent to *v* need to be updated only if *w* is in $(A \setminus X)$. If *w* is in *X* or *B*, its associated costs will not be used later in the execution of the vertex migration algorithm. The costs of `G[x-1]↑.edges[j]↑` with 0 ≤ `j` < `G[x-1]↑.nEdges` are therefore updated only if `G[x-1]↑.edges[j]↑.index` is greater or equal to `x`.

After the costs of `G[x-1]↑.edges[j]↑` have been updated, the external cost `G[x-1]↑.edges[j]↑.extC` may have changed. Let `ext` be equal to `G[x-1]↑.edges[j]↑.extC` and `ind` be equal to `G[x-1]↑.edges[j]↑.index`. If the costs of `G[x-1]↑.edges[j]↑` have been updated, `ind` is greater or equal to `x` and lower

```
i := x;
if x = y → s := gLen {C is empty}
□ x ≠ y → s := y
fi;
maxD := G[i]↑.difC; maxI := i; i := i + 1;
do i < s →
   c := G[i]↑.difC;
   if c ≤ maxD → skip
   □ c > maxD →
      maxD := c; maxI := i
   if;
   i := i + 1
od;
swap(G, maxI, x);
if x < y and maxI ≥ y → {G[maxI] must be put back in C}
   swap(G, maxI, y); y := y+1
□ x = y or maxI < y → skip
fi;
x := x + 1
```

Algorithm 7.16: "Move vertex from $(A \setminus X)$ to X".

than gLen. The vertex associated to G[x-1]↑.edges[j]↑ may:

- stay in C if ext > 0 **and** ind < y,
- be added to C if ext > 0 **and** ind ≥ y,
- be removed from C if ext = 0 **and** ind < y,
- stay out of C if ext = 0 **and** ind ≥ y.

Algorithm 7.17 describes "Update costs and C" (see Algorithm 7.13). Command "Set costs for n" can be used only if n is associated to a vertex from A. It is the case here because x ≤ n↑.index < gLen and the associated vertex is therefore in $(A \setminus X) \subseteq A$.

```
v := G[x - 1]; {Last vertex added to X}
j := 0;
do j < v↑.nEdges →
   n := v↑.edges[j];
   if n↑.index < x → skip {Vertex from B or X}
   □ n↑.index ≥ x →
      "Set costs for n";
      ext := n↑.extC; ind := n↑.index;
      if ext > 0 and ind < y → skip {stay in C}
      □ ext > 0 and ind ≥ y → {add to C}
         swap(G, index, y);
         y := y + 1
      □ ext = 0 and ind < y → {remove from C}
         swap(G, index, y - 1);
         y := y - 1
      □ ext = 0 and ind ≥ y → skip {stay out of C}
      fi
   fi;
   j := j + 1
od
```

Algorithm 7.17: "Update costs and C".

7.4 Load Balancing with an Adapted Tree Walking Algorithm

The *Tree Walking Algorithm* (TWA) was designed by Shu and Wu [70] in the context of dynamic scheduling on distributed memory computers. It can be used during balancing phase instead of a diffusive scheme-based method. It is able to accurately balance load using global information and remains scalable. It assumes that processors are organized as a tree, have all the same computational power and that scheduled tasks require the same execution time.

In our context, the work load is influenced by the number of sublattices associated to a given computer but also by the computational power of the computer (homogeneous computational power is not assumed). Therefore, the original TWA has to be slightly modified: instead of balancing the number of tasks associated to a computer, we balance the work load caused by a given number of sublattices associated to a computer. We assume that computers are organized as a tree (i.e. each node is a computer) called *Computers Tree* (CT).

Let n be a computer of the CT. T_n represents the computers subtree having n as root. Let c_n be the Contextual Computational Power (CCP, see Section 5.5) associated to computer n and s_n the number of sublattices associated to computer n.

Following values can be calculated for each computer n of the CT:

- The aggregated CCP K_n of T_n: $K_n = \sum_{m \in T_n} c_m$
- The aggregated number of sublattices Z_n of T_n: $Z_n = \sum_{m \in T_n} s_m$

With these values, the *average load* $\bar{L} = \frac{Z_r}{K_r}$ can be computed where r is the root of the CT.

The *quota* q_n of a computer n is then given by $\bar{L} \times c_n$. This value represents the ideal number of sublattices that should be associated to computer n.

An *aggregated quota* H_n can be associated to each computer n: $H_n = \sum_{m \in T_n} q_n$. It gives the number of sublattices that should be associated to the set of computers in subtree T_n.

Aggregated quota are used to schedule work migration in case of load imbalance:

1. Computer n waits for supplementary sublattices from:
   - its parent if $Z_n < H_n$,

- its child m if $Z_m > H_m$.

2. Computer n sends:

    - $(Z_n - H_n)$ sublattices to its parent if $Z_n > H_n$,
    - $(H_m - Z_m)$ sublattices to its child m if $Z_m < H_m$.

If $(Z_n = H_n)$ no sublattices have to be migrated to or from the parent. In the same way, n does not migrate sublattices to or from its child m if $(Z_m = H_m)$.

Shu and Wu have proved that this algorithm achieves good load balance and minimizes work migration and communications during balancing [70].

Average load, quota and aggregated quota are real numbers. In the same way than with diffusive scheme (see Section 7.2.1), there is a rounding issue to solve: the number of sublattices to migrate must be a natural.

7.4.1 Migration Scheduling and Rounding Issues

If a computer must send and receive sublattices, it first receives all awaited sublattices. Let z_n be the number of sublattices associated to computer n after it received all awaited sublattices.

Let x be the number of sublattices a computer n must send to a neighbor ($B_n - C_n$ if sending to parent, $C_m - B_m$ if sending to child m). If n sends $\lfloor x \rfloor$ sublattices, n may be overloaded after it sent all sublattices. If n sends $\lceil x \rceil$ sublattices, n may overload its neighbor. If n rounds x (apply ceiling function if $x \geq \lfloor x \rfloor + 0.5$, apply floor function if $x < \lfloor x \rfloor + 0.5$), one of the above issues may still arise.

If the calculated number of sublattices to send to a neighbor is x, $\lfloor x \rfloor + y$ sublattices could actually be sent where y is equal to 0 or 1. The value of y is decided such as, after all sublattice have been sent, n and its neighbors are as little overloaded as possible. Algorithms 7.18, 7.19 and 7.20 describe the commands executed by a computer n in order to avoid as much as possible the issues presented in previous paragraph when sending sublattices.

`M` is the number of children that computer n has in the CT. We suppose that the children of n are ordered, it is therefore possible to refer to the i^{th} child of computer n with $0 \leq i < M$. `nSubs` represents the number of sublattices associated to computer n and is initialized with z_n. `quota` is the quota q_n. `nErr` is the number of additional sublattices that can be sent to neighbors in order to decrease the additional load on computer n.

The `neighId` field of type `Entry` is lesser than 0 if the entry is associated to the parent. Otherwise, it gives the order number of a child (an integer between $0 \leq i < M$). `toSend` represents the actual number of sublattices that have to be sent to a neighbor. `error` is a real number between 0 and 1 excluded. The higher it is, the lesser the associated neighbor is overloaded if an additional sublattice is sent to it.

`B` is sorted in decreasing order regarding the `error` field of its entries. This way, additional sublattices are sent in priority to neighbors that will be less overloaded if an additional sublattice is sent to them.

```
type
   Entry = record
      neighId, toSend :  integer;
      error :  real;
   end
var
   B : array[0..M] of Entry;
   nErr, quota :  integer
begin
   "Initialize B, nErr and nSubs";
   "Sort B in decreasing order regarding error";
   i := 0;
   do nSubs > quota and nErr > 0 →
      B[i].toSend := B[i].toSend + 1;
      nSubs := nSubs - 1;
      nErr := nErr - 1;
      i := i + 1
   od;
   "Send sublattices"
end
```

Algorithm 7.18: Sending of sublattices to neighbors.

The commands `"Send B[0].toSend sublattices to parent"` and `"Send B[i].toSend sublattices to i-1 th child"` from Algorithm 7.20 implement the migration phase described in Section 7.2.2.

```
nErr := 0; nSubs := "Number of sublattices associated to computer";
x := "Number of sublattices to send to parent";
B[0].neighId := -1;
if x = 0 →
   B[0] := 0; B[0].error := 0
□ x ≠ 0 →
   B[0] := ⌊x⌋; B[0].error := x - B[0].toSend;
   nSubs := nSubs - 1; nErr := nErr + 1
fi
i := 0;
do i < M →
   x := "Number of sublattices to send to i th child";
   B[i+1].neighId := i;
   if x = 0 →
      B[i+1] := 0; B[i+1].error := 0
   □ x ≠ 0 →
      B[i+1] := ⌊x⌋; B[i+1].error := x - B[i+1].toSend;
      nSubs := nSubs - 1; nErr := nErr + 1
   fi;
   i := i + 1
od
```

Algorithm 7.19: "Initialize B, nErr and nSubs".

```
"Send B[0].toSend sublattices to parent";
i := 1;
do i ≤ M →
   "Send B[i].toSend sublattices to i-1 th child";
   i := i + 1
od
```

Algorithm 7.20: "Send sublattices".

7.4.2 TWA Distributed Implementation

The TWA can easily be implemented in a distributed way with computers only sending messages to their parent or their children in the CT:

1. Computers calculate K_n, Z_n through a bottom-up propagation of these values: $K_n = c_n + \sum_{m \in S_n} K_m$ and $Z_n = s_n + \sum_{m \in S_n} Z_m$ where S_n is the set of children of computer n. Note that for leaves, this set is empty and K_n and Z_n can directly be computed and propagated to parent.

2. Average load $\bar{L}$ is computed and broadcasted by the root computer to all computers using a top-down propagation in the CT. Each computer can then compute its quota and aggregated quotas for itself and for its children. Finally, work migration scheduling can be computed.

Unlike K_n and Z_n, H_n does not need an additional bottom-up propagation of messages because K_n value is known for all children of a computer (due to first bottom-up propagation) and $H_n = \bar{L} \times K_n$.

7.5 Dynamic Load Balancing Integration in LaBoGrid

This section is about the actual implementation of the methods presented in this chapter in LaBoGrid.

7.5.1 Computer Tree

The CT is built and maintained by the Controller. It is updated each time a new DA registers to the Controller (a new node is added) or when one or several failures are detected (nodes are removed).

The Controller maintains a representation of the complete CT. This representation allows him to send its adjacency information in the CT (parent and children) to any DA. The CT is represented by a height balanced binary tree. This ensures an optimal execution of the TWA.

7.5.2 Dynamic Load Balancing Implementation

Section 7.3.2 suggests that only partial sublattices graphs are needed to apply the migration algorithm described in Section 7.3.3. Partial mapping tables are therefore not needed during migration phase and can be calculated once for all after all migrations are done.

The following implementation of dynamic load balancing in LaBoGrid can therefore be used:

1. When dynamic load balancing process is triggered, our adapted TWA distributed algorithm is executed and each DA knows how many sublattices it should send or receive to or from its neighbors in the CT.

2. Each DA sends its partial sublattices graph to neighbors that should send sublattices to it.

3. When the migration process is finished for a DA, it sends its partial sublattices graph to the Controller which updates the partitions and mapping table (the partitions table provides the set of sublattices associated to a given DA and the mapping table provides the DA associated to a given sublattice, see Section 5.3).

4. When the migration process is finished for all DAs, the Controller updates the partial mapping tables of all DAs. Both partial sublattices graph and partial mapping table are up-to-date for all DAs and can be used on next simulation restart.

The actual data migration i.e. the migration of the state of the sublattices only occurs when a simulation (re)starts. The migration phase only implies the "exchange" of sublattices graph's vertices without the transfer of the associated data (sublattice's state).

In case of failure of a DA during migration phase, the partitions and mapping tables maintained by the Controller become inconsistent with the data from the partial sublattices graphs of remaining DAs.

To avoid this situation, the Controller duplicates the partitions and mapping tables before the load balancing process is triggered. Only one copy is updated during migration phase. Thus, in case of failure, the original partitions and mapping tables can be reused. The sublattices of the failed DA are then assigned to another DA and partitions and mapping tables updated accordingly. Partial sublattices graphs and partial mapping tables have to be updated as well.

When triggering a load balancing process, the Controller associates a sequence number to the process. This sequence number can be used to dismiss obsolete messages sent by DAs to update partitions and mapping tables.

7.5.3 Load Balancing Triggering

The dynamic load balancing process must be triggered when application or resource graph changes. Three events can cause resource graph changes and therefore trigger a dynamic load balancing process:

- a new simulation,
- a failure detection,
- a new DA registration.

The execution of a new simulation requires an initial distribution of a new sublattices graph on a resource graph because its weights have potentially changed (see Section 5.5).

When a failure is detected (i.e. a DA is down) or a new DA has registered, the DA is removed or added to CT. The CT is then restructured and new adjacency information is sent to DAs. The load balancing process must then be triggered because the resource graph has changed.

In case of failure, the sublattices that were associated to a DA that failed must be attributed to other DAs. In order to avoid to overload some DAs, this attribution should be computed by the dynamic load balancing process. After CT restructuring, the sublattices from the DA that failed are associated to the DA executed by the root of the CT. The load balancing process is then triggered in order to balance the load.

7.5.4 Initial Distribution of Sublattices

Before a new simulation can start, sublattices must be distributed among DAs. This initial distribution can be addressed with tools described in Chapter 5. Dynamic load balancing methods are only used during simulation execution. Another solution is to associate all sublattices to CT root computer and let the adapted TWA method distribute the sublattices.

7.6 Results

In this section, our adapted TWA mapping method is compared to a static mapper considered in Section 5.6 regarding mapping quality (i.e. the execution time of an LB simulation is compared when using the different mappers).

These results will show that:

1. the adapted TWA method achieves mapping qualities that are comparable to those obtained with heterogeneous SCOTCH (see Section 5.6) i.e. comparable execution times are obtained when executing distributed LB simulations with mappings produced by the adapted TWA and heterogeneous SCOTCH,
2. the TWA balancing scheme presented in Section 7.4 strongly reduces the number of exchanged messages when compared to a classical diffusion scheme (see Section 7.2.1),
3. the number of migrated sublattices in case of resource graph changes (failure or new available computer) is greatly reduced by using incremental mapping.

7.6.1 LB Simulation Execution Time using the Adapted TWA

The same setup as in Section 5.6 is used: a distributed LB simulation with the MRT collision operator and a lattice of size $(176, 176, 176)$ is executed on a cluster of 17 computers. The cluster is composed of 8 computers of type A (Pentium IV, see Section 5.6), 8 of type B (Celeron) and 1 of type C (Xeon). The mapping of 32, 64, 128, 192 and 256 sublattices is considered.

The adapted TWA is compared to the heterogeneous SCOTCH mapper which gives the best results when compared to other static mappers (see Section 5.6).

Figure 7.4 shows the execution time of distributed LB simulations using the mappings produced by heterogeneous SCOTCH (HeScotch) and the adapted TWA (ATWA).

We can see that ATWA leads to execution times comparable to the ones obtained with mappings produced by heterogeneous SCOTCH with 128 sublattices and more. ATWA performs better with smaller numbers of sublattices.

These results can be explained by the fact that mappers produce partitions of the application graph with small imbalances caused by the fact that the number of vertices in a partition is not necessarily optimal regarding the weight of the associated resource graph vertex. The problem in the context of distributed LB simulations is that a sublattice represents a load that is not negligible and therefore, small imbalances can lead to a significant increase in execution time.

The adapted TWA mapper produces small partitions imbalances caused by rounding issues (see Section 7.4.1). However, these imbalances are smaller than the imbalances produced by heterogeneous SCOTCH.

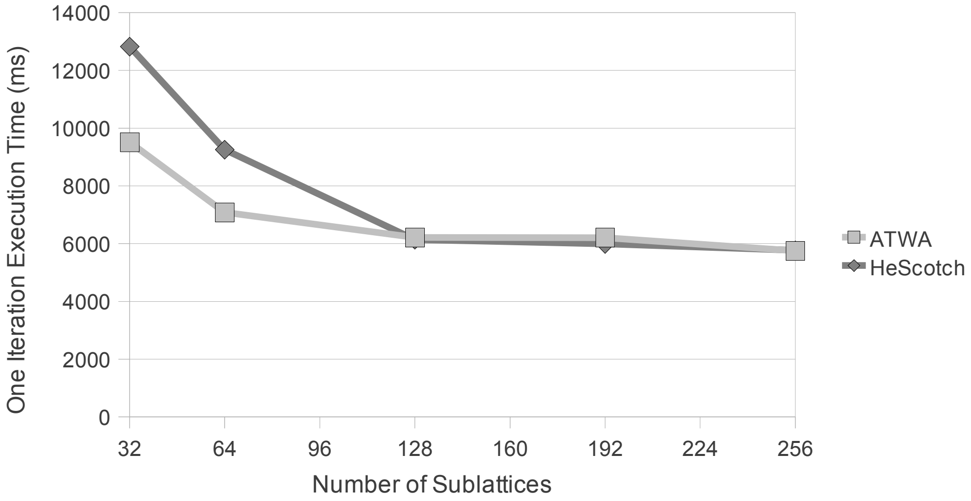

Figure 7.4: LB simulation execution times obtained when using increasing numbers of sublattices and two different mappers.

7.6.2 Exchanged Messages During Balancing Phase

During the balancing phase of local improvements methods, each computer exchanges messages with its neighbors. With a diffusion scheme, each computer sends load information to its neighbors in the migration graph. In the context of the TWA, each computer receives a message from its parent and sends a message to its children during top-down propagation and receives a message from each child and sends a message to its parent during bottom-up propagation. We define a *round* by the fact that all computers have sent one message to each neighbor and received one message from each neighbor.

In the context of the TWA, there are 1.5 rounds: a top-down or bottom-up propagation can be considered as a half round and three propagation passes are needed (a top-down propagation that triggers the bottom-up propagation that sets aggregated variables and the final top-down propagation) to complete the balancing phase.

With diffusion schemes, the number of rounds is not known in advance. It depends on the initial conditions (in our context, the number of sublattices associated to each computer), the number of computers participating to the balancing process and the way they are connected.

The number of rounds required by the diffusion scheme described in Sec-

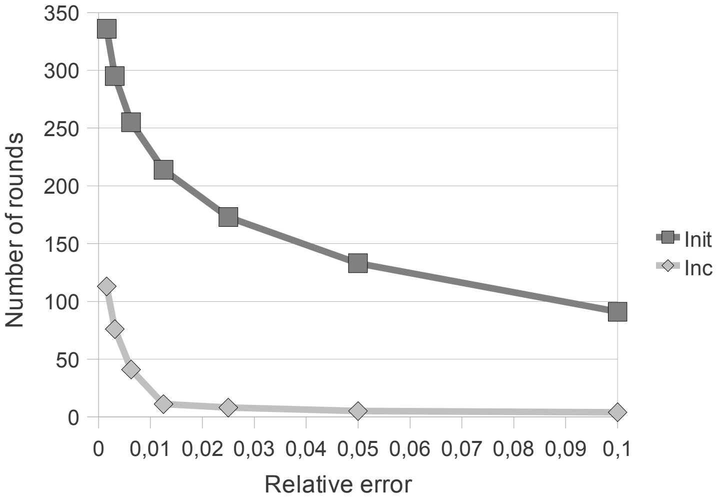

Figure 7.5: Number of rounds of diffusion scheme with two initial conditions.

tion 7.2.1 was evaluated when distributing 256 sublattices on a computer tree made of 54 computers: 27 of type A (see Section 5.4) and 27 of type B. Each computer has 3 neighbors: its parent and its two children. Two initial conditions where used:

1. all sublattices are associated to the root computer,
2. sublattices distribution comes from a previous TWA mapping.

First initial condition can be considered as a bad case with a high number of rounds for the diffusion scheme and the second initial condition as a good case with a low number of rounds.

When using a diffusion scheme, a stop criterion must be defined to decide when the variables w_i^t (and therefore n_i^t) have converged. All w_i^t variables should converge towards the same value, we therefore use following stop criterion:

$$\frac{w_{max}^t - w_{min}^t}{w_{max}^t} < \varepsilon$$

where $w_{max}^t = \max_i w_i^t$ and $w_{min}^t = \min_i w_i^t$. ε is called the relative error on w_i^t values.

Figure 7.5 shows the number of rounds required by the diffusion scheme in function of the relative error ε. The curve labeled “init” (respectively “inc”) gives the number of rounds when using initial condition 1 (respectively 2). In the best case (initial condition 2 and $\varepsilon = 0.1$), the diffusion scheme still requires 4 rounds to converge. However, with initial condition 1 and $\varepsilon = 0.1, 91$ rounds are required.

In any case, the diffusion scheme requires a number of rounds that is potentially far higher than TWA. In addition, the value of ε has to be chosen such as the number of rounds is minimized while the load balancing "quality" (i.e. the work load homogeneity) is maximized. The TWA does not require to solve this kind of optimization problem.

7.6.3 Sublattices Migrations

To observe sublattices migration in case of resource graph topology changes, a sequence of resource graphs is generated. A given number of computers are removed and then added to the resource graph. A given sublattices graph is mapped on each resource graph. The number of migrated sublattices between two subsequent mappings is evaluated.

This experiment is described by Algorithm 7.21. `M` computers are removed from and added to the resource graph at each iteration. `N` mappings are produced. The number of migrated sublattices is therefore evaluated `N-1` times.

The number of migrated sublattices between two subsequent mappings computed by command "`Evaluate the number of migrated sublattices between M1 and M2`" is the sum of the number of sublattice's states each computer must download after a new mapping has been produced. Let S_1^c (respectively S_2^c) be the set of sublattices associated to a computer c by mapping `M1` (respectively `M2`). The set $(S_1^c \setminus S_2^c)$ contains the sublattices that are not available on c and have to be downloaded from another computer in order to resume a simulation using the mapping `M1`. The total number of migrated sublattices between two subsequent mappings `M1` and `M2` is therefore given by:

$$\sum_{c \in R} |S_1^c \setminus S_2^c|$$

Note that S_2^c may be empty because c was not in `R` when mapping `M2` was produced.

The experiment is executed using PaGrid, heterogeneous SCOTCH and the adapted TWA mappers. The adapted TWA mapper is configured to produce incremental mappings (using the mapping on previous resource graph to generate the mapping on current resource graph) or classical mappings (previous mapping is ignored).

```
"Initialize resource graph R";
"Initialize sublattices graph A";
"Generate mapping of A onto R and store result in M1";
i := 1;
do i < N →
   if $(\mathtt{i} \bmod 2) \neq 0$ →
      "Remove M computers from R and store them into L"
   □ $(\mathtt{i} \bmod 2) = 0$ →
      "Add the M computers from L to R";
      "Clear L"
   fi;
   "Store mapping M1 in M2";
   "Generate mapping of A onto R and store result in M1";
   "Evaluate the number of migrated sublattices between M1 and M2";
   i := i + 1
od
```

Algorithm 7.21: Sublattices migrations experiment.

Figure 7.6 shows the number of migrated sublattices when using the three different mappers, TWA being configured to produce incremental (TWA) or classical (TWA-0) mappings. A sublattices graph of 256 sublattices is mapped on 10 resource graphs ($\mathtt{N} = 10$). The number of migrated sublattices is therefore evaluated 9 times. The sequence of resource graphs is produced such as R contains initially 54 computers and and M is equal to 1.

We can observe that even small changes in resource graph topology lead to nearly all sublattices being migrated between two subsequent mappings when previous mapping is ignored.

4-6 sublattices are associated to each computer of the resource graph. This means that at least 4-6 sublattices have to migrate when a computer is removed from the resource graph. Additional migrations are triggered to balance the work load. In Figure 7.6, we observe that the adapted TWA leads to 20-30 migrations when one computer is removed from or added to the resource graph.

Figure 7.7 shows the number of migrated sublattices with $\mathtt{M} = 1$, $\mathtt{M} = 5$, $\mathtt{M} = 10$, $\mathtt{M} = 25$ and $\mathtt{M} = 40$ using the adapted TWA mapper configured to produce incremental mappings. A sequence of 100 resource graphs is produced. The initial resource graph contains 54 computers. 256 sublattices are mapped onto

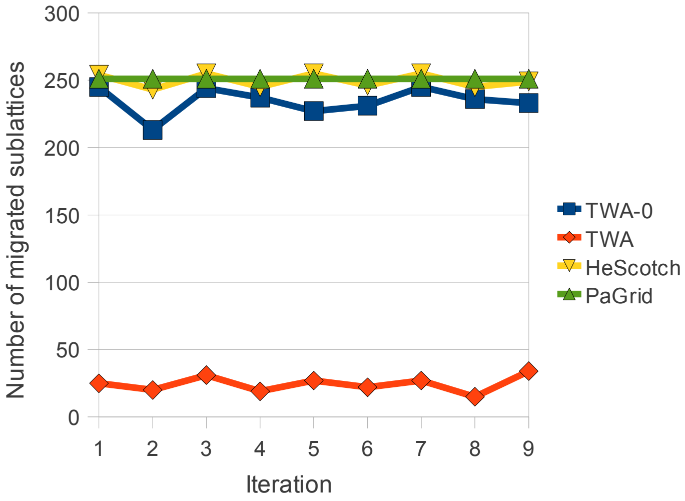

Figure 7.6: Number of migrated sublattices when mapping a sublattices graph containing 256 sublattices on a sequence of 10 resource graphs using four different mappers.

the resource graphs. In all cases, the number of migrated sublattices varies but remains around a central value.

Figure 7.8 shows the mean and standard deviation (SD) of the number of migrated sublattices for each curve of Figure 7.7 in function of $\mathtt{M}$. We observe that the SD remains bounded.

Note that even in the case with $\mathtt{M} = 25$ where the size of the resource graph is almost divided or multiplied by 2 between two subsequent resource graphs, the number of migrated sublattices remains smaller than the case where previous mapping is not taken into account.

7.7 Conclusion

In this chapter, we proposed a dynamic load balancing method inspired from local improvements methods that uses an adapted TWA [70]. The adapted TWA takes computer powers heterogeneity into account during balancing phase. KL-based refinements [48] on graph boundaries are implemented for the migration phase.

Regarding partitioning quality, the method was compared to a static mapper (heterogeneous SCOTCH [62]) introduced in Chapter 5. We observed that map-

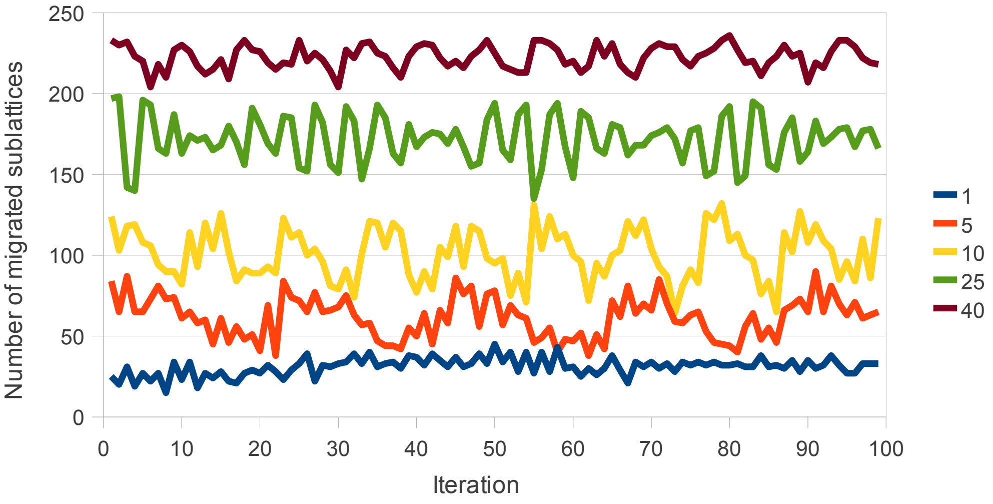

Figure 7.7: Number of migrated sublattices for a sequence of resource graphs using TWA and increasing the number of computers added to or removed from resource graphs.

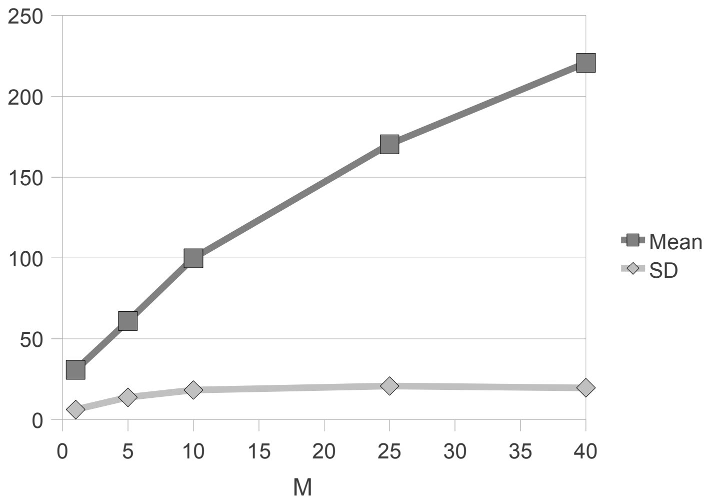

Figure 7.8: Mean and Standard Deviation (SD) of Figure 7.7's curves.

pings produced with our method lead to comparable execution times with high ($\geq$ 128) numbers of sublattices and better execution times with small (< 128) numbers of sublattices. This is due to the fact that our method produces smaller partition imbalances than heterogeneous SCOTCH.

The adapted TWA was compared to diffusion schemes regarding the number of exchanged messages before convergence is reached to find the optimal work load of computers. In all cases, our approach requires less messages (diffusion scheme requires from 2 to 200 times more exchanges before convergence than TWA for a cluster of 54 computers) to be exchanged in order to determine the optimal work load.

Finally, the number of sublattices migrated in case of resource graph topology changes was measured using different mappers including the adapted TWA. We showed that when previous mappings are ignored, almost all sublattices have to migrate. Our method clearly reduces the number of migrated sublattices while maintaining load balance. We observed that the number of migrations varies over time but remains in a bounded interval.

Chapter 8

Robust Distributed Control

8.1 Introduction

LaBoGrid is able to execute efficiently (see Chapters 3, 5 and 7) distributed LB simulations. In case of failure of one or several worker computers i.e. computers running each exactly one Distributed Agent (DA, see Section 4.4), simulations can continue thanks to the distributed checkpoint/restart mechanism presented in Chapter 6.

However, if the Controller is not accessible anymore for some reason, the execution of LaBoGrid cannot continue and the result of the currently executed simulation cannot be retrieved. The Controller is therefore a single point of failure.

To avoid this robustness problem, two approaches can be used:

1. the Controller disappears and the Controller Agent (CA, see Section 4.4) is executed by a DA which therefore "controls" the others. In case of failure of this DA, the CA is instantiated on another DA and continues its execution. The state of the CA must therefore be saved in order to be restored on another DA. A leader election problem must be solved by the DAs in order to decide which DA must instantiate the CA.

2. the Controller's implementation is distributed. This implementation must be fault-tolerant.

In addition to the robustness problem, the centralized architecture of the Controller implies a scalability problem: it is a potential bottleneck during its control operations.

We chose to combine the approaches presented above: the CA becomes a component that can be instantiated by any DA. However, to improve scalability and reduce CA migration cost, its implementation is mostly distributed.

Generic services partially or completely implemented by the Controller and used during a distributed LB simulation can be identified:

- DAs identification (association of a unique identifier to each DA),
- broadcasting,
- barrier synchronization (synchronization of all processes),
- distributed file system (see Section 6.6),
- dynamic load balancing (see Chapter 7).

Currently, these services are fully or partially implemented in a centralized way:

- Each DA initially connects to the controller to obtain a unique identifier,
- A message to broadcast is sent by the controller to all DAs.
- Barrier synchronization consists of all DAs sending a "reached" message to the Controller which releases the DAs when all "reached" messages are received.
- The distributed file system uses a central file location table maintained by the controller called Global File Location Table (gFLT, see Section 6.6.1).
- The dynamic load balancing method is mostly distributed but requires the construction of the Computer Tree (CT, see Section 7.5) which is done by the Controller.

In this chapter, we show how these services can be implemented in a distributed and robust way.

In addition to above services, a leader election mechanism must be implemented to choose the DA that will instantiate the CA.

8.1.1 Chapter Outline

Section 8.2 presents how DAs can identify themselves without an initial connection to the controller.

Section 8.3 describes tree-based algorithms to implement broadcasting, leader election and barrier synchronization services in a distributed way.

Section 8.4 introduces a robust and distributed system to represent the gFLT.

Section 8.5 describes a robust way to implement the tree-based algorithms of Section 8.3.

Section 8.6 briefly explains how distributed LB simulations uses the services.

Section 8.7 compares the execution times when centralized and distributed implementations of services are used.

Finally, Section 8.8 concludes this chapter.

8.2 Distributed Agents Identification

Initially, DAs connect to the Controller and receive a unique identifier (an integer). This integer can directly be used to identify partitions in the context of static (see Chapter 5) and dynamic (see Chapter 7) load balancing. Moreover, DA's initial registration signals a resource graph change and therefore triggers a load balancing process.

DAs identification by the Controller could be ignored: the TCP/IP address associated to a DA (i.e. the address on which the DA accepts connections) can be used as identification.

However, in Section 6.3, the detection of micro-interruptions of DAs is discussed and implies the comparison of the destination field of a message (which contains the identifier of a DA) to the identifier of the receiver DA. If the TCP/IP address is used as identifier, micro-interruptions are not detected anymore (two different DAs executed consecutively on the same computer have the same identifier).

The combination of the TCP/IP address of the computer executing the DA and a time stamp indicating when the DA was instantiated solves this problem[1]. In addition, this identifier can be generated locally by each DA.

[1] The only required property for a DA's clock is that the time stamp generated upon DA's instantiation is different of the time stamp generated before a micro-interruption on the same computer.

8.3 Tree-based Broadcasting, Leader Election and Barrier Synchronization

Broadcasting and barrier synchronization services can be implemented in a distributed way by organizing the computers executing the DAs into a rooted tree.

8.3.1 Leader Election

The leader election problem in dynamic topologies is generally reduced to the problem of finding a spanning tree [17, 54] because most of the distributed algorithms for finding a spanning tree produce a rooted spanning tree. The root of the tree is then chosen as the leader. If the rooted tree is available by construction, the leader election problem is trivially solved.

8.3.2 Broadcast

To efficiently broadcast a message to a large number of entities, the global number of exchanged messages and the actual broadcast time (the time between the first message is sent and the last entity receives it) should be minimized.

In a fully connected network, a centralized approach (the source sends directly the message to all other overlay members) can be used and minimizes the number of sent messages (if there are N entities, $N-1$ messages have to be sent). However, it is not scalable: when the number of entities becomes large (thousands up to millions of entities), each entity must know the address of all others which is clearly a problem in terms of memory usage. In addition, in case of topology change (an entity "appears" or "disappears"), all entities have to update their address table.

Broadcast trees also minimize the number of exchanged messages during broadcast and limit the broadcasting time: the message to broadcast traverses $O(\log(N))$ overlay members before any member is reached if the tree is balanced. Also, the size of the routing table of a tree member depends on the number of children of the member: each member maintains a link to its parent and its children. If each tree member has at most k children, the routing tables contain a number of entries that is at most $k+1$. k is generally much lower than N.

8.3.3 Barrier Synchronization

The scalable implementation of software barrier synchronization has been studied for large scale shared-memory multi-processor systems [57]. In particular, tree based methods [40] are scalable, minimize the number of transactions between processes and can easily be adapted to our context.

Let P be a set of processes organized into a rooted tree. In our context, the processes are executed by different computers. Each process $p \in P$ is the root of a subtree $S(p)$ and has at most K children (for example, Gupta et Hill [40] chose $K = 2$). A leaf process f has no child and is the root of a subtree $S(f)$ containing only one node, f. The root of the complete processes tree is noted r. The whole processes tree is therefore noted $S(r)$. If $|P| = 1$, the root is a leaf as well.

When all processes of a subtree $S(p)$ have reached the barrier, a message is sent by p to its parent. When the root process r receives this message from all its children or r has no child, all processes of P have reached the barrier and a message signaling that all processes can continue their execution is broadcasted by r to all processes using the tree structure.

Algorithm 8.1 describes the function `barrierWait` causing a process to wait until all other processes have reached the barrier. The processes exchange two types of messages: `REACHED` and `ALLREACHED`. The `REACHED` message is sent by a process p to its parent when all the processes of $S(p)$ have reached the barrier. The `ALLREACHED` message is broadcasted by root process when all processes have reached the barrier and causes all waiting processes to continue their execution.

8.4 Distribution of the Global File Location Table

The gFLT used by the distributed file system (see Section 6.6.1) provides, given a file unique identifier (FUID), the list of addresses representing the computers hosting the associated file. This table must be accessible by any computer running the distributed file system, hence the idea of using a generic, robust and scalable table service.

Distributed Hash Tables (DHT) [11, 13, 46, 56, 66, 67, 74, 80] are robust and scalable Peer-to-Peer (P2P) systems that act like a distributed structure allowing at least the insertion and the retrieval of data identified by a key:

- (*v-key*, *value*) entries can be inserted in a DHT,
- a *value* is retrieved given its associated *v-key*.

```
procedure barrierWait();
begin
   if "process has no child" → skip
   □ "process has at least one child" →
      "Wait for all children to have sent a REACHED message";
   fi;
   if "process is the root" →
      "Broadcast ALLREACHED message"
   □ "process is not the root" →
      "Send a REACHED message to parent process";
      "Wait for the reception of ALLREACHED message";
   fi
end
```

Algorithm 8.1: Barrier synchronization implementation.

An example of *v-key* is the FUID and an example of *value* is a list of addresses. The gFLT used by the distributed file system can therefore be implemented using a DHT. This would lead to a fully distributed implementation of the distributed file system.

In addition to insertion and retrieval, removal and update are required by our distributed file system implementation. When a file is removed from the DFS, its associated entry in the gFLT must be removed. When a file is replicated to several computers, new addresses need to be added to the associated list of addresses in the gFLT.

Section 8.4.1 presents the general principles common to all DHT systems. A particular DHT system is presented in Section 8.4.2. Existing DHT systems are compared in Section 8.4.3 and our choice of one of them is motivated. Finally, Section 8.4.4 briefly describes how the implementation of the chosen DHT system had to be adapted in order to be able to implement the gFLT.

8.4.1 General Principles of DHTs

In a centralized table service, a computer called *table server* maintains a *data table* containing key-value entries of the form (*v*-key, *value*). When another computer called *table client* wants to, for example, insert a new entry in the data table, it sends a request to the table server which potentially modifies the data table and sends a result to the table client (in the context of an insertion, if the insertion was successful or not; the definition of a successful insertion depends on the implementation).

In a DHT, the client-server model of the centralized approach is replaced by a P2P model where each computer is both client and server. In particular, the entries of the data table are not centralized anymore by a single computer but distributed among all the computers of the DHT. Each computer therefore maintains a *partial data table* i.e. a data table that does not contain all the entries. When a computer wants, for example, to retrieve a value, it must first search for the computer that might have this entry in its partial data table. It can then send a request to this computer which executes it and sends the result to the requesting computer (in the context of a retrieval, the *value* associated to the given v-key or nothing if there is no entry associated to the given v-key in the partial data table).

The member of a DHT is called a *DHT peer*. Each DHT peer has a unique identifier that we call p-key (for example, an IP address). The p-key of a DHT peer can be used to directly send a message to the peer. Let V be the v-keys space and P be the p-keys space. All DHT systems rely on a hash function h which associates a h-key to any v-key or p-key. h is defined as follows:

$$h : P \cup V \to H$$

where H is the space of h-keys. A partition A_i of H is associated to each DHT peer. Let $D \subseteq P$ be the set of all DHT peers of the DHT. The following property must be true:

$$H = \bigcup_{i \in D} A_i$$

A DHT peer i is *responsible* of all v-keys k such as $h(k) \in A_i$. A request associated to a particular v-key k must therefore be sent to the DHT peer that is responsible of k.

In order to find the DHT peer responsible of a particular v-key k, a routing table containing key-value entries of the form (h-key, *address*) could be associated to each DHT peer. There is an entry for each DHT peer of the DHT. A search algorithm could be defined to find the entry of the peer responsible of k in function of $h(k)$.

DHTs have been developed in order to be used at the Internet scale i.e. in a context where the potential number of DHT peers is very large ($|D| > 10^6$). The approach presented in previous paragraph is clearly not scalable because the set D of DHT peers may change over time (new DHT peers may join and other leave the DHT) and the maintenance cost of the routing table on each DHT peer may become prohibitive (each time a DHT peer joins or leaves, the routing table associated to all DHT peers must be updated).

In DHT systems, the routing table is distributed by organizing the DHT peers into a *Structured Overlay Network* (SON). A SON is an application-level network built on top of a physical network. A request associated to a particular v-key is forwarded in the SON by a routing algorithm executed on each DHT peer. The routing algorithm uses the *partial routing table* (i.e. a routing table that does not contain all DHT peers) of the DHT peer. The routing algorithm ensures that the request associated to a v-key k eventually reaches the DHT peer responsible of k.

When a DHT peer joins or leaves the SON, a number M of DHT peers have to update their partial routing table with $M \ll |D|$ (M is proportional to the size of the partial routing tables, see next paragraph).

The size of the partial routing table associated to a particular DHT peer should grow slowly in function of the total number of DHT peers $|D|$. This ensures the system remains scalable. Most DHT systems [74, 80, 67, 13, 56] use partial routing tables whose size is $O(\log(|D|))$. Some DHT systems [66] use constant size partial routing tables.

A request potentially traverses several DHT peers before it reaches the responsible DHT peer. The sequence of DHT peers a request traverses is called a *routing path*. The forwarding of a request by a DHT peer to the next DHT peer in the routing path is called a *hop*. In the same way than the size of the routing table, the number of hops needed before a request reaches the responsible DHT peer must grow slowly with the total number of DHT peers $|D|$ for a DHT system to be efficient. In most DHT systems [74, 80, 67, 13, 56], the number of hops before a request reaches the responsible DHT peer is $O(\log(|D|))$.

Another important aspect of DHTs is their robustness: when a DHT peer leaves the DHT unexpectedly, the routing of requests in the SON should continue. This is generally achieved by using a maintenance algorithm executed by each DHT peer that updates the partial routing table of the DHT peer when a DHT peer associated to an entry of the partial routing table unexpectedly leaves.

In addition, the partial data table associated to a DHT peer is replicated to several other DHT peers. This way, if a DHT peer leaves unexpectedly, the (v-key, *value*) entries of its partial data table are still available on another peer.

The principles presented in this section are common to most DHT systems. DHTs generally differ in the way they implement:

- the SON,

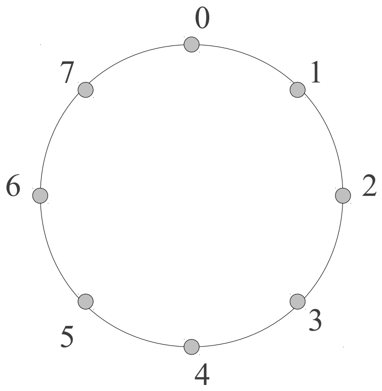

Figure 8.1: Chord identifier circle with $m = 3$.

- the routing algorithm on top of the SON,
- the maintenance algorithm,
- the partial data table replication process.

8.4.2 A Simple Example of DHT: Chord

In order to illustrate the general principles introduced in Section 8.4.1, the DHT system called Chord [73, 74] is presented in this section.

Chord assumes that H is circular and ordered, and that h-keys are represented on m bits. H can therefore be represented by an identifier circle with 2^m possible positions numbered from 0 to $2^m - 1$. An example of Chord identifier circle with $m = 3$ is shown in Figure 8.1. In this case, the hash function h is defined as follows:

$$h : P \cup V \to \{0, 1, 2, 3, 4, 5, 6, 7\}$$

The notation $)a, b]$ represents an interval of the Chord identifier circle where $0 \leq a, b < 2^m$. This interval is defined as follows:

$$)a, b] = \begin{cases}]a..b] & if \quad a \leq b \\ ]a..2^{m-1}] \cup [0..b] & if \quad a > b \end{cases}$$

and contains all the values encountered on the Chord identifier circle when going clockwise from a excluded to b included. For example, in the context of the Chord identifier circle of Figure 8.1, $)1, 4] = \{2, 3, 4\}$ and $)6, 2] = \{7, 0, 1, 2\}$.

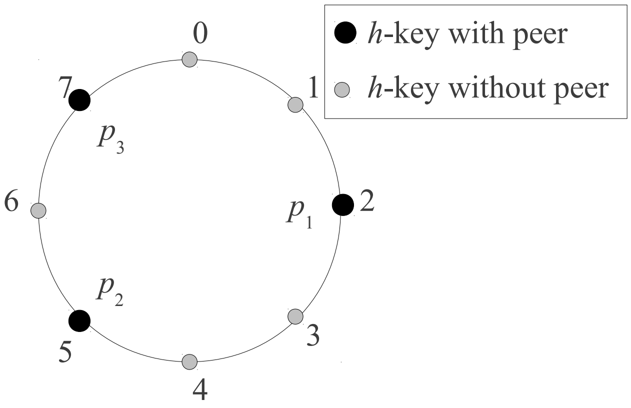

Figure 8.2: Chord identifier circle with $m = 3$ and 3 peers represented.

Let a Chord DHT involve N peers with $N \leq 2^m$ (in general, $N \ll 2^m$). Let p_i be the p-key of a Chord peer with $i = 0, 1, \ldots, N-1$. The order of the p-keys is chosen such as:

$$h(p_0) < h(p_1) < \ldots < h(p_{N-1})$$

We suppose that h never assigns the same h-key to two different p-keys.

The Chord peer with p-key p_i is responsible of v-key k if

$$h(k) \ \in \ )h(p_{(i-1) \bmod 2^m}), h(p_i)].$$

The peer with p-key $p_{(i-1) \bmod 2^m}$ is called the *predecessor* of h-key k and the peer with p-key p_i is called the *successor* of h-key k.

The functions $Succ : H \to P$ and $Prec : H \to P$ provide the p-key of respectively the successor and the predecessor of a given h-key. For example, in the situation presented in previous paragraph:

$$\begin{array}{rcl} Prec(k) & = & p_{(i-1) \bmod 2^m} \\ Succ(k) & = & p_i \end{array}$$

Figure 8.2 shows the same identifier circle as in Figure 8.1 but with three Chord peers represented on it. The three peers have p-keys p_1, p_2 and p_3 such as $h(p_1) = 2$, $h(p_2) = 5$ and $h(p_3) = 7$.

Table 8.1 shows the h-keys each peer of Figure 8.2 is responsible of.

Table 8.2 defines the functions *Succ* and *Prec* for the situation of Figure 8.2.

p-key	h-keys
p_1	$\{0,1,2\}$
p_2	$\{3,4,5\}$
p_3	$\{6,7\}$

Table 8.1: h-keys the peers of Figure 8.2 are responsible of.

k	$Prec(k)$	$Succ(k)$
0	p_3	p_1
1	p_3	p_1
2	p_3	p_1
3	p_1	p_2
4	p_1	p_2
5	p_1	p_2
6	p_2	p_3
7	p_2	p_3

Table 8.2: Definition of *Succ* and *Prec* for the situation of Figure 8.2

Structured Overlay Network

Let q be the p-key associated to a Chord peer. The partial routing table of the peer contains two entries: $Prec(h(q))$ and $Succ(h(q)+1)$. The first entry is the p-key of the predecessor of the peer and the second entry is the p-key of the successor of the peer.

The SON used by Chord can therefore be represented by a double linked ring called the Chord ring. The chord ring associated to the situation illustrated in Figure 8.2 is given in Figure 8.3.

Routing algorithm

The Chord ring is enough for the routing of requests towards the peer responsible of a particular h-key: the request is simply forwarded by a peer to its successor until the responsible peer is reached. However, this approach leads to a number of hops that is $O(N)$ where N is the number of peers. In addition, if one peer leaves unexpectedly, the routing of requests is not possible anymore.

Each peer therefore maintains an additional table called *fingers table*. The fingers table contains at most m p-keys. Let q be the p-key associated to a Chord peer and `finger[i]` be the i^{th} entry of the finger table of this peer. The content of

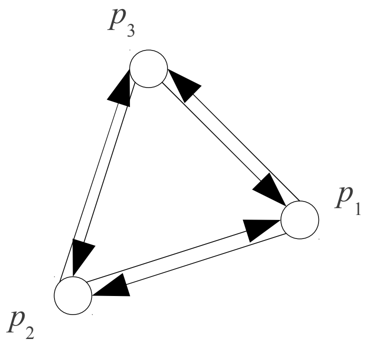

Figure 8.3: Chord ring for the situation illustrated in Figure 8.2.

the finger table is then defined as follows:

$$\texttt{finger[i]} = Succ(h(q)+2^i) \text{ where } 0 \leq i < m.$$

`finger[0]` gives the successor of the peer. The entries of the finger table are not necessarily distinct and equal entries do not need to be stored separately.

When a request is received by a Chord peer, the routing algorithm first tests if the peer is responsible of the v-key w associated to the request. Let $k = h(w)$ and q be the p-key of receiving peer. The peer is responsible of w if $k \in)h(Prec(q)), h(q)]$.

If it is the case, the request can be handled, possibly leading to one of the following operations applied on the partial data table of the peer: the insertion of a new entry, the retrieval of an entry or the removal of an entry.

If the peer is not responsible of k, the routing algorithm forwards the request to the closest predecessor of k it can find in its fingers table.

The routing algorithm executed by a Chord peer is described by Algorithm 8.2.

It has been shown that this routing algorithm leads to a number of hops that is $O(\log(N))$ where N is the number of peers in the Chord ring [73].

Peer Join

Let a peer that has p-key q join the Chord ring. If no peer is present in the ring, q has no predecessor and uses itself as successor. In order to join an existing Chord ring, q must know the p-key of at least one peer that is already in the ring. Let q' be the p-key of a peer that is already in the ring. q has no predecessor and uses

```
w := "$v$-key associated to received request";
k := $h$(w);
if k $\in$ $)h(Prec(p)),h(p)]$ $\rightarrow$
   "Handle received request"
□ k $\notin$ $)h(Prec(p)),h(p)]$ $\rightarrow$
   i := m - 1;
   do i $\geq$ 0 and finger[i] $\notin$ $)h(p),k]$ $\rightarrow$
      i := i - 1
   od;
   "Forward request to finger[i]"
fi
```

Algorithm 8.2: Efficient location of the peer responsible of h-key k.

the peer responsible of key $h(q)$ as its successor. This peer is found by forwarding a request to q' which executes Algorithm 8.2.

After the insertion process, the other peers of the DHT are not yet aware of the new peer. A stabilization algorithm is triggered periodically to update peers' routing tables in order to learn about newly joined peers. The stabilization algorithm also updates routing tables if a peer leaved unexpectedly.

Chord Fault Tolerance

If a peer leaves unexpectedly, remaining peers can loose their knowledge of their successor. In this case, the correctness of the Chord protocol is not guaranteed anymore [73]. In addition, the data entries the peer that leaved was responsible of are not available anymore.

To increase robustness, each Chord peer maintains a list of its r successors in the Chord ring. Its data entries are then replicated on the r successors. All data entries that were inserted into the DHT are therefore likely to be still available even if several peers failed simultaneously.

Chord peers' lists of successors are built by a slightly modified version of the stabilization algorithm.

Chord Features Summary

Chord is a scalable and robust DHT system. Its scalability comes from the fact that:

- the routing algorithm remains efficient even with large numbers of peers (the number of hops before the destination peer is reached is logarithmic in the number of peers in the system),
- the amount of routing information each peer maintains remains small regarding the total number of peers (the routing table of a peer contains a number of entries that is logarithmic in the total number of peers in the system).

The robustness of Chord is achieved through:

- data replication (data entries are replicated on several peers),
- link replication (each peer maintains a list of several successors),
- self-healing algorithms (the stabilization algorithm that updates routing tables in case of peers insertion and/or failures).

8.4.3 Comparison of Existing DHTs

DHT systems can be compared using following criteria [34]:

1. lookup efficiency: The number of hops needed to find the peer responsible of a given h-key.
2. routing table size: The number of entries a peer maintains in its routing table.
3. fault-tolerance: All DHT systems are fault-tolerant but they may differ in the way they achieve fault-tolerance. In particular, used methods can have different maintenance costs in terms of the number of sent messages.
4. implementation availability: to be easily integrated into LaBoGrid, a DHT system should be available in the form of a lightweight Java library.

The following systems are only presented regarding the defined criteria. The interested reader can refer to references for more details. In the following, N represents the number of peers in the DHT.

Chord [73, 74] was shortly presented in the previous section. The number of hops is $O(\log_2(N))$. The routing table contains $O(\log_2(N))$ entries. In case of peer insertion or departure (graceful or not), routing tables are updated by a stabilization algorithm. A Java library is available.

DKS [13, 14] can be seen as a generalization of Chord providing better lookup efficiency but requiring larger routing tables. The number of hops to find a peer is $O(\log_k(N))$ where k is a parameter of the method. There are $(k-1)\log_k(N)$ entries in the routing table of a peer. Overlay maintenance is achieved using two techniques: correction-on-use (COU) and correction-on-change (COC). With correction-on-use, routing table entries are not corrected until they are needed. However, it is assumed that the number of joins and leaves (or fails) is reasonably lower than the number of lookup messages (used to realize correction-on-use). If this assumption does not hold, correction-on-change is used: in case of join, leave or failure, all peers that need to update their routing tables are notified.

Pastry [67] and Tapestry [80] are similar both in underlying concepts and properties. $O(\log_\beta(N))$ hops are needed to find a peer (β is a parameter of the system). Each peer maintains $O(\alpha \log_\beta(N))$ routing entries where α is proportional to β. The routing tables are updated using a deterministic procedure in case of peer insertion or departure. Java libraries are available for both systems. A notable advantage of Pastry and Tapestry over other DHT systems is the minimization of latency in addition to the number of hops. This is achieved by forwarding search messages to peers that are physically close (i.e. geographically close).

Kademlia [56] is based on principles similar to Pastry and is comparable in terms of number of hops and number of entries in the routing table of peers. The maintenance of routing tables is done by analyzing the lookup traffic in order to minimize maintenance costs. A Java library is available. Like Pastry and Tapestry, latency is minimized.

CAN [66] (Content Addressable Network) maps keys onto a d-torus. The number of hops is $(d/4)(N^{(1/d)})$. The number of entries in the routing table does not depend on the number of peers in the system. A stabilization algorithm is used to handle peers failure. No library has been found for this DHT.

P-Grid [11] is a system based on randomized algorithms and reaches performances comparable to most DHTs in terms of number of hops ($O(\log(N))$) with a high probability. The routing table size is $O(\log(N))$. Overlay maintenance and construction are based on random interactions between peers.

DHT system	hops	routing entries	Maintenance	Library
Chord	$O(\log_2(N))$	$O(\log_2(N))$	Stabilization	Yes
DKS	$O(\log_k(N))$	$(k-1)\log_k(N)$	COU+COC	Yes
Pastry, Tapestry	$O(\log_\beta(N))$	$O(\alpha \log_\beta(N))$	Deterministic	Yes
Kademlia	$O(\log_\beta(N))$	$O(\alpha \log_\beta(N))$	Traffic analysis	Yes
CAN	$(d/4)(N^{(1/d)})$	k	Stabilization	No
P-Grid	$O(\log(N))$	$O(\log(N))$	Random	Yes
Koorde	$\frac{O(\log_2(N))}{O(\log_2(\log_2(N)))}$	$O(\log_2(N))$	Stabilization	Yes
Viceroy	$\frac{O(\log_2(N))}{O(\log_2(\log_2(N)))}$	$O(\log_2(N))$	Stabilization	No
DH	$\frac{O(\log_2(N))}{O(\log_2(\log_2(N)))}$	$O(\log_2(N))$	-	No
Ulysses	$\frac{O(\log_2(N))}{O(\log_2(\log_2(N)))}$	$O(\log_2(N))$	Stabilization	No

Table 8.3: Comparison of DHTs regarding lookup efficiency, peer routing table size, overlay maintenance method and Java library availability.

Koorde [46], Distance Halving (DH) [61], Viceroy [53] and Ulysses [51] need a number of entries in peers' routing table that does not depend on the total number of peers in the system (like CAN). They feature a number of hops that is $O(\log_2(N))$. However, a disadvantage of these methods is that some peers will have more traffic than others (congestion problem). To be fault-tolerant, these systems generally need to expand their routing table and reach a number of entries that is $O(\log_2(N))$. In this case, the number of hops can be reduced to $\frac{O(\log_2(N))}{O(\log_2(\log_2(N)))}$ and there is no more congestion. A stabilization algorithm is used to maintain routing tables in case of peers failure. A Java library is available for Koorde. No implementation was found for DH, Viceroy and Ulysses.

Table 8.3 summarizes above information. The "hops" column contains the number of hops needed by a lookup operation, the "routing entries" column contains the number of entries in the routing table associated to each peer of the DHT, the "maintenance" column contains the method used to update routing tables in case of peer failure, the "library" column indicates if a Java library implementing the DHT is available.

The best trade-off between lookup efficiency and routing tables size is provided by Koorde, Viceroy, DH and Ulysses. No implementation of Viceroy, DH and Ulysses was found. An existing Java toolkit called Overlay Weaver [7, 71] (OW) provides a Java implementation of Koorde. Koorde was therefore chosen to be integrated into LaBoGrid.

Note that DKS provides an interesting approach regarding overlay maintenance by using methods that minimize the cost in terms of exchanged messages. An adaptation of these methods to Koorde would be interesting but is out of the scope of this thesis.

8.4.4 A Missing Function: Update

All DHTs feature at least the basic *get* (retrieval of an entry given its key) and *put* (insertion of a new entry) functions. Most of them provide a *remove* function (removal of an entry given its key). Another interesting function is *update*: the value of an entry of a partial data table is modified in a way depending on provided parameters. Examples of updates are the addition of a stored number with another given one, the adding of a new element to a set or a list, etc. We have not found any DHT system providing this function. In particular, OW does not implement the update of a previously inserted value.

In the context of LaBoGrid, this function is required for the distributed file system implementation: when a file is replicated, a new location must be added to the list of available locations.

Adding this functionality to OW is not difficult, an update request is routed in the same way as a get, put or remove request would be towards a set of peers (the peer responsible of the entry and the peers holding a replica of the entry). When a peer receives this request, it retrieves the entry from its partial data table and updates it with given data.

8.5 MN-tree: A Multiple Purpose Tree Overlay

In Section 8.3, we described how broadcast and barrier synchronization services can be implemented in a distributed way by organizing the computers executing the DAs into a tree. If DAs are part of an overlay having a tree structure, described distributed implementations could take advantage of it.

A drawback of tree overlays is their lack of resistance to link or node failures: in this case, the overlay is disconnected. Robust tree overlays therefore require maintenance mechanisms. Large-scale systems for data dissemination or data lookup based on robust tree overlays exist [37, 22] and use mechanisms to repair the overlay in case of node failure.

For example Frey et al. [37] proposed a system where, if a peer detects that its parent failed, it searches for a new parent. If it does find one, it connects to the

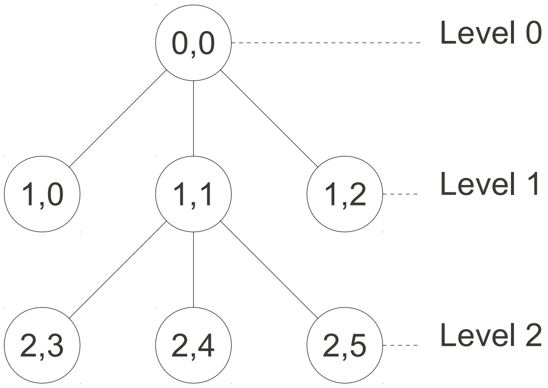

Figure 8.4: Illustration of the position of a node in a tree.

new parent. Otherwise, it declares itself as the root. A mechanism ensures that no cycles appear and that the degree (i.e. the number of connected peers) of the peers remains bounded.

Our approach is slightly different and is based on node (and therefore link) redundancy. This leads to simpler repair operations. We have developed the MN-tree structure for this purpose.

8.5.1 The Overlay

We can define the *position* of a node in a tree as a pair (x,y) where x is the level in the tree (i.e. the "distance" of the node to the root) and y the position of the node in its level. If the maximum number of children by node is bounded by K, at level l, there are K^l possible positions. Figure 8.4 illustrates the position of nodes in a tree where nodes have at most 3 children ($K = 3$). If the tree is not complete (like it is the case in Figure 8.4), some positions have no node associated to them.

We propose a tree overlay where several peers can be associated to a given position in the tree. The set of peers associated to a same position is called a *meta-node*. The tree overlay whose nodes are meta-nodes is called a meta-nodes tree or MN-tree.

The peers of a meta-node are fully connected. In addition, each peer of a meta-node is also fully connected to all peers of the parent meta-node and children meta-nodes.

Therefore, one interesting feature of MN-trees is that even if one or several peers fail, the tree structure is not altered (as long as at least one peer remains available per meta-node). This situation is illustrated in Figure 8.5 showing an

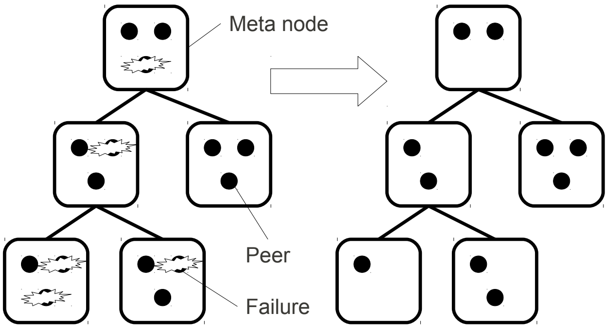

Figure 8.5: Availability of tree structure even in case of multiple failures.

MN-tree with five meta-nodes containing three peers each; five peers fail but the resulting MN-tree is still connected. A link between two meta-nodes means that at least one peer of a meta-node is connected to at least one peer of the other meta-node (i.e. there remains at least one peer per meta-node in the MN-tree).

The number of entries in peers' routing tables does not depend on the number of peers in the overlay. If each meta-node contains M peers and has K children, the routing table of a peer p in a meta-node n contains $M-1$ entries for the other members of n, M entries for the parent and M entries per child of the meta-node. The total number of entries in the routing table is therefore $(K+2)M-1$.

8.5.2 Overlay Construction and Joining

Let M be the set of meta-nodes of an MN-tree and P be the set of peers that are part of this MN-tree. The function $mn : P \to M$ provides the meta-node associated to a given peer i.e. if peer p is part of meta-node n, then $mn(p) = n$. The *size* of a meta-node is the number of peers associated to this meta-node.

There is a trade-off between robustness and scalability: an MN-tree made of one meta-node containing all the peers of the overlay is perfectly robust (the tree structure is always preserved provided that there remains at least one peer in the overlay). However, this approach is not scalable at all because of the size of the routing table of each peer and the maintenance cost of these tables in case of topology change. On the other hand, an MN-tree made of meta-nodes containing one peer each is not robust at all (if one peer fails, the overlay is disconnected) but implies small routing tables: $(K+1)$ entries with K being the number of children per meta-node.

A reasonable approach is therefore to use a parameter giving the number of peers a meta-node should contain in order to have a robust enough overlay but

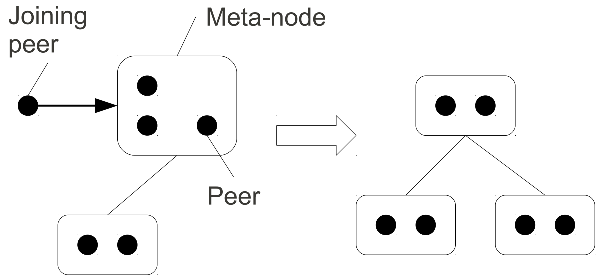

Figure 8.6: Peer joining a meta-node and triggering the creation of a new meta-node ($R = 2$).

leading to reasonably small routing tables.

A meta-node is considered as *reliable* if it contains at least R peers i.e. its size is greater or equal to R and *unreliable* otherwise. The R parameter is called *reliability threshold*. An MN-tree is *reliable* if it is only made of reliable meta-nodes and *unreliable* if it contains at least one unreliable meta-node.

For each meta-node, a *master peer* is defined. The master peer manages its associated meta-node topology. The function $mp : M \rightarrow P$ provides the master peer of a given meta-node.

Initially, the first peer of the MN-tree is the master peer of the root meta-node.

A peer that wants to join an existing MN-tree can send a request to any peer already part of the overlay. If the peer is the master peer of a meta-node, it tries to insert the joining peer in its meta-node, otherwise the join request is forwarded to the master peer of the associated meta-node.

The master peer $mp(n)$ inserts the joining peer in the meta-node n if n is not yet reliable or it has not yet K children. Otherwise, it forwards the join request to the master peer of one child of n. If the peer was inserted in n and the size of n is greater or equal to $2R$, a new reliable meta-node can be extracted from n and can become its child. The creation of a new meta-node is illustrated in Figure 8.6 with $R = 2$.

The creation of a new meta-node implies that peers already in a meta-node are moved to another meta-node. In this case, the routing table of a moved peer must be updated. If a peer of a meta-node n is moved to another meta-node m, the main peer of m sends the new routing table to the peer.

The insertion scheme ensures that only reliable meta-nodes are created and that the maximum number of children per meta-node does not exceed K. Indeed, a new child is added for a meta-node only if it has not already K children. If the peer has been inserted because the meta-node was not reliable, a new child cannot be created (because the size of the meta-node remains smaller than $2R$).

A join request message is forwarded at most $O(\log_K(N/R))$ times before the peer is inserted, where N is the number of peers in the MN-tree, K the number of children per meta-node and R the number of peers per meta-node.

For example, if $N = 10^4$, $K = 2$, $R = 5$ and a join request is sent to a peer of the root meta-node of a complete (all meta-nodes have K children except the leaves) reliable MN-tree, the join request is then forwarded 11 times before reaching a leaf. Each peer maintains a routing table containing at most 19 entries. MN-trees are clearly scalable regarding routing table size and joining.

8.5.3 Overlay Maintenance

In case of peer failure, an MN-tree remains connected. If the failed peer was a master peer, it is replaced by another peer of the same meta-node. We suppose that each peer has a unique identifier (e.g. an IP address and a TCP port) and that the peer identifier space is ordered. In this case, the peer that becomes the new master peer is simply the peer with the smallest identifier.

All the peers connected with the failed master peer also “know” the candidate peer (i.e. they have it in their routing table) that will replace it (by overlay construction). The update of the master peer of a meta-node is therefore a local operation that does not require an additional exchange of messages.

However, in case of peer failure, the MN-tree may become unreliable and therefore more vulnerable to subsequent peer failures. Additional maintenance methods have to be provided.

Passive Maintenance

The join process could be modified in order to route join requests towards unreliable meta-nodes. After peer insertions, the reliability of the MN-tree may be restored.

However, this approach requires that new peers frequently join the overlay. If it is not the case, the MN-tree may stay unreliable for a long period of time. In order to avoid this situation, an active maintenance method is proposed.

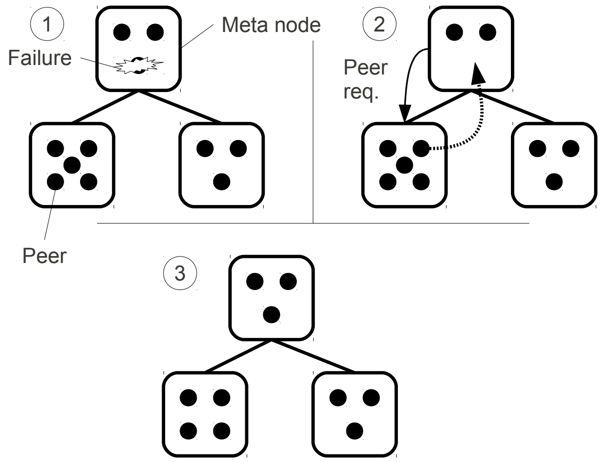

Figure 8.7: Active maintenance process in case of failure: 1) a meta-node becomes unreliable, 2) a peer request message is sent and a peer is moved, 3) the MN-tree is reliable again ($R = 3$).

Active Maintenance

The MN-tree construction method presented previously tends to build a complete tree with meta-nodes containing exactly R peers. The meta-nodes that do not already have K children may have a size greater to R. A meta-node that becomes unreliable could therefore take a peer from one of these meta-nodes to become reliable again and the providing meta-node remains reliable.

Let a meta-node become unreliable because one of its peers fails. If the meta-node is a leaf of the MN-tree (it has no child), it merges with its parent. Otherwise, it sends a peer request message to one of its children. This message is forwarded downside in the MN-tree until a meta-node having more than R peers or a leaf is reached. A peer is then moved from reached meta-node to the unreliable meta-node. If the reached meta-node is a leaf and becomes unreliable after peer move, it merges with its parent.

Active maintenance implies that peers already part of a meta-node are moved to another meta-node. In the same way than when peers are moved during the insertion of a new peer, the routing tables of moved peers must be updated.

Figure 8.7 illustrates the described maintenance process.

8.5.4 MN-trees Multiple Purposes

The MN-tree overlay is designed in order to:

- elect a leader peer,
- broadcast messages,
- provide a Computer Tree used for Dynamic Load Balancing (see Chapter 7),
- implement barrier synchronization service in a distributed way.

Leader election

The master peer of the root meta-node is selected as the leader. Therefore, in case of leader failure, the maintenance process described in Section 8.5.3 implicitly selects the new leader through the selection of the new master peer of a meta-node. This process is particularly efficient because it does not require directly the exchange of any message.

Reliable Broadcast

An efficient broadcast is easy to implement using MN-trees: initially, the source peer (the peer that initiates the broadcast) sends the message the master peer of its meta-node (if it is not the master peer). The classical tree broadcast method is then used to broadcast the message among the master peers of the MN-tree. Each master peer also forwards the message to the other peers of its associated meta-node. This process is illustrated in Figure 8.8.

This method is not reliable in case a master peer fails before it could deliver the message to all required peers. Any other peer can fail without affecting the reliability of the broadcast.

A possible solution would be to use multiple paths to broadcast a message. A broadcasted message is then received multiple times and even if one peer fails during message propagation and cuts one path, the message will still be delivered. This solution has the disadvantage of implying a substantial constant overhead even in the absence of failure.

We propose a protocol based on message retransmission in case of failure with a smaller overhead in terms of message transmissions in the absence of failure.

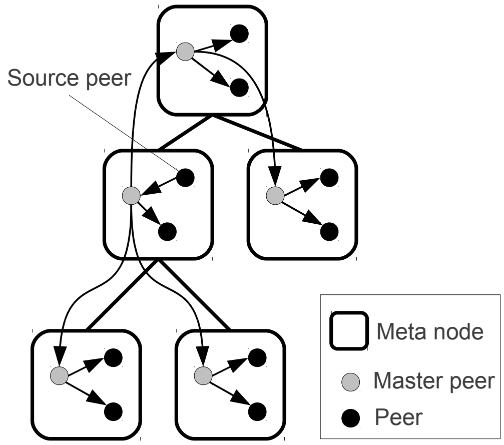

Figure 8.8: Simple broadcast in an MN-tree.

When a master peer p receives a broadcast message, it stores and forwards it to its neighboring master peers (i.e. the master peers of parent and children meta-nodes) and to the other peers of its meta-node which also store the message. When neighboring master peers have forwarded the message to all the other peers of their associated meta-node, they acknowledge the forwarding to p. p then signals to all the peers of its associated meta-node they can delete the acknowledged message. The only overhead in terms of message transmissions caused by this protocol in the absence of failure is caused by the acknowledgements.

If a master peer fails before the message could be delivered to all required peers (namely the peers from its associated meta-node, parent meta-node and children meta-nodes main peers), the new master peer sends the message again.

If the message to broadcast was not yet forwarded to all other peers of the meta-node when the master peer fails, the new master peer may not be able to send the message again because it did not receive it. In this case, it is sent again by the source peer to the new master peer.

Computer Tree

The Computer Tree (CT) is used during the balancing phase of the dynamic load balancing method presented in Chapter 7. This phase essentially consists in computing the optimal number of work units (sublattices in our particular case) to attribute to each computer of the tree.

The peers of the MN-tree can be used as nodes of the CT and it can therefore be directly derived from the MN-tree: the tree formed by the master peers serves

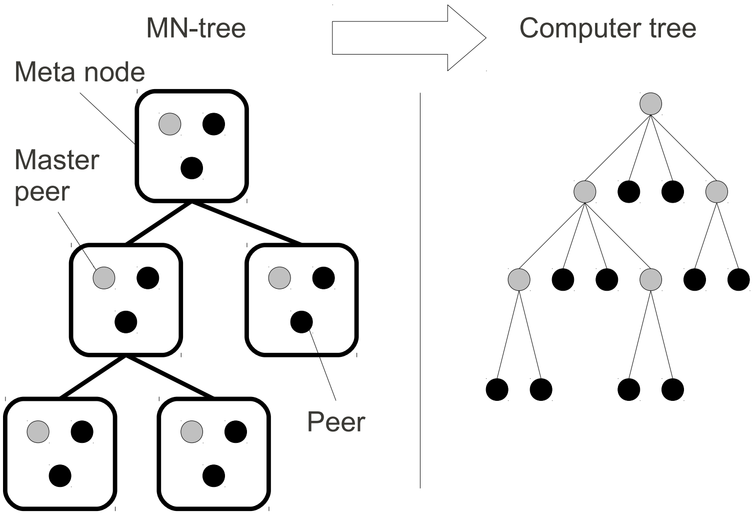

Figure 8.9: Computer tree derived from a given MN-tree.

as skeleton and the peers of a meta-node are the children of the master peer of their meta-node. Figure 8.9 shows the CT derived from a given MN-tree. Each meta-node contains three peers, the master peer of a meta-node with two children in the MN-tree has four children in the CT: the master peers of the children meta-nodes in the MN-tree and the two other peers of its meta-node. The master peer of a leaf meta-node has only two children in the CT: the other peers of its meta-node.

The construction of a new CT on topology changes (new computer available or computer fail) is implicit to the MN-tree construction and maintenance and requires no additional effort as long as the dynamic load balancing component is aware of the MN-tree topology changes.

Barrier Synchronization

The scalable tree-based barrier synchronization mechanism from Section 8.3 can use the MN-tree structure. As a reminder, this mechanism is based on two types of messages the processes exchange: `REACHED` and `ALLREACHED`. The `REACHED` message is sent by a process *p* to its parent once all the processes of the subtree whose root is *p* have reached the barrier. The `ALLREACHED` message is broadcasted by root process when all processes have reached the barrier and causes the processes to continue their execution.

If we suppose that each peer runs exactly one process that will reach a barrier, then we can use the tree based barrier synchronization mechanism with a tree organization of the processes similar to the Computer Tree derived in previous section.

8.6 Distributed LaBoGrid Control

In previous sections, we presented tools to distribute the central point of the architecture described in Chapter 4: the Controller. In this section, we present how these tools are integrated into LaBoGrid.

All DAs must join two overlays: the DHT overlay which allows the distributed access to shared data and the MN-tree overlay which provides leader election, broadcasting, implicit computer tree construction and barrier synchronization. As discussed in Section 8.2, DAs are able to generate themselves a unique identifier and do not need anymore to initially connect to the Controller in order to receive it. The first executed DA initializes the two overlays. Subsequent DAs take as argument a list of DAs already part of the overlays through which they can join.

The DA executed by the leader peer selected by the MN-tree is called the *leader* DA. It executes the Controller Agent (CA) which is stripped from all logic and data implemented by the DHT and the MN-tree. In the context of LaBoGrid, the minimal CA:

- generates the initial application graph (the graph representing a distributed application, see Section 5.2),
- triggers load balancing phases when needed (see Section 7.5.3) and broadcasts the mapping result to all DAs,
- updates the state of the experience (simulation number and last replicated iteration for current simulation).

Topology changes must be signaled to the CA when detected (the CA can then trigger a load balancing process and (re)start a simulation). It is the master peer of the meta-node in which the change occurred that signals the event (failure or insertion) to the CA. To signal a topology change, a message is reliably forwarded to the leader DA. A protocol similar to the reliable broadcast is used.

8.7 Results

In this section, we compare the efficiency of broadcast, barrier synchronization and shared table services implemented in a centralized non-robust way and using an MN-tree and a DHT.

An MN-tree has two parameters: the maximum number of children of a meta-node d and the reliability threshold r. Let N be the number of MN-tree peers. The MN-tree then contains $\lfloor N/r \rfloor$ meta-nodes and has depth $\lceil \log_d(\lfloor N/r \rfloor) \rceil$.

An MN-tree degenerates to the centralized case in two situations:

- when $r = 1$ and $d = N - 1$,
- when $r > (N/2)$.

In first situation, there are as many meta-nodes as peers, the MN-tree has depth 2 and is not robust: if the only peer of the root meta-node leaves, the overlay is disconnected. In last situation, the MN-tree has depth 1, contains only one meta-node and is robust.

With $d = 1$, the MN-tree degenerates into a meta-nodes chain.

8.7.1 Broadcast

In order to compare the centralized and distributed implementations of the broadcasting service, we measure the time required to broadcast 1000 messages. The experience is executed on a cluster of 51 computers ($N = 51$).

It takes 64 seconds to broadcast all messages with the centralized implementation. Table 8.4 shows the time to broadcast the messages using the distributed implementation in function of different MN-tree parameters. The distributed implementation of the broadcast service gives better results in all cases except the two highlighted ones in the table.

$d \backslash r$	1	2	4	10
1	25	14	28	**68**
2	11	12	22	48
4	15	19	26	**66**
10	37	41	48	57

Table 8.4: Time (in seconds) to broadcast 1000 messages using different MN-tree parameters.

When increasing r and/or d, the depth of the tree potentially decreases and the dissemination process is better parallelized, which should let the execution time unchanged or decrease it. However, we do not systematically observe this property.

The reason for this is that when increasing d or r, main peers have a supplementary source of acknowledgement messages which increases their load in terms of messages to handle. The choice of d and r values is therefore a trade-off between robustness and efficiency.

8.7.2 Barrier Synchronization

The centralized and distributed implementations of the barrier synchronization service are compared in the same way than broadcast service: the time required to synchronize all processes 1000 times is measured using the two implementations. The experience is executed on a cluster of 51 computers ($N = 51$).

It takes 137 seconds to sync all processes 1000 times with the centralized implementation. Table 8.5 shows the time to sync the processes using the distributed implementation in function of different MN-tree parameters.

$d\backslash r$	1	2	4	10
1	141	90	59	62
2	41	41	42	59
4	41	41	43	56
10	41	42	44	56

Table 8.5: Time (in seconds) to execute 1000 barrier synchronizations using different MN-tree parameters.

The MN-tree based implementation is more than 3 times faster most of the time ($d > 1$ and $r < 10$). When $d = 1$, the tree used by the synchronization algorithm described in Section 8.3 degenerates into a chain which explains the bad results.

When $d \geq 2$, good results are achieved but increasing r slows the execution down. This is because the load of the main peers in terms of messages to send and receive then increases. Therefore, like for the broadcast service, the choice of the MN-tree parameters is a trade-off between robustness and efficiency.

8.7.3 Table Service

In order to compare the centralized and distributed implementations of the table service used by the distributed file system, we executed an experience where each task executed by a DA inserts 2000 entries, then retrieves them and, finally, removes them from the table.

The distributed implementation uses a DHT to implement the table service. As stated in Section 8.4, we use a Koorde [46] DHT implemented by the Overlay Weaver [7] library.

The DHT based implementation of the table service leads to the lowest execution time: it takes 680 seconds to execute the experience with the centralized implementation and 578 seconds with the distributed implementation.

8.8 Conclusion

In order to remove the single point of failure and potential bottleneck in LaBoGrid's architecture, we proposed a mostly decentralized implementation of the controller.

This implementation is based on the definition of several services used by the components of LaBoGrid:

- leader election,
- broadcasting,
- barrier synchronization,
- distributed file system,
- dynamic load balancing.

A decentralized implementation was proposed for each of these services.

The distributed implementation of leader election, broadcasting and barrier synchronization services is based on MN-trees: an original robust tree-based overlay.

The distributed file system relies on the gFLT that was previously hosted by the Controller. The gFLT is now implemented by a Koorde [46] DHT provided by Overlay Weaver [7, 71], a Java toolkit that features an API for overlay algorithms design and high-level services. Overlay Weaver's API had to be slightly modified in order to add the "update" operation required by our distributed file system.

Finally, the dynamic load balancing service needs the computation of the Computer Tree (CT). The CT, previously generated by the Controller, is now built in a decentralized way by using MN-trees.

We observed that, in addition to being robust, the distributed implementation of most services is generally more efficient (i.e. decreases the required execution time for the same task) than the centralized implementation.

The efficiency of broadcast and barrier synchronization services depends on the MN-tree parameters: the reliability threshold (controlling the number of peers per meta-node) and meta-nodes degree (bounding the maximum number of children a meta-node can have).

Increasing the reliability threshold improves the overlay robustness but increases the load on the main peers of the meta-nodes. Increasing the meta-nodes degree improves the parallelization of broadcast and barrier synchronization services but also increases the load on the main peers of the meta-nodes. In the same way than explained in Section 6.7, if the probability of failures is known, the choice of these parameters can probably be optimized. However, this topic is not addressed in this thesis.

The services presented above are not intensively used in the context of distributed LB simulations. The use of their distributed implementation instead of the centralized implementation does therefore not lead to a significant reduction in execution time (at least, not in the simulations we executed). However, LaBoGrid is now better adapted to larger scale simulations and is completely robust.

Chapter 9

Conclusion

9.1 Summary

We have developed a distributed implementation of LB simulations presented in Chapter 2. In Chapter 3, this implementation was optimized using a method introduced by Murphy [58] that had to be adapted to the Java programming language and was improved by removing its memory overhead. An additional method was proposed in order to circumvent the memory locality problem introduced by Murphy's method regarding the collision operator implementation. As a result of these modifications, the execution time was almost divided by two regarding the initial implementation.

LaBoGrid is the distributed application designed in the context of this thesis. Its general architecture is presented in Chapter 4. LaBoGrid allows the user to easily run sequences of simulations with potentially different parameters and implementations for each simulation. These are provided to LaBoGrid by the user through an XML configuration file and, potentially, additional JAR files containing additional classes used to implement the simulations.

LaBoGrid is written using a generic framework based on asynchronously communicating components. This framework includes a communication layer that handles the transmission and the reception of messages through the network. We expected this generic framework to produce an execution time overhead when compared to a more specific implementation but we observed that this overhead remains acceptable. For example, the execution time of a distributed LB simulation on a $(128, 128, 128)$ lattice using an MRT collision operator executed by 32 processors is multiplied by a factor 1.17 when using LaBoGrid instead of the specific implementation.

In Chapter 5, we have shown that existing static load balancing tools called static mappers can be used to distribute an LB simulation in a way that minimizes the execution time for a cluster of computers with heterogeneous computational powers. This is achieved by distributing simulation data in function of the computational power of available computers and by minimizing the amount of data transmitted through the network. For example, the execution time of a distributed LB simulation on a $(176,176,176)$ lattice using the MRT collision operator was divided by 1.34 when distributing 256 sublattices on 17 computers of a heterogeneous cluster using the heterogeneous SCOTCH mapper instead of assigning a single sublattice to each computer (see Section 5.6).

The fault-tolerance problem is described in Chapter 6 where a method is proposed to execute robust distributed LB simulations. This method is based on the regular saving of the state of the simulation. The most recently saved state is then reloaded to restart the simulation if a simulation process fails. This general mechanism is called checkpoint/restart. The saving of simulation's state requires that all processes write their state to disk in a state file. The state files are then replicated in a distributed way to several computers in order to be still available if one or several computers fail.

LaBoGrid can be executed as a job of CanoPeer [1], a P2P Grid computing middleware created by Cyril Briquet who participated to the design of LaBoGrid's fault-tolerance system. The interest of this integration is mainly the resource discovery service provided by CanoPeer. However, CanoPeer's scheduling policy is an additional source of failures.

We observed that state replication causes a substantial overhead in execution time. However, the proposed distributed replication scheme is far more efficient than centralized replication (for example, centralized replication is almost 13 times slower that distributed replication with a single replication neighbor for a $(176,176,176)$ lattice distributed among 25 computers).

In order to evaluate the interest of using state replication instead of simply restarting the simulation in case of failure, we introduced the concept of mean execution time of a simulation which takes into account the probability of failure of one or several processes one or several times during the simulation's execution. Using the mean execution time, we have shown through a few examples that if the probability of failure of a process is high enough, it is indeed interesting to use state replication instead of simply restarting the simulation from the beginning in case of failure.

In addition, if the probability distribution of failures is known, it is possible to choose replication parameters (the number of replication neighbors and the replication period) in a way that minimizes the mean execution time.

The checkpoint/restart mechanism potentially triggers the execution of several load balancing phases during a simulation. The static load balancing tools from Chapter 5 are not adapted for the additional constraints introduced by the dynamic load balancing problem.

An original dynamic load balancing method combining an adapted Tree Walking Algorithm (TWA) [70] for load balancing and the KL criterion [48] for work migration is introduced in Chapter 7.

We observed that our method produced mappings leading to a reduction of simulations' execution time comparable to the reduction obtained with heterogeneous SCOTCH [62], a static load balancing tool introduced in Chapter 5. We also observed that, compared to a distributed implementation of diffusive scheme [30] generally used in dynamic load balancing methods, the adapted TWA significantly reduces the number of exchanged messages. For example, when distributing 256 sublattices among 54 computers and initially associating all sublattices to one computer, the number of exchanged messages is divided by a factor between 61 and 412. This factor depends on the expected quality of the distribution produced using the diffusion scheme. Finally, our method minimizes the amount of migrated work in case of incremental mappings.

Finally, we have shown in Chapter 8 that LaBoGrid's architecture could be expressed in terms of services:

- leader election,
- broadcasting,
- barrier synchronization,
- shared table,
- distributed file system,
- dynamic load balancing.

Most of these services were totally (broadcasting, barrier synchronization, shared table) or partially (distributed file system, dynamic load balancing) centralized and implemented by the Controller. The Controller was therefore both a single point of failure (robustness problem) and a potential bottleneck (scalability problem).

A distributed and mostly decentralized implementation is proposed for each of these services. We designed a robust tree-based overlay called MN-tree used in

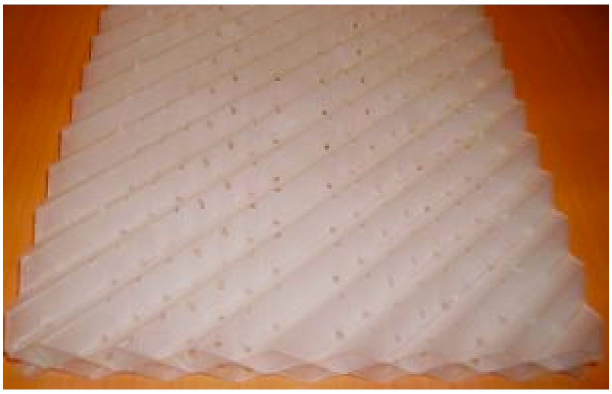

Figure 9.1: Example of structured packing used in distillation and reactive distillation columns.

the distributed implementations of leader election, broadcasting, barrier synchronization and dynamic load balancing services. We observed that, in addition to being robust, the distributed implementations are more efficient than centralized implementations, even for small scale clusters (50 computers). For example, when using MN-trees instead of a centralized implementation to organize 50 processes, broadcast is 2.5 times faster and barrier synchronization 1.5 times faster.

The distributed shared table service is based on Koorde DHT [46] whose implementation is taken from an existing toolkit called Overaly Weaver [7]. In addition to making the service robust, we observed slightly better results than the centralized implementation: a data intensive task consisting of 2000 insertions, retrievals and removals by each task of a set of 50 tasks is executed 1.17 times faster when using the distributed implementation instead of the centralized implementation.

9.2 Concluding Remarks

The main result of this thesis is LaBoGrid, the application currently used by the LGC at University of Liège. Several publications [19, 24, 55, 75, 18] include results produced using it.

In particular, LaBoGrid was used to produce a velocity field describing a fluid flow in structured packing used in distillation and reactive distillation columns (see Figure 9.1). Figure 9.2 shows a mesh representation of this kind of structured packing. The blue areas represent the inflow plane and the red areas the outflow plane. Figure 9.3 shows two slices of the velocity field computed from the result

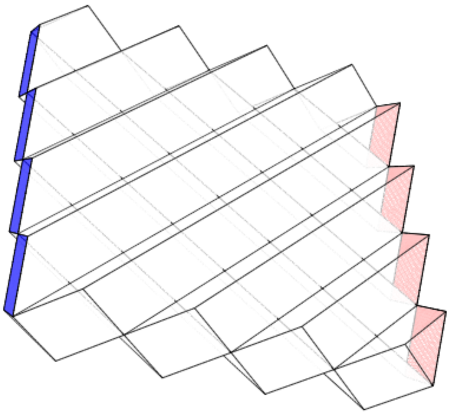

Figure 9.2: Mesh representation of a structured packing.

of an LB simulation of a single-phase gas flow through the structure shown in Figure 9.2. Upper slice is parallel to flow direction (and therefore perpendicular to inflow and outflow plane). Lower slice is perpendicular to flow direction (and therefore parallel to inflow and outflow plane). Only two channels out of the four represented in Figure 9.2 are shown in the lower slice.

LaBoGrid is currently not able to directly produce graphical representations like shown in Figure 9.3. These were obtained by post-processing LaBoGrid's result using MATLAB.

LaBoGrid is currently executed in three environment types:

- single desktop computer,
- very small homogeneous cluster of 3 powerful servers,
- small heterogeneous cluster of 50 desktop computers.

The first two environment types were generally chosen to execute small simulations for example when testing various implementations of collision operators or boundary conditions. The last environment was used to execute larger simulations involving lattices with several millions of sites (for example, $(400,400,400)$ lattices) and complex collision operators (for example, the turbulent viscosity model of Smagorinsky adapted to the MRT collision operator; see Section 2.2.2).

The execution environment must be taken into account when choosing LaBoGrid's parameters. In Section 5.6, we observed that using a number of sublattices that is high enough regarding the number of computers (in our experiments, we observed best results when the number of sublattices is more than 10 times

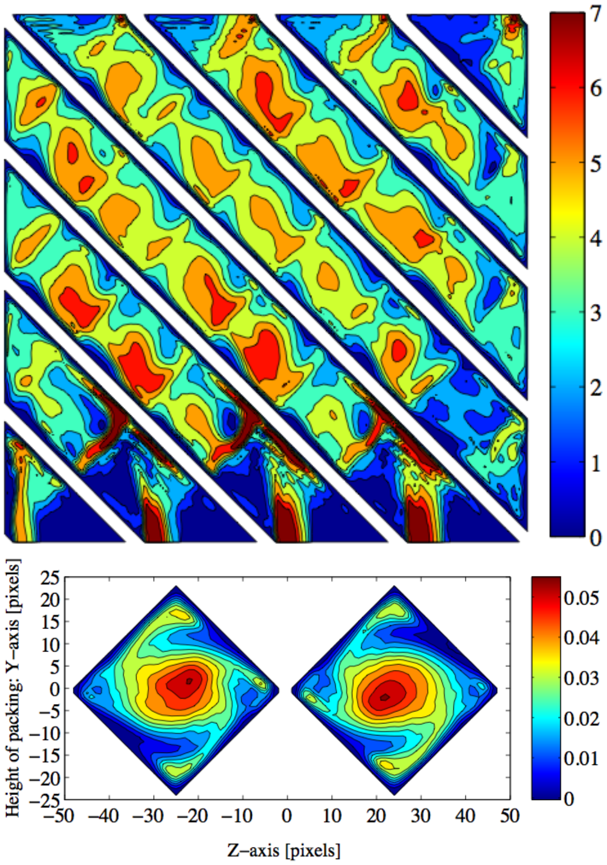

Figure 9.3: Slices of a velocity field.

higher than the number of computers) is interesting when computers have heterogeneous computational powers but implies an execution time overhead when computers have homogeneous computational powers (and, therefore, a single sublattice could be associated to each computer). The number of sublattices to produce should therefore be chosen in function of the execution environment: equal to the number of computers in case of homogeneous computational powers and higher otherwise (for example, 10 times higher for a $(176,176,176)$ lattice executed on 17 computers).

We have shown that state replication implies an important execution time overhead (see Section 6.7). It should therefore be disabled for simulations where the simulation execution time without replication enabled is small, and in reliable (i.e. where the probability of failure is low enough so that restarting the simulation in case of failure is acceptable) environments. In unreliable environments, the selection of replication parameters (number of replicas and replication period) can be optimized if failure probability distribution is known.

State replication brings robustness to distributed LB simulations. However, for simulations to be able to complete, the execution environment must feature enough stability periods, i.e. phases of the execution during which no failure occurs. The stability periods must be long enough to complete at least one simulation phase (the computation for time steps between two replications) followed by a replication. In following example, it is not the case: a failure occurs systematically after the same amount of time and this amount of time is smaller than the time required to execute a simulation phase followed by a replication. The simulation will then never complete because each time the failure occurs, the simulation is restarted from the beginning.

This situation can be avoided by choosing a very short replication period (for example, state replication occurs after each simulation time step). However, in this case, the execution time overhead becomes prohibitive.

To summarize, we designed a powerful and flexible software called LaBoGrid to execute large LB simulations. LaBoGrid is highly configurable which allows to adapt it to the execution environment. In particular, the execution environment can be heterogeneous (in terms of architecture, OS, but also computational power) and unreliable.

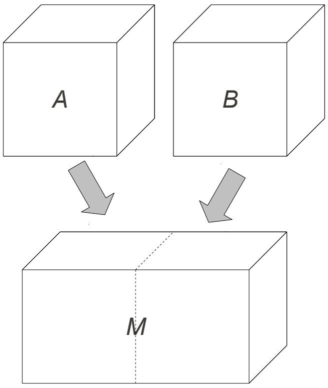

Figure 9.4: Grouping of two adjacent sublattices *A* and *B* into a meta-lattice *M*.

9.3 Future Work

LaBoGrid is currently an experimental software that provides “raw” results that need to be post-processed in order to extract a meaningful information such as a velocity field. A part or the whole post-processing could be embedded into LaBoGrid and executed in a distributed way in order to accelerate its completion.

LaBoGrid is based on a generic framework providing commonly used services. Other distributed applications than distributed LB simulations can surely benefit from this implementation. It is planned to separate the components of LaBoGrid that are generic from those that are specific to LB simulations and use the generic framework to implement other types of applications. This will probably arise the need for new services to be added to the existing framework.

In Section 5.6, we stated that increasing the number of sublattices per computer implies an execution time overhead caused by the fact that more incoming and outgoing densities have to be handled when compared to the single sublattice case because the number of sublattice-sublattice interfaces is increased. A method consisting of copying the contents of adjacent sublattices associated to the same computer in a single *meta-lattice* (which, similarly to sublattices, has the same properties as a lattice) decreases the number of sublattice-sublattice interfaces. Figure 9.4 illustrates the grouping of two adjacent sublattices *A* and *B* into a single meta-lattice. The densities that were copied from one sublattice to another when using only sublattices are then simply propagated in the meta-lattice.

The Java programming language was chosen to implement LaBoGrid because it is an acceptable trade-off between efficiency and portability. A native version of LaBoGrid could be produced for environments where portability is not an issue.

LaBoGrid was designed by keeping extendability in mind. In particular, it should be easy (from software engineer point of view at least) to develop new collision operators and boundary conditions, add support for multi-phase flows or create more complex lattice representations including more information than only the description of the fluid.

In Chapter 8, the distribution of the Controller was addressed. The two main reasons for this are robustness and scalability. There is still a scalability issue to solve. However, it is less important than with the centralized case: the structure of MN-trees implies that some computers have more load than others, in particular the main peer of the root meta-node. There is maybe a more generic structure that could replace partially or completely MN-trees and that would not have this drawback.

From the LB simulations point of view, before simulations can be executed in large scale environments on very large lattices (for example, a $(1000,1000,1000)$ lattice), another scalability issue must be solved: the solid representation. Currently, the solid is given in a file that is fully read into memory by the master process in order to be subdivided into subsolids before the simulation is executed. With very large lattices (and therefore solids), it is not possible anymore to read the whole file into memory (a $(1000,1000,1000)$ lattice requires around 1 gigabyte of memory). Several solutions are possible: another lighter representation for solids than bitmaps, a structured file representation allowing to directly extract parts of the solid (but this implies the access to the file from all computers), etc.

All these suggestions for future works mostly imply extensions or improvements of the existing implementation. However, the system could be further analyzed: a more general study on the choice of replication parameters (see Section 6.7) in function of the probability distribution of failures and simulation parameters would be interesting. The dynamic adaptation of replication parameters in reaction to failure events could also be an interesting topic.

The dynamic load balancing method presented in Chapter 7 should be studied on more general graphs than the application graphs representing a distributed LB simulation. For example, it would be interesting to compare it to other mappers when handling meshes produced for classical CFD methods.

9.4 Thesis Statement Fulfillment

We have shown that it is possible to design and develop a software system which is able to organize large scale clusters of inherently unreliable computers in an efficient, scalable and robust way for implementing large scale LB simulations:

- existing and original load balancing methods have been used to leverage the computational power of available computers in an optimal way,
- a distributed and scalable fault-tolerance mechanism was designed to make LB simulations adapted to unreliable execution environments,
- the master-slave model, on which our distributed implementation of LB simulations relies, is implemented in a distributed and mostly scalable way.

In previous section, suggestions were made to further improve the efficiency of distributed LB simulations (in particular, by using meta-lattices).

In addition, scalability issues remain; the most important being the solid representation. This issue must be solved in order to be able to execute very large scale LB simulations involving lattices composed of billions of sites.

Finally, our distributed implementation of the master-slave model is potentially not scalable for very large scale clusters (involving several millions of computers) because the tree organization of computers used as the basis of our implementation implies that computers that are close to the root endure a heavier load. Further research is needed to address this kind of execution environment in the context of LB simulations' execution.

Bibliography

[1] CanoPeer. http://www.canopeer.org/.

[2] Coda File System. http://www.coda.cs.cmu.edu/.

[3] El'Beem. http://elbeem.sourceforge.net/.

[4] Java specifications API. http://java.sun.com/reference/api/.

[5] Lustre File System. http://www.lustre.org/.

[6] MooseFS: A File System for Highly Reliable Petabyte Storage. http://www.moosefs.org/.

[7] Overlay Weaver: An overlay construction toolkit. http://overlayweaver.sourceforge.net/.

[8] Palabos: Parallel Lattice Boltzmann Solver. http://www.lbmethod.org/palabos/.

[9] PowerFlow. http://www.exa.com/pages/pflow/pflow_physics.html.

[10] Sailfish. http://sailfish.us.edu.pl/.

[11] K. Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *Proceedings of the Sixth International Conference on Cooperative Information Systems (CoopIS 2001)*, volume 2172/2001 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2001.

[12] S. Aferka. *Etude de l'hydrodynamique dans une colonne de distillation réactive : Mesure de la distribution des phases par tomographie à rayons X*. PhD thesis, University of Liège, 2009.

[13] L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS(N, k, f): A family of low communication, scalable and fault-tolerant infrastructures for P2P

applications. In *Proceedings of the 3st International Symposium on Cluster Computing and the Grid (CCGrid'03)*, pages 344–350, 2003.

[14] L. O. Alima, A. Ghodsiand, and S. Haridi. A framework for structured peer-to-peer overlay networks. In *Proceedings of Global Computing 2004*, volume 3267/2005 of *Lecture Notes in Computer Science*, pages 223–249. Springer, 2005.

[15] B. Amedro, V. Bodnartchouk, D. Caromel, C. Delbé, F. Huet, and G. L. Taboada. Current state of Java for HPC. Technical report, INRIA, August 2008.

[16] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An approach to easily assemble grids with equitable resource sharing. In *Proceedings of Workshop on Job Scheduling Strategies for Parallel Processing*.

[17] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 230–240. ACM, 1987.

[18] D. Beugre. *Etude de l'écoulement d'un fluide dans des géométries complexes rencontrées en Génie Chimique par la méthode de Boltzmann sur réseau*. PhD thesis, University of Liège, 2010.

[19] D. A. Beugre, S. Calvo, G. Dethier, M. Crine, D. Toye, and P. Marchot. Lattice Boltzmann 3D flow simulations in a metallic foam. *Journal of Computational and Applied Mathematics*, 2009.

[20] P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems. *Physical Review*, 94:511–525, 1954.

[21] C. Briquet. *Systematic Cooperation in P2P Grids*. PhD thesis, University of Liège, 2008.

[22] F. Buccafurri and G. Lax. TLS: A tree-based DHT lookup service for highly dynamic networks. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, volume 3290/2004 of *Lecture Notes in Computer Science*, pages 563–580. Springer, 2004.

[23] S. Calvo. *Caractérisation de l'hydrodynamique des écoulements gaz-liquide à contre-courant dans une colonne rectangulaire à empilage*. PhD thesis, University of Liège, 2010.

[24] S. Calvo, D. A. Beugre, M. Crine, A. Léonard, P. Marchot, and D. Toye. Phase distribution measurements in metallic foam packing using X-ray radiography and micro-tomography. *Chemical Engineering and Processing: Process Intensification*, 48:1030–1039, 2009.

[25] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 1996.

[26] H. Chen, S. Chen, and W. H. Matthaeus. Recovery of the Navier-Stokes equations using a lattice-gas Boltzmann method. *Physical Review A*, 45(8):R5339–R5342, April 1992.

[27] J. Chen and V. E. Taylor. Mesh partitioning for efficient use of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):67–79, 2002.

[28] C. Chevalier and F. Pellegrini. PT-SCOTCH: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6-8):318 – 331, 2008.

[29] W. Cirne, F. Brasileiro, N. Andrade, L.B. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 2006.

[30] G. Cybenko. Dynamic load balancing for distributed memory processors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[31] D. d'Humières. Multiple-relaxation-time Lattice Boltzmann models in three dimensions. *Philosophical Transactions of The Royal Society A*, 360(1792):437–451, March 2002.

[32] R. Diekmann, A. Frommer, and B. Monien. Efficient schemes for nearest neighbor load balancing. *Parallel Computing*, 25(7):789 – 812, 1999.

[33] J. Dongarra, I. Foster, G. Fox, W. Grop, K. Kennedy, L. Torczon, and A. White, editors. *Sourcebook of parallel computing*. Morgan Kaufmann Publishers Inc., 2003.

[34] S. El-Ansary and S. Haridi. *Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless and Peer-to-Peer Networks*, chapter 39. An Overview of Structured Overlay Networks. CRC Press, 2005.

[35] C. Engelmann and G. A. Geist. A diskless checkpointing algorithm for super-scale architectures applied to the fast fourier transform. In *Proceedings of CLADE'03, HPDC Workshops*, Seattle, WA, USA, June 2003.

[36] C. Engelmann and G. A. Geist. Super-scalable algorithms for computing on 100,000 processors. In *Proceedings of ICCS 2005*, Atlanta, GA, USA, May 2005.

[37] D. Frey and A. L. Murphy. Failure-tolerant overlay trees for large-scale dynamic networks. In *Eighth International Conference on Peer-to-Peer Computing*, pages 351–361. IEEE Computer Society, 2008.

[38] Q. Gao, W. Yu, W. Huang, and D. K. Panda. Application-transparent checkpoint/restart for mpi programs over infiniband. In *Proceedings of the International Conference on Parallel Processing (ICPP) 2006*, pages 471 – 478, 2006.

[39] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. SIAM, 2001.

[40] R. Gupta and C. R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *International Journal of Parallel Programming*, June 1989.

[41] H.-U. Heiss and M. Schmitz. Decentralized dynamic load balancing: The particles approach. *Information Sciences*, 84(1-2):115–128, May 1995.

[42] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184(2-4):485–500, April 2000.

[43] S. Hou, J. Sterling, S. Chen, and G. D. Doolen. A Lattice Boltzmann subgrid model for high reynolds number flows. In Raymond Kapral and Anna T. Lawniczak, editors, *Pattern formation and lattice gas automata*, pages 151–166, Boston, MA, USA, 1995.

[44] Y. F. Hu and R. J. Blake. An optimal dynamic load balancing algorithm. Technical report, Daresbury Laboratory, Warrington, WA4 4AD, UK, 1995.

[45] S. Huang, E. Aubanel, and V. C. Bhavsar. PaGrid: A mesh partitioner for computational grids. *Journal of Grid Computing*, 4(1):71–88, March 2006.

[46] F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, volume 2735/2003 of *Lecture Notes in Computer Sience*, pages 98–107. Springer, February 2003.

[47] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, August 1998.

[48] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:291–307, 1970.

[49] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. In *Proceedings of 1986 ACM Fall joint computer conference*, pages 1150 – 1158, Dallas, Texas, United States, 1986.

[50] M. Krafczyk, J. Tölke, and L.-S. Luo. Large-eddy simulations with a multiple-relaxation LBE model. *International Journal of Modern Physics B*, 17(01-02):33–39, 2003.

[51] A. Kumar, S. Merugu, J. Xu, E. W. Zegura, and X. Yu. Ulysses: A robust, low-diameter, low-latency peer-to-peer network. *European Transactions on Telecommunications*, 15(6):571 – 587, November 2004.

[52] S. Kumar, S. Das, and R. Biswas. Graph partitioning for parallel applications in heterogeneous grid environments. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 167, Florida, 2002. IEEE Computer Society.

[53] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st annual symposium on Principles of distributed computing*, pages 183 – 192. ACM, 2002.

[54] N. Malpani, J. L. Welch, and N. Vaidya. Leader election algorithms for mobile ad hoc networks. In *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 96–103, 2000.

[55] P. Marchot, D. A. Beugre, M. Crine, G. Dethier, A. Léonard, and D. Toye. Lattice Boltzmann 3D flow simulations in a hepa aerosol filter on a computing grid. In *Proceedings of ECCE6 - 6th European Congress of Chemical Engineering*, Denmark, september 2007.

[56] P. Maymounkov and D. Mazires. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, volume 2429/2002 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2002.

[57] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21 – 65, February 1991.

[58] S. Murphy. Performance of Lattice Boltzmann kernels. Master's thesis, University of Edinburgh, 2005.

[59] J. Nabrzyski, J. Schopf, and J. Weglarz, editors. *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, 2003.

[60] J. Nabrzyski, J. Schopf, and J. Werglaz. *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, 2003.

[61] M. Naor and U. Wieder. Novel architectures for P2P applications: The continuous-discrete approach. *ACM Transactions on Algorithms (TALG)*, 3(3), August 2007.

[62] F. Pellegrini and J. Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking (HPCN'96)*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer, 1996.

[63] J. S. Plank, Y. Kim, and J. J. Dongarra. Fault tolerant matrix operations for networks of workstations using diskless checkpointing. *Journal of Parallel and Distributed Computing*, 43:125–138, 1997.

[64] T. Plewa, T. Linde, and V. G. Weirs, editors. *Adaptive Mesh Refinement - Theory and Applications*, volume 41 of *Lecture Notes in Computational Science and Engineering*. Springer Berlin Heidelberg, 2005.

[65] Y. H. Qian, D. D'Humières, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *Europhysics Letters*, 17:479–484, 1992.

[66] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of the 2001 SIGCOMM Conference*, volume 31, pages 161–172. ACM, 2001.

[67] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms*, volume 2218/2001 of *Lecture Notes In Computer Science*, pages 329–350. Springer, 2001.

[68] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[69] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In Springer Berlin / Heidelberg, editor, *Euro-Par 2000 Parallel Processing*, volume 1900/2000 of *Lecture Notes in Computer Science*, pages 296–310, 2000.

[70] W. Shu and M.-Y. Wu. Runtime Incremental Parallel Scheduling (RIPS) on distributed memory computers. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):637–649, June 1996.

[71] K. Shudo. Overlay Weaver: An overlay construction toolkit. *AIST Today*, 6(6):22–23, 2005.

[72] J. Smagorinsky. General circulation experiments with the primitive equations - I. The basic experiment. *Monthly Weather Review*, 91(3):99–164, March 1963.

[73] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 SIGCOMM Conference*, volume 31, pages 149–160. ACM, 2001.

[74] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11(1):17–32, February 2003.

[75] E. Verdin, D. A. Beugre, C. Ania, J. Parra, D. Toye, P. Marchot, M. Crine, and A. Léonard. Simulation de l'écoulement dans des filtres à charbons actifs par la méthodologie des réseaux de Boltzmann. In *Actes du 12ème Congrès de la Société Française de Génie des Procédés*, number 98 in Récents progrès en Génie des procédés. Lavoisier - Technique et Documentation, 2009.

[76] C. Walshaw and M. Cross. Mesh partitioning: A multilevel balancing and refinement algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

[77] C. Walshaw, M. Cross, and M. G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47(2):102–108, December 1997.

[78] S. Wolfram. Cellular automaton fluids 1: Basic theory. *Journal of Statistical Physics*, 45(3-4):471–526, 1986.

[79] C. Xu, F. C. M. Lau, and R. Diekmann. Decentralized remapping of data parallel applications in distributed memory multiprocessors. *Concurrency: Practice and Experience*, 9(12):1351 – 1376, December 1998.

[80] B. Y. Zhao, L. Huang, S. C. Rhea, J. Stribling, A. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

[81] Q. Zou and X. He. On pressure and velocity boundary conditions for the Lattice Boltzmann BGK model. *Physics of Fluids*, 9:1591–1598, 1997.

Appendix A

Agent and Error Handler Class Diagrams

Figure A.1 gives the class diagrams of `Agent` class and `ErrorHandler` interfaces. The members of `Agent` class have the following definition:

- `queue`: the events queue of the agent.
- `agentThread`: the thread that executes the agent.

The methods of `Agent` class have following definition:

- `start()`: instantiates `agentThread` thread and starts it if the agent has not already been started.
- `stop()`: puts a stop event into agent's events queue.
- `join()`: waits until `agentThread` thread finishes its execution.
- `run()`: instructions executed by `agentThread` thread. These instructions represent the execution of the agent (initialization, events handling and closing).
- `signalError(error : Throwable, agent : Agent)`: implements the `ErrorHandler` interface by putting an error event into message queue. The members of the error event include `error` and `agent`.
- `submitEvent(e : Object)`: called by subclasses to put an event into queue.

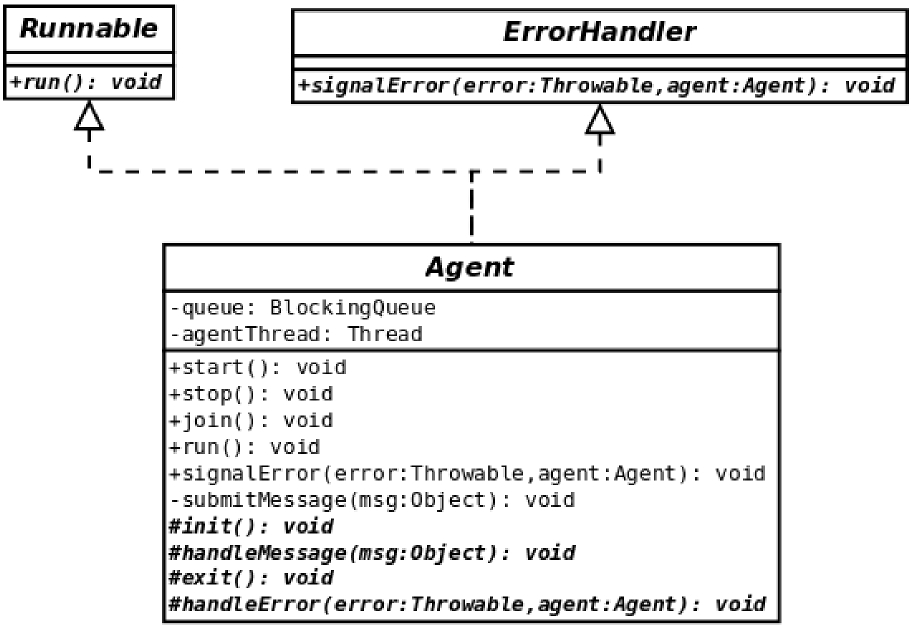
Runnable
+run(): void
ErrorHandler
+signalError(error:Throwable,agent:Agent): void
Agent
-queue: BlockingQueue
-agentThread: Thread
+start(): void
+stop(): void
+join(): void
+run(): void
+signalError(error:Throwable,agent:Agent): void
-submitMessage(msg:Object): void
#init(): void
#handleMessage(msg:Object): void
#exit(): void
#handleError(error:Throwable,agent:Agent): void

Figure A.1: Class diagrams of `Agent` and `ErrorHandler`.

Following methods have to be implemented by subclasses of `Agent`:

- `init()`: instructions of initialization phase of agent.
- `handleMessage(e : Object)`: handling code for `e` event extracted from messages queue.
- `exit()`: clean-up instructions executed when agent is closing.
- `handleError(error : Throwable, agent : Agent)`: handling code for a signaled error.

`Agent` class features no public method that allows another class to submit an event (except `stop` and `signalError` that are implemented by the insertion of a message into queue). It is the responsibility of subclasses to declare event submission methods and thus specify what kind of event can be submitted to the agent.

Appendix B

LaBoGrid's XML Configuration File

As a reminder (see Section 4.5), The XML configuration file is composed of three parts:

1. LB configurations,
2. Processing chains description,
3. Simulations description.

B.1 LB Configurations

This part contains a sequence of LB configurations. Each LB configuration has a unique identifier (a string) and provides:

- a specific lattice class name and lattice's size,
- a specific solid class name, a file name and the content type of the given file (binary or text),
- a specific partitions generator class name and the number of partitions to generate.

The size of the lattice must be equal to the size of given solid.

Figure B.1 shows an example of a single LB configuration. The associated LB simulation is executed on a $(32,32,32)$ D3Q19 lattice divided into 8 sublattices. The solid matrix is read from a file called "bin.solid" containing a binary representation.

```
<LBConfiguration id="conf0">
   <Lattice
      class="lb.lattice.d3.q19.D3Q19DefaultLattice"/>
      size="(32,32,32)"
   <Solid
      class="lb.solid.d3.D3SolidBitmap"
      fileId="bin.solid"
      type="bin"/>
   <SubLattices
      generatorClass="lb.modelGraph.d3.D3CuboidsGenerator"
      subLatticesCount="8"/>
</LBConfiguration>
```

Figure B.1: An example of LB configuration.

B.2 Processing Chains Description

An operator is described by its class name and a parameters string. The parsing of this parameters string is the responsibility of the operator.

The description of a logger is composed of:

- logger class name,
- logger parameters string,
- refresh rate,
- logger identifier,
- logger output class name (to choose the type of logging output),
- logger output parameters string.

Figure B.2 shows an example of processing chain description. This chain implies following operations each iteration:

1. send outgoing densities,
2. apply in-place propagation on lattice,

```
<ProcessingChain id="proc0">
   <Operator
      class="laboGrid.operators.BorderSender"
      parameters=""/>
   <Operator
      class="laboGrid.operators.InPlaceStream"
      parameters=""/>
   <Operator
      class="laboGrid.operators.BorderFiller"
      parameters=""/>
   <Operator
      class="lb.operators.d3.q19.D3Q19PressureOperator"
      parameters="0 1.001 0.099"/>
   <Operator
      class="lb.operators.d3.q19.D3Q19SRTCollider"
      parameters="1 0 0 0 0 20"/>
   <Logger
      loggerClass="laboGrid.logging.loggers.IterationLogger"
      loggerParameters=""
      rate="10"
      id="itLog"
      clientClass="laboGrid.logging.output.LocalFileLogOutput"
      clientParameters="/home/user/logs/"/>
</ProcessingChain>
```

Figure B.2: An example of processing chain description.

3. set incoming densities with received data,
4. apply pressure boundary conditions,
5. apply SRT collision operator,
6. log current iteration every 10 iterations to a local file.

B.3 Experiment Description

Figure B.3 shows an example of experiment description. This experiment is composed of two simulations sequences. First sequence contains three simulations:

1. First simulation lasts 3000 iterations using lattice, solid and partitioning defined by LB configuration "conf0". It applies processing chain "proc0". Starting iteration equal to 0 means that the fluid is initially at rest. The solid file is read from given input.
2. Second simulation uses as initial conditions the result from previous simulation and lasts 1000 iterations. Another processing chain ("proc1") is used.
3. Third simulation uses as initial conditions the result from previous simulation and lasts 1000 iterations. Processing chain "proc1" is used again but another LB configuration is set (partitioning and solid data are ignored, only the type of the lattice is taken into account).

Second sequence contains only one simulation: it uses as initial conditions the result of a previous simulation stored in folder "/home/user/io/in/". It starts its execution at iteration 5000 (this means the result used as initial conditions should be obtained with a simulation of 5000 iterations) and lasts 10000 iterations. The result of the simulation is stored in the folder "/home/user/io2/".

```
<Experiment>
   <SimulationSequence>
      <Simulation
         iterationsCount="3000"
         lbConfiguration="conf0"
         processingChain="proc0"
         startingIteration="0">
         <Input
            class="laboGrid.simIO.LocalFileInput"
            parameters="/home/user/io/"/>
      </Simulation>
      <NextSimulation
         iterationsCount="1000"
         lbConfiguration="conf0" processingChain="proc1"/>
      <NextSimulation
         iterationsCount="1000"
         lbConfiguration="conf1"
         processingChain="proc1">
         <Output
            class="laboGrid.simIO.LocalFileOutput"
            parameters="/home/user/io/"/>
      </NextSimulation>
   </SimulationSequence>
   <SimulationSequence>
      <Simulation
         iterationsCount="10000"
         lbConfiguration="conf1" processingChain="proc1"
         startingIteration="5000">
         <Input
            class="laboGrid.simIO.LocalFileInput"
            parameters="/home/user/io/in/"/>
         <Output
            class="laboGrid.simIO.LocalFileOutput"
            parameters="/home/user/io2/"/>
      </Simulation>
   </SimulationSequence>
</Experiment>
```

Figure B.3: An example of experiment description.