Sequential/parallel reusability study on solving Hamilton-Jacobi-Bellman equations

Florian Dang

UNIVERSITÉ DE VERSAILLES
SAINT-QUENTIN-EN-YVELINES

École doctorale Versailles Saint-Quentin-En-Yvelines

Sequential/parallel reusability study on solving Hamilton-Jacobi-Bellman equations

THÈSE

présentée et soutenue publiquement le 22 juillet 2015

pour l'obtention du

Doctorat de l'Université de Versailles Saint-Quentin-En-Yvelines

(spécialité Informatique)

par

Florian Dang

Composition du jury

Rapporteurs :	M. Vassil ALEXANDROV	Professeur des Universités, Barcelona Supercomputing Center
	M. Damien TROMEUR-DERVOUT	Professeur des Universités, Université Claude Bernard Lyon
Examinateurs :	Mme Laura GRIGORI	Directrice de recherche, INRIA, Laboratoire J.L. Lions
	M. William JALBY	Professeur des Universités, Université de Versailles Saint-Quentin en Yvelines
	M. Philippe RAVIER	Docteur-ingénieur, Directeur R&D Silkan
Directrice :	Mme Nahid EMAD	Professeur des Universités, Université de Versailles Saint-Quentin en Yvelines

Laboratoire PRISM

SILKAN

UNIVERSITÉ DE VERSAILLES
SAINT-QUENTIN-EN-YVELINES

École doctorale Versailles Saint-Quentin-En-Yvelines

Etude de la réutilisabilité séquentielle/parallèle pour la résolution des équations Hamilton-Jacobi-Bellman

THÈSE

présentée et soutenue publiquement le 22 juillet 2015

pour l'obtention du

Doctorat de l'Université de Versailles Saint-Quentin-En-Yvelines

(spécialité Informatique)

par

Florian Dang

Composition du jury

Rapporteurs :	M. Vassil ALEXANDROV	Professeur des Universités, Barcelona Supercomputing Center
	M. Damien TROMEUR-DERVOUT	Professeur des Universités, Université Claude Bernard Lyon
Examinateurs :	Mme Laura GRIGORI	Directrice de recherche, INRIA, Laboratoire J.L. Lions
	M. William JALBY	Professeur des Universités, Université de Versailles Saint-Quentin en Yvelines
	M. Philippe RAVIER	Docteur-ingénieur, Directeur R&D Silkan
Directrice :	Mme Nahid EMAD	Professeur des Universités, Université de Versailles Saint-Quentin en Yvelines

Laboratoire PRISM

SILKAN

Remerciements

C'est en français que j'ai le plaisir de placer ces quelques lignes au combien importantes dans la vie d'un thésard. Les sciences m'ont toujours fasciné et c'est naturellement que je me suis orienté vers une thèse. J'ai appris que pour réaliser une thèse, exercice long et méandreux, il fallait faire preuve d'une certaine persévérance. La difficulté d'une thèse, ne réside pas uniquement dans les raisonnements alambiqués auxquels on se retrouve souvent confronté. Je suis l'un de ces thésards qui admet avoir connu quelques états passagers de découragement, où l'on ressent cette étrange sensation de marcher dans le noir sans pouvoir atteindre la lumière. Ma motivation n'aurait jamais pu perdurer dans cette grotte sans cette passion qui m'anime pour ce domaine et sans mon entourage. Je remercie ainsi toutes les personnes qui se sont impliquées de loin comme de près à la confection de cette thèse, fruit de quatre longues années de reflexion.

J'aimerais tout d'abord remercier vivement ma directrice de thèse, Madame Nahid Emad de l'Université de Versailles, pour avoir supervisé ma thèse pendant toutes ces années. Je suis ravi d'avoir travaillé en sa compagnie, elle qui, toujours présente lorsque j'avais besoin d'aide, a su me soutenir et me conseiller.

Je tiens à remercier Messieurs Vassil Alexandrov et Damien Tromeur-Dervout, qui m'ont fait l'honneur d'être les rapporteurs de ma thèse. Leurs remarques pertinentes m'ont permis d'enrichir mon travail. J'exprime mes remerciements à l'ensemble des membres de mon jury : Madame Laura Grigori, Messieurs Philippe Ravier et William Jalby.

Je remercie infiniment Pierre Fiorini de Silkan, à l'origine de ce sujet si captivant, qui m'a permis de réaliser cette thèse. Je remercie les collègues de Silkan, qui ont toujours su m'accueillir à bras ouverts, en particulier Philippe Ravier, qui m'a prodigué des conseils avec un oeil différent. Merci encore à Silkan, sans qui cette thèse CIFRE n'aurait pas eu lieu.

Je remercie les Institutions académiques, mon école doctorale l'Université de Versailles Saint-Quentin-en-Yvelines ainsi que la maison de la simulation, où j'ai trouvé des collègues, élèves, amis. Merci donc à Zifan, Thomas, Maxime, Makarem, Miwako, Alexandre, Fan, Tarek, Langshi, Cihui, Pablo... La vie de recherche académique et d'enseignement est passionante. Je n'oublie pas les membres de l'équipe du laboratoire PRiSM, ainsi que les services de l'école doctorale en particulier Mme Delahaye qui est d'une efficacité redoutable.

Je n'oublie pas non plus la ACT team, mes collègues actuels, qui m'ont supporté, ils savent ce que c'est !

Merci à mes amis qui ont eu la délicatesse (ou pas) de me demander à chaque fois comment avançait ma thèse.

Merci à ma famille, mes proches qui sont toujours présents. Merci à mes parents, mes beaux-parents, mes soeurs...

Et last but not least, merci Iris pour avoir pu me porter jusqu'au bout, essuyé les moments difficiles tout comme les moments de joie. Cette thèse à ton empreinte.

Cảm Ơn !

Je dédie cette thèse
à mon père.

Contents

List of Figures

Introduction

1 Numerical era : modeling, discretization and simulation

In everyday life, scientists try to represent the world as governed by physical laws. They aim to explain how the world surrounding us is working.

For instance, numerical simulation is a field which can help us predict complex physical phenomena such as weather forecasting with the help of computer technology. We can divide three main parts in the process of numerical simulation :

- Modeling which is the process to interpret the behaviour of a complex system by using mathematics.
- Discretization where we translate the mathematical continuous models into discrete problems.
- Simulation which is the process of running the model on computers to obtain a result.

Mathematical modeling methods based on partial differential equations (PDEs) form an important part of contemporary science and play a key role in engineering and scientific applications. It reveals to be a challenge to implement fast solvers, especially for large scale simulations where simulations of such applications implies to work on consequent amount of data and computations Such problematics can be handled with the help of High performance Computing (HPC). HPC can be seen roughly as the science of supercomputing, and the art of computing simulations efficiently in parallel. By reducing time and cost, great advances in biology, chemistry, aeronautics, ecology, economy would not have occured without the simulation of PDEs with HPC. More specifically, Hamilton-Jacobi-Bellman equations, which are a class of PDEs, intervene in a wide range of applications, such as in aeronautics where we want to plan the trajectory of a plane, or in medical imaging where we want to help to propose visual representations of a body. In this dissertation, our aim is to propose numerical methods which would fit on parallel computers in order to simulate problems involving these equations.

1.1 From mathematical modeling to simulation

Numerical simulations are often based on mathematical models. To model physical phenomena such as heat transfer, wave propagation, structural stresses, or fluid dynamics, it is common to pose the involved problems as partial differential equations. A computer is not capable to solve anatically equations as a human could do. Interpreting these equations require to decompose them into discrete problems through a discretization process. Three most popular numerical techniques for solving partial differential equations include the finite difference, the finite element and the finite volume methods. In the finite difference approximation, the derivatives in a differential equation are replaced by difference quotients. The difference operators are usually derived from Taylor series and involve the values of the solution at neighbouring points in the domain. After taking the boundary conditions into account, a system of equations is solved over the domain points. The finite

differences method (FDM) is intuitive and quite straightforward to implement on regular domains. Unfortunately this method is difficult to apply for problems involving irregular geometries or unusual boundary conditions. The finite element method (FEM) provides an alternative which is better suited for such problems. In contrast to finite difference techniques, the finite element method divides the solution domain into simply shaped regions. An approximate solution for the PDE can be developed for each of these elements. The total solution is then generated by gathering the individual solutions while ensuring continuity at the interelement boundaries. The finite volume method (FVM) may also be applied on unstructured meshes. In this scheme the solution is represented as a series of piecewise constant elements. The discretised form of the PDE is found by integrating the equation over the elements (control volumes). For each control volume the area integral is converted into a line integral over its edges and the numerical flux at the boundaries also calculated.

1.2 Simulation of Hamilton-Jacobi-Bellman equations

Roughly, in this dissertation, the mathematical modeling would correspond to the problem interpretation into Hamilton-Jacobi equations, the discretization part to the finite element scheme and fast marching methods used to approximate accurate solutions, and the simulation aspect to the execution and results of our solver.

Numerical simulation is nowadays strongly linked with high performance computing (HPC). We are solving more and more complex problems which can require tremendous amount of data and/or computations. Adding to this complexity, recent years have shown that exploiting fully the computing capabilities of recent hardware architectures is not a simple task and require knowledge, skills regarding parallel programming. We present in the next section a brief introduction on HPC.

2 Intensive computing on parallel architectures

How can we describe computer evolution without mentionning Gordon Moore ? Moore originally stated in 1965 in an article entitled “Cramming more components onto integrated circuits” [Moore, 1965] that the number of transistors would increased at a rate of roughly a factor of two per year. This statement is commonly referred as Moore’s law and has remained applicable to the semiconductor industry for almost 50 years since its publication in 1965. The rate has kept being corrected through year but the idea behind is still be considered valid.

Since 2004, the clock speed and the central processing units (CPU) power of Intel and AMD processors has stopped increasing (fig. 1). Increasing the clock speed implies to increase energy consumption and heat dissipation which can become an important issue. Manufacturers have found a workaround by integrating multi-cores. Making these cores work in parallel reveals to be much more beneficial. Nowadays, the current trends have now moved from dual, tri, quad, hex, oct-core chips (multi-cores) to even hundreds of cores (many-cores).

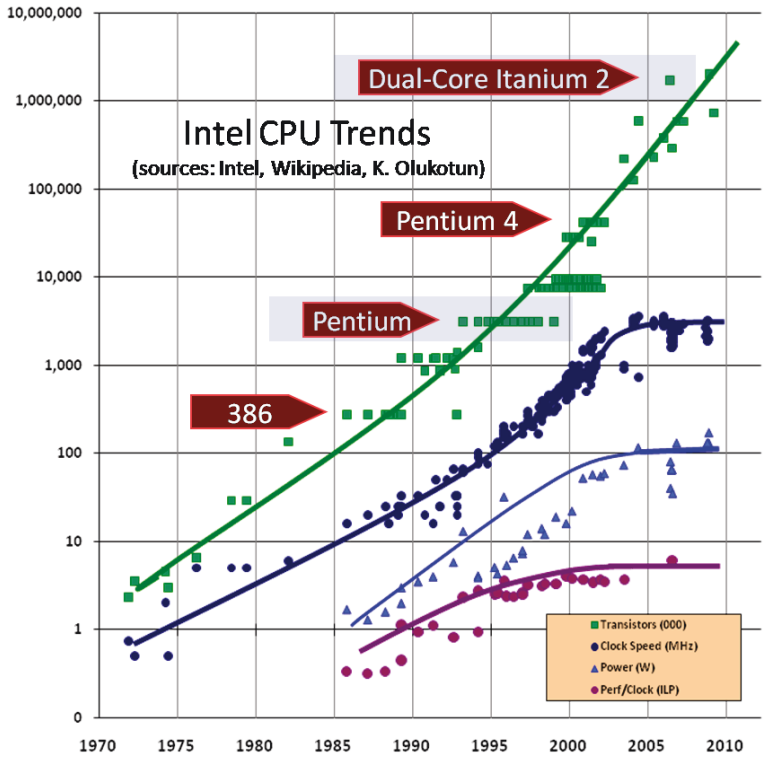

Figure 1: Intel CPU trends from 1970 to 2010

2.1 "The free lunch is over"

Taking advantages of the parallelism of each architecture can reveal to be quite a challenge, an application can not be necessarily parallelized and generally parallelized differently according to the environment. In March 2005, Herb Sutter started his article [Sutter, 2005] with the following words : "Your free lunch will soon be over" and adds later that "concurrency is the next major revolution in how we write software". The apparition of multi-core architectures have changed the way how programmers should write softwares. Prior to this date, programmers are poorly concerned about concurrency/parallelism and they mostly rely on Moore's law. Parallel architectures surround our every day life (laptop, cellphones, embeddded systems...) and we generally do not fully exploit them. With the advent of multi-cores, programmers have to be much more involved if they want to get the performance that their parallel architectures offer.

These trends do not only affect personal computers but also invades the embedded system world with current high-end smartphones which include vector processing units and also graphics processing unit (GPU).

Regarding the High Performance Computing (HPC) world, it becomes easy to see in the top500 list [Top500, 2014] that recent supercomputers own a wide variety of hardwares which have parallel architectures. Nowadays, supercomputers can reach 10^{15} operations per second (petaFLOPS) whereas fifteen years ago they hardly reach 10^{12} operations per second (teraFLOPS). Given the performance development shown in figure 2, supercomputers are projected to reach 10^{18} operations per second (exaFLOPS) by 2020.

These supercomputers are quite representative of the hardware evolution. Parallelism is multi-level and we can outline four groups :

- shared-memory systems ;
- distributed-memory systems ;
- hierarchical (hybrid) systems ;
- and hardware accelerators.

Basically, in shared memory systems, several processors can access globally to a shared memory. In distributed memory systems, each processor has its own private memory and potentially needs to communicate data. Hierarchical (hybrid) systems combine both distributed and shared memory. Nodes of a cluster in a hierarchical system can access to a large shared memory in addition to each node's limited non-shared private memory. Hardware accelerators such as graphical processing units (GPUs), field-programmable gate arrays (FPGAs), application-specific integrated circuits (ASICs) are specialized hardwares which are separated from the CPU. Depending on the algorithms used, the environnement available, one level of parallelism can be preferred over an other one. For instance, image processing algorithms such as image convolution might likely be more efficient on GPUs whereas ranking algorithms such as PageRank from Google would be more fitted on distributed memory clusters. One challenge in HPC is to investigate ways for some applications to combine efficiently different levels of parallelism in order to exploit fully the possibilities of the targetted system.

2.2 Parallel programming models

The needs in parallel computing are numerous. In adding to high performance computing purpose, it is now necessary in order to reduce power consumption and heat dissipation. Efficient parallel programs can be difficult to develop as we have mentionned previously. Nowadays, developpers are much more concerned about parallelism issues and can rely on several programming models which target specific level of parallelism.

Shared memory systems A shared-memory system consists of at least one multi-core processor or CPU, sharing the memory available on the system, meaning that all CPUs can access the same physical address space (see fig. (3)).

CPUs are in general multi-core nowadays. Therefore many programmers are concerned about ways to obtain parallel codes which can run on recent processors. Designing a scalable multi-core application is often challenging on architectures which has several cores. Traditionnaly, one way to exploit parallelism at this level is to use threads. Threads are attached to a processor, executed and potentially use shared-memory data. Threads can be created directly through libraries proposed by the operating system such as Threads POSIX (PThreads) for UNIX system. However, writing multithreaded programs can become cumbersome using PThreads since it is a low level API. Higher level abstraction APIs have appeared since such as OpenMP [Brunschen and Brorsson, 2000] or Intel

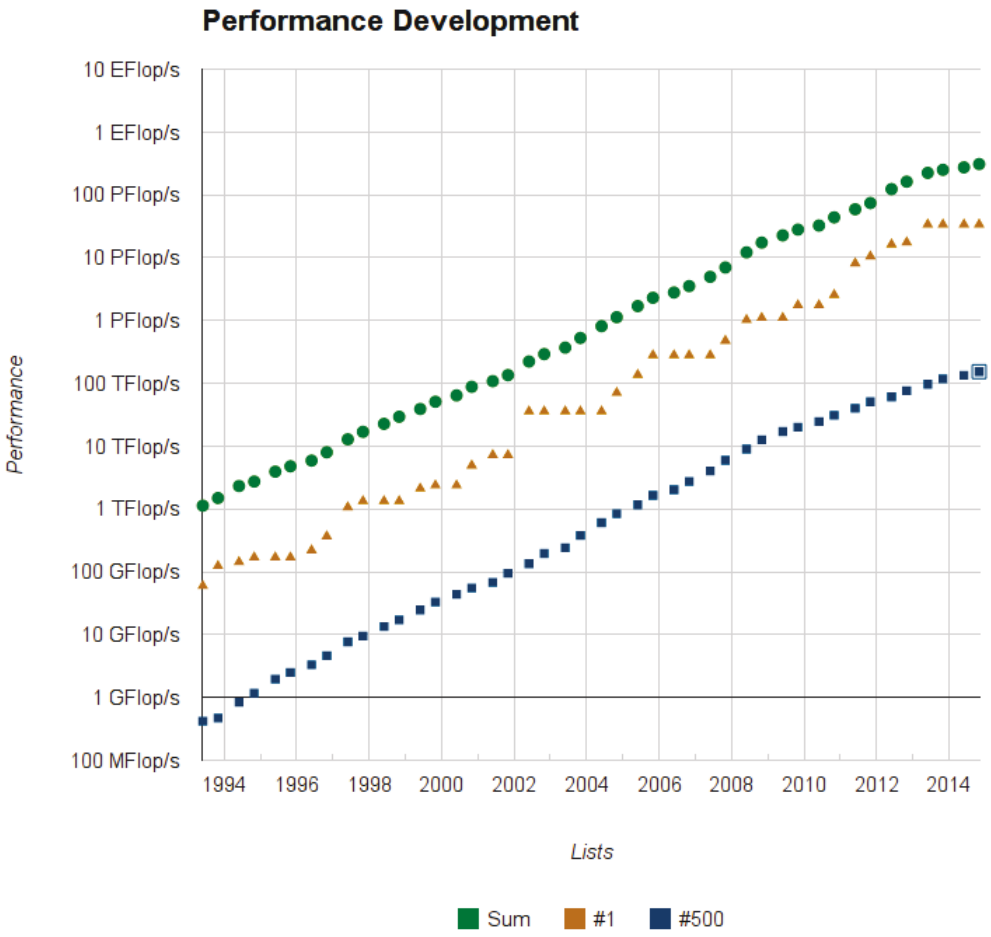

Figure 2: Evolution of the 1st, the 500th and the sum performance of supercomputers (source : top500.org)

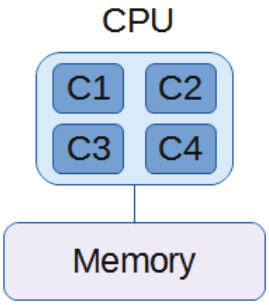

Figure 3: A unified memory access (UMA) system with a shared memory

Threading Building Blocks (TBB) [Reinders, 2007] which help popularize multi-core parallelism. Some of advantages to use these APIs include faster parallelization of codes with minor modifications, code portability (cross platform), and ease of use. However, achieving reasonable scaling for high core numbers is not a simple task.

Distributed memory systems A distributed-memory model connects different processors via a communication network. In a distributed-memory model, the memory is physically and logically distributed among individual processing units as illustrated on figure 4.

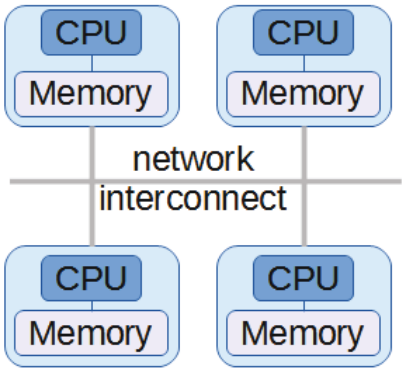

Figure 4: A distributed memory model

Any computation which can be done in parallel can also be realized by distributing work among these multiple computing nodes. Task parallelism is the simultaneous execution on multiple cores of many different functions across the same or different datasets. There are no restrictions on the types of operations each computing node can perform. However, one of the major pros of distributed systems is the overhead due to the communications. Indeed supporting distributed memory systems imply to adopt a paradigm based on message passing which can become costly if the nodes have to communicate often.

The message passing interface (MPI) [Forum, 1994] became the defacto standard for distributed memory systems and a dominant model used in HPC. MPI is a message-passing application programmer interface including point-to-point and collective communication which allows a wide range of abilities.

Hardware accelerators The advent of accelerators is rather recent. "Classic" parallel systems can be quite expensive, and the idea to reuse existing mainstream technologies such as graphic cards have emerged. The usage of such accelerators is quite different from what they were originally designed for. Graphics processing units (GPUs) are a typical example of accelerating hardwares which were initially designed for specific compute-intensive display function. General-purpose computing on graphics processing units (GPGPU) refers to the use of a GPU to perform computations which are traditionally done by the CPU. GPGPU has since played an important role in the use of parallel computing among scientifics. The hardware architectures of accelerator are many-cores

(including FPGAs and ASICs), devoting proportionally more transistors to arithmetic logic units, less to caches and flow control in comparison to CPUs. GPUs also typically have higher memory bandwidth compared to CPUs. Many problems are embarassingly parallel, and candidate for data parallelism. Briefly, data parallelism is the simultaneous execution on multiple cores of the same function across the elements of a dataset. One framework that is used to accomplish data-parallelism is "single instruction, multiple data" (SIMD), in which multiple processors execute the same instructions on different pieces of data. SIMD parallelization suits well accelerators such as GPUs, since flow control computation can be shared among processors.

About hybrid parallelism Every parallel paradigm has its pros and cons regarding a specific problem. Nowadays parallel architectures are hybrid and heterogenous. There are different muti-level parallelization possibilities which can be exploited by combining different parallel paradigms. Hence, developers need the knowledge to make them work together in order to exploit at the fullest the targetted architectures. Writing efficient code is a must go and reusability study should also be a concern.

2.3 On reusability and sequential/parallel reusability

Reusable code is essential to avoid duplicated effort in software development. Instead of rewriting software components from scratch, a programmer can make use of an existing component. Reusable code helps to obtain performance by making it possible to put reusable software components in specific libraries. We can therefore reuse code from specialists in their respective areas without being stuck to invent the wheel again. The concept of code reusability is not new. In a report from 1993 [Andreae et al., 1993], the authors make the distinction between two issues : the process of how to reuse code and the process of how to write more reusable code. The report focus on the way to write more reusable code which is our main concern. Software reusability has been a concern since we had to design consequent codes. The needs for industrial codes are numerous their importance differ according to the context. Codes have to be efficient in term of speed, relatively easy to use, maintainable, elegant (avoid code duplication, concise code)... This can be achieved by following software design strategies such as flexibility, modularity, orthogonality, genericity and sustainability.

Flexibility can be seen as the ease with which a system or its components can be modified for use in applications or environments other than those for which it was specifically designed [Eden and Mens, 2006]. A flexible software has the ability to adapt to future changes. Modularity emphasizes separating different functionalities of the sofware into independent and interchangeable modules. A modular system is far more reusable compared to a monolithic system since most of the modules can be reused. In an orthogonal design, modifying a component action neither creates nor propagates side effects to other components. Genericity proposes to write common code for functions, structures which can handle different types. Genericity can be achieved through abstraction which manage different levels of complexity and hide the most complex ones to the users. High level abstractions increase ease of use to the user. Finally, sustainability is the process to write

a code which would last long and would have the potential to be reused over time without (or with few) modifications.

Sequential/parallel reusability The tremendous variety of parallel architectures make the work harder for the programmer to aim for a generic code which could be efficient and parallelizable on any kind of architecture. Indeed, the different parallel paradigms show that parallelism is multi-level and approriate parallel algorithms should be chosen to target a specific parallel architecture. Sequential/parallel reusability tries to follow the design features mentionned previously and find a way to write a sustainable unique code which could work on several serial and parallel architectures without sacrifying performance.

The main objective of this Ph.D. dissertation is to define strategies for the design of reusable parallel libraries allowing to solve HJB equations. These strategies have to permit the reuse of sequential code into parallel contexts while enabling performance tuning.

3 Outline

The general idea is to design a library which would be able to follow several points where the library :

- should solve numerically classes of HJB equations ;
- should propose several solving methods and local schemes to the user ;
- should be efficient and exploit multi-level parallelism ;
- should tend to be reusable in terms of software reusability and also sequential/parallel reusability.

The first part of this document is dedicated to a recent state-of-the-art of the numerical resolution of HJB equations. We present in chapter 1 a brief overview of the mathematical theory behind the HJB equations. We emphasize on the widely known eikonal equation which is a particular case of HJ equations and we present some of its possible applications. In chapter 2, we focus on the numerical process to discretize the problem and approximate solutions of HJ equations. Locally, a first order Godunov scheme can be used which can be combined with global resolution method such as fast marching methods. We detail in chapter 3 the algorithms used in the latter methods and its derivates. We especially present the fast iterative method (FIM), which has a strong potential towards parallel computation.

The contribution of this dissertation is detailed in the second part. Chapter 4 is dedicated to the elaboration of new parallel methods for the FIM. Firstly, we show the validity of our early implementations illustrated with some applications. We then investigate the recent state-of-the art regarding parallel fast marching methods which show relatively poor results regarding their parallel scalability. Hence the idea to investigate other derivate methods such as the fast iterative method which is a method with a higher parallel potential. We propose a new method based on the fast iterative method, the

buffered fast iterative method (BFIM), whose strategy is based on active list partitioning. The parallel design can be extended for managing broader classes of HJB equations such as the semi-ordered fast iterative method which can handle anisotropic front propagations. We show promising results on parallel FIM and parallel SOFIM which prove that the proposed parallel strategies scales well particularly on shared-memory architectures. In chapter 5, we detail our library implementation and the steps achieved towards reusability both in terms of software reusability and sequential/parallel reusability. Sequential/parallel reusability is a recent concern and we propose some solutions to achieve compromises between performance, abstraction and maintainability.

Part I

Numerical solution for Hamilton-Jacobi equations

Chapter 1

Hamilton-Jacobi-Bellman equations

Contents

1.1 Hamilton-Jacobi equations

Partial differential equations (PDEs) are used to describe a large range of physical phenomena. Hamilton-Jacobi (HJ) and Hamilton-Jacobi-Bellman (HJB) equations constitute classes of PDEs where their numerical resolution is investigated throughout this thesis. We present the eikonal equation which is a well known special case of HJ equations. We then introduce the notion of viscosity solution and we finally give some real word applications.

1.1.1 The eikonal equation

The eikonal equation is widely used to simulate the propagation of a wave-front (fig. 1.1) or in general an interface. An interface is defined as the surface separating the area inside of the region from the area outside of the region. There are two different types of motion : an interface which is strictly expanding or contracting can be described by the boundary

value formulation whereas an interface which arbitrarily expands and contracts can be described by the initial value formulation.

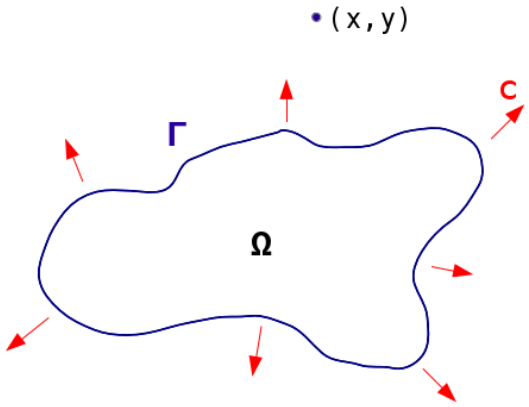

Figure 1.1: Front propagation problem

Let us take the example of a front propagation problem. In figure 1.1, c is a know speed (or velocity) function which can depend on several factors such as curvature, normal direction, shape of the front or other environment properties. We want to determine the first arrival time u as it crosses a point x in the domain. We then define $u(x) = \underset{t>0}{inf}(\Gamma_t)$ with $x \in \Gamma_t$. Therefore, we can describe the first arrival times of a moving interface with the following formulation (eq. (1.1)) :

$$\begin{cases} c(x).|\nabla u| & = 1, x \in \Omega \subset \mathbb{R}^n, c(x) > 0 \\ u(x) & = \phi(x) \text{ where } \phi : \Gamma \subset \partial\Omega \to \mathbb{R} \end{cases} \tag{1.1}$$

where Ω is an open subset of $\mathbb{R}^n$, c a positive speed function, Φ is a known function describing a surface Γ, ∇u represents the gradient of u and $|.|$ the Euclidian norm.

One interpration of the eikonal equation (and Hamilton-Jacobi equations) is to see a wave front which propagates by trying all possible directions from all possible point sources along a front. The sources and directions which propagate the wavefront forward the fastest are used to determine the future position of the wavefront. At every iteration, the wavefront tries every possible trajectories from the original source, and the position of the wavefront at a particular time is determined by all the places that can be reached in that time by the most efficient trajectories.

1.1.2 Static Hamilton-Jacobi equations

The eikonal equation is a a particular case of Hamilton-Jacobi equations. Given Ω an open domain of $\mathbb{R}^n$, first order static Hamilton-Jacobi equations take the following form :

$$H(x, u(x), \nabla u(x)) = 0, x \in \Omega \subset \mathbb{R}^n \tag{1.2}$$

where H is a continuous Hamiltonian defined on $\Omega \times \mathbb{R} \times \mathbb{R}^N$ and ∇u represents the gradient of u. The reader is invited to refer to the books [Barles, 1994, Bardi and Capuzzo-Dolcetta, 1997]

for more details on the theoritical aspects which are not detailed in this thesis. Given $H(x, \nabla u(x)) = c(x).|\nabla u(x)| - 1$ we get back the eikonal equation.

Viscosity solutions

Equation (1.2) is in general not well-posed (the solution does not necessarily exist or is not unique). It is possible to show several examples in which any classical solution (that is of class C^1 i.e. function is derivable and its derivate is continuous on its domain) exists or infinite weak solutions exist. A simple illustration of the problem on figure 1.2 is that at a given point it is possible to have a family of solutions which do not necessarily represent the physically correct evolution.

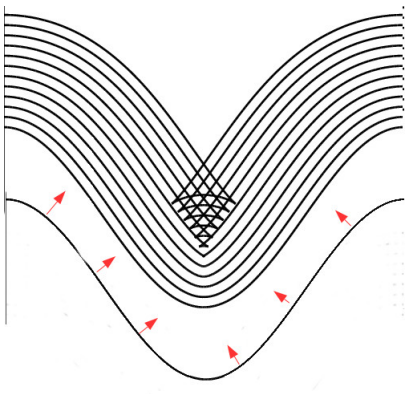

Figure 1.2: Swallow tail problem

In a one dimension case, we consider a very simple unidimensional HJ equation (here an eikonal equation) such as the following :

$$\begin{cases} |u'(x)| & = 1, x \in (-1, 1) \\ u(x) & = 0, x = \pm 1 \end{cases} \tag{1.3}$$

A general solution would be $u(x) = \pm x + C$ with C a constant to determine satisfying boundary conditions. Taking $u(x) = 1 - |x|$ satisfies the boundary solutions but not for the value $x = 0$ where the solution is not unique. We can find infinite multiple weak solutions which satisfy the differential equation almost everywhere. The following figure (1.3) illustrates two possible weak solutions.

The theory of viscosity solutions was developed in order to overcome these problems. We add a small viscosity term to equation (1.3) which gives a second-order equation :

$$\begin{cases} -\epsilon u_\epsilon(x)'' + |u_\epsilon(x)'| & = 1, x \in (-1, 1) \\ u_\epsilon(x) & = 0, x = \pm 1, \epsilon \geq 0 \end{cases} \tag{1.4}$$

Equation 1.4 has a unique solution with take the form :

$$u_\epsilon(x) = 1 + |x| + \epsilon e^{\frac{-1}{\epsilon}}(1 - e^{\frac{1-|x|}{\epsilon}})$$

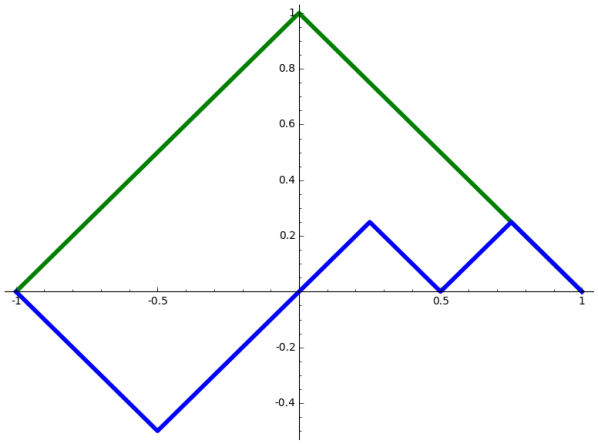

Figure 1.3: Two weak solutions for eq. 1.3

The solution of equation (1.3) can be recovered as the limit function $u = \lim_{\epsilon \to 0} u_\epsilon$ which converges to the viscosity solution. We will not detail the general case here but we will define some notions which would be useful in further sections such as subdifferential

Definition 1. We call respectively the super and subdifferentials of u at $x \in \Omega$ the close convex subspace of $\mathbb{R}^n$ defined as :

$$\begin{cases} D^+u(x) & = \{p \in \mathbb{R}^n | \limsup_{y \to x} \frac{u(y)-u(x)-(p,y-x)}{|y-x|} \leq 0\} \\ D^-u(x) & = \{p \in \mathbb{R}^n | \limsup_{y \to x} \frac{u(y)-u(x)-(p,y-x)}{|y-x|} \geq 0\} \end{cases}$$

Subdifferentials are necessary when computing with finite elements. We see later that their approximations imply to use mathematical series.

Definition 2. u is a *viscosity subsolution* of equation (1.2) if $\forall \varphi \in C^1(\Omega), \forall x_0 \in \Omega$ local maximum of $u - \varphi$,

$$H(x_0, u(x_0), \nabla u(x_0), \nabla^2 u(x_0)) \leq 0$$

u is a *viscosity supersolution* of equation (1.2) if $\forall \varphi \in C^1(\Omega), \forall x_0 \in \Omega$ local minimum of $u - \varphi$,

$$H(x_0, u(x_0), \nabla u(x_0), \nabla^2 u(x_0)) \geq 0$$

Definition 3. A continuous function u is a viscosity of equation (1.2) if and only if u is a *viscosity subsolution* and *viscosity supersolution*

Existence and uniqueness of the viscosity solution of equation (1.2) are shown in [Crandall and Lions, 1983, Lions, 1983].

1.1.3 To HJB equations : optimal control problems

The value function of Hamilton-Jacobi-Bellman equations represents the minimum cost for a controlled dynamical system. Richard Bellman introduces the theory of dynamic programming in order to solve these kind of equation. We will not focus on the detailed

mathematical aspects in this thesis of HJB equations which is a wide on-going research. Theories can be found in Crandral and Lions work [Lions, 1983, Crandall and Lions, 1983, Crandall et al., 1984]. HJB equations can be seen as an extension of HJ equations and the methods presented in this thesis are possible candidates to solve HJB equations. Further informations on fast methods and upwind based method working on more sophisticated HJB equations classes can be found in [Cacace et al., 2011, Cacace et al., 2012, Cacace et al., 2013].

1.2 HJB applications in real world

HJB equations arise in a wide range of domains such as finance [Wang and Forsyth, 2008] or chemistry [Dey and Ayers, 2009]. We present in this section a variety of applications where HJB equations can intervene and solved using fast marching methods.

1.2.1 Path planning : robotics, aeronautics

The goal of path planning problem is to find for an entity an appropriate path from a starting point to a destination point in a complex environment. Classic ways to solve the issue is to use A* algorithm which is widely used in path finding problems. The fast marching method (FMM) [1] appears to be a recent alternative and useful if we want to obtain high quality path planning. A comparitive work has been done in [Chiang et al., 2007] where to sum up, A* seems faster to generate continuous line path whereas FMM generates smoother and more precise path depending on the map resolution. In robotics for instance, constraints might apply such as in [Yu et al., 2013, Gomez et al., 2013] where we have to ensure that no collision happens between different robots for instance or that the path is far enough from an obstacle. In [Petres et al., 2005], the authors use the FMM for autonomous underwater vehicles. According to the authors, classic path planning algorithms are not designed to deal with these kind of continous environments. They also extend the FMM to take currents into account implying to make the FMM work both in isotropic and anistropic medias. It is also possible to combine other methods with the FMM such as frothing construction algorithm [Yu et al., 2013], or Voronoi diagram [Garrido et al., 2006].

1.2.2 Computer vision : photometric "stereo"

Photometric stereo is a technique which allow to estimate the surface normals of objects with the help of different lighting conditions using multiple 2D images. A particular case is the shape from shading problem where a single image is used. This case was firstly introduced by B.K.P. Horn in [Horn, 1989] (1989). Rouy and Tourin [Rouy and Tourin, 1992] proposed in 1992 a new approach based on HJB equations. The principle is to find, in certain conditions, an HJB formulation to represent the problem. In [Prados and Soatto, 2005, Yuen et al., 2007], the authors propose to use the fast marching method for the shape of shading problem. The technique uses the pattern of lights and

[1] fast marching methods will be studied in thie dissertation later precisely

shades to infer the shape of the surface. At each point, we can define the brightness map which depends on the reflectivity and the angle between incoming light and the surface normal.

1.2.3 Direct travel times computation

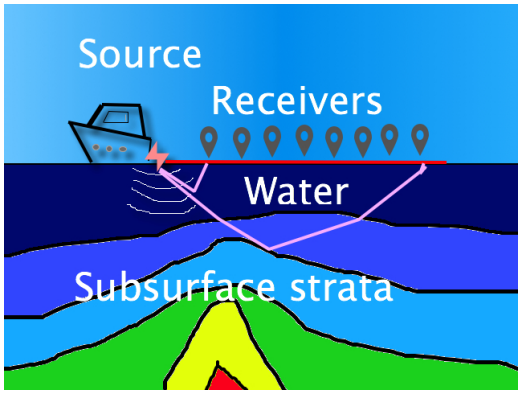

Figure 1.4: Seismic imaging

One direct application of the eikonal equations is to compute direct-arrival traveltimes. On figure (1.4), we represent a source which propagates sound waves which are reflected by the seabed and received to produce seismic image. A well-known way is to use ray tracing methods where the ray equations are solved for the characteristic curves of the eikonal equation. However, ray tracing becomes a burden and require too much computations when penetrating shadow zones. Eikonal solvers automatically extrapolate the wavefront into shadow zones, which are areas that ray tracing finds difficult to image. Indeed, using finite-difference based methods in order to solve the equation allow to avoid problems from ray tracing [Mo and Harris, 2002, Vidale, 1988].

However, as stipulated in [Rawlinson and Sambridge, 2004], where the authors propose to use fast marching method in a complex layered media, the fact that only first arrival traveltimes are computed can be seen as a weakness of eikonal solvers, because later arrivals are often important for high-quality imaging. In [Karlsen et al., 2000] and [Sharifi and Kelkar, 2014], the authors show fast marching method applications for oil reservoir simulations. In [Lelièvre et al., 2010], the authors show that it is possible, also with same method, to compute first-arrival seismic traveltimes on unstructured 3D tetrahedral grids. The works can be extended in seismic imaging to estimate seismic velocity [Cameron et al., 2007].

1.2.4 Image segmentation

Image segmentation is the process to partition an image into multiple regions, giving informations about differents objects, contours which compose the image. There are several methods to solve the issue and the FMM has been recently used for this purpose. Image segmentation using FMM is much used in biomedical domain such as in [Yan et al., 2004]

which analyze lymph node images or in [Roy Cardinal et al., 2003] which propose to work on ultrasound based image. FMM image segmentation also intervene in different contexts such as in historic preservation [Cerimele and Cossu, 2007] where the authors try to preserve ancient monuments by extracting degradation regions.

FMM in image processing keeps improving, in 2008 a generalized fast marching method is proposed [Forcadel et al., 2008], which can handle different velocity signs. The authors propose unassisted video segmentation using the FMM in [Stec and Domanski, 2003].

Chapter 2

Numerical schemes

Contents

2.1 Numerical approximations

As seen in the introduction, the simulation process requires a discretization step when working on continuous problem. Numerical analysis methods to solve PDEs are in constant evolution and based on three main methods : finite difference methods, finite volume methods and finite element methods. The methods presented in this thesis are based on finite difference methods. We first give details about the finite difference scheme used at the local level. We then give a brief overview on global methods such as fast marching methods, fast sweeping methods.

2.1.1 Discretization scheme

There are several different approaches to discretise Hamilton-Jacobi equations such as finite element [Bornemann and Rasch, 2006, Li et al., 2008] and finite difference [Vidale, 1988, Rouy and Tourin, 1992] discretizations. The most common approach is to use upwind finite difference schemes [Sethian, 1999b] since upwind stencils are accurate, efficient, and does not introduce much numerical diffusion. These shemes are detailed in furter subsections.

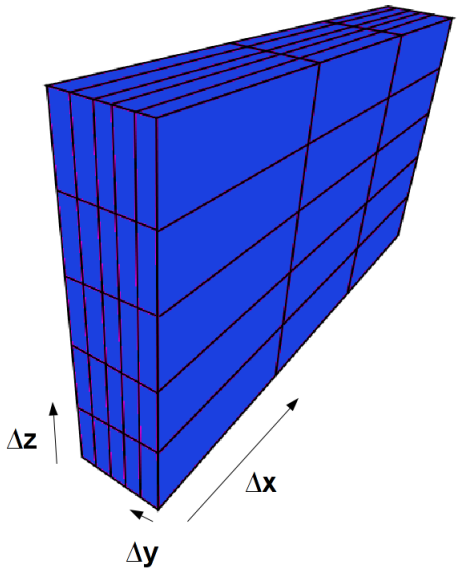

Figure 2.1: A 3D regular grid

Finite element methods decompose the computational domain into a group of cells called mesh or grid. In this thesis, we mainly consider to work on regular meshes (or grids) although most of the methods presented can also work with other meshes types such as unstructured grids [Sethian and Vladimirsky, 2000], or unstructured tetrahedral [Lelièvre et al., 2010]. In regular grids, the spacing, also called step or stride, between vertices regarding a dimension is constant but can differ to the one of an another dimension as shown on figure 2.1. If the strides are the same on every dimension, the cells are unit cubes, the grid is uniform and called a cartesian grid. Structured grids mostly used in finite difference methods since we can store conveniently derivatives of fields and the information describing the grid is minimal. Given a N dimension regular grid G with strides $(\Delta_x, \Delta_y, \Delta_z, ...)$ (fig. 2.1), we can approximate partial differential equations using Taylor series expansion :

$$u(x) = \sum_{n=0}^{\infty} \frac{(x - x_i)^n}{n!} \left(\frac{\partial^n u}{\partial x^n} \right)_i, \; u \in C^\infty$$

Following one dimension (x in this case), we obtain in first order :

$$\begin{cases} u(x + \Delta x) & = u(x) + \Delta x \frac{\partial u(x)}{\partial x} + o(\Delta x) \\ u(x - \Delta x) & = u(x) - \Delta x \frac{\partial u(x)}{\partial x} + o(\Delta x) \end{cases}$$

which gives the first order approximation :

$$\begin{cases} \frac{\partial u(x)}{\partial x} = \frac{u(x+\Delta x) - u(x)}{\Delta x} \\ \frac{\partial u(x)}{\partial x} = \frac{u(x) - u(x-\Delta x)}{\Delta x} \end{cases}$$

We can now note the four differentiation operations for the two dimensional case at (i, j) grid point :

$$\begin{cases} D_x^+ u_{i,j} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x}; & D_x^- u_{i,j} = \frac{u_{i,j} - u_{i-1,j}}{\Delta x} \\ D_y^+ u_{i,j} = \frac{u_{i,j+1} - u_{i,j}}{\Delta y}; & D_y^- u_{i,j} = \frac{u_{i,j} - u_{i,j-1}}{\Delta y} \end{cases} \tag{2.1}$$

with $(u_{i+1,j+1}) = u(x_i + \Delta x, y_j + \Delta y)$ where $u_{ij} = u(x_i, y_j)$.

2.1.2 Local upwind schemes

Upwind discretization methods compute the values using the directions from which the information should be coming. Let $u_{i,j}^k$ be the computed solution at the grid point (i, j) at iteration or time step k. We have the following explicit schemes in time.

$$\begin{cases} u_i^{k+1} & = u_{ij}^k - \Delta x . D_x^+ u_{ij}^k \text{ (forward scheme)} \\ u_i^{k+1} & = u_{ij}^k - \Delta x . D_x^- u_{ij}^k \text{ (backward scheme)} \end{cases}$$

For two-dimensional domains, the upwind method uses the gradient direction in order to select which differentiation operator to use. In Figure 2.2 we illustrate only two possible situations, all the other cases being similar up to a rotation in the system of coordinates. In the first case (a) we use the backward differences scheme in x and y and define the third quadrant as the upwind side. In the second case (b) case we use backward differences in x and forward differences in y and the upwind side is quadrant two.

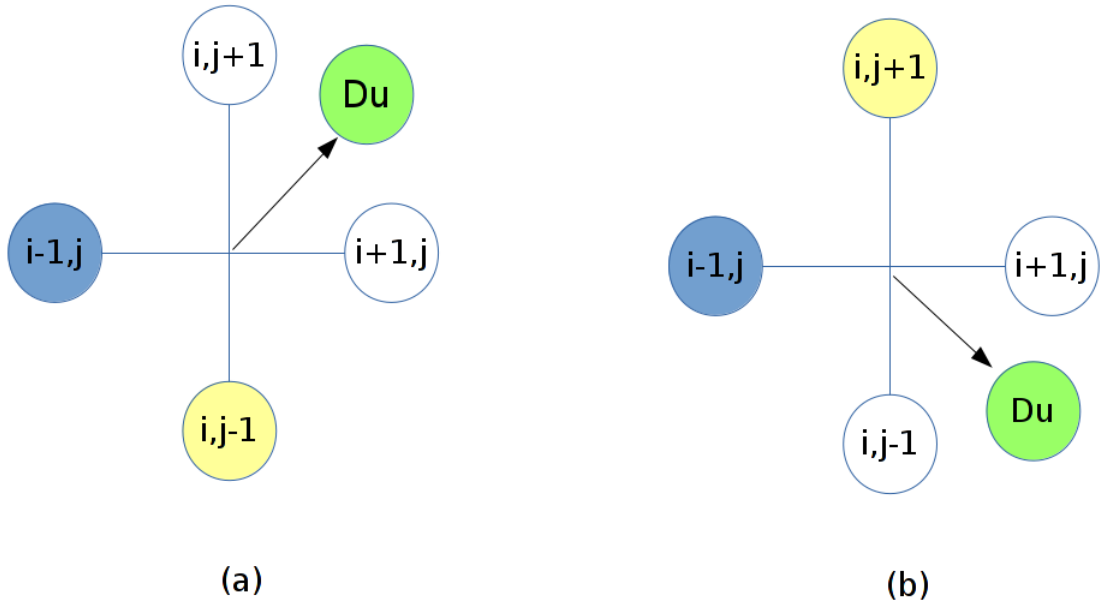

Figure 2.2: 2D case upwind discretization

Applying the local scheme on every grid points, we want to solve the equation presented below. Let $g = (g_{ij}) : \mathbb{R}^n \to \mathbb{R}$ defined by $g_{ij}(a, b, c, d) = \sqrt{\max(a^+, b^-)^2 + \max(c^+, d^-)^2} - c_{ij}$. A numerical approximation u_{ij} of the eikonal equation will satisfy :

$$\begin{cases} u_{ij} = \phi(x_i, y_j) \\ g_{ij}(D_x^- u_{i,j}, D_x^+ u_{i,j}, D_y^- u_{i,j}, D_y^+ u_{i,j}) = 0 \end{cases}$$

The upwind method uses the gradient direction in order to select which differentiation operator to use. In our implementations, the Godunov scheme will be used with the following formula :

$$\max(D_x^- u_{i,j}, -D_x^+ u_{i,j}, 0)^2 + \max(D_y^- . u_{i,j}, -D_y^+ u_{i,j}, 0)^2 = c_{i,j}^2$$

Injecting equations of the upwind local differentations (2.1) with the above Godunov scheme eq. (2.1.2)) leads to quadratic equations to solve. We detail the calculus in chapter 5 section 5.2 for a generic multi-dimensions case.

This commonly used scheme is chosen in our implementations in order to keep a consistent scheme while having the possibility to change between different global methods.

2.2 Global solving methods

We present in this section, methods to solve the problem on the whole grid which use the approximation schemes described above.

The arrival time of a propagating front is often described by non-linear static Hamilton-Jacobi equations. The origin of numerical methods on HJ equation take root from a paper by [Tsitsiklis, 1995] in 1995 where a Djikstra-like algorithm is presented to solve efficiently the problem. Since, different methods have emerged such as the fast sweeping method (FSM) and the fast marching method (FMM).

2.2.1 Rouy-Tourin algorithm

A straightforward algorithm to implement discretized HJ equations is to repeat iteratively the process on the whole grid until a stationary solution is reached (algorithm (1)). This "brute-force" like algorithm is sometimes called the Rouy-Tourin iterative scheme (in a simplified version) which is presented in [Rouy and Tourin, 1992],

Note that local scheme can also be used such as semi-Lagrangian scheme (Falcone and Ferretti) [Falcone and Ferretti, 2002] which can also be combined with the FMM [Cristiani and Falcone, 2007]. Using the Semi-Lagrangian scheme is known to be less efficient but more accurate compared to the finite difference scheme. In this thesis, we will stick with the original Godunov scheme. The Rouy-Tourin algorithm is way inefficient since its complexity is $O(n^2)$ where n is the grid size on the first dimension. Also the algorithm, does not take advantage that the information propagates from smaller to larger values of u.

In [Rouy and Tourin, 1992], the authors show that the scheme defined by 2.1.2 converges toward the viscosity solution of 1.2.

Algorithm 1: Rouy-Tourin Algorithm

Data: u^0
Result: u^{final}
begin
  for *all node* x *in grid* G **do**
    $u(x) \leftarrow 0$ on Γ^0
    $u(x) \leftarrow \infty$ elsewhere
  while *convergence is not reached* **do**
    foreach *node* $x \in G$ **do**
      $res \leftarrow$ solution of eq. 2.1 at node $x \in G$
      if $res < u(x)$ **then**
        $u(x) \leftarrow res$;

2.2.2 Fast sweeping methods (FSM)

Even though the fast marching method (FMM) is arguably mostly preferred among scientists. There is also an other "fast" method referred as the fast sweeping methods (FSM) [Tsai et al., 2003, Zhao, 2005, Kao et al., 2005] which proposes a different strategy for solving numerically HJ equations. The general idea is that the algorithm sweeps the whole domain with 2^D alternating orderings repeatedly where D is the domain dimension. The method alternates sweep ordering of Gauss-Seidel iterations on the whole grid until convergence. For instance, for the two dimensional case, we have four alternating orderings (N_d represents the number of vertices along dimension d where $1 \leq d \leq D$) :

$$\begin{cases} (1)i & = 1, ..., N_x; j = 1, ..., N_y \\ (2)i & = N_x, ..., 1; j = 1, ..., N_y \\ (3)i & = 1, ..., N_x; j = N_y, ..., 1 \\ (4)i & = N_x, ..., 1; j = N_y, ..., 1 \end{cases}$$

The number of sweeps needed for convergence depends on the geometry of the domain since obstacles can force distances to be measured along curved paths. Sweeping methods can reveal to be faster compared to tracking methods on simple examples [Chiang et al., 2007]. However, tracking methods are faster when the domain or velocity formulations are nontrivial. Unlike the ordering of updates in tracking methods, the order of point updates is entirely predefined in a sweeping method.

Chapter 3

Fast marching methods

Contents

3.1 Fast marching methods

Handling partial differential equations such as the wave equation implies to follow the principle of causality. We present in this section the notion of causality principle which are the basis in the way front tracking methods work. We present the fast marching method (FMM) and its variants which approximate the solution of eikonal equations. We then present the fast iterative method (FIM) an efficient FMM alternative with a higher parallel potential.

3.1.1 Front tracking methods

The principle of causality stipulates briefly that current position of a front cannot affect earlier positions of the front. In other terms, for a monotone front propagation, smaller values do not depend on larger ones. This property is known as the causality principle. Hence, it is not necessary to compute new values in areas already passed by the front nor in areas far ahead of the front. The amount of computations can therefore be reduced if points are updated in an order corresponding to this causality observation. The solution cand be constructed in an increasing order which is widely used by several methods know as front tracking methods. Fast marching methods are part of these front tracking methods and widely used for solving static HJ equations.

3.1.2 FMM basic idea

The FMM is closely related to Dijkstra's algorithm for computing the shortest path on a network. The principle is to compute the solutions values at grid points in the order in which the wavefront passes trough the grid points.

The FMM is a single pass method, a point in the grid may be visited and be updated only once following definition 3.1.2.

Definition 1. *(single-pass). An algorithm is said to be single-pass if each mesh point is re-computed at most x times, where x depends only on the equation and the mesh structure independently to the total number of vertices in the mesh.*

The local single-pass notion is also introduced in definition 3.1.2 :

Definition 2. *(local single-pass). A single-pass algorithm is said to be local if the computation at any mesh point involves only the values of nearest adjacent neighboring nodes.*

For that purpose, Sethian proposes to divide the grid into three regions [Sethian, 1999a] (fig. 3.1) : the accepted points or frozen points (FZ points), the narrow band points (NB points) and the far away points (FA points).

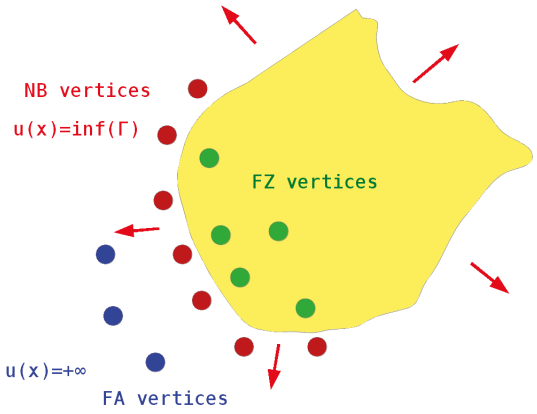

Figure 3.1: FMM three regions

The definition of these regions is as followed :

- Accepted points (or frozen) correpond to points of the mesh which are part of the initial front and which have already converged.
- Narrow band points correspond to the points which are the neighors of the accepted nodes at a given iteration.
- Far away points correspond to the points which are not yet computed.

The accepted vertices have already been reached by the front. Their solution has been computed and their value will not change in the future. Narrow band vertices are where the computation actually takes place and they might be updated at the following iterations. The narrow band forms a thin region between the accepted points and the far away points. Finally, the far away vertices have never been reached by the front. The FMM follows the basic idea of Dijkstra' method which is :

1. label the initial front nodes as accepted nodes ;
2. label neighbors of these nodes as narrow band nodes ;
3. compute the reaching cost for each neighbors ;
4. remove from the narrow band and put in the accepted region the smallest cost of the neighbors and return to the second step until all nodes are accepted ;

We suggest the reader to refer to Sethian's books and work [Sethian, 1999a, Sethian, 1999b] for details about the algorithm. The method is a one-pass method, each point being touched only once. This corresponds to a computational cost of $O(N)$.

A basic illustration of the method is available in figure 3.2. The red squares represent the current narrow band points, the black ones the far away points and the green points the accepted points. On the first frame, the single green point represents the initial source front. At every step the current point which is being updated is guaranteed to own the smaller value function in the narrow band.

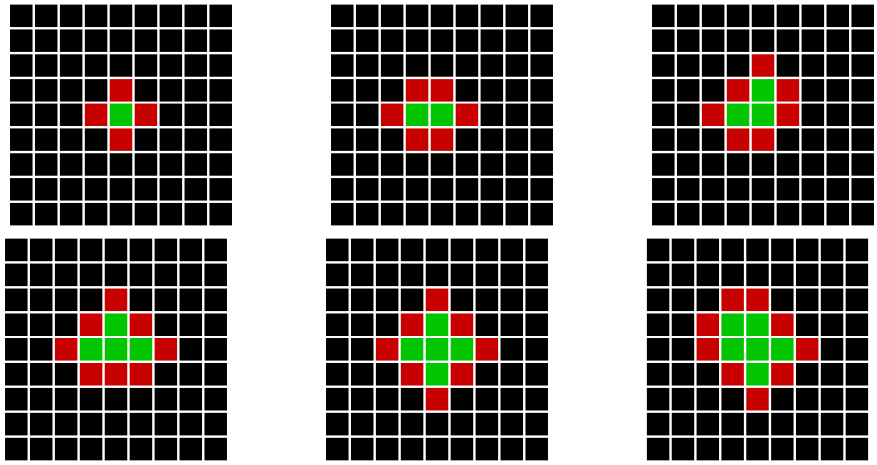

Figure 3.2: Fast marching method adding points in the narrow band

3.1.3 FMM data structure

In order to easily detect the smallest value in the narrow band, the FMM proposes to use a min heap data structure such as the one illustrated in figure 3.3 to store the narrow band vertices. This structure is found to be the most efficient [Sethian, 1999a]. We will

Algorithm 2: Fast marching method algorithm

```
Data: u^0
Result: u^final
UpdateNarrowBand(x)
    res ← solveEikonal(x)
    if x is far away then
        Add x to NB
    else
        Remove x from NB and add x to FZ
    if res < u(x) then
        u(x) ← res;
begin
    Initialization
    foreach node x ∈ G do
        if x ∈ Γ^0 then
            Add x to FZ
            u(x) ← 0
            foreach neighbor x_Nx of x do
                if x_Nx ∉ Γ^0 then
                    UpdateNarrowBand(x_Nx)
    Main loop
    while NB ≠ ∅ do
        x_min ← (x| min(u(x), x ∈ NB)))
        Add x_min in FZ
        foreach neighbor x_Nx_min of x_min do
            if x_Nx ∉ FZ then
                UpdateNarrowBand(x_Nx_min)
```

take a binary heap structure as implementation since it is the most used and the most simple one to implement.

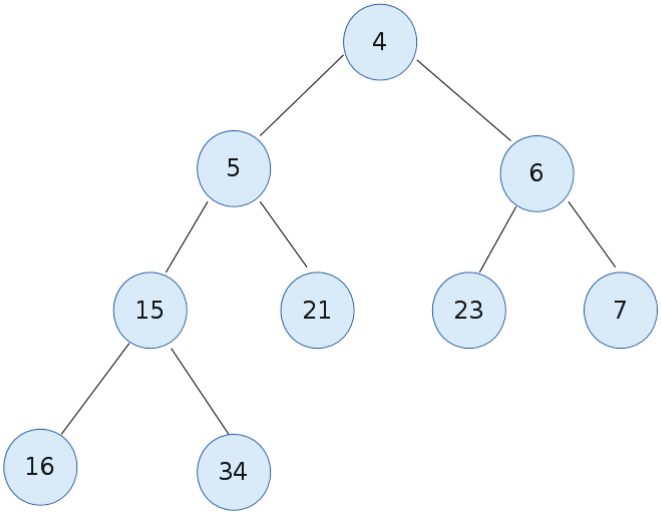

Figure 3.3: A min-binary heap

The implementation provides the following key operations in order to manage the min heap structure :

- find-minimum which takes the minimum value ($O(1)$ complexity) ;
- insertion which adds a new key to the heap ($O(log(N))$ complexity) ;
- delete-minimum which removes the minimum root value ($O(log(N))$ complexity) ;
- decrease-key which updates a heap node in the tree ($O(log(N))$ complexity).

At each iteration, computing the minimum value makes the global algorithm complexity up to $O(N.log(N))$ since inserting an element takes $O(log(N))$. The FMM is an efficient method to solve eikonal equation on sequential architectures. However, managing the heap structure is a hindrance for performance causing bottlenecks since the heap has to be updated whenever a new vertex in the narrow band added. The causality principle forbids the simultaneously update of several points at the same time. A basic heap management has a complexity of $O(N(\log(N))$. In our implementation, we use a min heap structure which reduce the cost to $\log(N)$. Even with this improvement, if we divide the narrow band points on parallel architectures of p processors, we would have a $\log(N) - \log(p)$ complexity. Given a large scale problem, it would still be insignifiant.

We present below on figure 3.4 and 3.5 the earliest FMM outputs realized by our solver Par4HJB representing two interfaces which are merging. The results were obtained at different iteration time where it is possible to follow the "correct" evolution of the interface with the FMM. We will see that this might be no longer the case with different methods.

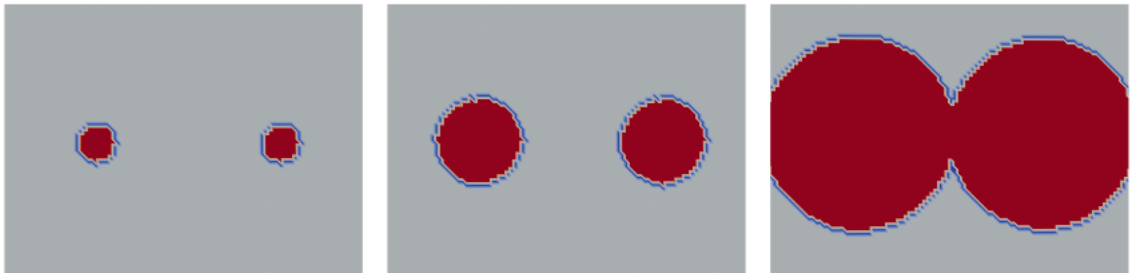

Figure 3.4: Two circles propagation

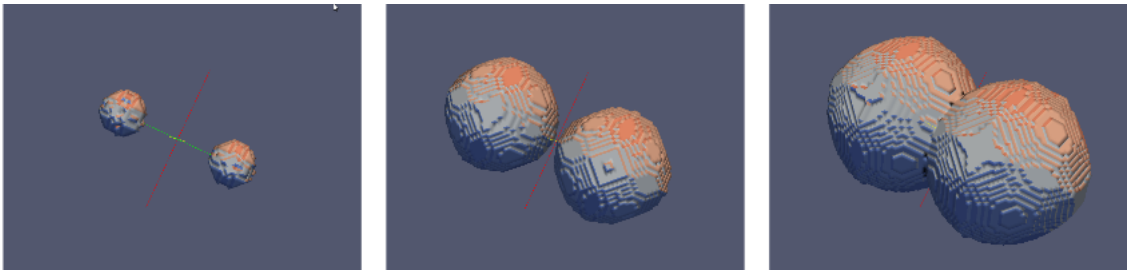

Figure 3.5: Two spheres propagation

FMM has been extensively improved in different aspects such as solving wider classes of HJ problems or for performance purpose. For instance, the group marching method (GMM) introduced by Kim [Kim, 2000, Kim and Folie, 2001] labels points as accepted differently compared to the FMM in order to compute more efficiently the solution. In some cases, GMM can reveal to be faster. The generalized fast marching method [Forcadel et al., 2008] is capable to handle no sign restriction on the velocity. Other fast marching method alternative proposed in [Cristiani, 2009] which is based on semi-Lagrangian schemes can be used to solve Hamilton-Jacobi-Bellman and Hamilton-Jacobi-Isaacs equations. Recently, the fast iterative method (FIM) reveals to be a efficient alternative to the classical FMM especially in a parallel environment.

3.2 Fast iterative method

As Jeong and Whitaker [Jeong and Whitaker, 2008] the authors of the method point out, designing fast parallel algorithms for solving the eikonal equation on parallel architectures should be greatly investigated.

3.2.1 FIM a method with a high parallel potential

The use of heterogeneous data structures, irregular data updating schemes hinder performance on parallel architectures. Therefore, Jeong and Whitaker proposed a new method called the fast iterative method (FIM) to solve the Eikonal equation efficiently on parallel architectures. The FIM is still interesting since we generally have a speedup between 6

and 100 compared with the FMM according to the authors [Jeong and Whitaker, 2008]. The FIM is faster than the FMM in sequential and more scalable for parallel purpose.

The FIM imposes to follow three main points : no particular update order, avoid separate, heterogeneous data structure for sorting and enable multiple points to be updated simultaneously. The FIM can be seen as a compromise between the FMM and the Rouy-Tourin method. We present the FIM algorithm 3. Indeed, the FIM keeps the idea of the narrow band (called the active list in the FIM) and manage to solve iteratively this list with the Rouy-Tourin scheme. One of the major differences is that the FIM allows blocks of active list to be updated at the same time. Note that points in the active list can remain during several iterations and can be computed many times until they have converged. Therefore, the narrow-band in the FMM propagates differently compared to the active list in the FIM. One of the drawbacks of the FIM is that we cannot follow the evolution of the front at a given time during the simulation (like the Rouy-Tourin algorithm).

FIM is a flexible method which can be extended. For instance, we will investigate parallel semi-ordered fast iterative method which is able to work for anisotropic problem in section 4.6.

Algorithm 3: Fast Iterative Method algorithm

Data: u^0
Result: u^{final}
begin
  Initialization
  foreach *node* $x \in G$ **do**
    if $x \in \Gamma^0$ **then**
      Add x to FZ;
      $u(x) \leftarrow 0$;
  foreach *node* $x \in G$ **do**
    foreach *neighbor* x_{Nx} *of* x **do**
      if $x_{Nx} \in \Gamma^0$ **then**
        Add x_{Nx} to NB
  Main Loop
  while $NB \neq \emptyset$ **do**
    foreach $x \in NB$ **do**
      $p \leftarrow u(x)$
      $q \leftarrow$solution of eq. 2.1 at x
      $u(x) \leftarrow q$
      if $|p - q| < \epsilon$ **then**
        foreach *neighbor* x_{Nx} *of* x **do**
          if $x_{Nx} \notin NB$ **then**
            $res \leftarrow$ solution of eq. 2.1 at x_{Nx}
            if $res < u(x_{Nx})$ **then**
              $u(x_{Nx}) \leftarrow res$
              Add x_{Nx} to NB;
        Remove x from NB

Part II

Contributions

Chapter 4

Parallel computing strategies

Contents

The causality principle in the FMM seems not well fitted for parallel purpose which lead to bottlenecks. So, how can we obtain efficient parallelism while solving HJ equations ? This chapter propose solutions to such problematics. One of the main idea is that changing the algorithm and the method itself may be a way to achieve efficient parallelism. As we have seen in section 3.2, FIM has a higher parallel potential compared to other FMM like methods. Section 4.1 proposes multi-level parallel strategies including a new method called the buffered fast iterative method (BFIM) which present an efficient parallel scalability for shared-memory architectures. One of the challenge in this dissertation is to keep efficiency combined with a reusable code. We investigate in this chapter ways which can achieve reusability in terms of sofware reusability and also sequential/parallel reusability.

4.1 Implementation of the fast iterative method

Eikonals solvers outputs can be represented via geodesic maps. Geodesic and euclidian maps differentation are detailed in this section and we present applications results in shape from shading and path finding. We discuss on how we evaluate the errors during our simulations.

4.1.1 Geodesic distance map

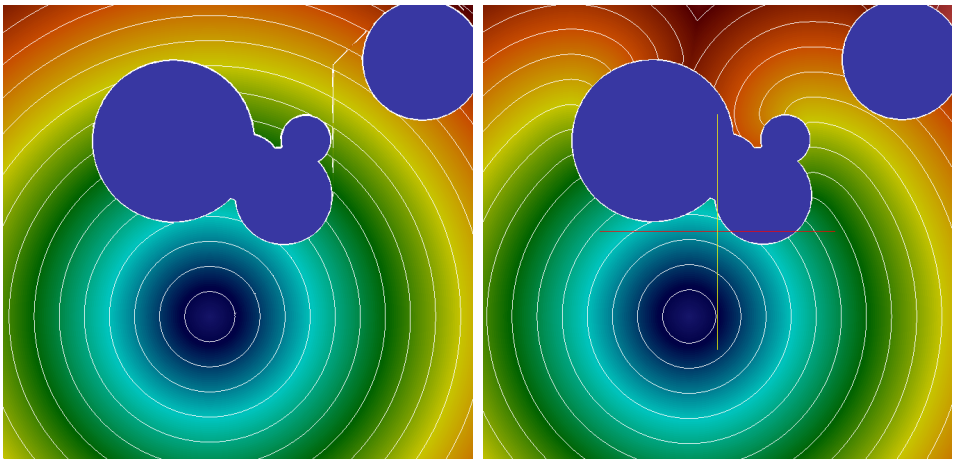

Figure 4.1: Euclidian (left) and geodesic (right) distance

The output we obtain when solving the eikonal equation with fast methods is a geodesic distance map, which is different from the euclidean distance, as shown in fig. 4.1. We can observe on the top of the figure that the geodesic distance is more representative of a wave front propagation when obstacles are encountered. The geodesic distance is the length of the shortest path between the source and a given point in a grid whereas the

euclidean distance is a straight line distance. Geodesic map preserves geodesics, which were in the former sense, the shortest routes between two points on the Earth's surface.

In our implementation we obtain the following execution time comparison on table 4.1. The parameters used for the simulation are the same as described in section 4.5 with three initial fronts, a monotone velocity and obstacles.

Data Size	RTM	FMM	FIM
100x100	0.22 s	0.057 s	0.02 s
600x600	44.49 s	8.49 s	2.37 s
1000x1000	223.66 s	38.02 s	11.25 s

Table 4.1: Sequential methods comparison (on Intel Core 2 Duo CPU E8400)

The results confirm that the sequential FIM is indeed faster than the sequential FMM as told previously in subsection 3.2.1. We illustrate in further subsections some applications obtained with our implementation.

4.1.2 Application : path finding

The rendered output given after calling the solver is a geodesic distance map. Shortests paths can be obtained by backtracking from a final point to the initial source using gradient descent. In an anisotropic environment, a characteristic descent can be used instead.

On figure 4.2, we obtain two different paths when choosing two different ending arrival points. The algorithm will choose the optimal path between the arrival point and its nearest starting point.

The gradient descent works as following. Basically, for each dimension, we compute finite differences while taking care to handle possible infinite differences. In order to normalize, we divide the gradient of all dimensions by the maximum gradient. The listing code in appendix D illustrates in details an implementation of the gradient descent in one of our library.

Path finding can reveal to be useful for motion planning in a complex environment. The experiments are executed on real world datas using the shapefile format. The ESRI shapefile format is a popular geospatial vector data format for storing geometric location. The technical details of the format are available in [Esr, 1998]. One shapefile is restricted to contain the same type of shape such as points, polygons, lines... Combined shapefiles can represent well detailed maps such as illustrated in figure (4.3).

Par4HJB and Hamijac own some basic functionnalities to manage shape geometries allowing the user to use interoperate easily with datas from OpenStreetMap. In Par4HJB, the user is in charge for writing the the piece of code which allow him to generate the final expected output. In Hamijac, the library integrates some functionalities which can communicate to other third party tools or data. For instance, the "external" namespace of Hamijac can read directly shapefiles and propose the minimal domain to work in with corresponding shape fronts. An example with circles shape is given in figure 4.4 where a path is computed starting from bottom left to reach up right.

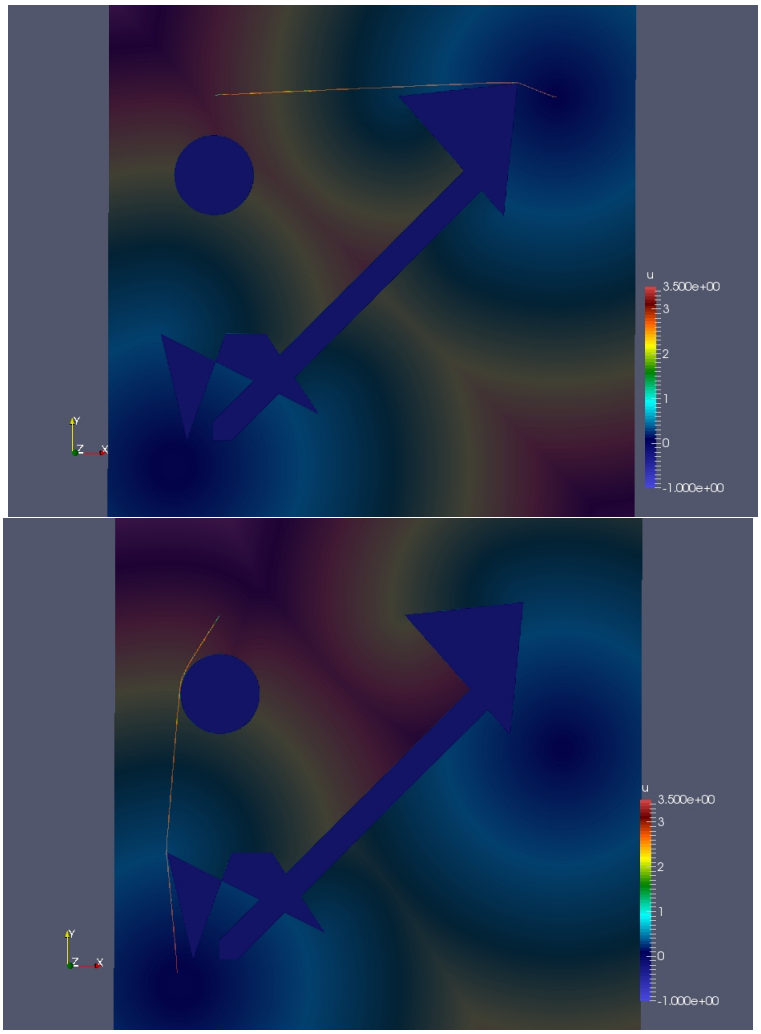

Figure 4.2: Path finding with gradient descent

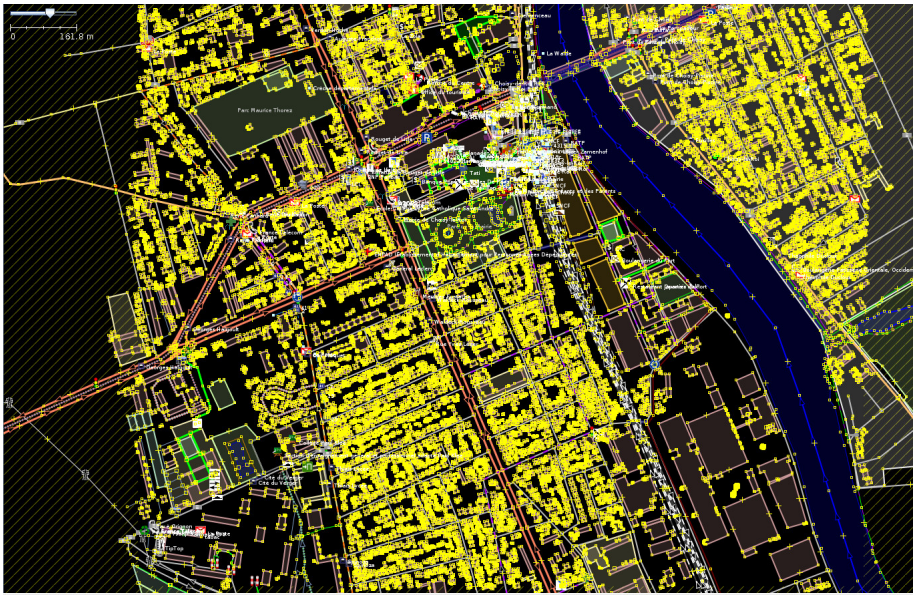

Figure 4.3: Thiais city shapefiles from OpenStreetMap

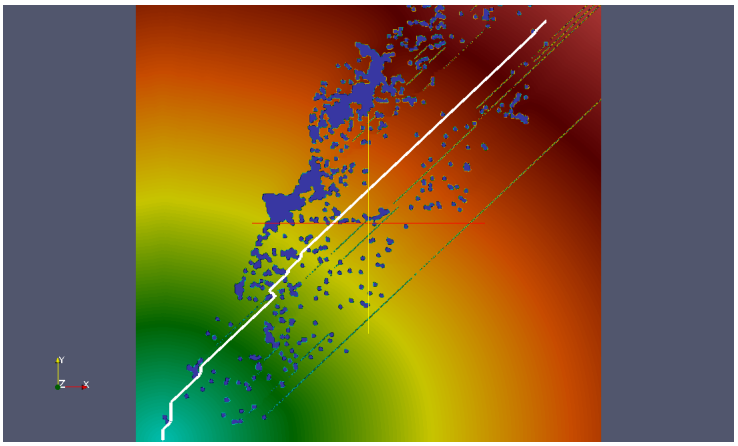

Figure 4.4: Simple path from a shapefile provided by OpenStreetMap

There are basically three steps for making the library work in this type of application :

- interpret shapefile files as inputs for the HJ library solver (external namespace);
- solve the HJ problem (library core functionalities);
- use a gradient descent on the given output (application namespace).

More details are available on the design of the library in next section.

4.1.3 Application : shape from shading

We recall that the aim of the shape from shading problem is to recconstruct a three dimensional shape from one or more two dimensional images.

Rouy and Tourin proposed in [Rouy and Tourin, 1992] (1992) a shape-from-shading representation in terms of the Hamilton-Jacobi equation. Methods to solve HJ equations are becoming more and more efficient since. In [Yuen et al., 2007] (2007), the authors solve the shape-from-shading problem by using fast marching methods. The particularity of shape-from-shading is that on the contrary to technique based on multiple images used in photometric stereo, it is using a PDE formulation based on a single image. However, the algorithm would work correctly under certains conditions. Indeed a number of assumptions have to be made in order to have revelant results such as :

- the image has to reflect the light uniformly
- the intensity of the reflected light is proportional to the scalar product between the direction of the light and the normal of the surface (lambertian material)
- there are no hidden regions (the scene is visible by the camera)

We show an overview of how FIM and SOFIM can behave for the shape from shading problem. The input is a grayscale image of a human face. We can recognize on figure 4.5 the shape of the face, where we have added a depth dimension according to the u function values. We consider simply an orthographic project, the light path is orthogonal, thus interpreting the velocity as $c(x) = \sqrt{\frac{1}{I(x)^2-1}}$ where $I(x)$ represents the intensity of the point x.

The velocity choice formula comes from the reflection geometry model. We consider that the image plane is x, y plane and the optical axis of the camera is aligned with z-axis, where the reflectance map R of specular shape from shading follows $R = \frac{1}{\sqrt{1+||\nabla z}}$ where z represents the z-axis. Initial sources values are chosen in a set of local minimum singular points.

Using SOFIM or FIM gives us the same results. The parallel implementations of both methods shown later is encouraging since it demonstrates interesting perspectives on efficient 3D shape reconstruction from a single 2D image.

Measuring the validity of such applications results is not detailed in this thesis. However, the numerical schemes and the methods which have been used can be evaluated with error analysis.

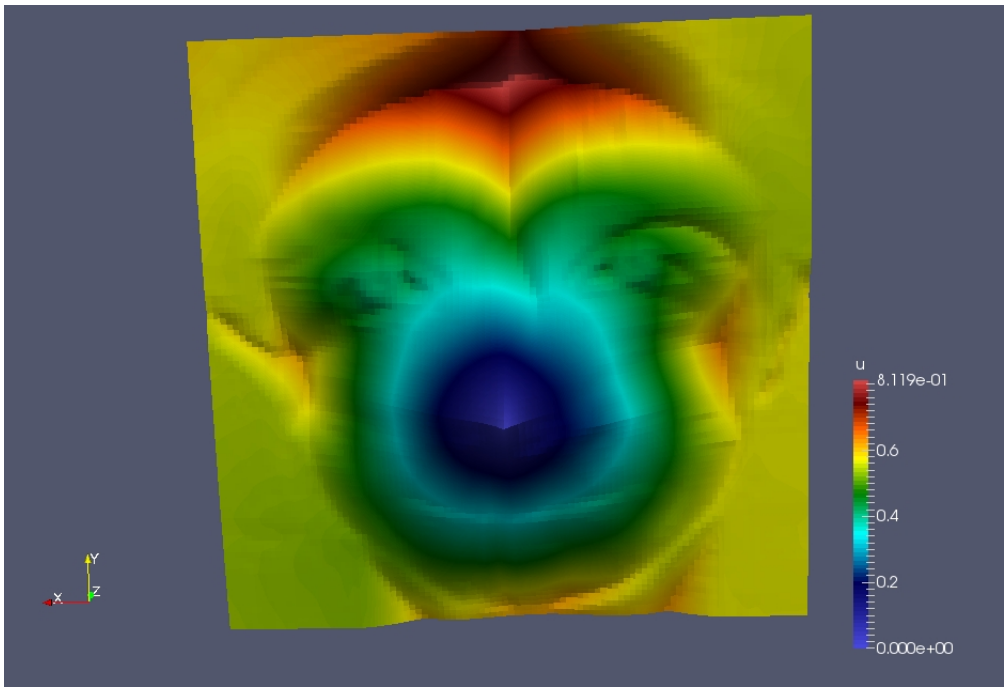

Figure 4.5: Shape from shading experimentation

4.1.4 Error analysis

Ensuring numerical quality in simulation is a compulsory step towards real world applications. Numerical errors can lead to false and dangerous results. A computer scientist needs to be aware of such problems and adapt his methods according to the environment. The *What Every Computer Scientist should known about Floating-Point Arithmetic* [Goldberg, 1991] article is a nice introduction regarding these difficulties.

In order to limit numerical errors, it is important to make the distinction between accuracy and precision. Accuracy refers to how close we are from the exact value. For instance, in our case, increasing the order of the finite difference scheme should increase the accuracy. The precision refers to how close the computed values agree with each other. In our case, we can compare the precision of our different methods if they give different results for the same test case. We must be aware that increasing the accuracy or the precision in a code is computationnaly more expensive. Therefore, we must find the best setup depending on the problem we want to solve.

We can separate numerical errors into two categories : rounding errors and truncation errors. Rounding error occurs because computers can only represent numbers using a fixed and limited number of significant figures. Irrationnal numbers cannot be represented exactly in computer memory. In our case, when the problem requires a high computing precision, we can reduce the effects of rounding errors by using double precision which uses 64 bits instead of simple precision which uses 32 bits. Truncation error occurs when the numerical method implemented represents itself an approximation of a series which is truncated to a few number of terms (e.g. Taylor series, Fourier series...). We can predict these errors since they depend of the algorithmic implementation.

The errors we evaluate concern both truncation errors (numerical discretization) and rounding errors (computer floating point). We measure errors with three different L_1, L_2 and L_∞ norms :

$$\begin{cases} \|x\|_1 = \sum_i |x_i| \\ \|x\|_2 = \sqrt{\sum_i |x_i|^2} \\ \|x\|_\infty = \max_i |x_i| \end{cases}$$

Let $\hat{x}$ be the approximation, the absolute error is defined by

$$E_a(x) = \|\hat{x} - x\|$$

and the relative error by

$$E_r(x) = \frac{\|\hat{x} - x\|}{x}$$

.

Our sequential FIM gives the same results as the RTM which is our reference since it applies directly the approximation scheme on the whole grid points.

So, computing errors may happen in the local numerical scheme used. One way to simply check the validity the correctness of our algorithms (approximation schemes and methods) is to compare our geodesic map values with respective simple euclidian map values.

We can see observe that the precision rises as the grid size has larger size and steps.

The interesting reader can take a look at [Higham, 2002, Montan, 2013] for more details on numerical errors.

Local finite difference scheme errors We have seen that we can easily verify the correctness of the global method used (FMM, FIM) by comparing it with a brute-force like algorithm (RTM) which update the whole point grids at every iteration.

Several parameters can impact the results errors such as :

1. the use a different local scheme (Godunov, semi-Lagrangian, ENO/WENO) ;
2. the order of the stencil scheme applied ;
3. the floating point precision used.
4. the mesh refinement

We notice that some parameters have stronger impacts. For instance, changing the floating precision (float or double) is negligeable compared to changing the mesh refinement. The context is a center test which consists of propagating a single center point in a $[-10, 10]^2$ domain using first order scheme with brute-force algorithm (RTM). Table 1.2 show different mesh refinements and their impact on numerical errors.

Computing accurately and precisely is an important step in the elaboration of a solver. Once the validity of the methods and schemes is confirmed, we can focus on the performance part and the parallelism.

Error/Vertices	400^2	1000^2	2000^2
$E_{r\|.\|_1}$	0.006750	0.003228	0.001819
$E_{r\|.\|_2}$	0.000050	0.000011	0.000004
$E_{r\|.\|_\infty}$	0.006298 s	0.002969	0.001650

Table 4.2: Error measure with different mesh refinement using floating point precision (double point precision hardly change the measures in this cases)

4.2 Study of available parallel fast methods

4.2.1 Classical domain decomposition for the FMM

In [Herrman, 2003], the author proposed a domain decomposition tuned for the FMM. The whole computational grid Ω is divided between p processors, giving each processor access to only its own sub-domain Ω_k where k is the process rank, and use message passing strategy to communicate between different processors. Drawbacks of these strategies are that each sub-grids Ω_k has to compute its smallest close value u^k_{min} for the FMM and has to share ghost nodes between other neighboring sub-domains leading to boundary communications (fig. 4.6). Indeed, given a ghost node u_{ij} between a sub-domain Ω_k and its neighbor Ω_{k+1}, if the smallest value in Ω_k is greater than the ghost node value computed from the neighbor sub-domain Ω_{k+1} i.e. $u^{k+1}_{ij} \leq u^k_{min}$, then it is more likely that greater values in Ω_k might be wrong since they can depend on u^{k+1}_{ij}. It is then necessary to rollback these nodes in the narrow band to compute them again and allow consistent algorithm. Some sub-domains can have few work to do, when the narrow band is not represented in the sub-domain for instance. Boundaries synchronizations, rollbacks can reveal to be costly in some situations.

This method is well known and generally efficient on distributed systems when a few rollback operations occur. In HJ problems, this is however not the case, and complex simulations can require an important amount of rollback operations.

4.2.2 Adaptive domain decomposition for the FMM

An improvement is proposed in [Herrman, 2003] with an adaptive domain-decomposition which is more focused on the narrow band management. The initial front is partitioned at the initialization and then each processor solves a processors, when some vertices from a sub partition can overlap in other sub partitions for instance. One option to overcome this is to redivide the narrow band vertices again. Applying this strategy to the FMM is costly since we generally want to minimize synchronizations the most at a fine-grained level. Changing the algorithm and the method itself can be a good idea in order to be efficient in parallel computing. The active list is more fitted for parallel purpose but still requires a careful partitioning. We cannot compute the vertices in the active list independently following the original algorithm.

Results from [Herrman, 2003] are available on figure 4.6 where the authors compare their method with the one proposed by Herrmann. The test case used is a random test case, composed of 32 random single points with monotone speed function. The domain

is $\Omega = [-5.0, 5.0]^2$ with 2000^2 vertices. Using 16 cores, the speedup reaches almost 9 whereas Herrmann's one hardly attain 7.

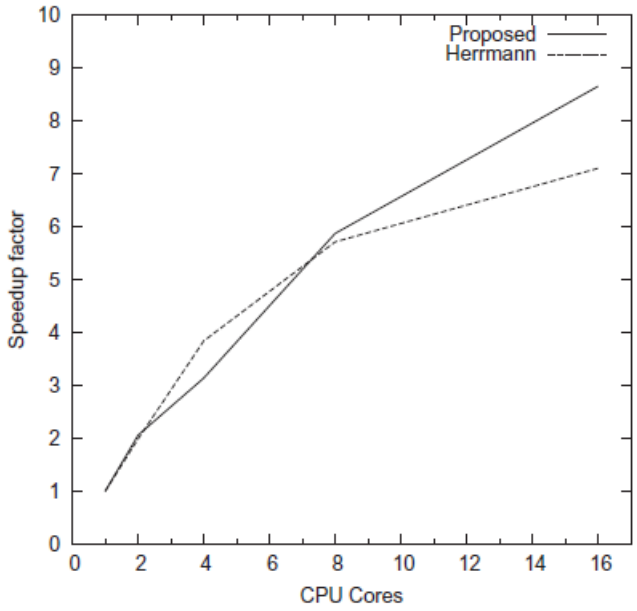

Figure 4.6: Parallel FMM : random test

4.2.3 Parallel fast sweeping method

Compared to the FMM, the parallelism in the FSM [Zhao, 2007, Detrixhe et al., 2013] is more interesting since the Gauss-Seidel computations can be proceed according to different directions. However, it is still limited by the number of directions which is not convenient when using several processes. Benchmarks from recent work on parallel FSM [Detrixhe et al., 2013] are presented in figure 4.7.

The test case used is the same as the one used for the parallel FMM except the domain is 3D and shows different data size simulations. We notice that the parallel speedup is much more interesting compared to the FMM on consequent simulations with high number of vertices. Using 320^3 vertices give a speedup of 11 when using 16 cores. On 30 cores, a speedup of 18 is almost reached.

4.3 Fine-grained parallel strategy for the fast iterative method

The work to obtain an acceptable parallel speedup as shown in [Herrman, 2003], requires a non negligeable work. Regarding the FSM, the parallelism is limited to the number of sweepings done. The speedup is strongly dependent of the latters. We propose in this section to modify and tune the original algorithm for multi-core architectures.

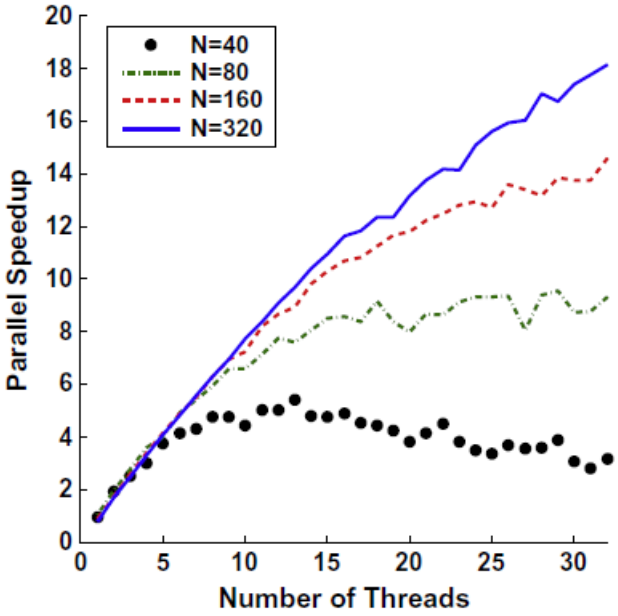

Figure 4.7: Parallel FSM : random test

4.3.1 From GPU to multi-core parallelization

[Jeong and Whitaker, 2007] GPU parallelization on the FIM is based on a block-based update scheme which can be compared to a domain decomposition. Synchronization issues are managed by reduction calls at every iteration. Using this strategy on other parallel systems than GPUs-like ones is likely to give poor performance since the synchronization calls among every threads would be too costly and would not be recovered by the computing process. We propose to use a different strategy based on an narrow band partitioning which target shared-memory architectures such as multi-core processing. We limit the use of any grid flags compared to the FMM and FIM original algorithms. Hence points update are independent in every partitioned narrow band. We propose the following fine-grained parallel FIM. We manage to make vertices computation independent in every sub active lists for the multi-core strategy. This requires to use a local active list and avoid synchronizing every sub partition minimum values. Given an initial front Γ_0 partitioned in $p > 1$ subsets Γ_0^k where k represents the rank of the subset, we have to arrange data that at each iteration t, every Γ_t^i can be solve independently and verify : $\forall i \in [[0; p]]$, $\begin{cases} \bigcup_{k=1}^{p} \Gamma_t^k & = \Gamma_t \\ \bigcap_{k=1}^{p} \Gamma_t^k & = \emptyset \end{cases}$. This arrangement is shown possible as presented in the buffered fast iterative method.

4.3.2 The buffered fast iterative method (BFIM)

In comparison with recent parallel works on the FMM, we remark that load balancing is a real concern in [Breuss et al., 2009, Breuss et al., 2011] where the authors have proposed to choose the sets in such a way that the emerging wave fronts ideally cover nearly the same portions of the computational domain. However, this assumes that we know the behavior

of the solution. One possible solution is to use a hierarchical domain decomposition scheme such as kd-trees to assign neighboring regions to the initial subsets. It is also mentioned that during simulations where unbalance can become larger (one sub-front can move ahead of another), we can redivide all the narrow band vertices rebalancing the jobs. The authors omitted this option completely as they mentioned since it is costly. In our shared-memory model from algorithm (4), we have decided to use this option and compensate the potential lack of performance by differing from the algorithm proposed in [Breuss et al., 2011] where every thread manages a local narrow band and solves its own sub-front. We decide in our strategy to share the narrow band NB among every thread avoiding potential synchronization needs and combine sub-fronts. Thus we do not have to split domains at the initialization in such way to obtain a good load-balancing. Load balancing would be done at every iteration. In order to ensure data coherency, if two different threads p_k and p_l want to write on the same value u_{ij}, we put well placed critical section and we ensure that u_{ij} take the minimum value between $u_{ij}^{p_k}$ and $u_{ij}^{p_l}$. This method allows us to limit rollback operations and the use of any hierarchical domain decomposition. Even though critical sections can be costly we minimize their impacts by well placing them in the lowest level possible. We can see their impacts in the next section 4.5. This method has the merit to be more direct forward to implement since we do not have to be as much concerned about synchronizations possible issues and offers a dynamic load-balancing.

First work on fine-grained parallel FIM is available in [Dang et al., 2013]. We have refined the model since and introduce a modified main-loop (one iteration) algorithm for shared-memory FIM. The refined method called the buffered fast iterative method, is presented in algorithm 4.

Note that the method has to use temporary arrays for the grid point values and the narrow band values in order to update points indepedently. This does not increase much cost, since we can swap the original and temporary array at the end of an iteration (since the temporary array will be erased). In our model, unlike in [Breuss et al., 2011], we do not have to be concerned about having to look for efficient way to split the grid at the initialization. The parallelization is straightforward to implement. The partitioning and load-balancing on the narrow band can be managed by a shared memory multiprocessing programming API such as OpenMP. We do not have to be concerned about rollback operations. Critical sections are necessary and placed at the lowest level possible. They do not hinder performance as we can see in section 4.5.

4.4 Coarse-grained parallel strategy for the fast iterative method

We now present a different parallel strategy possible for the FIM which is less straightforward to implement and based on subgrid decompositions. The strategy is quite similar to the decomposition found in subsection 4.2.1 but the ghosts points management differ since the FIM narrow band is different than the FMM. We also propose a way to distribute efficiently the work among processes which is compliant with code reuse. Reusability issues

Algorithm 4: Fine-grained buffered fast iterative method

```
Function Initialization()
    ∀x ∈ Ω, u(x) = +∞
    ∀x ∈ InitialFront, u(x) ← 0 and add neighbor vertices of x in NB
Function Main loop()
    Clear NewNB
    while NB ≠ ∅ do
        NewU = u
        foreach x ∈ NB in parallel do
            p_outer ← u(x)
            q_outer ← solveEikonal(x)
            if |p_outer − q_outer| < ϵ then
                foreach neighbor x_neighbor of x do
                    if x_neighbor ∉ NB then
                        p ← u(x_neighbor)
                        q ← solveEikonal(x_neighbor)
                        if q < p then
                            NewU(x_neighbor) ← q
                            add x_neighbor to NewNB ;        // critical section
            else
                if q_outer < NewU(x) then
                    add x to NewNB ;                         // critical section
                NewU(x) = q_outer
        NB = NewNB
        u = NewU
```

will be outlined in the next chapter 5). We use Message Passing Interface (MPI) routines in our code which is adaptated for distributed parallel systems.

4.4.1 Splitting the workflow and the dataflow

A simple way of distributing the work is to split the grid into subgrids where the work can be done separately from another subgrid. We can compare the u grid distribution problem as the convolution problem where we need to exchange borders from subgrid neighbors before the computation.

Regarding the narrow band distribution, a naïve way to manage it among all processes would be to share a global narrow band at every iteration. The working processes need to communicate new narrow band vertices everytime one is added in their subdomain. An other way would be to have a local narrow band for every processes and at the end of an iteration send it to a dedicated process which would manage a global narrow band. Neither

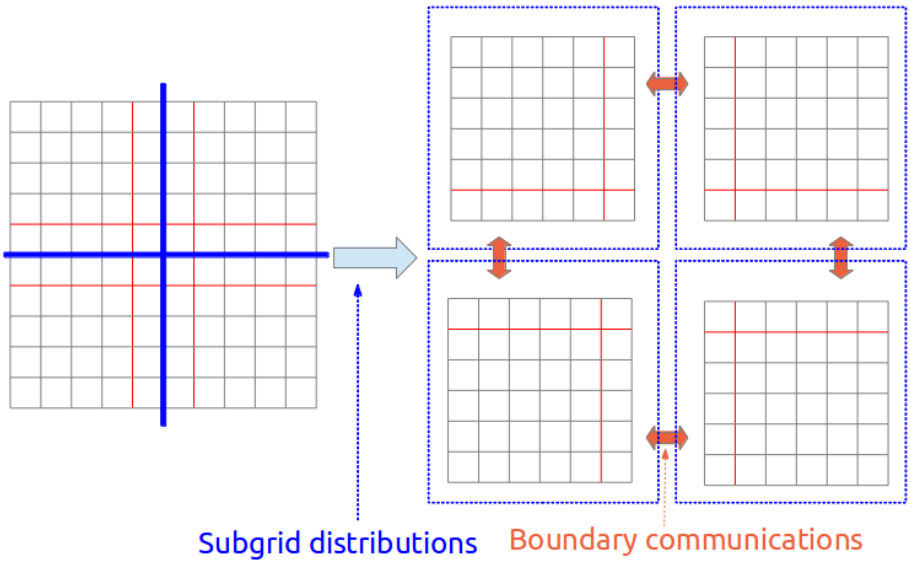

Figure 4.8: Simple coarse-grained FIM strategy

of these strategies are recommended. The first is subject to obvious communications bottlenecks. In the FIM, adding new grid points in the narrow band is much more recurrent compared to the FMM (the narrow band takes into account a wider range of points). Every time a point is added, we need to ensure data coherency between processes which is way to costly.

We propose that every narrow band becomes local to every subgrid during the whole simulation.

4.4.2 Managing ghost areas

At every iteration, one subgrid needs to send and receive future potential ghost points. Figure (4.9) illustrates parallel simulations where ghost exchange are not taken into account. At every node areas border, there is clearly a loss of information which leads to wrong computations since each node seem to work on independant subproblems. In our implementation, narrow band indices are stored as local in the subgrid they are part of.

We now present an algorithm (5) which is a simple distribution for the FIM. Let p be the number of processes, P_0 the master process and $P_{i\in[1;p-1]}$ different processes. The grid G is divided into p subgrids. We just scatter and gather the grid only one time (before and after the main loop) in order to avoid costly communications.

In order to determine whether new narrow band points are potential ghost points, we place these points in well placed buffers which can be sent to the corresponding neighbor processes. The prefix New indicates temporary buffers which allow to proceed in parallel efficiently. This technique allows us to be performant (we do not have to check the whole narrow band to send the ghost points) and minimize the parallel implementation impact in the code.

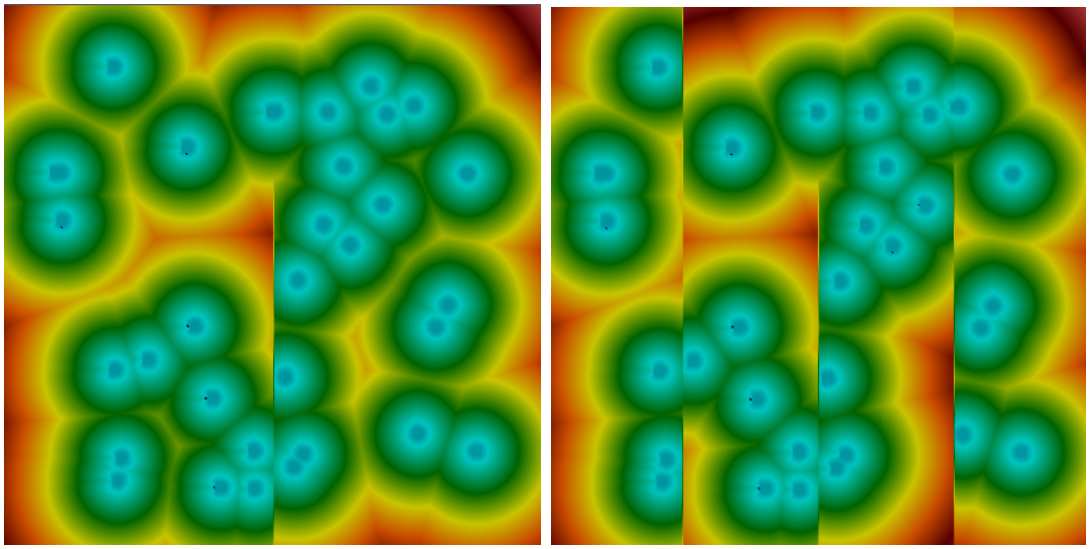

Figure 4.9: No ghost exchange using two nodes (left) and four nodes (right)

Algorithm 5: Distributed fast iterative method main loop

begin
- Scatter u in p chunks u_i
- Process P_i sends borders of subgrid G_i
- Reduce $local\_convergence = \Omega(NBblock_i)$ to $global\_convergence$
- Broadcast $global\_convergence$ to all processes
- **while** $NBblocks_i \neq \emptyset$ *i.e.* $global\_convergence > 0$ **do**
  - $NewU_i = u_i$
  - P_i sends G_i borders to corresponding subgrid neighbors
  - P_i receives borders from corresponding subgrid neighbors
  - Proceed as in algo 5 $NBblocks_i$
  - Send new NB_i ghost points to corresponding neighbor processes.
  - Receive new NB_i ghost points to corresponding neighbor processes.
  - Reduce $local\_convergence = \Omega(NBblock_i)$ to $global\_convergence$
  - Broadcast $global\_convergence$ to all processes
  - $NB_i = NewNB_i$
  - $u_i = NewU_i$
- Gather u_i to u

4.4.3 An improvement : Master worker model

The first case is obviously subject to communications bottlenecks and in the second case, we would have to synchronize the global narrow band with local narrow bands and distribute back new local narrow bands to worker processes which is costly. We propose a different approach based on a master-worker distribution, using local narrow bands and synchronizing only the narrow band vertices which are in ghost areas to the corresponding neighbor process. The method has the benefits to be reusable since we do not have to change the FIM algorithm main loop. Let p be the number of processes, P_0 the master process and $P_{i\in[1;p-1]}$ a worker process. The grid G is decomposed into $nblocks$ $G_{k\in[0;nblocks]}$ where $nblocks > 2(p-1)$ for load-balancing purposes. Indeed, we have to avoid collective, blocking communications since our problem is dynamic (a subdomain can have few works to do compared to an other one).

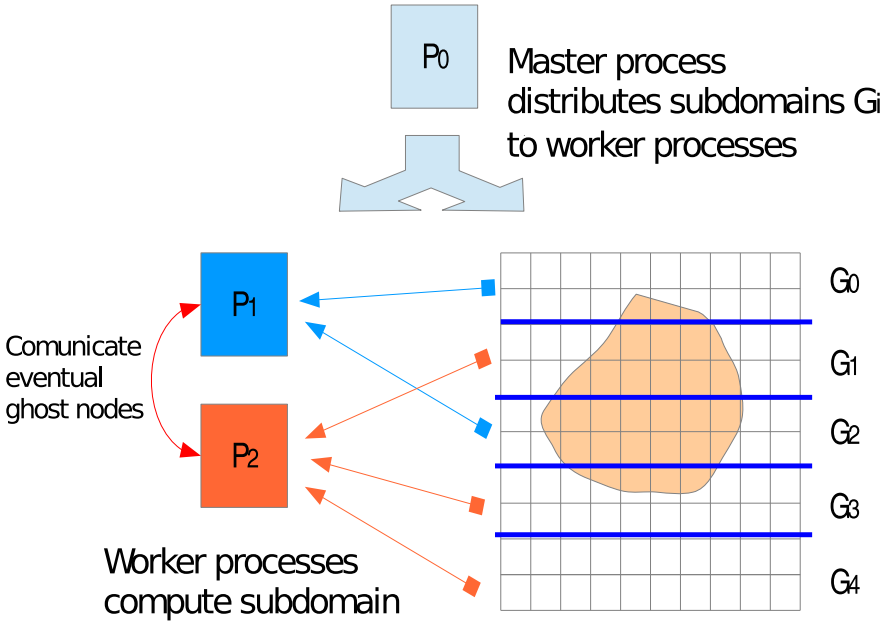

Figure 4.10: Load-balanced coarse-grained fast iterative method model

The model illustrated on fig. (4.10) with 3 processors works as the following. The master process P_0 gives some subdomains (G_0 and G_1) to compute to working process. As soon as one working process has finished (computing work is not static) it sends its work back to the master process which give immediatly an other work to compute (G_2 on P_1 and G_3, G_4 on P_2). We remark that subdomain G_4 is given to processor P_2 since P_2 might finish to compute its two first subdomains before P_1 (less narrow band vertices to check). If a new narrow band vertex is a ghost vertex then we have to communicate it to the corresponding neighbor process. We present the distributed FIM model algorithm

(6) below for one iteration :

Algorithm 6: Master-worker fast iterative method model

Process P_0 `master()`
- P_0 sends subdomains $G_{k\in[1;p-1]}$ to compute to worker process $P_{i\in[1;p-1]}$
- **while** $k < nblocks$ *i.e. there are stParallel semiill subdomains not computed yet* **do**
  - P_0 send subdomain G_k to compute to available P_i
  - Increment k

Processes $P_{i\in[1;p-1]}$ `worker()`
- P_i computes work G_k
- `Main loop()` of buffered fast iterative method algorithm where NB_i and u_i are local to the process P_i.
- **if** *vertices in* $NewNB_i$ *are in ghost zones* **then**
  - P_i sends $NewNB_i$ ghost vertices to corresponding neighbor process $P_{i-1} or P_{i+1}$

Advantage of this strategy is that we can minimize blocking and global communications, permitting to overlap communications with computations. Drawbacks are that hiding communications especially when exchanging ghost points is hard to manage and can become cumbersome. We illustrate the previous parallel levels on figure (4.11) where we can see how hybrid parallel computations can occur.

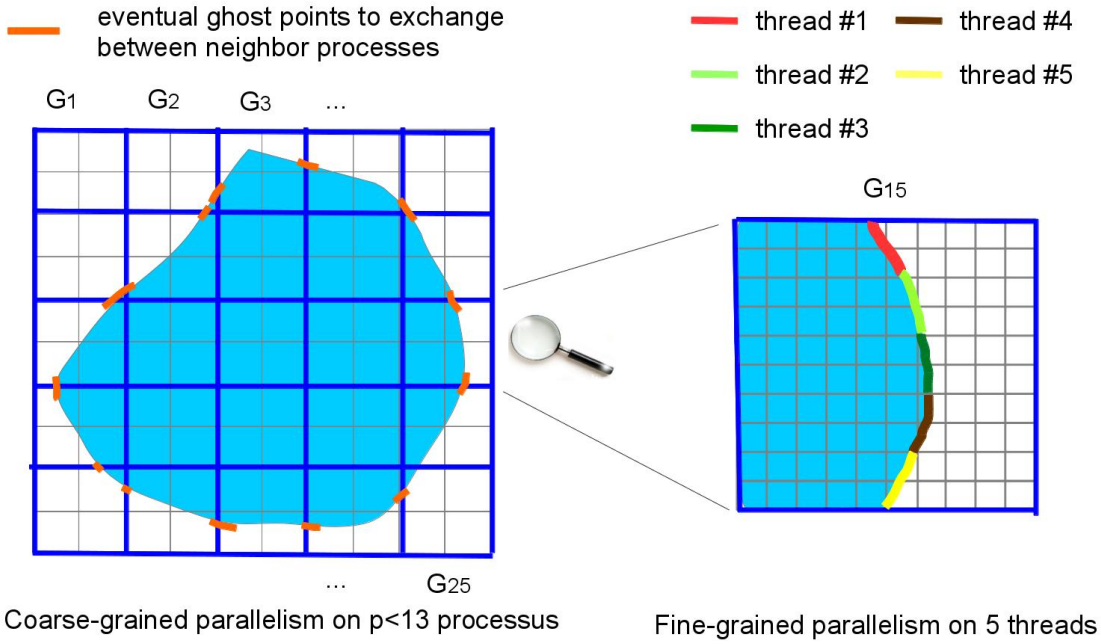

Figure 4.11: Multi-level parallelism for the fast iterative method

4.5 Experiments

We present several case tests running in different dimensions for the fine-grained parallel approach i.e. the parallel buffered FIM. We run the tests using OpenMP at the HPC@LR Montpellier Resource Center which is a shared-memory system composed of 2 processors Intel Xeon X5650 at 2.66 GHz with 6 physical cores each. Hyper-threading at HPC@LR is deactivated hence 12 logical cores in total are available. The presented execution performances are based on the median of at least five execution timings. We verify that the results obtained in parallel are the same as the sequential simulations.

4.5.1 Center, wall and random test

We show parallel scalability of our parallel algorithm on three different cases which are quite similar with the test cases done in [Breuss et al., 2009], including a center test, a wall test with two fronts and a random seed test composed of 32 initial fronts. The domain is $\Omega = [-5.0, 5.0]^2$ and the grid size is composed $N = 4$ million points. We consider monotone propagation (slowness field is constant with $F = 1$). The first test "center test" is composed of one single circle initial front at the center. The second test "wall test" illustrates the algorithm behaviour with an obstacle. The last test "random test" is more complex, which simulates a more realistic environment composed of 32 random seed initial fronts (fig (4.12)).

Figure 4.13 show that the parallel model is scalable with a smooth speedup even for the center and wall tests where the narrow band is small. When the narrow band is wider, we obtain better results as shown with the random test with an efficiency above 0.8. Real applications are likely to behave much more like the random test and should also therefore scale well.

We also monitor the parallel data-size scalability on different problem size. The second graph on figure 4.14 which represent the data size on axis x and parallel speedup using 12 threads on axis y show that working on larger problem size gives a better parallel speedup. Therefore, large scale applications should benefit from this.

4.5.2 Three dimensional case

In three dimensions the proposed algorithm stays the same, only the upwind scheme has to be modified in order to manage neighbors for another dimension. The simulation (fig. (4.15)) reproduces the center test in the domain $\Omega = [-5.0, 5.0]^3$.

The results show that the parallel model scales well in 3D. Efficiency is above 0.8 and can even reach 0.9 for large data size. Increasing the size of the problem tends to improve the parallel scalability as in 2D. We can remark a little scalability drop for $N = 100^3$ which can be neglected since the efficiency loss is minor (still above 0.8). This behaviour may be explained with partionned narrow band size which do not fit caches in the best conditions.

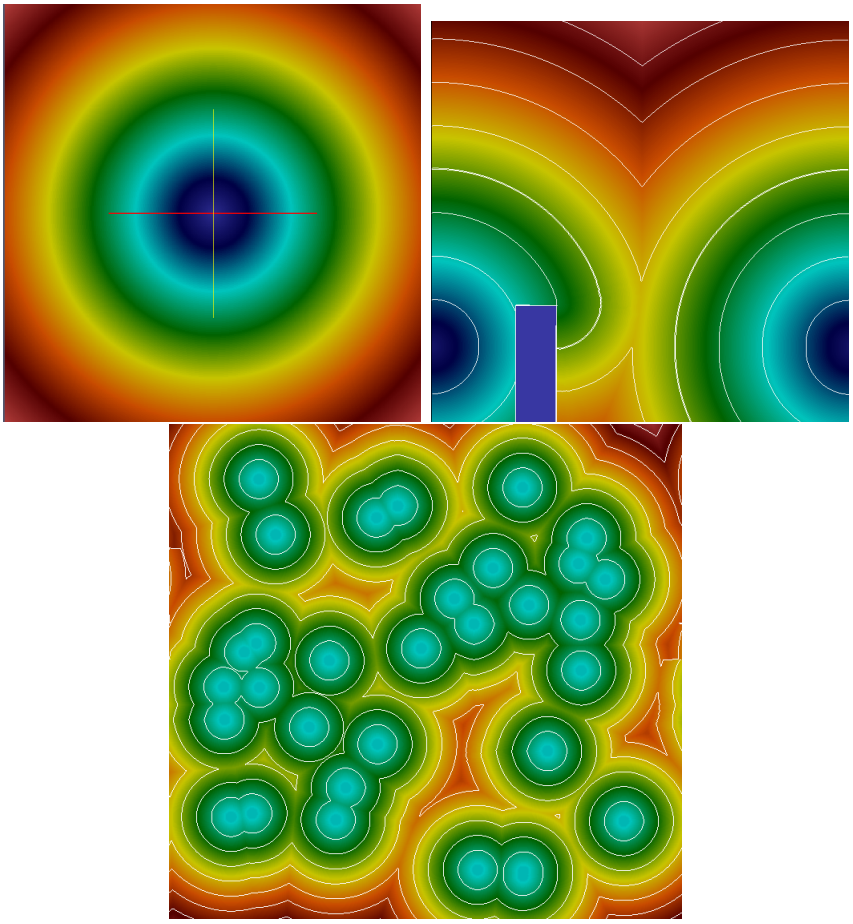

Figure 4.12: Three different test cases : center, wall and random

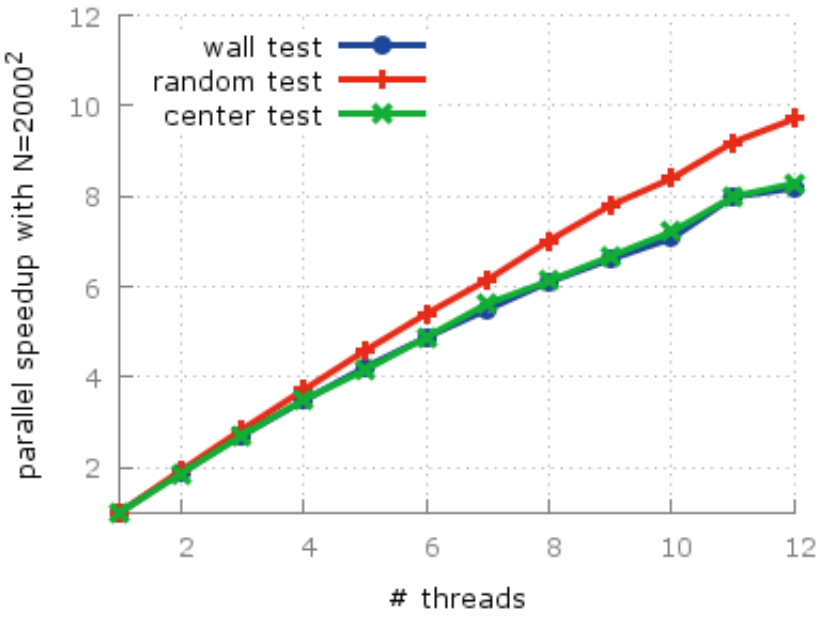

Figure 4.13: Parallel speedup scalability for different 2D test cases

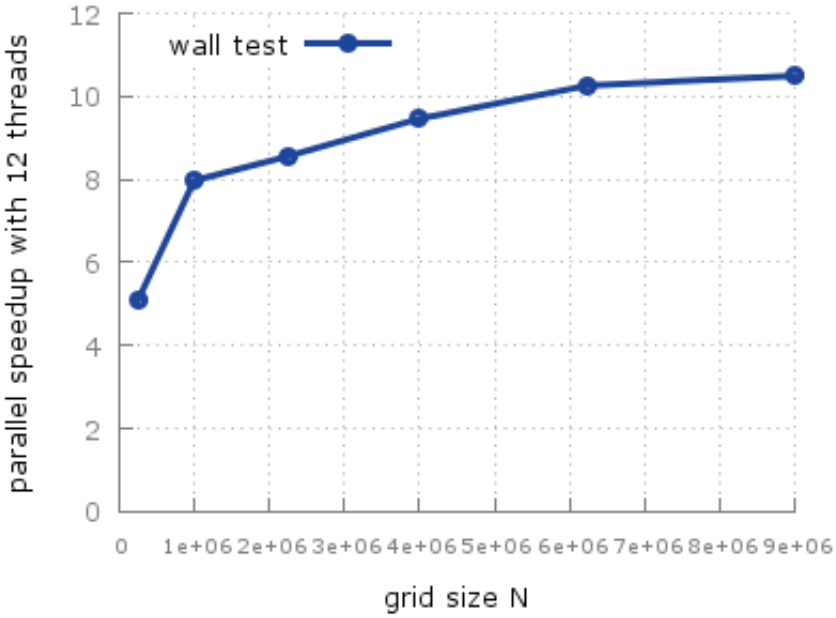

Figure 4.14: Parallel datasize scalability for the 2D wall test

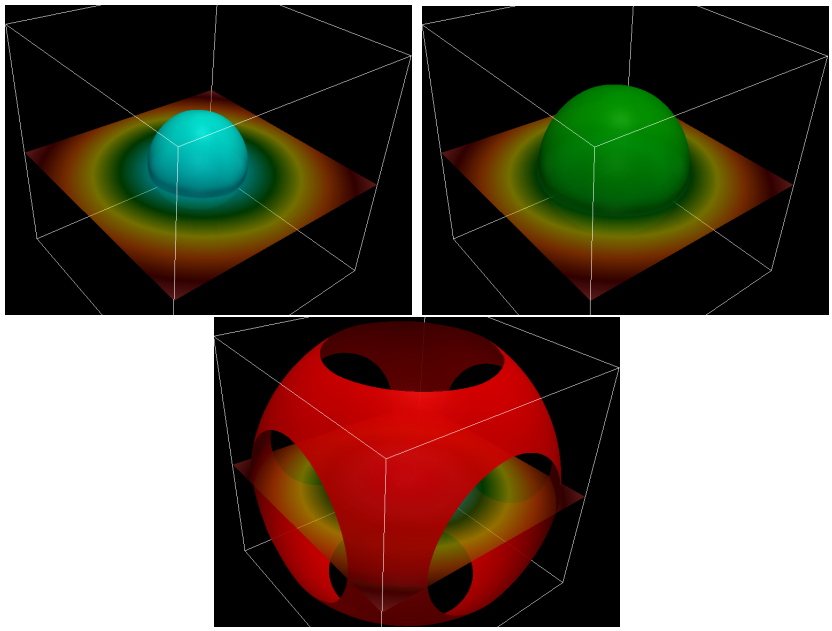

Figure 4.15: 3D center test taken at geodesic distance 2.0, 3.0 and 6.0

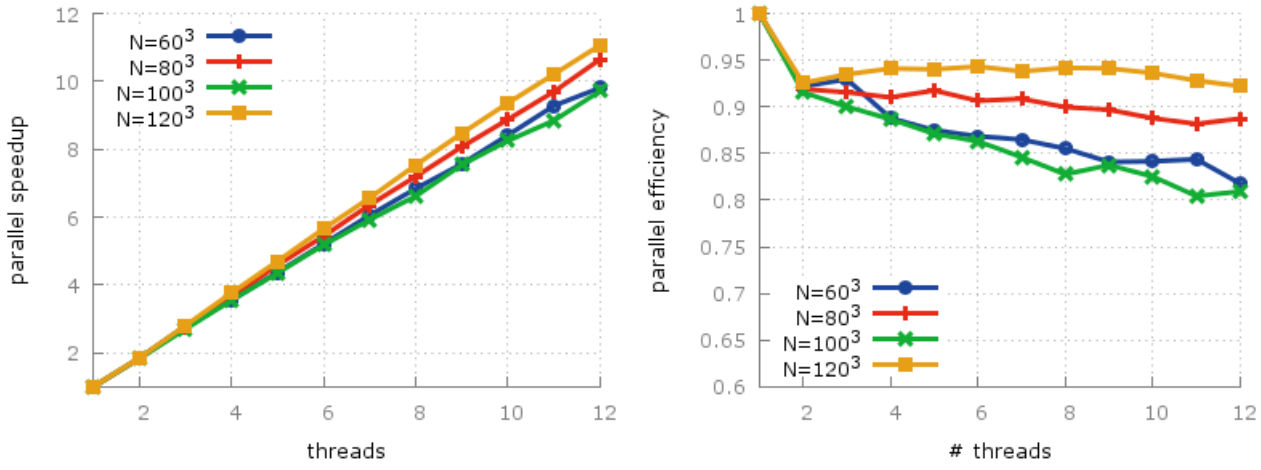

Figure 4.16: Parallel speedup and efficiency for the 3D center test on different data size

4.5.3 Discussions

We have proposed multi-level parallel strategies when updating upwind scheme using the FIM. Our parallel scalability behaves like the parallel FSM proposed in [Detrixhe et al., 2013] with an increasing speedup when the context is close to a realistic environment. Our results show a better speedup compared to recent works in [Breuss et al., 2011, Detrixhe et al., 2013]. We prove that the FIM is indeed fitted for parallel computing. Our fine-grained parallel model of BFIM targets mainly shared-memory architectures and our coarse-grained parallel model targets distributed systems. The fine-grained parallel strategy is straightforward to implement with the proposed algorithm without performance loss when running in sequential.

According to our experiments, the "scattered" FIM proposition gave the same results when executed in parallel as in sequential. Using two computing nodes gave a speedup of 1.8. However, using several nodes did not give any interesting speedup. The distributed FIM needs more investigation and work to be efficient.

4.6 Parallel semi-ordered fast iterative method

The idea behind the parallelization of the FIM is also reproductible for FIM-like method such as the semi-ordered fast iterative method [Gillberg, 2011]. The following work takes root as a combination of several works done in [Dang and Emad, 2014a, Gillberg, 2011, Weinbub and Hössinger, 2014] which propose an efficient parallel fast method targetting wider classes of HJ equations.

4.6.1 SOFIM principles

The SOFIM was originally proposed by Gillberg [Gillberg, 2011]. The method combines the FIM and the two-queue method [Gillberg, 2011].

The particularity of SOFIM is that the method pauses some of the awaiting updates according to a cutoff criterion based on statistical in-situ analysis of the solution values rather than computing all active points in parallel (as it is done by the FIM). Therefore, the computational resources are not fully used, obviously limiting the potential for parallel speedup relative to the FIM. However, the SOFI method offers excellent performance which has been shown for two-dimensional, sequential problems. In turn, the two-queue method also pauses nodes to get a partially ordered technique, but it is only applicable to isotropic problem formulations, whereas the SOFI method supports also anisotropic problems.

The parallel algorithm extends the original SOFI algorithm [Gillberg, 2011], albeit offering additional handling of shared-memory parallel programming aspects and an advanced cutoff method suitable for three-dimensional problems. We first introduce the required general data objects, discuss the initialization step and the actual parallel algorithm, and finally conclude with an analysis of the developed semi-automatic cutoff criterion.

The central algorithm entities, which have already been introduced previously with the FIM, are : the set of grid points x, the set of source nodes Γ, the active list aL, the paused list pL, the solution list u, the cutoff factor av, and the average solution m_{k} (k refers to the iteration counter) of the nodes in pL. Additionally, we propose to reuse the buffered parallel approach from the FIM by creating a temporary aL_{temp} to avoid expensive deletion processes of aL during the compute-intensive iterations. An essential aspect of the algorithm is the determination whether a node has been already added to aL or pL. To avoid an expensive lookup step, which would require finding the node in question within aL or pL, we use a tag-based system. To that end, we employ the aL_{tags} and pL_{tags} data structures, which provide us with element-based tag lookup for the expense of additional memory overhead. The coefficients c, $relax$, m_{k}, and $m_{\mathrm{k-1}}$ are required for our improved automatic cutoff computation, which will be explained later on.

4.6.2 Fine-grained parallel SOFIM

The initialization of the parallel SOFI algorithm sets all coefficients to zero, and the solution field is preloaded with an arbitrarily high number (e.g. 10^{12}). However, the solution of each source point is initialized with zero, whereas the source points themselves are added to the active list aL. This is different to the FIM, where not the source nodes themselves are added to the active list, but instead the neighboring nodes.

Algorithm 7 introduces the actual parallel SOFI algorithm. The main parallel loop is processing the active list aL as in the FIM. We use a *guided* scheduling method, as it has shown to be the best performing scheduling procedure, due to the irregular workload inside the parallel loop demanding a dynamic load balancing. The tag system ensures that the same nodes are not added to aL/pL again during an iteration. However, nodes might be reprocessed later on during a subsequent iteration, such is the general procedure of iterative methods. The use of write guards in form of atomic locks has been minimized to three spots. The neighbor (nb) iteration is required to generate the required 7-point stencil, which is used to discretize the eikonal equation's differential operator in three dimensions. Parallel write access to the pL and aL_{temp} data structures has been realized

via thread-exclusive containers (Lines 13,21), which - although requiring a serial merging step at the end of the parallel for loop - scales better for increasing thread numbers than guarding central data structures with additional critical sections. A similar technique is used for the cutoff procedure's essential coefficients t_{sum} and t_{sqsum}.

For the SOFIM to perform well, the algorithm for computing the cutoff coefficient av is essential, as av controls the assignment of a point to either the aL or the pL (Line 8). The cutoff level enforces an ordering of the nodes to be updated, in order to reduce the number of iterations needed. When too many points are activated (i.e. added to aL), the number of computations is high and the numerical solvers are slow. Similarly, if too many points are paused (i.e. added to pL), too few nodes are computed, as the ordering is too strict. Empirical investigations have shown best performance when approximately 80% of the nodes are activated[Gillberg, 2011].

The original method used for computing av is based on the average solution value of the paused nodes [Gillberg, 2011]. However, this approach does not perform well for general problems in three dimensions. The ordering enforced from this simple method tends to be too weak, since too many nodes are activated by being put into `aL`. When that happens, the additional cost of ordering computations outgrows its benefits. Therefore, we use a different method to compute the cutoff av, being based not only on the average solution value m_{k} but also on the standard deviation σ of the nodes in pL.

Assuming a normal distribution of the solution values of nodes in pL, we would activate approximately 84% by assigning a cutoff av of the average plus a standard deviation. However, a large spread (i.e. large σ) within pL indicates that a stricter ordering is needed. We estimate the average shift in cutoff level, by the difference between the current m_{k} and the previous: $\Delta m_{\text{k}} = m_{\text{k}} - m_{\text{k}-1}$. The original SOFI relaxation method is to have a cutoff level as a *relaxed* average by using $av = m_{\text{k}} + 1.5\Delta m_{\text{k}}$.

The additional coefficient c and $relax$ are used as additional parameters to adjust the cutoff computation, by investigating the pL-ratio, i.e. the number of nodes added to pL relative to the total number of nodes added to pL and aL (Lines 33-39). The used thresholds and coefficients have been shown to work best for the presented examples, but may be adjusted for more realistic devices, hence the designation *semi-automatic*.

Another mechanism to ensure that the computed cutoff level is reasonable is proposed, which is based on monitoring the number of iterations. If too many iterations are detected, it is assumed that the cutoff level is not optimal. Therefore, the cutoff procedure is restarted by triggering a recomputation of the cutoff level, increasing the chance of upholding a high convergence rate.

4.6.3 SOFIM Benchmarks

We investigate the performance of our parallel SOFI implementation relative to a reference FIM implementation. Our benchmarks cover different three-dimensional problems with varying problem sizes (100^3 and 200^3 Cartesian cube grids), speed functions, and single/multiple-source configurations (a single center source node versus 100 source nodes spread over the entire simulation domain).

Regarding speed functions, we investigate three different configurations which are :

1. constant speed (F_{const}), where for the entire domain $F = 1$ is used;
2. checkerboard speed (F_{check}), where the computational domain is divided into eleven equally sized cubes in each direction and the velocity is alternated between $F = 1$ to $F = 2$ from cube to cube [Gillberg et al., 2014][Chacon and Vladimirsky, 2012];
3. oscillatory speed (F_{osc}), where the speed function is modeled by a highly oscillatory continuous speed function [Chacon and Vladimirsky, 2012] :

$$F = 1 + \frac{1}{2} sin(20\pi x) sin(20\pi y) sin(20\pi z)$$

.

The benchmark platform is composed of a dual-socket node with two Intel Xeon E5-2620 (SandyBridge EP) 6-core (2 threads per core) processors with 128 GB of main memory. The parallel algorithm introduced in Section 2 has been implemented in C++. The presented execution performances are based on the median of five execution timings. The threads have been pinned to the individual physical cores via the likwid [Treibig et al., 2010] library to avoid thread-core reassignments, which would otherwise potentially introduce a performance penalty.

Figures from 4.17 to 4.19 depict the isosurfaces of the solutions of the individual test configurations for the 100^3 simulation grids. The results for the 200^3 are similar, albeit offering an increased resolution. To verify the correctness of the solutions, the FIM and the SOFI method results of the single source problem with constant speed for a 100^3 grid have been compared to an analytic solution given by the Euclidian distance function. The error norms of both methods are the same, being $L_1 = 29 \cdot 10^{-3}$, $L_2 = 10^{-3}$, and $L_\infty = 36 \cdot 10^{-3}$, indicating that the SOFI method computes the same result as the FIM. If indeed the results would be different, note that the ε used in the FIM's algorithm can prevent full convergence of the algorithm.

Figures from 4.20 to 4.22 compare the execution times and the parallel efficiency between our SOFI method and FIM implementation for a 100^3 grid. The SOFI method outperforms the FIM both for the single and multiple source configurations; for the more important multiple source setups (as these cases resemble real-world applications more closely) and 12 threads, a factor of 1.5 for F_{const}, 2 for F_{check}, and 1.7 for F_{osc} is achieved. The parallel efficiency of the single source test setups is by far inferior to the FIM, although both methods suffer in general from efficiency limiting factors typical for stencil computations, being cache misses and memory latency. This stems from the fact that the SOFI method inherently does not favor single source problems, as in this case no ordering is needed, thus introducing unnecessary overhead. However, for the more important multiple source cases, the scalability is reasonable: for 12 threads efficiencies of around 60% can be achieved for the highly challenging F_{check} and F_{osc} problems. The results show load balancing problems, which can be identified by the somewhat erratic parallel efficiency behavior. This fact is to be attributed to an unbalanced utilization of the aL and pL containers, triggered by insufficiencies in our automatic cutoff calculation.

Figures from 4.23 to 4.25 continue the investigation for an increased computational domain size, being 200^3, which allows to judge the performance under increased load.

Again, execution timings show that the SOFI method is faster than FIM. For the multiple-source cases and 12 threads, a factor of 1.9 for F_{const}, 2 for F_{check}, and 2.6 for F_{osc} is achieved. The parallel efficiency is comparable to the 100^3 grid results, being around 60% for the F_{check} and F_{osc} problems.

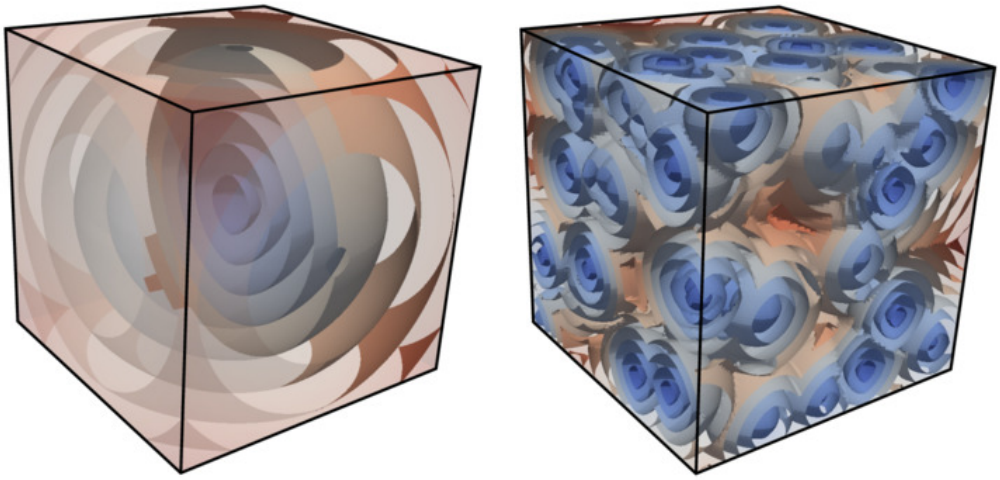

Figure 4.17: Isosurfaces of the F_{const} solution on a 100^3 domain for a single center source (left) and multiple sources (right)

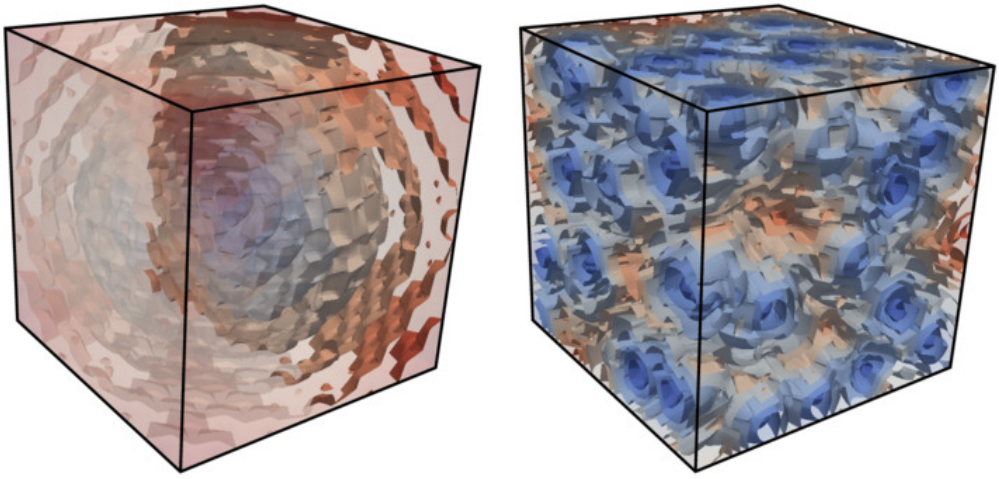

Figure 4.18: Isosurfaces of the F_{check} solution on a 100^3 domain for a single center source (left) and multiple sources (right)

Overall, the previously mentioned inferior parallel potential of the SOFI method relative to the FIM is reflected in the results, albeit being still reasonable, especially for more relevant multiple source scenarios. However, the execution time is what matters in real world applications. The parallel SOFI method is significantly superior to the parallel FIM, underlining the potential of parallel SOFI methods as a compelling alternative for solving the eikonal equation.

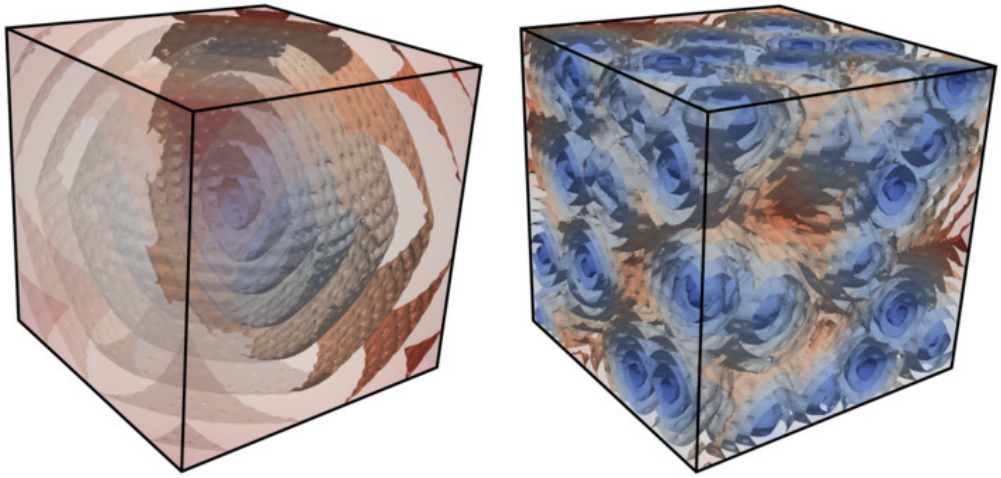

Figure 4.19: Isosurfaces of the F_{osc} solution on a 100^3 domain for a single center source (left) and multiple sources (right)

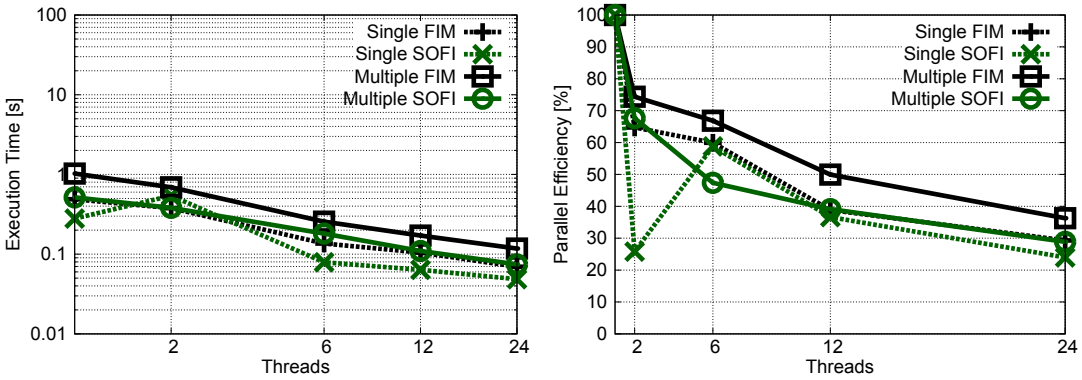

Figure 4.20: Execution times (left) and parallel efficiencies (right) of the F_{const} problem on a 100^3 domain

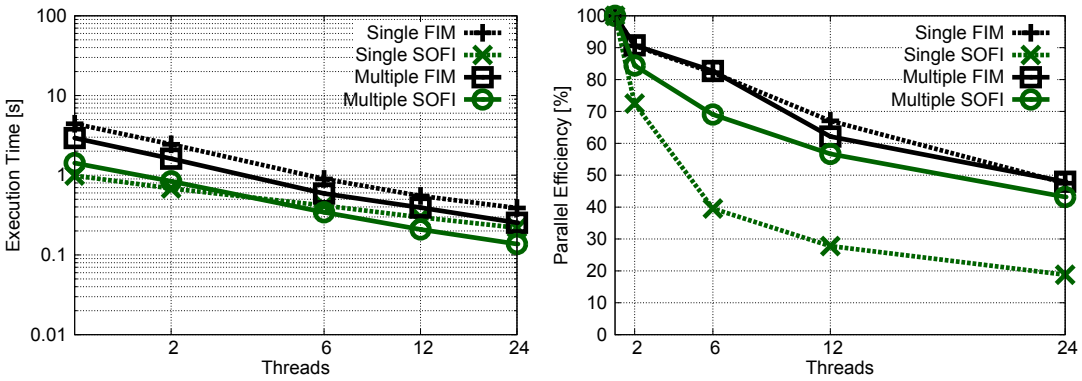

Figure 4.21: Execution times (left) and parallel efficiencies (right) of the F_{check} problem on a 100^3 domain

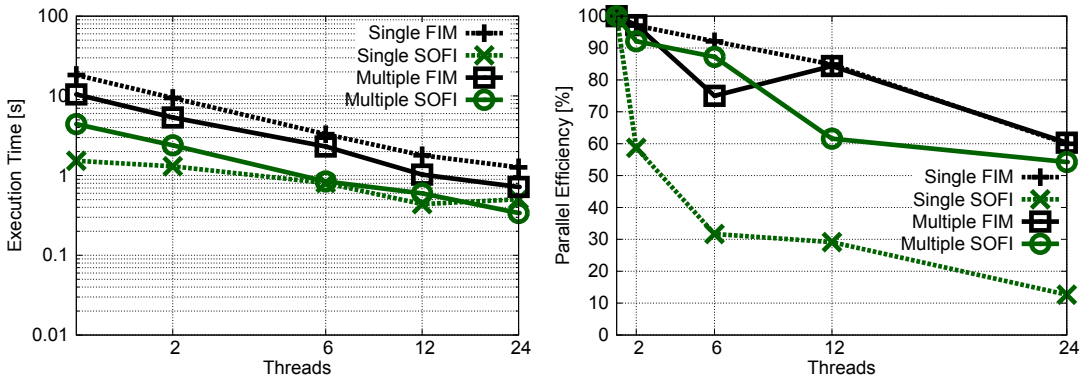

Figure 4.22: Execution times (left) and parallel efficiencies (right) of the F_{osc} problem on a 100^3 domain

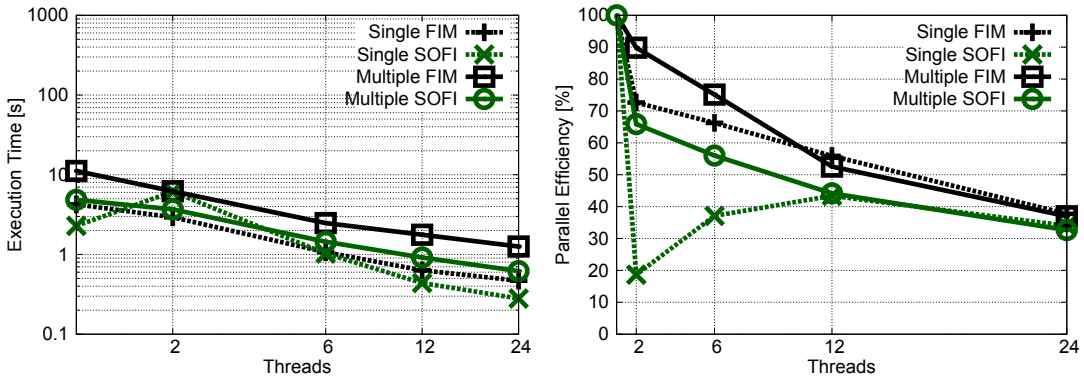

Figure 4.23: Execution times (left) and parallel efficiencies (right) of the F_{const} problem on a 200^3 domain

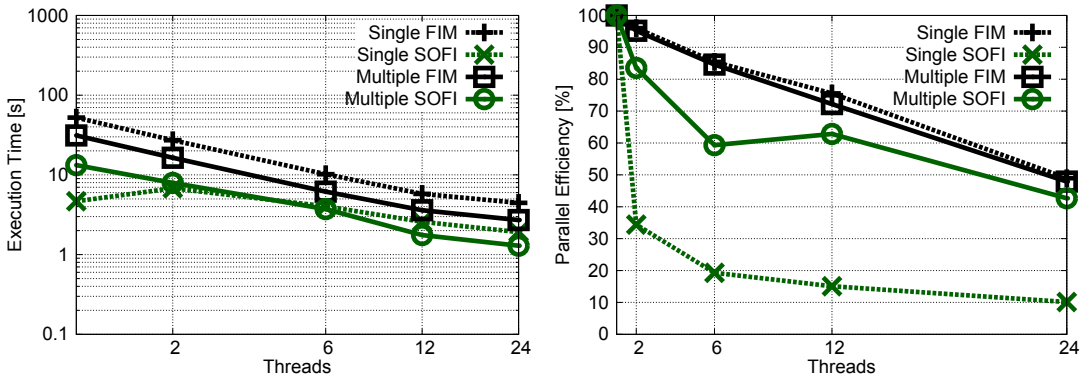

Figure 4.24: Execution times (left) and parallel efficiencies (right) of the F_{check} problem on a 200^3 domain

4.7 Summary on parallel BFIM and parallel SOFIM

The parallel buffered fast iterative method presented in section (4.3) has been developped in order to overcome the lack of efficient parallel algorithms to solve Hamilton-Jacobi

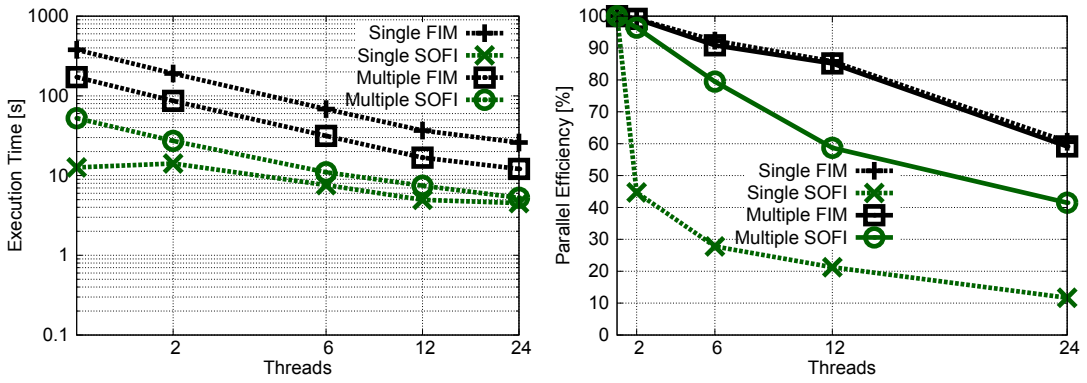

Figure 4.25: Execution times (left) and parallel efficiencies (right) of the F_{osc} problem on a 200^3 domain

equations. The method takes root with the original fast iterative method and manage to compute indepedently points in the narrow band with the help of intermediary buffers. The results show execution time and parallel scalability which are not obtained up to now compared to other recent parallel methods such as in [Detrixhe et al., 2013, Breuss et al., 2011].

A coarse-grained strategy for the FIM is also proposed. While the method gives good results and some speedup for few processes, it is not yet scalable for large scale computations. A slave-master model has therefore been proposed and has yet to be implemented efficiently. The achievement of this step would allow an efficient hybrid strategy combined with the BFIM.

In section (4.6), we have proposed an approach for parallelizing the SOFI method. The SOFI algorithm differs from the FIM by handling larger classes of HJ equations. The idea of the approach is to iterate the BFIM strategy over the SOFIM. Several experiments have been made in order to find a good compromise in order to have fast and scalable results. Beyond the parallel strategy, we have also worked on improving an alternative cutoff method supporting three-dimensional problems for semi-automatically driving the applied iterative Two-Queue technique. Our parallel SOFI algorithm offers superior execution performance relative to a reference FIM implementation for different speed functions and problem sizes, while offering reasonable parallel efficiency. The detailed benchmarks in (4.6.3) shows the excellent capabilities of the SOFI method for tracking front propagation.

Algorithm 7: Parallel SOFIM algorithm

```
begin
  while aL ≠ ∅ do
    foreach x ∈ aL in parallel do
      aL_tags(x) ← 0;                                          // critical
      foreach neighbor x_nb of x do
        if u(x) < u(x_nb) then
          u_new ← SolveEikonal(x_nb)
          if u(new) < u(x_nb) then
            if u(new) > av then
              if pL_tags = 0 then
                pL_tags ← 1 ;                                  // critical
                u_sum ← u_sum + u_new
                u_sqsum ← u_sqsum + u²_new
                add x_nb to pL
              else
                u_sum ← u_sum + u_new − u(x_nb)
                u_sqsum ← u_sqsum + u²_new − u(x_nb)²
            else
              if aL_tags(x_nb) = 0 then
                aL_tags(x_nb) ← 1;                             // critical
                add x_nb to aL_temp
          u(x_nb) ← min(u(x_nb), u_new);                       // critical
    merge(aL_temp) and swap(aL, aL_temp)
    aL_temp ← ∅
    if aL − swaps > ∛size(x)/10 then
      av ← 0.0
    if av > m_k then
      if pL − ratio < 0.5% and aL − swaps > 5 then
        c ← 0.8c
        av ← m_k − relax
      else if pL − ratio > 99% and aL − swaps < 5 then
        c ← 2.0c
        av ← m_k − relax
    if aL = ∅ then
      merge(pL); swap(aL, pL) and swap(aL_tags, pL_tags)
      pL ← ∅
      m_k ← Σ u_sum/size(aL)
      σ ← √(Σ u_sqsum/size(aL))
      relax ← c(2(m_k − m_{k−1} + σ))²/(6σ²)
      u_sum ← 0.0 and u_sqsum ← 0.0
```

Chapter 5

Sequential/parallel reusable library

Contents

5.1 Reusable libraries for solving HJ equations

We present in this section a state of the art of the actual libraries which are closely related to HJB equation and we investigate the reusability notion, its evolution, its meaning in scientific libraries.

Scientific computing concerns a very wide large range of applications from chemistry, fluids dynamics, weather forecasting to geoscience... As we have seen in section 1, numerical simulation generally implies to discretize the problem leading to numerical analysis and/or linear algebra problems. These problems can be solved using different softwares/libraries which are specialized towards specific problems. Nowadays, tendency in scientific libraries is to propose different ways to interact between other softwares and reuse specialized codes.

5.1.1 Brief reusability overview in scientific libraries

Reusable code is essential to avoid duplicated effort in software development. Instead of rewriting software components from scratch, a programmer can make use of an existing component. In addition, reusable code helps performance by making it possible to put reusable software components in shared libraries. The concept of code reusability is not new. In a report from 1993 [Andreae et al., 1993], the authors make the distinction between two issues : the process of how to reuse code and the process of how to write more reusable code. The report focus on the way to write more reusable code which is our main concern.

In numerical linear algebra, where matrix computation is a concern (e.g. eigenvalues computation, linear system resolution), libraries such as BLAS [Dongarra et al., 1988] and Lapack [Anderson et al., 1999] are considered as references in the scientific community and used in many industrial contexts. These libraries implement efficient computations routines which are designed to be efficient. However, using these raw libraries do not take into account the complex environment, architectures on which they are run on. Jack Dongarra himself shows that BLAS does not allow multi-level parallelism and the way BLAS is implemented does not allow high level abstraction (no data type abstraction such as matrix). In adding to that, these libraries also does not allow reuse between the parallel and sequential versions of the applications. The subroutines of the solvers are not able to adapt their behaviors depending on the data types. Those subroutines must be defined once for use in sequential and once again in parallel.

Therefore, generally BLAS are undirectly used via an API or other applications which take care of the BLAS optimization according to the hardware. Higher level linear al-

gebra libraries which propose to wrap BLAS routines have emerged since with different usage, such as Eigen [Guennebaud et al., 2010], PETSc [Balay et al., 2012] or Trilinos [Heroux et al., 2005]. Also most of high level computation softwares such as Mathematica [Wolfram, 2003], Matlab [Higham and Higham, 2000] or Scilab [Scilab Enterprises, 2012] make use of BLAS. We investigate more in details the parallel/sequential reusability aspect in the next subsection.

5.1.2 Sequential/parallel reusability : a recent challenge

New parallel technologies are changing computing needs as seen in the introduction. A technology lifetime can sometime be shorter than the time it takes to port applications. Therefore programmers need support to design reusable applications which can fully exploit without necessarily being parallel programming or parallel architecture experts.

Parallel reusability is a recent concern where new parallel architectures such as new CPUs, or GPUs are much more accessible to the masses. Industrials can now use these architectures at a low cost for their needs. Also, parallel paradigm languages becomes much more user friendly to program such as OpenACC for accelerating hardwares (GPGPU) or OpenMP for shared memory multiprocessing. Therefore, it seems important to write algorithms and codes in a way that they can be parallelized easily on any parallel architectures. This is not generally the case and often, a specialist has to rewrite a huge amount of code and only for a specific parallel architectures. In our library, we aim to have a generic approach, where you can find parallel codes which do not interfere with sequential codes. Both codes use the same core functions. Parallel design pattern [Kjolstad and Snir, 2010] are helpful to design parallel code which can be algorithmically reproduce in a same appropriate context.

Although recent scientific libraries provide possibilities for parallel computing, they often lack potential parallel reusability. Packages in Trilinos for instance have been written originally in a sequential way implying a considerable potential addition of work both in parallel algorithmics and implementations.

High level libraries or computation have to find compromises and cannot fulfill every needs perfectly. For instance, high level computation softwares which propose user friendly interface for the end user do not propose efficient multi-level parallelism. PETSc and Trilinos show efforts to propose to the user tools to manage parallelism. These libraries enforced drastically the modularity, interoperability and reusability of high level components within the libraries as well as in the user applications. Using PETCs or Trilinos, the application specifies the building blocks of the solver. However the solver is provided by the library and the application code no more contains the logic of the method. It provides parallel and sequential solvers and allows to make use of one and/or the other in the same application. The parallel and sequential solvers still use different application codes [Dandouna, 2012, Noulard and Emad, 2001].

5.1.3 State of the art of libraries for HJB equations

In finite element libraries such as deal.II [Bangerth et al., 2007], DiffPack [Langtangen, 1999], Getfem++ [Fournié et al., 2010], libMesh [Kirk et al., 2006], DUNE [Bastian et al., 2008],

the finite element computations are often separated from linear algebra computations. The latters are either implemented directly or use a specialized linear algebra packages as seen previously (PETSc, Trilinos). In [Kronbichler and Kormann, 2012], the authors challenge the view of separating linear algebra from finite element assembly routines and show that it can become arguable from a performance perspective. Regarding numerical libraries for solving HJ equations, there is no such code publicly available to our knowledge. The difficulty to find numerical code for HJ equations solution comes to the numerical approximation which requires development of a significant code base to support gridding, initial conditions, approximation of spatial and temporal derivatives/integration and visualization. [Mitchell and Templeton, 2005] is an attempt to propose a toolbox for solving Hamilton-Jacobi using MATLAB. The authors argue that there is no such collection of code publicly available. The toolbox has many features such as dimension flexibility, high order accuracy, handle level set methods... However, the toolbox is a MATLAB code which targets numerical scientists, prototypes and is hardly parallelizable nor usable for industrial codes.

Regarding industrial codes, the libraries mentionned at the beginning of this subsection can be used by software engineers. They are written in C++ language, they are fast, and relatively parallelizable. It would be possible for most of the libraries to handle some HJ problems but we would still have to implement the whole method in order to be efficient. Fast methods are too specific to use for a particular problem. We present briefly Trilinos and deal.II, two different libraries which are used in the scientific community, and their position regarding sequential/parallel reusability. Trilinos [Heroux et al., 2005] is a collection of open source software libraries, does not currently have efficient way to manage structures meshes for finite element purposes. For the interested reader, we can check the Intrepid et PAMGEM packages from Trilinos. deal.II [Bangerth et al., 2007] is a quite recent library which is targetted on the computational solution of partial differential equations using adaptive finite elements. deal.II offers scientists the possibility to add their own finite elemente codes. The project is highly collaborative, and support parallelization (Intel TBB and MPI). In deal.II, the authors show that they are much concerned about parallel computing on different architectures such as shared-memory [Kronbichler and Kormann, 2012] or massively parallel processors (MPPs) [Bangerth et al., 2012]. However their code examples require to be quite familiar with parallel issues and do not provide simple ways of dealing with them. Despite modularity and reusability of their high level components, these object oriented libraries rarely allow the simultaneous reusability of components between the sequential and the parallel versions of an application.

These two libraries rely on community (Trilinos, deal.II) and offer the possibility to any experts in a specific area to submit a package or function which can be integrated. Contribution in open-source softwares are the key to their success and the modularity approach is an important key to make codes which last long.

5.1.4 Par4HJB and Hamijac C and C++ libraries for solving HJ equations

With these key points in mind, we manage to design a library which would aim to propose multi-level parallelism to the user directly for their needs. The design ideas could be reused for other PDE solvers.

Figure 5.1 shows a overview of how the libraries Par4HJB and Hamijac are basically designed in a functional point of view.

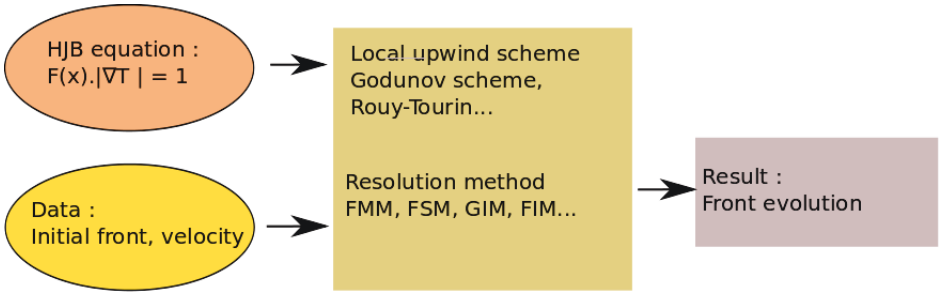

Figure 5.1: Basic libraries functional overview

Par4HJB is written and was realized so that it can be easily integrated in any bigger numerical solver. The C language was chosen for two main reasons :

1. For integration in a C or C++ solver
2. Being able to be used with Par4all which would generate GPU code generation [Amini et al., 2012, Amini, 2012]

However, Par4all is not capable to offer an efficient parallelization for Par4HJB since the data parallelization is has a complexity which can be hardly exploitable with Par4all. Furthermore, the lack of object-oriented paradigms, make a reusable code harder to write and maintain. Hence, we also began to write a new C++ library Hamijac based on the work done on Par4HJB where we aim to enhance syntactic sugar in a way that the library syntax is designed to be even more readable and usable. Using the C++ Standard Library helps us to make use of efficient containers and methods where our own made structures would be surely slower. In terms of code reusability, C++ should be clearly preferred over C where the use of objects, inheritance, encapsulation and polymorphism makes the life much easier for the developper. The concept of encapsulation is to hide some information and prevent unwanted external direct access to functions or structures which are not supposed to use or modify the information. This concept limits potentially programming errors and can hide some complexity to the user. Inheritance promotes code reuse through a hierarchical system where a class can be derived from a base class and share some common properties. Finally polymorphism is the ability to provide a single interface which can be used for different types. Genericity can be achieved through polymorphism.

The intested reader can take a look at the paper [Siff and Reps, 1996] which propose a strategy to convert existing C programs to C++ to obtain a better genericity. In our case we have managed to rethink our design and start from a different mindset.

Sometimes, we will present different codes with C or C++ syntax for easier lisibility. The C syntax refers to Par4HJB and the C++ syntax refers to the Hamijac library. Both libraries basically share the same principles and functionnalities. Their design architecture is what make them essentially different. When we talk about Par4HJB, it can thus refer to Par4HJB and Hamijac.

5.2 Algorithmic reusability

Numerical functions are sometimes meant to be reusable for other methods. We present in this section some algorithmic generic functions, which can be reused at any level. Making some routines available to the high end user, can allow him to make use of these functions in other extern libraries. We see that algorithmic genericity has a price and imply to think about some formulae or models before implementations.

5.2.1 Local numerical scheme for high dimensions

For futher implementation in a n-dimensional case, we generalize the numerical Godunov scheme. Let $u_{i,min} = \min(u_{i-1,j}, u_{i+1,j})$, be the minimum time between two neighbors of $u(x)$ on dimension i. Thus, we have $\max(D_x^- u_{ij}, D_x^+ u_{ij}, 0) = max(\frac{u-u_{i,min}}{h_i}, 0)$. For instance, in a three dimension domain we obtain $g(x)$ an approximation of $H(x, \nabla u)$ with a first order Godunov discretization :

$$\begin{aligned} g(x) &= \left[\frac{\max(u(x) - u_{i,min}(x), 0)}{h_x}\right]^2 + \left[\frac{\max(u(x) - u_{j,min}(x), 0)}{h_y}\right]^2 \\ &+ \left[\frac{\max(u(x) - u_{k,min}(x), 0)}{h_z}\right]^2 - \frac{1}{F^2(x)} \end{aligned}$$

which gives supposing $u(x) - u_{min}(x) > 0$

$$g(x) = \left(\frac{1}{h_x^2} + \frac{1}{h_y^2} + \frac{1}{h_z^2}\right).u^2 + 2.\left(\frac{u_{imin}}{h_x^2} + \frac{u_{jmin}}{h_y^2} + \frac{u_{kmin}}{h_z^2}\right).u + \left(\frac{u_{imin}^2}{h_x^2} + \frac{u_{jmin}^2}{h_y^2} + \frac{u_{kmin}^2}{h_z^2}\right) - \frac{1}{F^2(x)}$$

After calculations we obtain in a n-dimension space, with regular strides i.e. $\forall i \in \{1, ..., n-1\}, h_i = h_{i+1}$:

$$g(x) = n.u^2 - 2.\left(\sum_i u_{i,min}\right).u + \sum_i u_{i,min}^2 - \frac{h^2}{F^2} \tag{5.1}$$

Considering a regular stride, we illustrate a simple generic scheme which can be used for any dimension in algorithm (8). The function returns a number corresponding to the potential new value of a point mesh at *idx*. *vel* corresponds to the velocity field F. Note that we take into consideration a regular grid here where step grids can change depending on dimension rank.

Algorithm 8: A simplified generic local scheme

$a, b', c \leftarrow 0.0$
$neighbors \leftarrow findNeighboursAt(idx)$
for $i \leftarrow d = 0$ **to** $D - 1$ **do**
  $left \leftarrow neighbors[2 * d]$
  $right \leftarrow neighbors[2 * d + 1]$
  $D_- \leftarrow u(left)$
  $D_+ \leftarrow u(right)$
  $u_{min} \leftarrow min(D_-, D_+)$
  if $u_{min} < u(idx)$ **then**
    $a \leftarrow a + 1/\Delta_d^2$
    $b' \leftarrow b' + u_{min}/\Delta_d^2$
    $c \leftarrow c + u_{min} * u_{min}/\Delta_d^2$
return *Quadratic solution of :* $a * x^2 - 2 * b' * x + c$

With irregular strides i.e. $\exists i \in \{1, ..., n-1\}, h_i \neq h_{i+1}$,we obtain the following formula :

$$g(x) = \sum_i h_i^2 . u^2 - 2. \sum_i \left[\left(\prod_j \frac{h_j}{h_i^2}^2 \right) . u_{i,min} \right] . u + \sum_i \left(\prod_j \frac{h_j}{h_i^2}^2 . u_{i,min}^2 \right) - \frac{\prod_j h_j^2}{F^2}$$

which leads to

$$g(x) = \frac{\sum_i h_i^2}{\prod_j h_j^2} . u^2 - 2. \sum_i \left(\frac{u_{i,min}}{h_i^2} \right) . u + \sum_i \left(\frac{u_{i,min}^2}{h_i^2} \right) - \frac{1}{F^2}$$

We then have to compute and get the sign of $\Delta' = \left(\sum_i \frac{u_{i,min}}{h_i^2} \right)^2 - \frac{\sum_i h_i^2}{\prod_j h_j^2} . \left[\sum_i \left(\frac{u_{i,min}^2}{h_i^2} \right) - \frac{1}{F^2} \right]$ where $a = \frac{\sum_i h_i^2}{\prod_j h_j^2}$; $b' = \left(\sum_i \frac{u_{i,min}}{h_i^2} \right)$ and $c = \left[\sum_i \left(\frac{u_{i,min}^2}{h_i^2} \right) - \frac{1}{F^2} \right]$ in order to give the value function solution at the given vertex.

Thanks to this type of algorithm there is no need to declare several functions according to a specific dimension size. The algorithm is generic and avoid code redundancy and any method can reuse this local scheme.

5.2.2 A multi-dimensional mesh proposition

Mesh management follows the same generic idea. Meshes in the program are created by defining lower/top boundaries which are shown on figure (5.2).

For instance, a 3D regulard grid can be represented thanks to its dimension $D = 3$, its lower boundaries $[x_{min}, y_{min}, z_{min}]$, max boundaries $[x_{max}, y_{max}, z_{max}]$ and its steps size $[\Delta_x, \Delta_y, \Delta_z]$ or its number of vertices per side $[n_x, n_y, n_z]$.

For performance and scalability purpose, the mesh is not stored in nested arrays but in a linear one dimensional array in such as way that choosing the right ordering (row-major here in C language) provide a contiguous access to the elements. The approach we propose is the following.

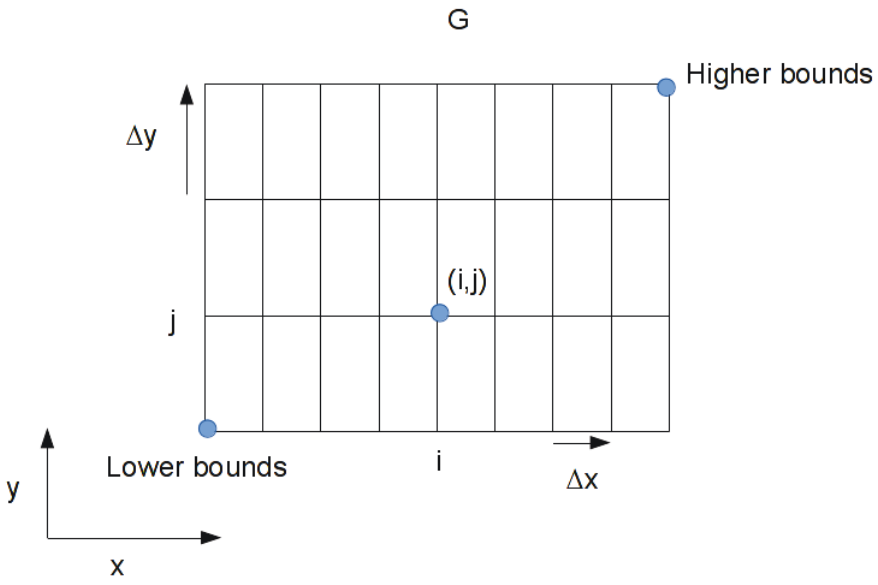

Figure 5.2: Mesh management for regular grids

Let G be a D dimension regular grid where $(\Delta_i)_{i\in[1,D]}$ is the stride on the i-th dimension and $(N_d)_{d\in[1,D]}$ the number of vertices on one dimension i. The lower and upper bounds are called *lower* and *upper* respectively. The total number of vertices in the grid is $N = \prod_{i=1}^{D} N_d$. A representation of the array is proposed in figure 5.3.

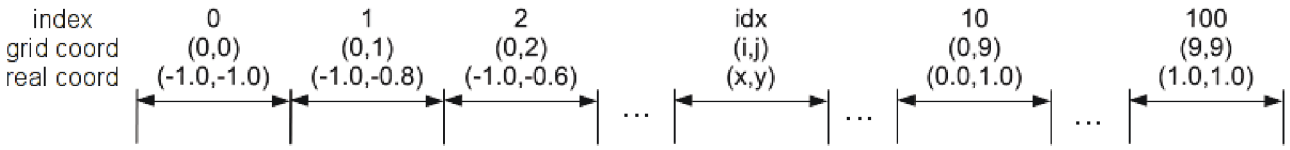

Figure 5.3: One dimensional storage for a 2D 10×10 grid with $[-1.0, -1.0]$ lower bounds and $[1.0, 1.0]$ upper bounds

This implementation in our library allow to provide generic algorithms in order to easily access the elements in the array such as :

- the index, which represents the memory offset of the element in the array, grid coordinates or real coordinates.
- the grid coordinates which are specified by a tuple of indices $(i_1, i_2, ..., i_D)$

- the real coordinates generated from the lower and upper bounds which are represented by a tuple of real values $(x_1, x_2, ..., x_D)$

Accessing to the index from the grid coordinates is done by computing with D representing the problem dimension :

$$idx = \sum_{j=1}^{D}(\prod_{k=j+1}^{D} N_k)N_j$$

Getting the grid coordinates from the index requires more efforts and should be called wisely in any algorithms because of the potential computation cost.

Algorithm 9: Get the grid coordinates from the index offset array

Data: (idx): integer offset index
Result: $(i_1, i_2, ..., i_d)$: indices of the grid
$a \leftarrow idx$
for $i \leftarrow 1$ **to** $D - 1$ **do**
  $b \leftarrow \prod_{k=i+1}^{D} N_k$
  $q_i \leftarrow a|b;$ `/* perform euclidian division */`
  $r \leftarrow a\%b$
  $a \leftarrow q_i$
return $(q_1, q_2, ..., q_D - 1, r)$

The interested reader can refer to the appendice C for further accessors implementation such as the one used for real coordinates. Thanks to these accessors, value function or fields of a particular point are easily accessible.

A modern C++ implementation is also available in the appendice C listing C which provides average $O(1)$ access. The memory usage use static storage for efficiency thanks to template programming. Index to coordinates access uses hash tables. We intend to limit memory usage in the future by providing direct compile-time access.

5.2.3 Managing first and two orders finite element discretization

In order to preserve the multi-dimensionnal generic aspect in the solver, and optimize code reuse, the neighbors are stored in a specific manner in a one-dimensional vector such as shown on fig. 5.4 for 2D example with a second-order stencil scheme.

We consider only adjacent neighbors (hence no diagonal vertices are considered). Given a nth-order stencil scheme in D dimensions, vector V size is $length(V) = 2 \times D \times n$ where each element of its element V verifies

$$V[2.d.n + idx] = u_{d+idx-n+\delta}$$

where $\begin{cases} \delta = 1 \text{ if } idx \geq n \\ \delta = 0 \text{ else} \end{cases}$ and current dimension d verifies $0 \leq d < D$.

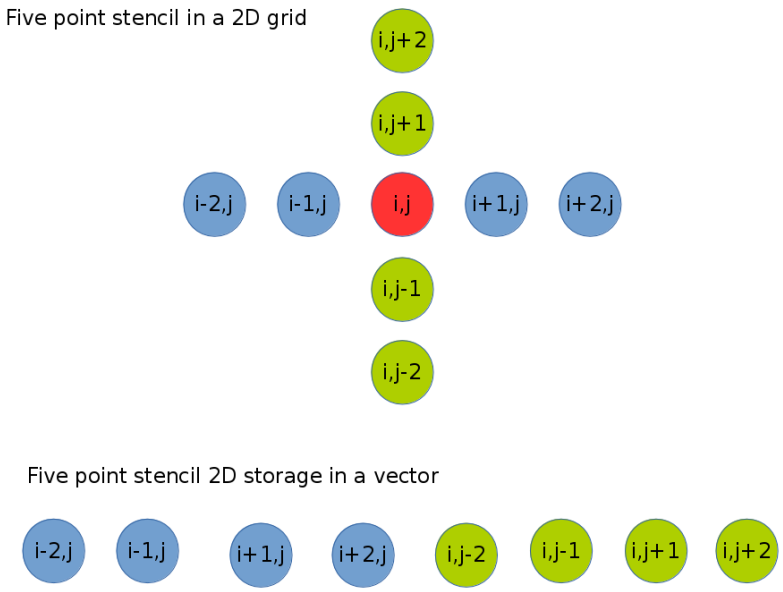

Figure 5.4: Multi-stencil management example in 2D with five-point stencil

5.3 Software reusability in Par4HJB

Par4HJB aims to allow reusability of code. We can represent reusability as an ensemble of practical design features such as :

- Orthogonality guarantees that modifying one part of a code neither creates nor propagates side effects to other parts of the program. The language has to be easy to describe, learn and to implement.
- Flexibility allows to change the setup of an application with minimal effort.
- Usability refers to the ease with which users interact with the software.
- Maintainability allows to uphold the code quality.
- Expandability denotes the ability to add functionality.

Writing a reusable code requires high level abstractions which imply to think carefully about the code design. For instance, the Unified Modeling Language (UML) a general-purpose modeling language which intends to provide a standard way to visualize the design of a system can be required before writing any code. This process is quite common before producing any industrial code since there are many benefits including readability (collaboration work in a team), testability (debugging will be more efficient) and also maintainability (less time consuming to change, extend the code).

As shown on figure 5.5, graphical possibilities to represent the code design are key tools to comprehend complex codes. Par4HJB uses Doxygen [Van Heesch, 2004] to generate code documentation which is also used in softwares such as deal.II or Trilinos. The utility

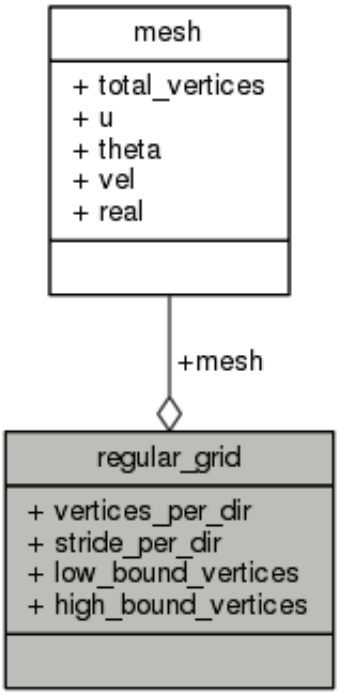

Figure 5.5: Hierarchy generated by the Par4HJB library documentation

can concern both the users and the developers. In the next subsection we will try to identify the different type of users.

5.3.1 Making the difference between the end user, the advanced user, and the developer

We can differentiate types of software users : the end user, the advanced user, and the developer. The end user is a person who utilizes software tools and is not interested in the technical details of the tool. The end user is only interested about the input/output and considers the software as a black box. For instance, a scientist interested in using the fast marching can use Par4HJB routines which are recognizable with the prefix “PHJ” and does not have to know how the library is implemented in details. The provided examples show that we intend to simplify the simulation process for the user by providing simple, few, straightforward functions. From a development point of view, the end user requires particular treatment regarding the usability where technicalities must be hidden as good as possible, and some automatisms have been provided. The advanced user is similar to the end user. The difference is that the advanced user has more skills to investigate software tools in details, can comprehend the global technical aspect of the library and can exploit the subtilities of the library such as different parameter options. For instance, in Par4HJB, an advanced user would be able to take the output of the fast marching method results and implement his/her own gradient descend to compute a shortest path. The advanced user is able to modify some parts of the library where he/she is specialized in. The developer develops software used by either advanced or end users and is aware

about the detailed technical implementation. Developers often require low-level access to the software but any contributors to the Par4HJB or Hamijac libraries can be considered as developers as they have produced code which would be used.

5.3.2 Towards a generic library

In terms of development, Par4HJB functions intends to be reusable. Solving methods (FIM, FMM, RTM) use the same numerical scheme function. It is possible to provide different scheme other than the default Godunov one. The numerical scheme itself can use functions provided by the library such as stencil coding functions or grid management.

In Par4HJB which is written in C language, the benefits from an object-oriented programming are recreated in the library with the use of function pointers, void pointers or virtual tables (vtables) which can provide run-time polymorphism possibilities. Indeed, generic pointers allow to point to data of different types at different times. For instance, the qsort C standard library uses these techniques. Other parameters are defined however at compile-time such as the data type (float, double) and other typedefs for convenience.

For illustration purpose, let us check in details the prototype of our generic initializing front function.

```
t_uint PHJ_init_front(t_float *source, void *front_data,
    FRONT_FUNC_PTR front_function, s_mesh mesh);
```

This fonction takes several parameters : *source* is an array representing the center or vertices of the front. $*front\_data$ contains information about the geometry front (radius for a circle, thickness for a segment, number of vertices for a polygon...). $front\_function$ precise the front function to apply depending of the front nature ($front\_cirle$, $front\_polygon$, $front\_square$...). *mesh* represents the mesh structure which can be cartesian or regular.

Regarding the datatypes, methods, the function returns the number of vertices in the initial narrow band. The type is a t_uint which is a typedef where a user can change it to an other data type. $*front\_data$ is a void pointer since we only know at runtime the type of the front used. $front\_data$ can indeed be an integer representing the number of vertices for a polygon or a float representing the radius of the circle front. We reuse existing definitions which are standards such as boolean values defined in <stdbool.h> and also functions such as $fabs()$ defined in <math.h>.

5.4 Code evolution for reusability purpose

Whenever we want to create a new way of encapsulating data of arbitrary type, that is a type constructor, we ask ourselves how can we reuse our existing libraries on data that is encapsulated through this type constructor. In C, having polymorphism and abstraction implies to use virtual tables (also called vtables). Par4JHB which is written in C, heavily relies on them. C++ offers an object-oriented programming (OOP) aspect which has many advantages for complex high-level code. Classic OOP mostly relies on virtual methods to implement abstraction and polymorphism. Using virtual abstraction is convenient but we must be aware that there is a run-time overhead which can be explained

with two reasons [Driesen and Hölzle, 1996]. One reason is that virtual methods are implemented via vtables (presented in the precedent paragraph) in which function pointers are stored. A call to a virtual method implies to check the address of the function to call from the vtable which takes some time. An other reason is that the compiler generally cannot known which function will be called and inlining or other kind of optimizations are not possible.

The Standard Template Library (STL) [Austern, 1998] manages to combine code performance and abstraction by using complex techniques based on templates. Several techniques have been proposed for programmers to obtain efficient generic programming such as in [Alexandrescu, 2001]. Hamijac tries make uses of such advanced C++ techniques. We present in figure (5.6) the UML design used in Hamijac where the design proposed does use abstraction through template, functors and avoid the use of inheritance on several levels. The design is therefore flatten, and still provide strong possibilities through the use of techniques such as shown in subsection (5.5.4).

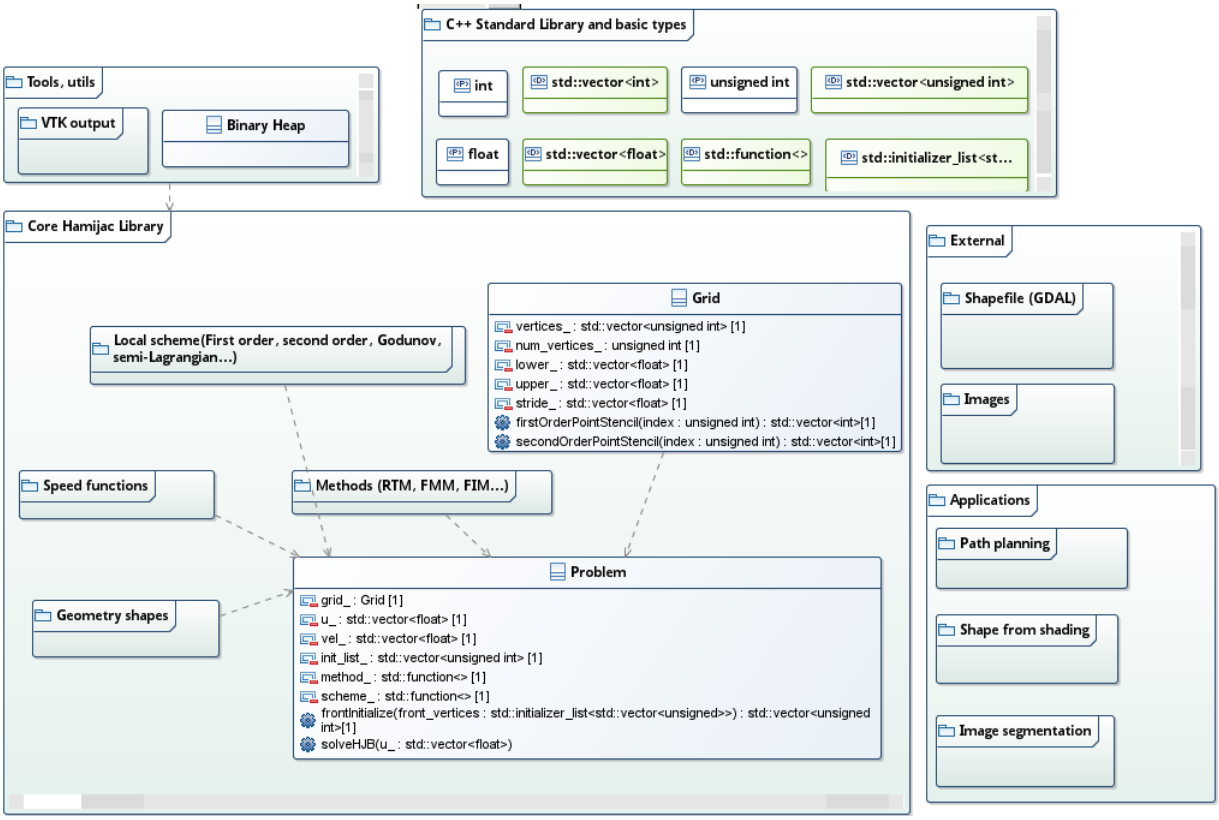

Figure 5.6: Simple UML design of Hamijac a modular library

The core namespace should propose to the user basic functions to simulate the wanted problem. Notice that the external and applications namespaces do not have dependencies with the core namespace. Indeed, the namespace "external" can be considered as interfaces to provide bridges for the user who would like to interpret datas such as shapefile or images into a Hamijac problem. The namespace "applications" provide helpful tools to render and post process the simulation into a concrete context.

5.4.1 Par4HJB and Hamijac make use of design patterns

The C and C++ community has seen good practices emerging to obtain performant reusable code. For instance, C++ programmers are aware of design patterns. In sofware engineering, design patterns are formalized originally in [Gamma et al., 1995]. The ideas take root from work of the architect Christopher Alexander who describes the term "pattern" with these words : "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

For instance, our library uses the facade pattern. The Facade Pattern hides the complexities of the system by providing an interface to the client from where the latter can access the system through a unified interface. Facade defines a higher-level interface that makes the subsystem easier to use. For instance, the problem class allows the user to work in a complex setup for his HJB problems while providing him simple methods to use to change his parameters.

A strong concept in software design is also to reproduce certain steps of an algorithm without changing the algorithm's structure. This behaviour can be obtained thanks to the the template method pattern which defines the program skeleton of an algorithm in a method, called template method, which defers some steps to subclasses. The strategy pattern (also called the policy pattern) enables an algorithm to be selected at runtime. For instance in Hamijac, it is possible to call different methods (FMM, RTM, FIM) and schemes (change order, stencil) in the problem class. This can be achieved through the STL class $std :: function$ which is a general-purpose polymorphic function wrapper and allows to minimize the creation of supplementary classes while being intuitive to use.

5.4.2 Libraries implementation : a brief overview

Both libraries Par4HJB and Hamijac are aimed to be built in such a way scientists can easily pick the grid, methods, numerical schemes in the solver depending on their needs. A modular approach is privileged.

We present below a simple example call of our library par4HJB.

```
// Include the par4hjb library
#include <par4hjb/par4hjb.h>

int main()
{
  // We set the umber of nodes per dimension/direction
  // Step size will be automatically computed with PHJ_cartesian_grid
       function
  int num_nodes_per_dir = 100;

  // We define the extreme points in the grid
  double bottom_left[2] = {-5.0, -5.0};
  double top_right[2] = {5.0, 5.0};

  // We set the dimension of the problem (default is 2D)
```

```
  PHJ_set_dimension(2);

  // We create a cartesian grid
  PHJ_s_regular_grid reg_grid = PHJ_cartesian_grid(num_nodes_per_dir,
      bottom_left, top_right);

  // We give the center of an initial front source
  double source[2] = {0.0, 0.0};

  // We create a circle initial front with the pointer function
     front_circle
  PHJ_init_front(source, 1.0, PHJ_front_circle, reg_grid.mesh);

  // Uncomment the following line if you want to verbose grid
     information
  // PHJ_disp_regular_grid(reg_grid, LOG_INFO);
  // You can also set PHJ_set_verbose(LOG_INFO) for instance for the
     whole simulation

  // Use the Fast Iterative method to solve the problem
  PHJ_FIM(reg_grid);

  // Do not forget to free the grid when you don't need it anymore !
  PHJ_free_regular_grid(reg_grid);
}
```

End-user functions are recognizable with the prefix "PHJ" of the library. As we can see, informations about the simulation has to be processed through function parameters. The latters allow abstraction thanks to pointer functions such as front initialization. More concrete examples are available in the samples provided with the library [Dang, 2014] which illustrate more possibilities.

Hamijac is an object-oriented library written in C++ which take features of Par4HJB with a different designs. Hamijac make use of different techniques : pattern design, template metaprogramming in order to achieve abstraction.

We present below an example of a call with our library Hamijac :

```
// Include the Hamijac library
#include <hamijac.h>

void test_front_obstacle()
{
  RegGrid<2, float, 1000> grid(-2.0f, 2.0f); // a 2D 1000x1000
     regular grid
  auto point = createPoint(grid);
  auto polygon = createPolygon(grid);
  auto circle = createBall(grid);
  auto line = createSegment(grid);

  auto problem = createProblem(grid);

  auto init_point = point({1.4, 0.2});    // reaching point for path
     finding
  auto circle_init = circle({-1.5, -1.6}, 0.1);
```

```
  problem.frontInitialize({init_point, circle_init}); // two initial
      fronts

  auto polygon_1 = polygon({{0.2, 1.2}, {1.0, 0.3}, {1.1, 1.3}}); //
       triangle
  auto polygon_2 = polygon({{-0.4,-1.2}, {-1.0, -0.3}, {-1.4, -1.4},
      {-1.6, -0.6}}); // quadrilateral
  auto circle_obs = circle({-1.2, 0.6}, 0.3);
  auto line_1 = line({{-1.2,-1.4},{1.0,0.8}}, 0.1);

  problem.setObstacle({polygon_1, polygon_2, circle_obs, line_1}); //
       using std::initializer_list
  problem.setHJBScheme(&scheme::simpleFirstOrderCartesian<2,float,
      1000>);

  t_uint num_iterations = problem.solveHJB(&method::
      SemiOrderedFastIterative<2, float, 1000>);
  std::cout << "SOFIM done in " <<  num_iterations << " iterations."
      << std::endl;
  saveVTK(num_iterations, "SOFI", problem);

  // Finding the path using gradient descent
  auto path = application::gradientDescent({-1.2f,1.2f}, problem);
  saveVTK(num_iterations, "Path", problem, path); // print found path
}
```

Hamijac intends to use the potential of the new C++11 standard [Kumar et al., 2012]. For instance, using the *auto* keyword allow minimize the impact of type specifications for the user. An other feature is the use of variadic templates in the declaration of the regular grid class $template < t\_uint DIM = 2, typename RealT = double, t\_uint ... v > class RegGrid$ which precise the number of vertices on a specific dimension allows to compute efficiently some operations known at compile time. The use of $std :: initializer_list$ permits the user to set obstacle in one go without having to precise the number of shapes given to $setMethod()$. Good use of efficient STL functions and containers significantly ease the work for the developer producing often a more efficient, safer code. The user also does not have to concerned about destruction of allocated ressources. There are other interesting features where Hamijac will allow compared to Par4HJB.

5.5 Abstraction POO examples with Hamijac

Let's take the example of front generation which is revelant in our context. Both Par4HJB and Hamijac libraries propose structures to represent any shape fronts as presented on the geodesic map obtained on figure 5.7.

So we can design a base class Shape which can be derived into several subclasses (circle/sphere, square/cube, polygons...). We present here different ways and their evolutions to obtain abstraction in object-oriented language such as C++.

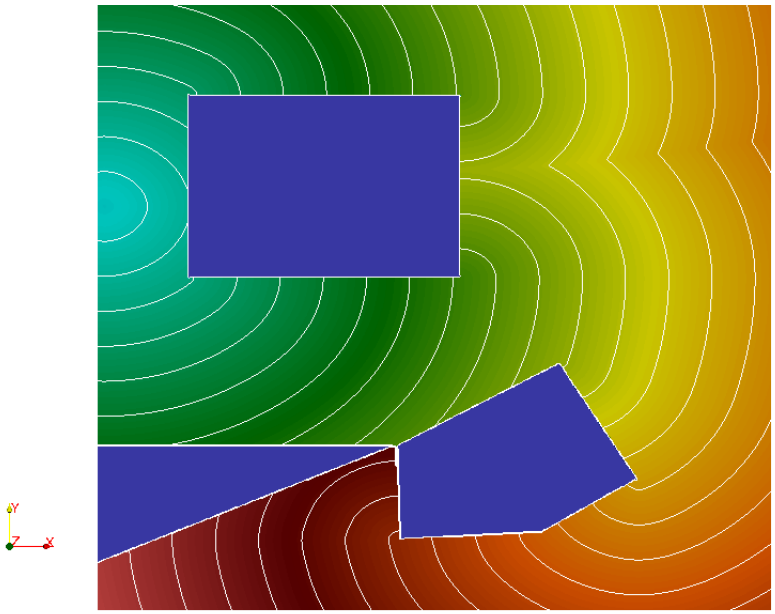

Figure 5.7: Different obstacle shape

5.5.1 Using classical virtual abstraction

In our first implementation in C++, we manage to handle a class base Shape where derived classes will present several methods. For illustration purpose, we choose to compute the *volume*() of any shape. In our solver, we have similarly a generic method which compute the vertices which are inside the shape. Note that in this case, *Shape* is an abstract class since *volume*() is a pure virtual function and thus cannot be instantiated. The user has to specifically determine the derived shapes she/he wants to use.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <cassert>

template<typename T>
class Shape {
public:
  Shape(std::vector<T> base_point) : base_point_(base_point) {}
  virtual T volume() const = 0; // pure virtual function => class is
      abstract
protected:
  std::vector<T> base_point_;
};

template<typename T>
class Circle : public Shape<T> {
public:
  Circle(std::vector<T> base_point, T radius) : Shape<T>(base_point),
      radius_(radius) { assert(base_point.size() == 2); }
  virtual T volume() const { return static_cast<T>(M_PI) * radius_ *
```

```
    radius_; } ;
private:
  T radius_;
};

template<typename T>
class Sphere : public Shape<T> {
public:
  Sphere(std::vector<T> base_point, T radius) : Shape<T>(base_point),
      radius_(radius) { assert(base_point.size() == 3); }
  virtual T volume() const { return 4 / static_cast<T>(3.0) *
      static_cast<T>(M_PI) * radius_ * radius_ * radius_; }
private:
  T radius_;
};

int main() {
  Circle<float> circle({0.2f,0.3f}, 4.0f); // or Sphere<double>
      sphere({0.0,0.0,0.0},2.0);
  std::cout << circle.volume() << std::endl;
}
```

We have so far genericity over the scalar type but not really with generic dimensions where our subclasses Circle and Sphere are not safe enough hence the use of assertions. Note that the use of exceptions instead should be more appropriate in this case. The next improvement will prevent us from using them.

5.5.2 Using template parameters and full template specialization

We now would like to manage generic dimensions at compile time by merging Sphere and Circle derived classes into one Ball derived classes. The code would be for instance the following by specializing the template functions.

```
#include <iostream>
#include <array>
#include <cmath>

template<typename T, std::size_t DIM>
class Shape {
public:
  Shape(std::array<T,DIM> base_point) : base_point_(base_point) {}
  virtual T volume() const = 0;
protected:
  std::array<T,DIM> base_point_;
};

template<typename T, std::size_t DIM>
class Ball : public Shape<T,DIM> {
public:
  Ball(std::array<T,DIM> base_point, T radius) : Shape<T,DIM>(
      base_point), radius_(radius) {}
```

```
  virtual T volume() const;
private:
  T radius_;
};

template<>
float Ball<float,2>::volume() const { return static_cast<T>(M_PI) *
   radius_ * radius_; }

template<>
float Ball<float,3>::volume() const { return 4/static_cast<T>(3.0) *
   static_cast<T>(M_PI) * radius_ * radius_ * radius_; }

template<>
float Ball<double,2>::volume() const { return static_cast<T>(M_PI) *
   radius_ * radius_; }

template<>
float Ball<double,3>::volume() const { return 4/static_cast<T>(3.0) *
    static_cast<T>(M_PI) * radius_ * radius_ * radius_; }

// Only full function template specialization is allowed so the
   following does not work

// template<typename T>
// T Ball<T,2>::volume() const { return static_cast<T>(M_PI) *
   radius_ * radius_; }

// template<typename T>
// T Ball<T,3>::volume() const { return 4 / static_cast<T>(3.0) *
   static_cast<T>(M_PI) * radius_ * radius_ * radius_;  }

int main() {
  Ball<float,2> circle{{0.2f,0.3f}, 4.0f};
  std::cout << circle.volume() << std::endl;
}
```

Knowing any shape dimensions at compile time allows us to avoid branch if conditions by using template specialization for every template parameters T and DIM. The compiler knows the method to call (in our case $floatBall < float, 2 >:: volume()$), and the execution can proceed without indirection regarding the dimension size or type.

5.5.3 Using curiously recurring template pattern and type to type mapping

However, we still can improve the code design. As we can see in the above code 5.5.2, C++ forbids partial template specialization of functions.

We can bypass this difficuly by using overload and delegate to a dummy class the specialization. This technique is referred in [Alexandrescu, 2001] as type to type mapping. We propose to pass through an other class $BallVolume$ in order to compute the volume and avoid code redondancies if we change the scalar type. Obviously, the volume

computation should be determined for each dimension size. Note that we could also use a generic formula for computing the volume of n dimensions ball $V_n(r) = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)} r^n$, where r is the radius of the ball. However, we will lose some performance here since our partial template class specializations avoid branch instruction conditions and using the formula above requires more operations.

An other improvement is to avoid virtual call costs by using the "Curiously Recurring Template Pattern" (CRTP). This technique helps us to mimic a virtual call of an abstract method. The subclass Ball derives from the Shape class which use Ball as template argument. This idiom is also known as F-bounded polymorphism where the subtype constraint is parametrized itself by one of the binders. The term "F-bounded polymorphism" takes root in [Cardelli and Wegner, 1985].

In the following code 5.5.3, we illustrate the techniques which solve the two mentionned points.

```
// Using type to type mapping since partial specialization of
   function templates is not allowed
template <typename T, std::size_t DIM>
struct BallVolume;

template <typename T>
struct BallVolume<T, 2> {
  static T compute(T radius) { return static_cast<T>(M_PI) * radius *
      radius; }
};

template <typename T>
struct BallVolume<T, 3> {
  static T compute(T radius) { return 4 / static_cast<T>(3.0) *
     static_cast<T>(M_PI) * radius * radius * radius; }
};

template<typename T, std::size_t DIM, typename Derived>
class Shape {
public:
  Shape(std::array<T,DIM> base_point) : base_point_(base_point) {}
  T volume() { static_cast<Derived *>(this)->volume(); }
protected:
  std::array<T,DIM> base_point_;
};

template<typename T, std::size_t DIM>
class Ball : public Shape<T,DIM,Ball<T,DIM>> {
public:
  Ball(std::array<T,DIM> base_point, T radius) : Shape<T,DIM,Ball<T,
     DIM>>(base_point), radius_(radius) {}
  T volume() const { return BallVolume<T, DIM>::compute(radius_); }
private:
  T radius_;
};
```

5.5.4 Abstraction "without polymorphism" using functors

We propose in this subsection a different point of view in order to use efficient abstraction. Since we are interested in one common method which is *volume*(), we can use functors classes where the *operator*() would return the same type. Simply put, functors can be seen as classes which behave like a function. For instance, we can rewrite our code using functors.

```
#include <iostream>
#include <array>
#include <cmath>

template <typename T, std::size_t DIM>
struct BallVolume;

template <typename T>
struct BallVolume<T, 2> {
  static T compute(T radius) { return static_cast<T>(M_PI) * radius *
      radius; }
};

template <typename T>
struct BallVolume<T, 3> {
  static T compute(T radius) { return 4 / static_cast<T>(3.0) *
     static_cast<T>(M_PI) * radius * radius * radius; }
};

template<typename RealT, std::size_t DIM>
class Ball {
public:
  Ball(const std::array<RealT,DIM>& base_point) : base_point_(
     base_point) {}
  RealT operator()(const RealT radius) const { return BallVolume<
     RealT, DIM>::compute(radius); };
private:
  std::array<RealT,DIM> base_point_;
};

// To avoid template parameter type duplication
template<typename RealT, std::size_t DIM>
Ball<RealT,DIM> createBall(const std::array<RealT,DIM>& base_point) {
  return Ball<RealT,DIM>(base_point);
}

int main() {
  std::array<float, 3> center{0.0f, 1.0f, 3.0f};
  auto sphere = createBall(center); // auto type deduction avoiding
     Ball<float,3> sphere{center};
  std::cout << sphere(2.0f) << "," << sphere(4.0f) << std::endl;
}
```

We have decided for illustration purposes to make *base_point* a private member of the functor Ball. Whenever an instantation of Ball is reused (in this case $sphere(2.0f)$ and

sphere(4.0*f*)), the functor keeps some information from the constructor (*base_point*) and compute different values according to parameters (*radius*). This can happen to be much useful if we want to compute several volumes of circles of same centers with different radiuses. In Hamijac, the grid class is stored as private member and different shapes are therefore instantiated only once.

Functors have several pros. They are straightforward to implement, does not have indirection calls since the operator is called directly according to its corresponding class and can be more easily inlined by the compiler compared to function pointers. Furthermore, functors have the ability to maintain a state that affects operator() between calls. We can now consider functions as a kind of type which are not containers but can still hide a value which is the value they return when calling the function.

The reader can take a look directly in an example code of Hamijac in 5.4.2 and see how functors improves the library usability. When managing several fronts, the user can aggregate the return arrays from the functors, combine them and reuse them for future use.

5.5.5 Choosing a compromise between performance, abstraction and maintainability

Design patterns, idioms, metaprogramming and many other techniques can help to try achieving generic, reusable code which is still efficient. As we have seen, code abstraction allows to get generic code. A generic code requires more efforts to write, is generally harder to maintain for the developpers since advanced techniques are used. However, a reusable library should also be able to be improved by others. Writing maintainable code imply to write software which can be comprehensible by peers, and should be sustainable. The difficulties rise as soon as we want to have an efficient code in term of speed execution. Advanced techniques are sometimes needed to achieve that. For instance, using expression templates allow to create domain-specific embedded language (DSEL) and perform lazy evaluation of C++ expressions [Kirschenmann et al., 2012]. These kind of techniques are often difficult to implement, to understand and to modify afterwards. Hence choosing an appropriate balance between performance, abstraction, maintainability should be done the sooner possible, and should always be a concern.

Writing the most generic code is not necessarily the most reusable code As we have seen in subsection 5.5.3, having a generic algorithm to compute a shape volume for any dimension can hinder performance. More branch conditions, operations can be needed when executing generic algorithms We would also miss the opportunity to use different optimization techique for a specific dimension size.

Our algorithm below allow to get the grid coordinates $(i, j, ...)$ from the vertex index in the grid. This algorithm has the merit to be convenient in some cases but should not be used in regions where the computation is intense. Indeed, lack of specialization implies a loss of some optimization possibilities. Therefore, algorithmic generic functions should be used wisely in an appropriate context.

```
//! Give grid coordinates from grid index
```

```
/*!
  idx->(i,j,...)
*/
const VectorUint indexToCoord(const t_uint& idx) const
{
  VectorUint coord(DIM);
  t_uint a = idx, b = num_vertices_, r = 0; //a = b.q + r

  for (t_uint i = 0; i <= DIM - 2; i ++)
  {
    b /= vertices_[i];
    coord[DIM - i - 1] = a / b;  // Mirror effect : use coord[i]
    r = a % b;
    a = r;
  }
  coord[0] = r;                  // Mirror effect : use coord[DIM - 1]
  return coord;
}
```

Furthermore, having generic codes often imply to write more complex code which can be difficult to debug. Adding abstraction in the code hinders maintainability and should be considered greatly. We can illustrate an actual difficulty we had in our algorithm "indexToCoord" (9) where interverting rank i with $DIM - i - 1$ introduces a mirror bug effect. Indeed, on figure (5.8), there should be only one initial front at the bottom right of the map. However, we can detect that an other front is propagating taking source out of the domain. This bug can be rather difficult to detect in some simulations since there is no real miscomputations. The initial problem does not correspond to the expected one.

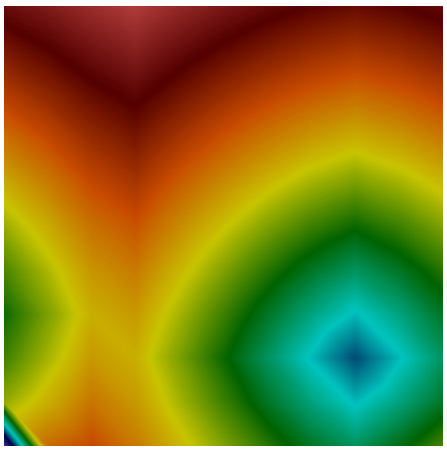

Figure 5.8: Mirror bug effect

Thus, overusing abstraction can lead to hardly maintainable and not performant code. Reusability is a matter of compromise between these concepts where each design, solution depends highly on the context, the purpose of the library. A reusable parallel library for instance, is capable to perform well on parallel architectures, while being intuitive to use both by the users and developers.

5.6 Sequential/parallel reusability in Par4HJB

Getting a multi-level parallel code for complex algorithms is generally not an easy task. Parallelizing algorithms are not only about programming issues and can require to change the original algorithm as we have seen with the fine-grained FIM 4.3. In addition to that, parallel paradigms can also limit theoritical performances. An example are OpenMP issues found in Lapack [Addison et al., 2003] where version 1.1 provides a code hard to maintain and cumbersome. The programmer therefore needs to be aware about the last technologies in order to get the best performance. In a reusable point of view we cannot afford to tune every applications for every parallel architectures. We think, because of the amazing evolution of these technologies, we should provide the most generic parallel code possible while giving great performance.

There are many strategies used in industrial codes for parallelizing a code. To sum up we can outline two main strategies :

1. write a sequential code and specialized parallel codes regarding different architectures
2. write a generic code which can handle both sequential and parallel architectures

Both strategies are arguable to use according to the context. In term of reusability, the second one is obviously preferred since it requires to think directly about the way we want to design our algorithm for future parallelization. Furthermore, the end user and advanced user would be allowed to have no particular skill in parallel computing using the second strategy.

Having a non-intrusive code for the non parallel expert can become an important key to have a usable code. A physicist who want to change the numerical scheme, the stencil, or other core functions wants to get good performance with its new code without changing any parallel parts of the code.

Our fined-grained parallel implementation is transparent in this sense since the sequential and parallel code are the same and only involve “pragma” directives. In order to obtain the same code, works were necessary with the aim to limit code size and execution time penalties.

Coarse-grained implementation for the FIM is more elaborate regarding reusability. We have succeeded into proposing a unique code for both sequential and parallel executions.

5.6.1 A parallel reusable numerical library design model

Most of previously mentioned libraries suffer from many problems. Imperative numerical libraries lack portability, modularity, interoperability. Despite their aim to promote modularities and reusability of their high level components, most object oriented libraries such as PETSc, Trilinos do not allow the simultaneous reusability of components between the sequential and the parallel versions of an application. We notice that all these libraries lack an additional level of abstraction which is necessary to achieve such a kind of reusability.

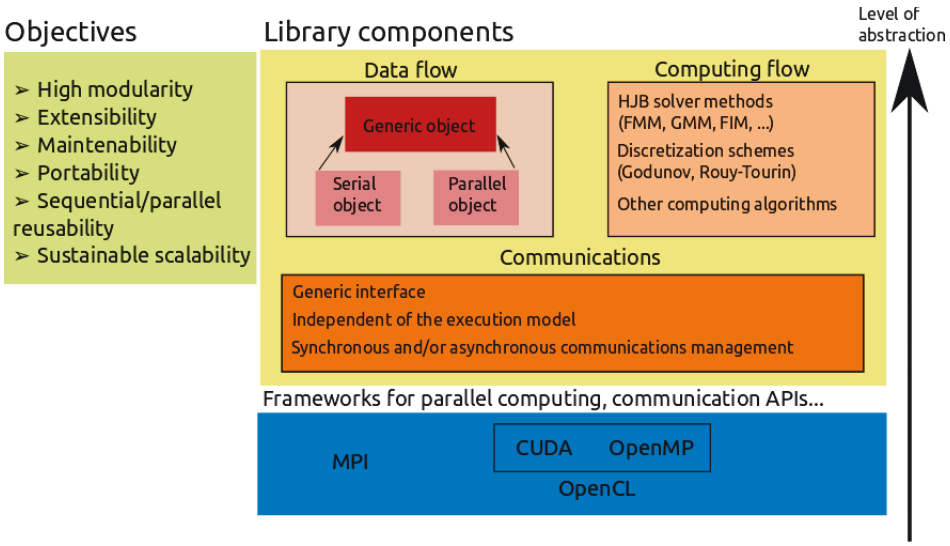

Figure 5.9: Design architecture for Par4HJB a reusable parallel library

To remedy to these problems, we propose a library design model based on three levels of abstraction. That means, a model which separates strictly the computation aspect, the data definition and the communication actions of applications (see figure 5.9). The data definition includes data types abstraction. The computation aspect represents all computation components. These two components communicate through the communication actions. Our main goal is to achieve the simultaneous reusability between sequential and parallel components, so in data definition part we encapsulate the parallelism in a common generic object which has the same interface in parallel and in serial. Then, parallel objects can be used polymorphically. Components of the computation part will be clients of these objects. We want to allow the code to be the same between the sequential and parallel versions of an application. Thereby every function is implemented once and used either in sequential or in parallel. Additionally, the maintainability of the library implemented according to this model would be simplified using this approach.

The library sequential/parallel reusability aim to follow different keypoints in particular it should :

- separate the data flow, the computing flow and the communications
- allow to maintain a unique code for both sequential and parallel implementations
- write specific functions which would take subdomains instead of the whole domain in parameter
- be performant and allow the possibility to overlap communications and computations
- if possible, allow parallelization to be done at a higher level software such as YML

The interested reader can take a look at the whitepaper from The Parallel Computing Research at Illinois which give an overview of good practices for parallel design patterns ideas from specialists [Sarita V. Adve et al., 2008].

5.6.2 Parallel pattern for distributed FIM

One difficulty in distributed FIM is the way we have to handle sub active independent lists. We have two strategies : the master process saves and distributes local copies of sub active lists at every iteration Λ_i ; or the whole sub-active lists are shared among all processes. In terms of reusability, the first choice is interesting since it would require minor modifications on the computing and data workflow. However, communications would be more costly considering worker processes have to exchange their sub active list with with the master process in the idea of map reduce. In adding to that, other communications are likely to happen when handling ghost points between neighbor processes. Therefore, the second choice appears to be a better candidate in terms of performance point of view but would require to change the way we manage the sub active list. We propose here parallel pattern design for the coarse-grained parallelism in the idea of design pattern used in POO language. Managing ghost cells is a difficult task and depends on the methods used [Kjolstad and Snir, 2010, Kronbichler and Kormann, 2012].

The coarse-grained subgrid decomposition for the FIM from section 4.4 is illustrated on figure (5.10). One of the difficulties is to to manage the indices of the narrow band partitions and subgrids. Indeed, several strategies can be used. Keep an index relative to the subgrid where a vertex in a subgrid can have the same index as a vertex in an other grid ; or keep a global index where every vertex has a unique identifier in the whole grid. In order to maintain a unique code for both sequential and parallel implementations, we manage to call functions which could take any subdomains in parameter instead of the whole domain. Therefore, the first strategy is chosen since calling a generic parallel function imply to consider any subgrid independently. The strategy chosen allows for instance to reuse a single function which is applicable for any subdomain. Hence, basic collective communications such as MPI Gather and MPI Scatter can be called directly.

The distributed parallelism is complex to implement. To our knowledge, it is not possible to design a code which could be parallelized by a third-party software without having to modify the inner kernel computations for exchange points. Managing the ghost points and different sub narrowbands impact both the data, computation flow and communication flow. At a certain level, to ensure data coherency, separation of these layers is hardly possible.

For instance, in Par4HJB, at every iteration the data (value function u) are scattered in a manner that there are several buffers on the narrow band, in order to avoid communications and allow efficient ghost exchanges.

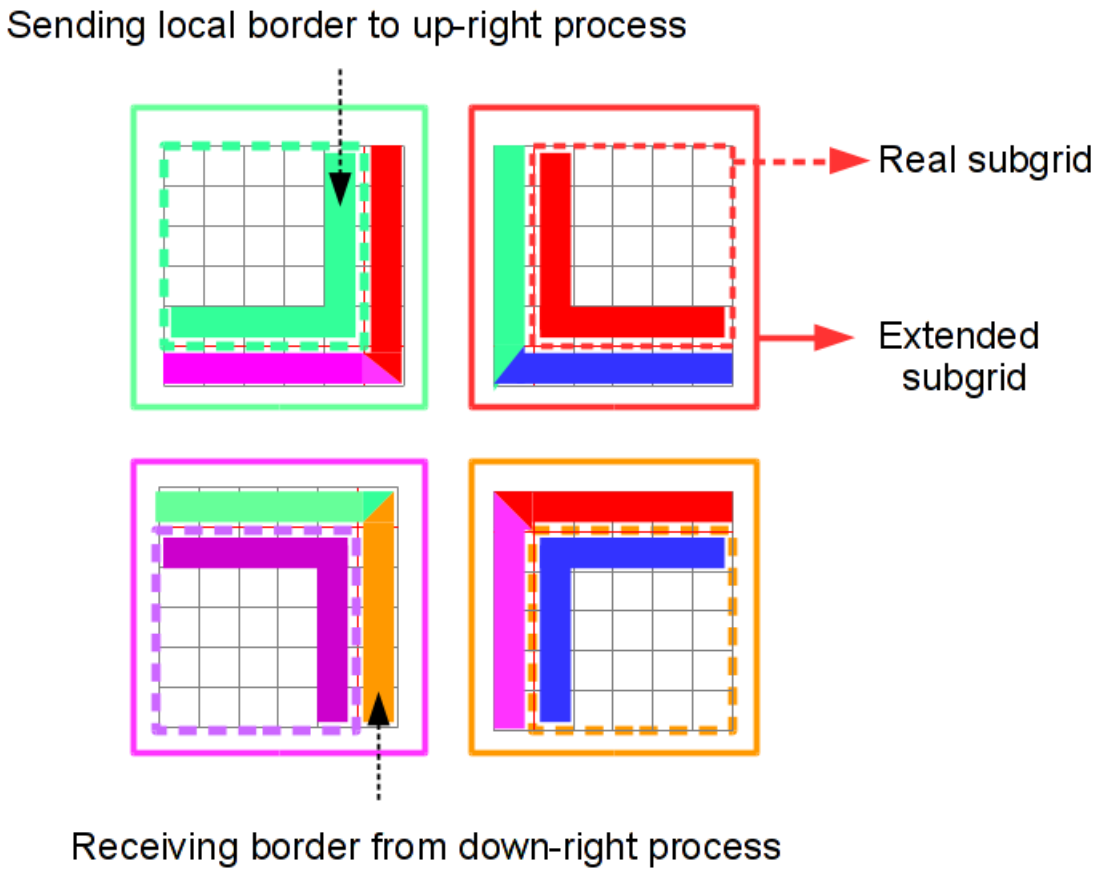

Figure 5.10: Ghost exchange design

5.7 Summary on reusable library implementation for solving HJB equations

We review reusability state-of-art in 5.1. We tackle both software reusability and sequential/parallel reusability aspects. Sequential/parallel reusability is a recent concern and we present how recent libraries are. We present two reusable libraries Par4HJB written in C and Hamijac, written in C++11, aiming to propose high end functionalities for solving HJB problems. To our knowledge, there is no such library publicly available. Both libraries are concerned about reusability and their design follow principles listed in 5.1.1 and 5.3. Adding to the classical software reusability, we also propose reusable algorithms such as a multi index access in a multi-dimensional grid 5.2. These features allow the libraries to be much more flexible in several contexts. In addition to their adaptability, both libraries are user-friendly and propose convenient functions, structures as shown in the listings presented in 5.4.2. Such ergonomy is possible through code abstraction thanks to the use of advanced and modern programming techniques (5.5). Finally we show how we design sequential/parallel reusability by presenting a parallel pattern for our libraries in 5.6.1.

Contributions, conclusion and future work

Herb Sutter's statement [Sutter, 2005] is more than ever a burning issue. Varieties of parallel hardware is surrounding us, heterogeneous and challenging to program. The evolution of these architectures does not allow to have a unified parallel paradigm for the moment. Great ideas have emerged recently but often target a specific type of parallel architectures such as dataflow programming language for many-core processors [Aubry et al., 2013], framework for global distributed computing platforms [Choy et al., 2009]. Developpers should therefore focus on efficient ways of writing parallel reusable code. Computational science has consequent and stronger needs in HPC. Solving PDEs such as Hamilton-Jacobi-Bellman equation can become computationaly intensive regarding real life applications. Recent parallel architectures become more and more complex. Scientific code needs to be adaptable not only regarding sofware reusability but also regarding sequential/parallel reusability. Writing an efficient sequential/parallel library require both to know the recent complex numerical methods to approximate the solutions and also to be aware of the technological evolution.

Contributions

Throughout this thesis, we have tried to propose solutions to such issues. The fourth chapter is dedicated to the development of efficient parallel numerical methods such as the buffered fast iterative method regarding the resolution of HJB equations. Multi-level parallel strategies are proposed targetting different parallel architectures in HJB context. The fifth chapter outlines the reusability implementation of the libraries Par4HJB and Hamijac in terms of software reusability and sequential/parallel reusability.

The buffered fast iterative method Throughout this thesis we have shown that efficient sequential numerical methods such as the fast marching method are not necessarily the best to use in a parallel context. For instance, the fast iterative method which was originally tuned for GPUs [Jeong and Whitaker, 2007], strongly proves that changing the former algorithm can become beneficial. Hence we have proposed in this thesis a fine-grained parallel fast iterative method [Dang et al., 2013, Dang and Emad, 2014a] called the buffered fast iterative method (BFIM). The BFIM uses temporary buffers in order to handle efficiently concurrent points which reside in the narrow band. The parallel computations are therefore more efficient and reduce the scope of critical sections. We have shown benchmarks targetting shared-memory architectures where the BFIM gives the best parallel scalability and execution time compared to current state-of-the-art parallel methods.

Parallel semi-ordered fast iterative method The model has been refined through collaborative work [Weinbub et al., 2015] and extended for the semi-ordered fast iterative method (SOFIM) which can handle broader classes of HJB equations such as in anisotropic environment. Two main improvements have been achieved : on the parallel algorithm and on the cutoff factor. Regarding the parallel aspect, once more, SOFIM results show to our knowledge the best parallel scalability and execution time compared to the recent state

of the art on parallel methods. In addition, SOFIM is capable of solving more general HJ(B) equations as illustrated with the several test cases provided.

A reusable parallel pattern for distributed parallelism Concerning the distributed parallelism, we have proposed a reusable parallel pattern, which tries to separate the three following workflows : communication, computing and data workflow [Dang et al., 2012]. Based on subdomains decomposition, the pattern employed minimize communications and ghost areas exchanges. Due to the complexity of the algorithms, a distinct separation of the workflows proposed is hardly possible. In order to ensure data coherency, a specific strategy has been adopted to handle sub active independent narrow bands. The strategy chosen allows for instance to reuse a single function which is applicable for any subdomain, so that a change in the kernel computation should be independent and would not force to change any parallel implementation. The current distributed parallel implementation gives correct result but does not give a sufficient speedup. An improved model has been proposed in order to deliver an efficient load balancing. The presented parallel pattern can be reused in the context of other numerical methods.

A reusable parallel library In addition to the sequential/parallel reusability aspect, we have also investigated the software reusability. The library design aims to be modular, flexible and in the same time try to propose a compromise between performance, abstraction and maintainability. The C++ Hamijac library was created in order to compensate the lack of native object-oriented programming features in the C library Par4HJB. Par4HJB had to mimic for instance polymorphism, genericity via the use of void pointers leading to a hard maintainable code. The C++ library Hamijac proposes convenient classes, helper functions for the end user eager to simulate HJB problems. The Hamijac UML shows how the library is based on a components strategy. Every components intend to be generic while being performant, which is achieved by several ways. One approach is to call generic algorithms such as the multi-indexes access utilities functions which work for multi-dimensional grids. Thanks to advanced and modern programming techniques, Hamijac, for instance, proposes core classes which improve both reusability performance by providing a container like usage with some compile-time convenient features.

Conclusion and future work

The libraries are still an ongoing work and subjects to several changes. This thesis has exposed their basic fundations and we propose some various perspectives.

Improve the multi-level parallelism The parallelism is improvable at every level. At the fine-grained level, the multi-buffered approach used for FIM and SOFIM show the best results in term of parallel scalability. At the coarse-grained level, the distributed approach needs to be perfected in order to get a decent parallel scalability, and would allow an interesting hybrid parallelism. The latter can be efficient for instance for a cluster of nodes which would make use of the coarse grained approach to communicate between nodes and the fine-grained approach for multi-core parallelism on one node. Multi-GPUs computation would also be possible if we combine the GPGPU original method with the coarse-grained approach. Large scale computation should greatly benefits from these multi-level parallel strategies.

Extended parallel FIM and SOFIM Possible improvements of both methods are still numerous. Not only regarding parallelism but also in their algorithms. Finding a better cutoff criterion in the SOFI method should allow to get a better accuracy and consistency in the algorithm. Some work on the algorithms can allow both FIM and SOFIM to handle broader classes of HJB equations. Hence testing these methods on longer real world applications will also be greatly beneficial in that sense. The work done for these fast marching like methods can lead to make scientists gain interest in these methods. For instance, in seismic imaging, the fast iterative method can reveal to be an alternative to other methods such as parallel reverse time migration algorithms [Abdelkhalek et al., 2012].

Library design The solver library design can be improved by adding interfaces between the problem and its requirements (grid, method, scheme, velocity, initial fronts...). This would allow to enforce encapsulation by avoiding the use of any setters, and provide to the user only functions which will be useful for him. Furthermore, modifications should be made in order to minimize memory cost, structure copies. For instance, the use of smart pointers, metaprogramming should greatly improve in Hamijac library. A modern C++14 prototype is available at appendix C listing C and show fast access to a multi-dimensional grid while keeping informations about the geometry involved. Further improvements would allow to even get access to the datas at compile-time. Memory usage and performance would be much improved. The library will also offer an ergonomic usage. Some functions can be hidden or available depending on the type of user, by adding layers. These further improvements are easy to implement thanks to the library modularity and flexibility.

Towards large scale applications Model designs proposed in this dissertation are aimed to work for large scale applications. Sequential/parallel reusability help achieve

parallel scalability at different levels. The need to use accessible tools for parallel computations is more than ever a concern. Therefore, model designs have to be adaptable, flexible in order to ease future parallel implementations. Parallelization of an application can also be "automatic" as we have seen with Par4all [Amini et al., 2012] for GPGPU and "transparent" with YML [Choy et al., 2009] on distributed systems. Indeed manually developing parallel codes is a time consuming, complex, error-prone and iterative process. However "automatic" parallelization can give unexpected results, poor performance, and is not necessarily applicable for complex problems. Choices depend on the context and the needs. In any cases, computive intensive codes should be written and be concerned with the issues outlined in this dissertation.

Appendix A

Solving quadratic equations numerically

Solving quadratic equation is a necessary steps in the simulation of fast marching methods and more precisely at the Godunov scheme level in our case. The latter scheme exposes a final formula which take the form of a quadratic equation where the u value function is the unknown. We present in this annex some numerical implementations used in Par4HJB and Hamijac to overcome some problem of floating points errors which occur in computer simulations.

Quadatric equations take the following form

$$ax^2 + bx + c = 0$$

where $(a, b, c) \in \mathbb{R}^3$ are constants and x the unknown to compute.

One familiar way to compute the solutions is to use the well-known expressions

$$x_{minus}, x_{plus} = \frac{-b \pm \sqrt{\Delta}}{2a} \text{ with } \Delta = b^2 - 4ac, \Delta \geq 0$$

.

However, this approach is numerically not stable. Floating-point operations on a computer can introduce rounding errors, cancellation, overflows... For instance, catastrophic cancellation can happen when $b^2 \gg 4ac$ since we compute $-b \pm \sqrt{\Delta}$. Cancellation can also occur when evaluating $b^2 - 4ac$. Also, overflow can happen when $b^2 - 4ac$ becomes difficult to represent in the floating-point precision system used. Adding two terms of different signs can therefore be dangerous in some situations.

Massive cancellation

Massive cancellation is an important source of problems and should be avoided. Catastrophic cancellation occurs when substracting two nearly equal numbers. One way to overcome this difficulty is to add two terms of the same sign. For instance, we propose to use the classical quadratric formula for one root and the Citardauq Formula for the other one.

If $b \geq 0$

$$x_{minus} = \frac{-b - \sqrt{\Delta}}{2a} \text{ and } x_{plus} = \frac{2c}{-b - \sqrt{\Delta}}$$

If $b > 0$

$$x_{minus} = \frac{2c}{-b + \sqrt{\Delta}} \text{ and } x_{plus} = \frac{-b + \sqrt{\Delta}}{2a}$$

Note that when we get a root we indeed compute without lose of signifiance the other root using : $x_{minus}.x_{plus} = c.a$

An other approach proposed is to compute first :

$$q = -\frac{1}{2}(b + sign(b)\sqrt{\Delta})$$

Then we can compute the roots with :

$$x_{minus} = \frac{q}{a} \text{ and } x_{plus} = \frac{c}{q}$$

The first approach was chosen to be used in our implementations. Below we present some difficulty which can arise. However, there are not revelant in our cases.

Cancellation when $b^2 \approx 4ac$

This difficulty may be eliminated by computing Δ using higher precision arithmetic. For instance, we can compute Δ with double precision when a,b,c are simple precision. We can also find roots with Newton's iteration method.

Overflow

Overflowing problems are none of concerns in the equations we have to solve. If this is an issue we suggest the reader to take look at Goldberg article [Goldberg, 1991] and the whitepaper Scilab is not naïve [Michael Baudin, 2010].

Appendix B

Geometry functions in Hamijac

We present in this annex some functions which are used to determine different shapes. The illustrating codes come from our C++ library Hamijac which is faster in many ways. Indeed, we do not browse the whole grid to determine whether a point is inside a shape or not. We optimize the process by using a bounding box in order to determinate which vertex is inside the ball. For that purpose, the *incrementCoords*. was created to handle only vertices inside this bounding box.

Bounding box

```
bool incrementCoords(std::vector<t_uint>& current, const std::vector<
    t_uint>& lower, const std::vector<t_uint>& upper)
{
    for (auto i = current.size(); i != 0;)
    {
        i --;
        ++ current[i];
        if (current[i] != upper[i] + 1)
        {
            return true;
        }
        current[i] = lower[i];
    }
    return false;
}
```

Ball (N dimension)

A point M is inside a ball of center C and radius r if and only if $|\overrightarrow{AC}| < r$ where $|.|$ represents the euclidian distance This function works for any dimension. Following this point allows us to evaluate whether a point in a bounding box is inside the ball or not.

```
//! Functor class : return points inside a ball (sphere in 3D or
    circle in 2D)
/*!
        Example : auto circle = Ball({0,1,0},3);
*/
template<t_uint DIM, typename RealT, t_uint... v>
class Ball
{
public:
        explicit Ball(const RegGrid<DIM,RealT,v...>& reg_grid) :
            grid_(reg_grid) {}

        std::vector<t_uint> operator()(const std::vector<RealT>&
            center, const RealT& radius)
        {
                if (center.size() != DIM)
                        throw std::invalid_argument("Ball definition
                            has not good dimensions !");
                std::vector<RealT> low_bound_box(DIM);
                std::vector<RealT> up_bound_box (DIM);
                // max/min for out of bounds
                // using epsilon to make sure to include border
                    vertices
                for (size_t i = 0; i < DIM; i ++)
                        low_bound_box[i] = std::max(grid_.lower()[i]
                            + Constant<RealT>::k_epsilon, center[i] -
                            radius);
                for (size_t i = 0; i < DIM; i ++)
                        up_bound_box[i] = std::min(grid_.upper()[i] -
                             Constant<RealT>::k_epsilon, center[i] +
                            radius);

                auto coord_low_box = closestVertex(low_bound_box,
                    grid_);
                auto coord_up_box  = closestVertex(up_bound_box,
                    grid_);

                std::vector<t_uint> inside_points;
                auto current = coord_low_box;

                do
                {
                        if (distanceEuclidian(grid_.coordToReal(
                            current), center) <= radius)
                        {
                                inside_points.push_back(grid_.
                                    coordToIndex(current));
                        }
                } while (incrementCoords(current, coord_low_box,
                    coord_up_box));

                return inside_points;
        }
```

```
private:
        const RegGrid<DIM,RealT,v...>& grid_;
};
```

Segment front (2D)

Given a segment $[AB]$ any point C is in the segment [AB] with thickness T if and only if

$$0 \leq \overrightarrow{AC} \cdot \overrightarrow{AB} \leq AB^2 \text{ , and } \qquad AB^2 AC^2 \leq T^2 AB^2 + \left(\overrightarrow{AC} \cdot \overrightarrow{AB}\right)^2$$

```
//! Functor class : return points around a 2D segments (depending on
    thickness)
/*!
        Example : auto polygon = Segment
            ({1.1,2.0},{2.1,3.0},{1.1,2.0});
*/
template<t_uint DIM, typename RealT, t_uint... v>
class Segment
{
public:
        explicit Segment(const RegGrid<DIM,RealT,v...>& reg_grid) :
            grid_(reg_grid) {}

        bool isAroundSegment(const std::vector<RealT>& real,
                const std::pair<std::vector<RealT>,std::vector<RealT
                    >>& points, RealT thickness)
        {
                // dotprod AB.AC
                RealT dotprod = (real[0] - points.first[0]) * (points
                    .second[0] - points.first[0]) +
                        (real[1] - points.first[1]) * (points.second
                            [1] - points.first[1]);
                RealT AB_squared = (points.first[0] - points.second
                    [0]) * (points.first[0] - points.second[0]) +
                        (points.first[1] - points.second[1]) * (
                            points.first[1] - points.second[1]);
                RealT AC_squared = (points.first[0] - real[0]) * (
                    points.first[0] - real[0]) +
                        (points.first[1] - real[1]) * (points.first
                            [1] - real[1]);

                if (dotprod < 0 || dotprod > AB_squared)
                        return false;
                if (AB_squared * AC_squared > thickness * thickness *
                     AB_squared + dotprod * dotprod)
                        return false;

                return true;
        }
```

```
        std::pair<std::vector<t_uint>,std::vector<t_uint>>
            findBoundingBoxes(const std::pair<std::vector<RealT>,std::
            vector<RealT>>& points)
        {
                // DIM should be 2
                std::vector<RealT> min_value(DIM);
                std::vector<RealT> max_value(DIM);
                for (size_t i = 0; i < DIM; i ++)
                {
                        min_value[i] = std::min(points.first[i],
                            points.second[i]);
                        max_value[i] = std::max(points.first[i],
                            points.second[i]);
                }
                auto coord_low_box = closestVertex(min_value, grid_);
                auto coord_up_box  = closestVertex(max_value, grid_);
                return std::pair<std::vector<t_uint>,std::vector<
                    t_uint>>(coord_low_box,coord_up_box);
        }

        //! Give vertices which are inside a segment with specific
            thickness
        /*!
                \param points     pair of two vector points which are
                    the extremum of the segment.
                \param thickness  thickness size depends on the
                    problem grid. Should be > step/stride.
                \return a vector of vertices indexes
        */
        std::vector<t_uint> operator()(const std::pair<std::vector<
            RealT>,std::vector<RealT>>& points, RealT thickness)
        {
                std::pair<std::vector<t_uint>,std::vector<t_uint>>
                    bound_box = findBoundingBoxes(points);
                std::vector<t_uint> inside_points;
                auto current = bound_box.first;
                do
                {
                        if (isAroundSegment(grid_.coordToReal(current
                            ), points, thickness))
                        {
                                inside_points.push_back(grid_.
                                    coordToIndex(current));
                        }
                } while (incrementCoords(current, bound_box.first,
                    bound_box.second));

                return inside_points;
        }

private:
        const RegGrid<DIM,RealT,v...>& grid_;
};
```

Hypercube (N dimensions)

```
//! Functor class : return points inside a hypercube (cube in 3D or
    square in 2D)
/*!
        Example : auto cube = Hypercube({0,-2,0},{2,-1,3});
*/
template<t_uint DIM, typename RealT, t_uint... v>
class Hypercube
{
public:
        explicit Hypercube(const RegGrid<DIM,RealT,v...>& reg_grid) :
            grid_(reg_grid) {}

        std::vector<t_uint> operator()(const std::vector<RealT>&
           low_bound_box, const std::vector<RealT>& up_bound_box)
        {
                if (low_bound_box.size() != DIM || up_bound_box.size
                    () != DIM)
                        throw std::invalid_argument("Hypercube bounds
                            must have good dimensions !");
                for (size_t i = 0; i < DIM; i ++)
                {
                        if (low_bound_box[i] >= up_bound_box[i])
                                throw std::domain_error("Upper bound
                                   must be strictly higher than lower
                                    bound !");
                }
                // max/min for out of bounds
                for (size_t i = 0; i < DIM; i ++)
                        low_bound_box[i] = std::max(grid_.lower()[i]
                           + Constant<RealT>::k_epsilon,
                           low_bound_box[i]);
                for (size_t i = 0; i < DIM; i ++)
                        up_bound_box[i] = std::min(grid_.upper()[i] -
                            Constant<RealT>::k_epsilon, up_bound_box[
                           i]);

                auto coord_low_box = closestVertex(low_bound_box,
                    grid_);
                auto coord_up_box  = closestVertex(up_bound_box,
                    grid_);

                std::vector<t_uint> inside_points;
                auto current = coord_low_box;

                do
                {
                        inside_points.push_back(grid_.coordToIndex(
                           current));
                } while (incrementCoords(current, coord_low_box,
                    coord_up_box));
```

```
                return inside_points;
        }


private:
        const RegGrid<DIM,RealT,v...>& grid_;
};
```

Polygon (2D)

```
//! Functor class : return points inside a 2D polygon
/*!
        This is faster than in Par4HJB since this is not a brute
           force in O(n^DIM) where
        n is the number of vertices in one dimension.
        We use a bounding box in order to determinate which vertex is
            inside the ball.
        Example : auto polygon = Polygon2D
           ({1.1,2.0},{2.1,3.0},{1.1,2.0});
*/
template<t_uint DIM, typename RealT, t_uint... v>
class Polygon2D
{
public:
        explicit Polygon2D(const RegGrid<DIM,RealT,v...>& reg_grid) :
            grid_(reg_grid)
        {
                if (DIM != 2)
                        std::cout << "Warning ! Polygon construction
                           will only work in 2D !" << std::endl;
        }

        bool isInPolygon(const std::vector<RealT>& real_coord, const
           std::vector<std::vector<RealT>>& poly_vertices)
        {
                bool is_inside = false;
                for(t_uint i = 0, j = poly_vertices.size() - 1; i <
                   poly_vertices.size(); j = i ++)
                {
                        if( ((poly_vertices[i][1] >= real_coord[1])
                           != (poly_vertices[j][1] >= real_coord[1]))
                            &&
                                (real_coord[0] <= (poly_vertices[j
                                   ][0] - poly_vertices[i][0]) * (
                                   real_coord[1] - poly_vertices[i
                                   ][1]) /
                                (poly_vertices[j][1] - poly_vertices[
                                   i][1]) + poly_vertices[i][0] )
                                )
                                is_inside = !is_inside;
                }
```

```
                return is_inside;
        }

        std::pair<std::vector<t_uint>,std::vector<t_uint>>
           findBoundingBoxes(const std::vector<std::vector<RealT>>&
           poly_vertices)
        {
                std::vector<RealT> min_value({poly_vertices[0][0],
                   poly_vertices[0][1]});
                std::vector<RealT> max_value(min_value);
                for (size_t i = 1; i < poly_vertices.size(); i ++)
                {
                        if (poly_vertices[i][0] < min_value[0])
                                min_value[0] = poly_vertices[i][0];
                        else
                                max_value[0] = poly_vertices[i][0];

                        if (poly_vertices[i][1] < min_value[1])
                                min_value[1] = poly_vertices[i][1];
                        else
                                max_value[1] = poly_vertices[i][1];
                }
                auto coord_low_box = closestVertex(min_value, grid_);
                auto coord_up_box  = closestVertex(max_value, grid_);
                return std::pair<std::vector<t_uint>,std::vector<
                   t_uint>>(coord_low_box,coord_up_box);
        }

        std::vector<t_uint> operator()(const std::vector<std::vector<
           RealT>>& poly_vertices)
        {
                std::pair<std::vector<t_uint>,std::vector<t_uint>>
                   bound_box = findBoundingBoxes(poly_vertices);
                std::vector<t_uint> inside_points;
                auto current = bound_box.first;
                do
                {
                        if (isInPolygon(grid_.coordToReal(current),
                           poly_vertices))
                        {
                                inside_points.push_back(grid_.
                                   coordToIndex(current));
                        }
                } while (incrementCoords(current, bound_box.first,
                   bound_box.second));

                return inside_points;
        }

private:
        const RegGrid<DIM,RealT,v...>& grid_;
};
```

Appendix C

Multidimensional regular grid functions

We present in this annex some of the generic data access functions used in Par4HJB. These functions are not optimized but show the generic algorithm possibilities.

In Hamijac C language

Grid coordinates to real coordinates access

```
const VectorScal coordToReal(const VectorUint& coord) const
{
        assert(coord.size() == DIM);
        VectorScal real_coord(DIM);
        for (t_uint i = 0; i < DIM; ++ i)
                real_coord[i] = lower_[i] + static_cast<RealT>(coord[
                    i]) * stride_[i];
        return real_coord;
}
```

```
t_uint real_to_index(t_float *real_coord, PHJ_s_regular_grid
    regular_grid)
{
        t_uint i, idx;
        idx = 0;

        for (i = 0; i < PHJ_g_dimension - 1; i++)
                idx = (idx + (t_uint) ceil( (- regular_grid.
                    low_bound_vertices[i] + real_coord[i])
                         / regular_grid.stride_per_dir[i]) )
                         * regular_grid.vertices_per_dir[i + 1];

        idx += (t_uint) ceil((- regular_grid.low_bound_vertices[
            PHJ_g_dimension-1] +
                real_coord[PHJ_g_dimension-1]) / regular_grid.
                    stride_per_dir[PHJ_g_dimension-1]);
```

```
        return idx;
}
```

Grid coordinates to index access

Give the index $T(idx) = T(x_i)_{i\in[0,d-1]} = (v_0, v_1, ..., v_{d-1})$ at the $(x_0, x_1, ..., x_{d-1})$ knowing the coordinates in the grid G.
 $idx = x_0 + \sum_{i=1}^{d-1} x_i.n_{i-1}$ where n_i is the number of vertex at the i dimension.

```
t_uint coord_to_index(t_uint *grid_coord, PHJ_s_regular_grid
    regular_grid)
{
        t_uint i, idx;
        t_uint prod;

        // idx = grid_coord[PHJ_g_dimension - 1];
        idx = grid_coord[0];
        prod = 1;

        for (i = 1; i < PHJ_g_dimension; i++)
        {
                prod *= regular_grid.vertices_per_dir[i];
                idx += grid_coord[i] * prod;
        }

        return idx;
}
```

Index to real coordinates access

```
t_float *index_to_real(t_uint idx, t_float **real_coord)
{
        t_float *real_vertex;
        real_vertex = (t_float *) malloc(PHJ_g_dimension * sizeof (
            t_float));
        memcpy(real_vertex, real_coord[idx], PHJ_g_dimension);  //
            Copy

        return real_vertex;
}
```

Index to coordinates access

```
t_uint *index_to_coord(t_uint idx, t_uint *vertices_per_dir)
{
        t_uint *coord;
```

```
        coord = (t_uint *) malloc(PHJ_g_dimension * sizeof (t_uint));

        // a = b.q + r
        t_uint a, b, r; // q will be stored in coord directly
        t_uint i, j;

        a = idx;
        r = 0;

        for (i = 0; i <= PHJ_g_dimension - 2; i ++)
        {
                // We can optimize that by dividing each iteration
                   the product of every vertices_per_dir
                // cf coord_to_index
                b = 1;
                for(j = i + 1; j <= PHJ_g_dimension - 1; j ++)
                        b *= vertices_per_dir[j];

                // coord[i] = a / b; // no !
                coord[PHJ_g_dimension - i - 1] = a / b;

                r = a % b;
                //printf("%u = %u * %u + %u\n", a, b, grid_coord[
                   PHJ_g_dimension - 1 - i], r);
                a = r;
        }

        // coord[PHJ_g_dimension - 1] = r;  // no !
        coord[0] = r;

        return coord;
}
```

A C++14 prototype for a multi-dimensional grid

This multi-dimensional grid class has several features :

- store multi-dimensional grid values into a flatten array
- provide grid values access and flat index access from multi dim coordinates with O(1) average complexity (worst case is O(N))
- provide grid values access and coordinates access from flat index with O(1) complexity
- provide iterators so that grid behaves like a STL container
- statically-sized arrays, efficient storage, similar to C-style array thanks to metaprogramming

```
#include <iostream>
#include <array>
#include <vector>
#include <unordered_map>
#include <type_traits>

#include <algorithm> // just for std::generate in main()

// std::unordered_map needs std::hash specialization for std::array
namespace std {
    template<typename T, size_t N>
    struct hash<array<T, N> > {
        using argument_type = array<T, N> ;
        using result_type = size_t;
        result_type operator()(const argument_type& a) const {
            hash<T> hasher;
            result_type h = 0;
            for (result_type i = 0; i < N; ++i) {
                h = h * 31 + hasher(a[i]);
            }
            return h;
        }
    };
}

// pretty-print for std::array
template<class T, size_t N>
std::ostream& operator<<(std::ostream& os, const std::array<T, N>&
   arr) {
    os << "{";
    for (auto && el : arr) { os << el << ";"; }
    return os << "\b}";
}

// meta functions
template<typename T>
constexpr T meta_prod(T x) { return x; }

template<typename T, typename... Ts>
constexpr T meta_prod(T x, Ts... xs) { return x * meta_prod(xs...); }

template<typename T, typename E>
constexpr T meta_pow(T base, E expo) { return (expo != 0 ) ? base *
   meta_pow(base, expo-1) : 1; }

// Compute the total number of elements 2x2x2 for two usage
// for Grid<3, float, 2, 2, 2>  (specify all size dimensions)
template<size_t DIM, size_t... NDIM> constexpr
std::enable_if_t<sizeof...(NDIM) != 1, size_t>
num_vertices() { return meta_prod(NDIM...); }

// for Grid<3, float, 2> (specify one size dimension and consider the
    same size for other dimensions)
```

```
template<size_t DIM, size_t... NDIM> constexpr
std::enable_if_t<sizeof...(NDIM) == 1, size_t>
num_vertices() { return meta_pow(NDIM...,DIM); }

template<size_t DIM, typename T, size_t... NDIM>
class MultiGrid {
public:
    static_assert(sizeof...(NDIM) == 1 or sizeof...(NDIM) == DIM,
        "Variadic template arguments in Multigrid do not match
           dimension size !");

    using ArrayValues     = std::array<T,num_vertices<DIM,NDIM...>()
       >;
    using ArrayCoord      = std::array<size_t,DIM>;
    using MapIndexToCoord = std::array<ArrayCoord,num_vertices<DIM,
       NDIM...>()>;
    using MapCoordToIndex = std::unordered_map<ArrayCoord,size_t>;

    using value_type      = typename ArrayValues::value_type;      //
        T
    using reference       = typename ArrayValues::reference;       //
        T&
    using const_reference = typename ArrayValues::const_reference; //
        const T&
    using size_type       = typename ArrayValues::size_type;       //
        size_t
    using iterator        = typename ArrayValues::iterator;        //
        random access iterator
    using const_iterator  = typename ArrayValues::const_iterator;

    MultiGrid() : MultiGrid(ArrayValues{}) {} // default constructor
       use delegating constructor
    MultiGrid(const ArrayValues& values)
        : map_idx_to_coord_(fill_map_idx_to_coord())
        , map_coord_to_idx_(fill_map_coord_to_idx())
        , values_(values)
        {}

    iterator       begin()        { return values_.begin();  }
    const_iterator begin()  const { return values_.begin();  }
    const_iterator cbegin() const { return values_.cbegin(); }
    iterator       end()          { return values_.end();    }
    const_iterator end()    const { return values_.end();    }
    const_iterator cend()   const { return values_.cend();   }

    reference       operator[] (size_type idx)       { return values_
       [idx]; };
    const_reference operator[] (size_type idx) const { return values_
       [idx]; };

    reference       operator[] (const ArrayCoord& coord) {
        return values_[map_coord_to_idx_.at(coord)];
    };
    const_reference operator[] (const ArrayCoord& coord) const {
```

```
        return const_cast<reference>(static_cast<const MultiGrid&>(*
            this)[coord]);
    };

    auto get_coord_from_index(size_type idx) const {
        return map_idx_to_coord_.at(idx);
    }

    auto get_index_from_coord(const ArrayCoord& coord) const {
        return map_coord_to_idx_.at(coord);
    }

private:
    auto  fill_map_idx_to_coord() const {
        MapIndexToCoord coord;
        std::array<size_t,DIM> size_per_dim{{NDIM...}};
        if (sizeof...(NDIM) == 1) { size_per_dim.fill(size_per_dim
            [0]); }
        for (size_t j = 0; j < num_vertices<DIM,NDIM...>(); j ++) {
            size_t a = j, b = num_vertices<DIM,NDIM...>(), r = 0;
            for(size_t i = 0; i <= DIM - 2; i ++) {
                b /= size_per_dim[DIM - i - 1];
                coord[j][DIM-i-1] = a / b;
                r = a % b;
                a = r;
            }
            coord[j][0] = r;
        }
        return coord;
    }
    auto fill_map_coord_to_idx() const {
        MapCoordToIndex mapping(num_vertices<DIM,NDIM...>());
        for(size_t i = 0; i < num_vertices<DIM,NDIM...>(); i ++) {
            mapping.emplace(map_idx_to_coord_[i],i); // reuse the
                previous mapping
        }
        return mapping;
    }
    friend auto &operator<<(std::ostream &os, const MultiGrid& that)
        {
        os << "Values : {";
        for (auto&& v : that.values_)  { os << v << ";"; }
        os << "\b}\nMapping index to coord :\n";
        static size_t count{0};
        for (auto&& m : that.map_idx_to_coord_) { os << count ++ << "
            :" << m << "\t"; }
        os << "\nMapping coord to index :\n";
        for (auto && m : that.map_coord_to_idx_) { os << m.first << "
            ->" << m.second << "\t"; }
        return os << "\n";
    }

private:
    MapIndexToCoord map_idx_to_coord_;    // O(1) access flat index
```

```
        -> dim coordinates
    MapCoordToIndex map_coord_to_idx_;    // O(1) average acess dim
        coordinates -> flat index (worst case : O(N))
    ArrayValues     values_;              // same behaviour as
        declaring  `float values_[meta_prod(NDIM)];`
};

int main() {
    // Create a 4D grid with 3x2x3x5 vertices
    MultiGrid<4,float,3,2,3,5> grid;
    // grid behaves like a STL container and we can fill values with
        std::generate
    std::generate(grid.begin(), grid.end(), []() {static float n{0.0f
        }; return n+=0.5f;} );
    std::cout << grid << std::endl;

    // get coordinates from index
    std::cout << "get_coord_from_index(43) = " << grid.
        get_coord_from_index(43) << std::endl;
    // and vice versa
    std::cout << "get_index_from_coord({{2,0,2,3}}) = " << grid.
        get_index_from_coord({{2,0,2,3}}) << std::endl;
    // print value at specific coordinates
    std::cout << "Grid[{{2,0,2,3}}] = " << grid[{{2,0,2,3}}] << std::
        endl;
    // print value at specific index
    std::cout << "Grid[42] = " << grid[42] << "\n\n";

    MultiGrid<2, float, 2> little_grid;
    std::cout << little_grid << std::endl;
}
```

Appendix D

Gradient descent implementation in Hamijac

The following gradient descent is used in order to find a shortest path between a source and a given point in the grid.

```
template<t_uint DIM, typename RealT, t_uint... v>
vector<RealT> gradientDescent(const vector<RealT>& destination, const
    Problem<DIM, RealT, v...>& problem)
{
  vector<RealT> res(problem.num_vertices(), 0.0); // result map

  // gradient map
  vector<RealT> grads(DIM, 0.0);
  t_uint idx = problem.realToIndex(destination);

  vector<RealT> current_point(destination);
  vector<vector<RealT>> path; // the path

  RealT time = 0.0;
  path.push_back(current_point);
  res[idx] = time;

  const double step = problem.stride()[0];

  while (problem.u()[idx] > Constant<RealT>::k_epsilon)
  {
    vector<t_int> neighbors = problem.firstOrderPointStencil(idx);
    RealT max_grad = 0.0;

    for (t_uint d = 0; d < DIM; d ++)
    {
      grads[d] = 0.0;
      grads[d] -= problem.u()[neighbors[2 * d]] / 2;
      grads[d] += problem.u()[neighbors[2 * d + 1]] / 2;
      if (std::isinf(grads[d]))
        grads[d] = sgn<RealT>(grads[d]);
      if (std::abs(max_grad) < std::abs(grads[d]))
```

```
            max_grad = grads[d];
        }

        // Updating points
        for (t_uint d = 0; d < DIM; d ++)
        {
          current_point[d] = current_point[d] - step * grads[d] / std::
              abs(max_grad);
        }

        path.push_back(current_point);

        RealT new_idx = problem.coordToIndex(closestVertex(current_point,
            problem.grid()));
        idx = new_idx;
        time += k_time;
        res[idx] = time;
      }

      return res;
}
```

Publications

International conference articles (peer-reviewed with proceedings)

A fine-grained parallel model for the fast iterative method in solving eikonal equations
Florian Dang, Nahid Emad, Alexandre Fender
In Proceedings of the 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC '13, pages 152–157. IEEE Computer Society, 2013.
[Dang et al., 2013]

Fast Iterative Method in Solving Eikonal Equations : A Multi-level Parallel Approach
Florian Dang, Nahid Emad
Procedia Computer Science, Volume 29, 2014, Pages 1859-1869, ISSN 1877-0509.
[Dang and Emad, 2014a]

Multi-level parallel upwind finite difference scheme for front propagation
Florian Dang, Nahid Emad
VECPAR 2014 11th International Meeting High Performance Computing for Computational Science.
[Dang and Emad, 2014b]

Shared-Memory Parallelization of the Semi-Ordered Fast Iterative Method
Josef Weinbub, Florian Dang, Tor Gillberg, Siegfried Selberherr
Proceedings of the 23rd High Performance Computing Symposium (HPC), pp. 8, 2015.
[Weinbub et al., 2015]

International conference communications (abstracts)

Toward reusable numerical library for solving Hamilton-Jacobi-Bellman equations
Florian Dang, Nahid Emad and Pierre Fiorini
7th International Workshop on Parallel Matrix Algorithms and Applications
PMAA'2012, Birkbeck University of London, UK, 2012.
[Dang et al., 2012]

Invited talks

Toward reusable numerical library for solving Hamilton-Jacobi-Bellman equations
Florian Dang
7th Seminar on High Performance Numerical Computing @ Maison de la Simulation. CEA Saclay, France. March 8th 2012.

Poster

Journée des doctorants de l'Université de Versailles Saint-Quentin-en-Yvelines. Septembre 2012.
[Dang, 2012]

Glossary

A list of acronyms which can be found in this thesis.

aL : Active List
API : Application Programming Interface
BFIM : Buffered Fast Iterative Method
CPU : Central Processing Unit
CRTP : Curiously Recursive Template Pattern
FEM : Finite Element Method
FIM : Fast Iterative Method
FMM : Fast Marching Method
FSM : Fast Sweeping Method
GDAL : Geospatial Data Abstraction Library
GPGPU : General Purpose Graphical Processing Unit
GPU : Graphical Processing Unit
HPC : High Performance Computing
NB : Narrow Band
OUM : Ordered Upwind Method
OSM : OpenStreetMap
PDE : Partial Differential Equation
pL : Paused List
OOP : Object-Oriented Programming
RTM : Rouy-Tourin Method
SMP : Symmetric Multi-Processing
SOFIM : Semi-Ordered Iterative Method
UML : Unified Modeling Language

Index

Thesis index

Bibliography

[Abdelkhalek et al., 2012] Abdelkhalek, R., Calandra, H., Coulaud, O., Latu, G., and Roman, J. (2012). Fast seismic modeling and reverse time migration on a graphics processing unit cluster. *Concurrency and Computation: Practice and Experience*, 24(7):739–750. 101

[Addison et al., 2003] Addison, C., Ren, Y., and van Waveren, M. (2003). OpenMP Issues Arising in the Development of Parallel BLAS and LAPACK Libraries. *Sci. Program.*, 11(2):95–104. 92

[Alexandrescu, 2001] Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 81, 87

[Amini, 2012] Amini, M. (2012). *Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators*. PhD thesis. 73

[Amini et al., 2012] Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., Mcmahon, J. O., Pasquier, F.-X., Péan, G., and Villalon, P. (2012). Par4all: From Convex Array Regions to Heterogeneous Computing. In *IMPACT 2012 : Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, Paris, France. 2 pages. 73, 102

[Anderson et al., 1999] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition. 70

[Andreae et al., 1993] Andreae, P., Biddle, R., and Tempero, E. (1993). Understanding code reusability: Experience with C and C++. Technical report. 9, 70

[Aubry et al., 2013] Aubry, P., Beaucamps, P.-E., Blanc, F., Bodin, B., Carpov, S., Cudennec, L., David, V., Dore, P., Dubrulle, P., Dinechin, B. D. d., Galea, F., Goubier, T., Harrand, M., Jones, S., Lesage, J.-D., Louise, S., Chaisemartin, N. M., Nguyen, T. H., Raynaud, X., and Sirdey, R. (2013). Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor. *Procedia Computer*

Science, 18(0):1624 – 1633. 2013 International Conference on Computational Science. 99

[Austern, 1998] Austern, M. H. (1998). *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 81

[Balay et al., 2012] Balay, S., Brown, J., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. (2012). PETSc Web page. http://www.mcs.anl.gov/petsc. 71

[Bangerth et al., 2012] Bangerth, W., Burstedde, C., Heister, T., and Kronbichler, M. (2012). Algorithms and Data Structures for Massively Parallel Generic Adaptive Finite Element Codes. *ACM Trans. Math. Softw.*, 38(2):14:1–14:28. 72

[Bangerth et al., 2007] Bangerth, W., Hartmann, R., and Kanschat, G. (2007). deal.II – a General Purpose Object Oriented Finite Element Library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27. 71, 72

[Bardi and Capuzzo-Dolcetta, 1997] Bardi, M. and Capuzzo-Dolcetta, I. (1997). *Optimal Control and Viscosity Solution of Hamilton-Jacobi- Bellmann Equations*. Birkhauser. 16

[Barles, 1994] Barles, G. (1994). *Solutions de viscosité des équations de Hamilton-Jacobi*. Springer-Verlag. 16

[Bastian et al., 2008] Bastian, P., Blatt, M., Dedner, A., Engwer, C., Kloefkorn, R., Kornhuber, R., Ohlberger, M., and Sander, O. (2008). A Generic Grid Interface for Adaptive and Parallel Scientific Computing. Part II: Implementation and Tests in DUNE. *Computing*, 82(2-3):121–138. 71

[Bornemann and Rasch, 2006] Bornemann, F. and Rasch, C. (2006). Finite-element Discretization of Static Hamilton-Jacobi Equations based on a Local Variational Principle. *Computing and Visualization in Science*, 9(2):57–69. 23

[Breuss et al., 2009] Breuss, M., Cristiani, E., Gwosdek, P., and Vogel, O. (2009). A Domain-Decomposition-Free Parallelisation of the Fast Marching Method. Universitat des Saarlandes, preprint. 49, 56

[Breuss et al., 2011] Breuss, M., Cristiani, E., Gwosdek, P., and Vogel, O. (2011). An adaptive domain-decomposition technique for parallelization of the Fast Marching Method. *Applied Mathematics and Computation*, 118(1):1–206. 49, 50, 59, 66

[Brunschen and Brorsson, 2000] Brunschen, C. and Brorsson, M. (2000). OdinMP CCp - a portable implementation of OpenMP for C. *Concurrency - Practice and Experience*, 12(12):1193–1203. 6

[Cacace et al., 2012] Cacace, S., Cristiani, E., and Falcone, M. (2012). A Local Ordered Upwind Method for Hamilton-Jacobi and Isaacs Equations. In *18th IFAC World Congress*, volume 18, Milano, Italie. 19

[Cacace et al., 2013] Cacace, S., Cristiani, E., and Falcone, M. (2013). Can local single-pass methods solve any stationary Hamilton-Jacobi-Bellman equation? *ArXiv e-prints*. 19

[Cacace et al., 2011] Cacace, S., Cristiani, E., Falcone, M., and Picarelli, A. (2011). A patchy Dynamic Programming scheme for a class of Hamilton-Jacobi-Bellman equations. *ArXiv e-prints*. 19

[Cameron et al., 2007] Cameron, M. K., Fomel, S. B., and Sethian, J. A. (2007). Seismic velocity estimation from time migration. *Inverse Problems*, 23(4):1329. 20

[Cardelli and Wegner, 1985] Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM COMPUTING SURVEYS*, 17(4):471–522. 88

[Cerimele and Cossu, 2007] Cerimele, M. M. and Cossu, R. (2007). Decay regions segmentation from color images of ancient monuments using fast marching method. *Journal of Cultural Heritage*, 8(2):170–175. 21

[Chacon and Vladimirsky, 2012] Chacon, A. and Vladimirsky, A. (2012). Fast Two-scale Methods for Eikonal Equations. *SIAM Journal on Scientific Computing*, 34(2):A547–A578. 62

[Chiang et al., 2007] Chiang, C. H., Chiang, P. J., Fei, J.-C., and Liu, J. S. (2007). A comparative study of implementing Fast Marching Method and A* SEARCH for mobile robot path planning in grid environment: Effect of map resolution. In *Advanced Robotics and Its Social Impacts, 2007. ARSO 2007. IEEE Workshop on*, pages 1–6. 19, 27

[Choy et al., 2009] Choy, L., Delannoy, O., Emad, N., and Petiton, S. G. (2009). Federation and Abstraction of Heterogeneous Global Computing Platforms with the YML Framework. In *2009 International Conference on Complex, Intelligent and Software Intensive Systems, CISIS 2009, Fukuoka, Japan, March 16-19, 2009*, pages 451–456. 99, 102

[Crandall et al., 1984] Crandall, M. G., Evans, L., and Lions, P.-L. (1984). Some properties of viscosity solutions of Hamilton-Jacobi-Bellman equations. *Trans. Amer. Math. Soc. 282 J. Sci. Comput. 27*, pages 487–502. 19

[Crandall and Lions, 1983] Crandall, M. G. and Lions, P.-L. (1983). Viscosity solutions of Hamilton-Jacobi-Bellman Equations. *Transactions of the American Mathematical Society*, 277(1):1–42. 18, 19

[Cristiani, 2009] Cristiani, E. (2009). A Fast Marching Method for Hamilton-Jacobi Equations Modeling Monotone Front Propagations. *Journal of Scientific Computing*, 39(2):189–205. 34

[Cristiani and Falcone, 2007] Cristiani, E. and Falcone, M. (2007). Fast Semi-Lagrangian schemes for the Eikonal equation and applications. *SIAM*, 45(5):1979–2011. 26

[Dandouna, 2012] Dandouna, M. (2012). *Librairie numérique pour le calcul distribué à grande échelle*. PhD thesis. 71

[Dang, 2012] Dang, F. (2012). Toward a parallel reusable numerical library for solving Hamilton-Jacobi-Bellman equations. 124

[Dang, 2014] Dang, F. (2014). Par4hjb a parallel reusable library for solving Hamilton-Jacobi-Bellman equations. http://www.prism.uvsq.fr/~flod/par4hjb. 83

[Dang and Emad, 2014a] Dang, F. and Emad, N. (2014a). Fast Iterative Method in Solving Eikonal Equations: A Multi-level Parallel Approach. *Procedia Computer Science*, 29(0):1859 – 1869. 2014 International Conference on Computational Science. 59, 99, 123

[Dang and Emad, 2014b] Dang, F. and Emad, N. (2014b). Multi-level parallel upwind finite difference scheme for front propagation. 123

[Dang et al., 2013] Dang, F., Emad, N., and Fender, A. (2013). A Fine-Grained Parallel Model for the Fast Iterative Method in Solving Eikonal Equations. In *Proceedings of the 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 3PGCIC '13, pages 152–157. IEEE Computer Society. 50, 99, 123

[Dang et al., 2012] Dang, F., Emad, N., and Fiorini, P. (2012). Toward Reusable Numerical Library for Solving Hamilton-Jacobi-Bellman Equations. In *7th International Workshop on Parallel Matrix Algorithms and Applications (PMAA 2012), 2012, Birkbeck University of London, UK*. 100, 124

[Detrixhe et al., 2013] Detrixhe, M., Gibou, F., and Min, C. (2013). A parallel fast sweeping method for the Eikonal equation. *Journal of Computational Physics*, 237(0):46–55. 48, 59, 66

[Dey and Ayers, 2009] Dey, B. K. and Ayers, P. W. (2009). Computing the chemical reaction path with a ray-based fast marching technique for solving the Hamilton-Jacobi equation in a general coordinate system. *Journal of Mathematical Chemistry*, 45(4):981–1003. 19

[Dongarra et al., 1988] Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. (1988). An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17. 70

[Driesen and Hölzle, 1996] Driesen, K. and Hölzle, U. (1996). The Direct Cost of Virtual Function Calls in C++. *SIGPLAN Not.*, 31(10):306–323. 81

[Eden and Mens, 2006] Eden, A. and Mens, T. (2006). Measuring software flexibility. *Software, IEE Proceedings* -, 153(3):113–125. 9

[Esr, 1998] Esr, I. (1998). *ESRI Shapefile Technical Description*. Environmental Systems Research Institute, Inc. 41

[Falcone and Ferretti, 2002] Falcone, M. and Ferretti, R. (2002). Semi-Lagrangian Schemes for Hamilton-Jacobi Equations, Discrete Representation Formulae and Godunov Methods. *Journal of Computational Physics*, 175(2):559–575. 26

[Forcadel et al., 2008] Forcadel, N., Guyader, C., and Gout, C. (2008). Generalized fast marching method: applications to image segmentation. *Numerical Algorithms*, 48(1-3):189–211. 21, 34

[Forum, 1994] Forum, M. P. (1994). MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA. 8

[Fournié et al., 2010] Fournié, M., Renon, N., Renard, Y., and Ruiz, D. (2010). CFD Parallel Simulation Using Getfem++ and Mumps. In D'Ambra, P., Guarracino, M. R., and Talia, D., editors, *Euro-Par (2)*, volume 6272 of *Lecture Notes in Computer Science*, pages 77–88. Springer. 71

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 82

[Garrido et al., 2006] Garrido, S., Moreno, L., Abderrahim, M., and Martin, F. (2006). Path Planning for Mobile Robot Navigation using Voronoi Diagram and Fast Marching. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2376–2381. 19

[Gillberg, 2011] Gillberg, T. (2011). A Semi-Ordered Fast Iterative Method (SOFI) for Monotone Front Propagation in Simulations of Geological Folding. In *MODSIM2011, 19th International Congress on Modelling and Simulation*, pages 641–647. Modelling and Simulation Society of Australia and. 59, 60, 61

[Gillberg et al., 2014] Gillberg, T., Bruaset, A., Hjelle, Ø., and Sourouri, M. (2014). Parallel solutions of static Hamilton-Jacobi equations for simulations of geological folds. *Journal of Mathematics in Industry*, 4(1). 62

[Goldberg, 1991] Goldberg, D. (1991). What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv.*, 23(1):5–48. 45, 104

[Gomez et al., 2013] Gomez, J. V., Lumbier, A., Garrido, S., and Moreno, L. (2013). Planning robot formations with fast marching square including uncertainty conditions. *Robotics and Autonomous Systems*, 61(2):137–152. 19

[Guennebaud et al., 2010] Guennebaud, G., Jacob, B., and others (2010). *Eigen v3*. 71

[Heroux et al., 2005] Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A.,

and Stanley, K. S. (2005). An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423. 71, 72

[Herrman, 2003] Herrman, M. (2003). A domain decomposition parallelization of the Fast Marching Method and applications to image segmentation. Annual Research Briefs, Center for Turbulence Research. 47, 48

[Higham and Higham, 2000] Higham, D. J. and Higham, N. J. (2000). *MATLAB Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. 71

[Higham, 2002] Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition. 46

[Horn, 1989] Horn, B. K. P. (1989). Shape from Shading. pages 123–171. MIT Press, Cambridge, MA, USA. 19

[Jeong and Whitaker, 2007] Jeong, W.-K. and Whitaker, R. T. (2007). A Fast Iterative Method for a Class of Hamilton-Jacobi Equations on Parallel Systems. Technical Report UUCS-07-010, University of Utah. 49, 99

[Jeong and Whitaker, 2008] Jeong, W.-K. and Whitaker, R. T. (2008). A Fast Iterative Method for Eikonal Equations. *SIAM J. Sci. Comput. Vol.30 No.5*, pages 2512–2534. 34, 35

[Kao et al., 2005] Kao, C.-Y., Osher, S., and Tsai, Y.-H. (2005). Fast Sweeping Methods for static Hamilton-Jacobi equations. *SIAM J. Numer. Anal.*, 74(250):2612–2632. 27

[Karlsen et al., 2000] Karlsen, K., Lie, K.-A., and Risebro, N. (2000). A fast marching method for reservoir simulation. *Computational Geosciences*, 4(2):185–206. 20

[Kim, 2000] Kim, S. (2000). An O(N) Level Set Method. Technical report. 34

[Kim and Folie, 2001] Kim, S. and Folie, D. (2001). An O(N) Level Set Method for Eikonal Equations. *SIAM Journal on Scientific Computing*, 22(6):2178–2193. 34

[Kirk et al., 2006] Kirk, B. S., Peterson, J. W., Stogner, R. H., and Carey, G. F. (2006). libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations. *Eng. with Comput.*, 22(3):237–254. 71

[Kirschenmann et al., 2012] Kirschenmann, W., Plagne, L., and Vialle, S. (2012). Multi-Target Vectorization with MTPS C++ Generic Library. In Jónasson, K., editor, *Applied Parallel and Scientific Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 336–346. Springer Berlin Heidelberg. 90

[Kjolstad and Snir, 2010] Kjolstad, F. B. and Snir, M. (2010). Ghost Cell Pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 4:1–4:9, New York, NY, USA. ACM. 71, 94

[Kronbichler and Kormann, 2012] Kronbichler, M. and Kormann, K. (2012). A generic interface for parallel cell-based finite element operator application. *Computers & Fluids*, 63(0):135 – 147. 72, 94

[Kumar et al., 2012] Kumar, A., Sutton, A., and Stroustrup, B. (2012). Rejuvenating C++ programs through demacrofication. In *ICSM*, pages 98–107. IEEE Computer Society. 84

[Langtangen, 1999] Langtangen, H. P. (1999). *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming.*, volume 2 of *Lecture Notes in Computational Science and Engineering*. Springer. 71

[Lelièvre et al., 2010] Lelièvre, P. G., Farquharson, C. G., and Hurich, C. A. (2010). Computing first-arrival seismic traveltimes on unstructured 3-D tetrahedral grids using the Fast Marching Method. *Geophys. J. Int. (2011) 184*, pages 885–896. 20, 24

[Li et al., 2008] Li, F., Shu, C.-W., Zhang, Y.-T., and Zhao, H. (2008). A Second Order Discontinuous Galerkin Fast Sweeping Method for Eikonal Equations. *J. Comput. Phys.*, 227(17):8191–8208. 23

[Lions, 1983] Lions, P.-L. (1983). On the Hamilton-Jacobi-Bellman Equations. *Acta Applicandae Mathematicae*, 1:17–41. 18, 19

[Michael Baudin, 2010] Michael Baudin (2010). Scilab is not naïve. 104

[Mitchell and Templeton, 2005] Mitchell, I. and Templeton, J. (2005). A Toolbox of Hamilton-Jacobi Solvers for Analysis of Nondeterministic Continuous and Hybrid Systems. In Morari, M. and Thiele, L., editors, *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 480–494. Springer Berlin Heidelberg. 72

[Mo and Harris, 2002] Mo, L.-W. and Harris, J. M. (2002). Finite-difference calculation of direct-arrival traveltimes using the eikonal equation. *GEOPHYSICS*, 67(4):1270–1274. 20

[Montan, 2013] Montan, S. (2013). *Sur la validation numérique des codes de calcul industriels*. PhD thesis, UPMC. 46

[Moore, 1965] Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8). 4

[Noulard and Emad, 2001] Noulard, E. and Emad, N. (2001). A key for reusable parallel linear algebra software. *Parallel Computing*, 27(10):1299 – 1319. 71

[Petres et al., 2005] Petres, C., Pailhas, Y., Petillot, Y., and Lane, D. (2005). Underwater path planing using fast marching algorithms. In *Oceans 2005 - Europe*, volume 2, pages 814–819 Vol. 2. 19

[Prados and Soatto, 2005] Prados, E. and Soatto, S. (2005). Fast Marching Method for Generic Shape from Shading. In Paragios, N., Faugeras, O., Chan, T., and Schnörr, C., editors, *Variational, Geometric, and Level Set Methods in Computer Vision*, volume 3752 of *Lecture Notes in Computer Science*, pages 320–331. Springer Berlin Heidelberg. 19

[Rawlinson and Sambridge, 2004] Rawlinson, N. and Sambridge, M. (2004). Multiple reflection and transmission phases in complex layered media using a multistage fast marching method. *Geophysics*, 69(5):1338–1350. 20

[Reinders, 2007] Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition. 8

[Rouy and Tourin, 1992] Rouy, E. and Tourin, A. (1992). A viscosity solutions approach to shape-from-shading. *SIAM J. Numer. Anal.*, 29(3):867–884. 19, 23, 26, 44

[Roy Cardinal et al., 2003] Roy Cardinal, M.-H., Meunier, J., Soulez, G., Thérasse, E., and Cloutier, G. (2003). Intravascular Ultrasound Image Segmentation: A Fast-Marching Method. In Ellis, R. and Peters, T., editors, *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2003*, volume 2879 of *Lecture Notes in Computer Science*, pages 432–439. Springer Berlin Heidelberg. 21

[Sarita V. Adve et al., 2008] Sarita V. Adve, Vikram S. Adve, Gul Agha, Matthew I. Frank, Maria Jesus Garzaran, John C. Hart, Wen-mei W. Hwu, Ralph E. Johnson, Laxmikant Kale, Rakesh Kumar, Darko Marinov, Klara Nahrstedt, David Padua, Madhusudan Parthasarathy, Sanjay Patel, Grigore Rosu, Dan Roth, Marc Snir, Josep Torrellas, and Craig Zilles (2008). Parallel Computing Research at Illinois the UPCRC Agenda Nov 2008. Technical report. 94

[Scilab Enterprises, 2012] Scilab Enterprises (2012). *Scilab: Le logiciel open source gratuit de calcul numérique*. Scilab Enterprises, Orsay, France. 71

[Sethian, 1999a] Sethian, J. A. (1999a). Fast Marching Methods. *SIAM Review Vol. 41 No. 2*, pages 199–235. 30, 31

[Sethian, 1999b] Sethian, J. A. (1999b). *Level Set Methods and Fast Marching Method*. Cambridge University Press. 23, 31

[Sethian and Vladimirsky, 2000] Sethian, J. A. and Vladimirsky, A. (2000). Fast methods for the Eikonal and related Hamilton– Jacobi equations on unstructured meshes. *Proceedings of the National Academy of Sciences*, 97(11):5699–5703. 24

[Sharifi and Kelkar, 2014] Sharifi, M. and Kelkar, M. (2014). Novel permeability upscaling method using Fast Marching Method. *Fuel*, 117, Part A(0):568 – 578. 20

[Siff and Reps, 1996] Siff, M. and Reps, T. W. (1996). Program Generalization for Software Reuse: From C to C++. In *SIGSOFT '96, Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, California, USA, October 16-18, 1996*, pages 135–146. 73

[Stec and Domanski, 2003] Stec, P. and Domanski, M. (2003). Two-Step Unassisted Video Segmentation Using Fast Marching Method. In Petkov, N. and Westenberg, M., editors, *Computer Analysis of Images and Patterns*, volume 2756 of *Lecture Notes in Computer Science*, pages 246–253. Springer Berlin Heidelberg. 21

[Sutter, 2005] Sutter, H. (2005). The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3). 5, 99

[Top500, 2014] Top500 (2014). Top 500 Supercomputer Sites. 5

[Treibig et al., 2010] Treibig, J., Hager, G., and Wellein, G. (2010). LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 207–216, Washington, DC, USA. IEEE Computer Society. 62

[Tsai et al., 2003] Tsai, Y.-H., Cheng, L.-T., Osher, S., and Zhao, H.-K. (2003). Fast Sweeping Methods for a class of Hamilton-Jacobi-Bellman equations. *SIAM Vol. 41 No.2*, 41(2). 27

[Tsitsiklis, 1995] Tsitsiklis, J. (1995). Efficient algorithms for globally optimal trajectories. *Automatic Control, IEEE Transactions on*, 40(9):1528–1538. 26

[Van Heesch, 2004] Van Heesch, D. (2004). *Doxygen*. 78

[Vidale, 1988] Vidale, J. (1988). Finite-difference calculation of traveltimes. *Bulletin of the Seismological Society of America*, 78(6):2062–2076. 20, 23

[Wang and Forsyth, 2008] Wang, J. and Forsyth, P. A. (2008). Maximal Use of Central Differencing for Hamilton-Jacobi-Bellman PDEs in Finance. *SIAM J. Numer. Anal.*, 46(3):1580–1601. 19

[Weinbub et al., 2015] Weinbub, J., Dang, F., Gillberg, T., and Selberherr, S. (2015). Shared-Memory Parallelization of the Semi-Ordered Fast Iterative Method. In *Proceedings of the 2015 Spring Simulation Multi-Conference (SpringSim'15)*, pages 599–606. 99, 123

[Weinbub and Hössinger, 2014] Weinbub, J. and Hössinger, A. (2014). Accelerated Redistancing for Level Set-Based Process Simulations with the Fast Iterative Method. *Journal of Computational Electronics*, 13(4):877–884. DOI: 10.1007/s10825-014-0604-x. 59

[Wolfram, 2003] Wolfram, S. (2003). *The Mathematica book (5. ed.)*. Wolfram-Media. 71

[Yan et al., 2004] Yan, J., Zhuang, T.-g., Zhao, B., and Schwartz, L. H. (2004). Lymph node segmentation from CT images using fast marching method. *Computerized Medical Imaging and Graphics*, 28(1–2):33–38. 20

[Yu et al., 2013] Yu, C., Qiu, Q., and Chen, X. (2013). A hybrid two-dimensional path planning model based on frothing construction algorithm and local fast marching method. *Computers & Electrical Engineering*, 39(2):475–487. 19

[Yuen et al., 2007] Yuen, S. Y., Tsui, Y. Y., and Chow, C. K. (2007). A fast marching formulation of perspective shape from shading under frontal illumination. *Pattern Recognition Letters*, 28(7):806–824. 19, 44

[Zhao, 2005] Zhao, H. (2005). A Fast Sweeping Method for Eikonal equations. *Mathematics of Computation Vol.74 No.250*, pages 603–627. 27

[Zhao, 2007] Zhao, H. (2007). Parallel implementations of the Fast Sweeping Method. *Journal of Computational Mathematics*, 25(4):421–429. 48

Résumé

La simulation numérique est indissociable du calcul haute performance. Ces vingt dernières années, l'informatique a connu l'émergence d'architectures parallèles multi-niveaux. Exploiter efficacement la puissance de calcul de ces machines peut s'avérer être une tâche délicate et requérir une expertise à la fois technologique sur des notions avancées de parallélisme ainsi que scientifique de part la nature même des problèmes traités.

Le travail de cette thèse est pluri-disciplinaire s'appuyant sur la conception d'une librairie de calcul parallèle réutilisable pour la résolution des équations Hamilton-Jacobi-Bellman. Ces équations peuvent se retrouver dans des domaines diverses et variés tels qu'en biomédical, géophysique, ou encore robotique en l'occurence sur les applications de planification de mouvement et de reconstruction de formes tri-dimensionnelles à partir d'images bi-dimensionnelles. Nous montrons que les principaux algorithmes numériques amenant a résoudre ces équations telles que les méthodes de type fast marching, ne sont pas appropriés pour être efficaces dans un contexte parallèle. Nous proposons la méthode buffered fast iterative qui permet d'obtenir une scalabilité parallèle non obtenue jusqu'alors. Un des points sensibles relevés dans cette thèse est de parvenir à trouver une recette de compromis entre abstraction, performance et maintenabilité afin de garantir non seulement une réutilisabilité dans le sens classique du domaine de génie logiciel mais également en terme de réutilisabilité séquentielle/parallèle.

Mots-clés: calcul haute performance, équations Hamilton-Jacobi-Bellman, réutilisabilité séquentielle/-parallèle, fast marching method, fast iterative method

Abstract

Numerical simulation is strongly bound with high performance computing. Programming scientific softwares requires at the same time good knowledge on the mathematical numerical models and also on the techniques to make them efficient on today's computers. Indeed, these last twenty years, we have experienced the rising of multi-level parallel architectures. The work in this thesis dissertation is multidisciplinary by designing a reusable parallel numerical library for solving Hamilton-Jacobi-Bellman equations. Such equations are involved in various fields such as in biomedical, geophysics or robotics. In particular, we will show interests in path planning and shape from shading applications. We show that the methods to solve these equations such as the widely used fast marching method, are not designed to be used efficiently in a parallel context. We propose a buffered fast iterative method which gives an interesting parallel scalability. This dissertation takes interest in the challenge to find compromises between abstraction, performance and maintainability in order to combine both software reusability and also sequential/parallel reusability. We propose code abstraction allowing algorithmic and data genericity while trying to keep a maintainable and performant code potentially parallelizable.

Keywords: high performance computing, Hamilton-Jacobi-Bellman equations, sequential/parallel reusability, fast marching method, fast iterative method