Particle-in-Cell Algorithms for Plasma Simulations on Heterogeneous Architectures

Xavier Sáez Pous

Advisors:

Prof. José María Cela Espín
Prof. Alejandro Soba Pascual

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONA**TECH**
Departament d'Arquitectura de Computadors

A dissertation submitted to the Universitat Politècnica de Catalunya (UPC)
for the degree of Doctor

Abstract

During the last two decades, High-Performance Computing (HPC) has grown rapidly in performance by improving single-core processors at the cost of a similar growth in power consumption. The single-core processor improvement has led many scientists to exploit mainly the process level parallelism in their codes. However, the performance of HPC systems is becoming increasingly limited by power consumption and power density, which have become a primary concern for the design of new computer systems. As a result, new supercomputers are designed based on the power efficiency provided by new homogeneous and heterogeneous architectures.

The growth in computational power has introduced a new approach to science, Computational Physics. Its impact on the study of nuclear fusion and plasma physics has been very significant. This is because the experiments are difficult and expensive to perform whereas computer simulations of plasma are an efficient way to progress. Particle-In-Cell (PIC) is one of the most used methods to simulate plasma. The improvement in the processing power has enabled an increase in the size and complexity of the PIC simulations. Most PIC codes have been designed with a strong emphasis on the physics and have traditionally included only process level parallelism. This approach has not taken advantage of multiprocessor platforms. Therefore, these codes exploit inefficiently the new computing platforms and, as a consequence, they are still limited to using simplified models.

The aim of this thesis is to incorporate in a PIC code the latest technologies available in computer science in order to take advantage of the upcoming multiprocessor supercomputers. This will enable an improvement in the simulations, either by introducing more physics in the code or by incorporating more detail to the simulations.

This thesis analyses a PIC code named EUTERPE on different computing platforms. EUTERPE is a production code used to simulate fusion plasma instabilities in fusion reactors. It has been implemented for traditional HPC clusters and it has been parallelized prior to this work using only Message Passing Interface (MPI). Our study of its scalability has reached up to tens of thousands of processors, which is several orders of magnitude higher than the scalability achieved when this thesis was initiated.

This thesis also describes the strategies adopted for porting a PIC code to a multi-core architecture, such as introducing thread level parallelism, distributing the work among different computing devices, and developing a new thread-safe solver. These strategies have been evaluated by applying them to the EUTERPE code. With respect to heterogeneous architectures, it has been possible to port this kind of plasma physics codes by rewriting part of the code or by using a programming model called OpenMP Super-scalar (OmpSs). This programming model is specially designed to make this computing power easily available to scientists without requiring expert knowledge on computing.

Last but not least, this thesis should not be seen as the end of a way, but rather as the beginning of a work to extend the physics simulated in fusion codes through exploiting available HPC resources.

Acknowledgements

"De gente bien nacida es agradecer los beneficios que reciben, y uno de los pecados que más a Dios ofende es la ingratitud."

Don Quijote de la Mancha
MIGUEL DE CERVANTES

I owe thanks to many people who have made this thesis possible, so I really hope not to forget anybody in this text. Firstly, I thank my thesis advisor José María Cela for giving me the opportunity to enter in the *CEPBA-IBM Research Institute* (which later became the *Barcelona Supercomputing Center*) and for suggesting the topic for this thesis. I also thank my thesis co-advisor Alejandro Soba for his persistent faith in me, since although he had to return to Argentina for personal reasons, he has stayed in touch with me. Behind that tough breastplate there is a great heart. I only have words of gratitude for you.

I want to express my gratitude to the chief of my group (*Fusion Energy*), Mervi Mantsinen, who in this short period of time has given me a lot of advices, has put her trust in me and has isolated me from the surrounding issues in the last months to finish the thesis.

I want to thank people of Barcelona Supercomputing Center (BSC) with whom I have collaborated to perform my work at one point or another. In particular, I would like to give special thanks to the following three teams. Firstly to the *Programming Models* Team, where Sergi Mateo, Xavier Martorell, Diego Nieto, Xavier Teruel and Roger Ferrer have shared time on searching for solutions to issues that I found using OmpSs and also for all the suggestions provided by them. I also do not wish to forget the *Performance Tools* team, where Harald Servat, German Llort, Pedro Gonzalez, Judit Giménez and Eloy Martínez have always found the time to solve my questions. And last but not least, the *Operations* team, specially David Vicente and Javier Bartolomé who have addressed my questions and issues related with the machines at our center.

Among collaborations with other research centers, I would like to thank Edilberto Sánchez at Centro Investigaciones Energéticas, Medioambientales y Tecnológicas (CIEMAT) for the experimental data used in the tests and also for the useful discussions. He has always been willing to give a hand to us. I wish also to thank Paco Castejón at CIEMAT for his support to perform this work. And obviously, special thanks to Ralf Kleiber and all members of Garching Max-Planck-Institut für Plasmaphysik (IPP) group for having allowed me to work with the EUTERPE code.

My heartfelt gratitude to my work-mates of CASE department for their demonstrations of affection and support on day-to-day and for making lunch an oasis in the working day, especially Eva Casoni, Cristóbal Samaniego and Toni Artigues. Of the many people who deserve thanks, there is one particularly prominent, which is my friend Raúl de la Cruz. Twelve years ago we started to work together as research scholars and we have shared the same path up to the present day. We have spent together hours and hours of working, with many great times and also some difficult ones, and I know that we will be there to support each other.

Some of the developments done in this thesis have been carried out in the framework of some European projects, so I would like to acknowledge support given by the EUFORIA project, the PRACE project and specially by the Mont-Blanc project under the grant agreement nº 288,777 and 610,402.

També voldria dedicar unes línies per agrair als meus pares tot l'esforç que han realitzat perquè pogués arribar fins aquí i tot el suport que m'han donat sempre. Aquest triomf és tan d'ells com meu. Us estima el vostre fill. I com no, a la meva estimada Marta, amb qui he tardat en retrobar-me a la vida, però amb la que desitjo formar una família per caminar junts per la vida. Gràcies pel teu suport d'aquest darrers anys, el teu amor i la felicitat que m'has donat. T'estima el teu Xavier.

And finally, thanks to the first person who hummed a melody, without music all would have been tougher.

Preface

This thesis describes my research on various aspects of the Particle-in-cell methods, centred around the EUTERPE program and heterogeneous architectures at Barcelona Supercomputing Center (BSC).

Doctoral candidate

Student:	Xavier Sáez Pous
NIF:	43.443.709-K
Email:	xavier.saez@gmail.com
Programme:	Computer architecture and technology
Department:	Computer architecture

Thesis advisors

Prof. José María Cela Espín
Prof. Alejandro Soba Pascual

Contents

To Marta and my parents, with love.

Chapter 1

Introduction

"I would like nuclear fusion to become a practical power source. It would provide an inexhaustible supply of energy, without pollution or global warming"

Time Magazine, 15/11/2010
STEPHEN HAWKING

This chapter explains the motivations that encouraged us to start the thesis, spells out the objectives and the main contributions made, and lists all my publications during this period. Finally, it describes briefly the contents of the remaining chapters.

1.1 Motivation

In physics, normally there is a mathematical theory that explains a system's behaviour, but often finding an analytical solution to its theory's equations in order to get an useful prediction is not practical due to its complexity. The great growth over the last few decades in the computational power of computers has introduced a new approach to science. *Computational physics* is the study and the implementation of numerical algorithms to solve problems in physics in a reasonable amount of time [1].

Nevertheless, although computers have become very powerful, they are still unable to provide a solution to most problems without approximations to the descriptive equations. For that reason, in a simplified manner we can say that scientific grand challenges, such as bioinformatics, geophysics, fusion and other types of compute-intensive applications, are the driving force for the evolution of High-Performance Computing (HPC) since they require increasing amounts of computing power to get precise simulations and achieve results (Fig. 1.1).

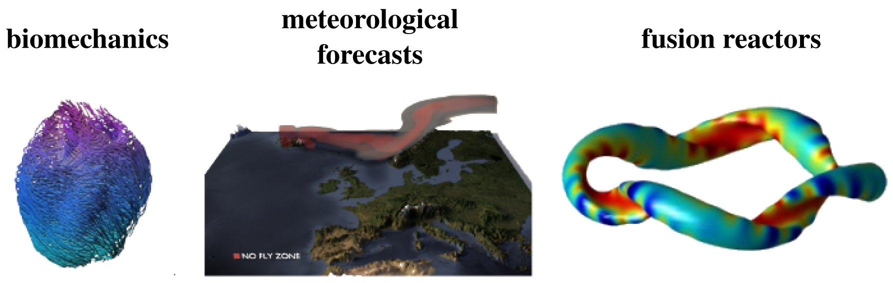


Figure 1.1: Examples of compute-intensive simulations.

The impact of computational physics in the study of plasma physics has been very significant. Firstly, while the laws that describe plasma's behaviour (laws of Newton and Maxwell) are well known, applying them to 10^{20} particles (or more) is not feasible to determine. Moreover, the experiments are difficult and expensive to perform, as for example, building a large device for reaching the desired plasma conditions. Due to this, computer simulations of plasma are an efficient way to progress [2].

Particle-In-Cell (PIC) is one of the most used methods in plasma physics simulations [3]. A large number of PIC codes have been designed to simulate various aspects of the plasma behaviour trying to approximate to real conditions of a fusion reactor [4–6]. The quality of the results achieved by this method can significantly increase the probability that future fusion reactor projects (such as International Thermonuclear Experimental Reactor (ITER) [7] and DEMOnstration Power Plant (DEMO) [8]) will be successful. These results will help us to understand how to avoid the degradation of plasma confinement which is the key in the sustaining of the fusion reactions and therefore in their commercial feasibility.

During the last two decades, HPC has grown rapidly in performance by improving single core processors at the cost of a similar growth in power consumption. For this reason, many scientists have only exploited the process level parallelism in their codes.

However, nowadays the performance of HPC systems is limited by power consumption and power density. Energy has become one of the most expensive resources and, as a consequence, the energy expenses in an HPC center can easily exceed the cost of infrastructure after a few years of operation. Therefore, power efficiency is already a primary concern for the design of any computer system and, looking to the future, new designs of supercomputers will be based on the power efficiency provided by new homogeneous and heterogeneous architectures. One example of this new trend is the Mont-Blanc project [9], as we shall see below in Chapter 4.

At this point, most PIC codes have been designed with a strong emphasis on the physics and have traditionally included only process level parallelism. Therefore, these codes exploit in a non-efficient way the new computing resources and, as a consequence, they are still limited to dealing with simplified models.

In fact, European Union founded a project with the acronym "EUFORIA" (EU Fusion fOR Iter Applications) [10] to improve this situation. The main target of this project was to increase the performance of a core set of fusion simulation codes for using Grid Computing and High-Performance Computing.

The PIC codes require intensive computation to perform their simulations. Therefore, they need to adapt constantly to new computing platforms and this is the aim of this work. This will enable an improvement in the simulations, either by introducing more physics in the code or by incorporating more detail to the simulations.

EUTERPE [11] is one of these PIC codes that simulates fusion plasma instabilities in fusion reactors. It has been written for traditional HPC clusters so it has been parallelized using only Message Passing Interface (MPI). We have selected EUTERPE as a representative of PIC-based production codes to study how to include the latest technologies available in computer science to take advantage of new multiprocessor supercomputers under development, and finally extend this knowledge to other PIC codes.

1.2 Main Objective

Traditionally, PIC simulations have suffered from the defects of dealing with highly idealized configurations due to lack of computing power. When I began this thesis work, the parallel PIC codes were running on several hundreds of processors and the two-dimensional (2D) simulations started to be standard [2]. Back then, the most prevalent computer architecture was based on homogeneous multi-core processors and the heterogeneous architectures began to emerge (e.g the Cell processor).

During the thesis, supercomputers have increased its computational power by several orders of magnitude, which has enabled to face an increase in the size and complexity of the PIC simulations (e.g. three-dimensional (3D) simulations are becoming a standard practice and it is covering more physics to be more realistic). However, even with today's very powerful computers, it is not possible to simulate as many particles as to fully represent a real plasma in this field of interest.

The main objective of this thesis was porting the PIC algorithms to a heterogeneous architecture. In other words, the purpose was to summarize the strategies applied to adapt a PIC code to an heterogeneous architecture and to improve the bottlenecks that would arise in that process. Moreover, this work would also be applicable to any physical phenomena that can be described numerically using PIC codes such as high energy studies, astrophysical numerical, etc.

Next we list the work done during the making of this thesis that have allowed us to reach the main objective:

- Before starting the migration to any new architecture, the state of art of PIC algorithm was studied in order to have a deep knowledge to perform possible optimizations oriented to the new environment.
- A study of the influence of noise into the stability of the system was performed to evaluate the necessity of a filtering method for preventing instabilities in the results of large simulations.
- The communication schemes and the load balancing have been studied in large simulations to evaluate the performance of a PIC application on a large number of processes.
- All the forms of parallelism provided by multi-core architecture have been exploited, including the thread level parallelism.
- A PIC application has been ported to an heterogeneous architecture in order to reach a good profit of its possibilities. Moreover it has been accompanied by an explanation of how to exploit this architecture based on the gained experience during this work.

1.3 Contributions

The main contribution of this thesis can be summarized in the development of several versions of a PIC-based production code using different programming models to take advantage of the different HPC that one can find currently or in a near future. This contribution may be expressed as the sum of the following achievements:

- For the first time, it has performed *a parametric analysis of the EUTERPE code*. In this study, we evaluated the possibility to optimise the simulations through careful selection of the input data and its effect in the quality of the results.
- For the first time, *the scalability of the EUTERPE code was analysed up to 61,440 processors*. Up to that moment, the study of its scalability had not reached a thousand of processors [12].
- EUTERPE only had process level parallelism, thus we decided to introduce thread level parallelism into the code. As a result, we developed *an original hybrid version (OpenMP+MPI) of a PIC-based production code* that allowed us to take advantage of all levels of parallelism that a multi-core architecture offers.
- During the development of the hybrid version of the EUTERPE code, we found that the solver used was not thread-safe. Therefore, we had to implement *an hybrid version (OpenMP+MPI) of the Jacobi Preconditioned Conjugate Gradient and Block Jacobi Preconditioned Conjugate Gradient solvers* to overcome this issue.
- In order to exploit the potential of new heterogeneous architectures, we implemented *a new version (OpenMP+OpenCL) of EUTERPE* that was tested in a new architecture based on ARM processors.
- Finally, we have analysed a new proposal, called OmpSs, to adapt scientific applications to a new architectures (such as heterogeneous) for the purpose of reducing the programming effort and avoiding the rewriting of the code. We wrote *a new version of the EUTERPE code using OmpSs*. It was a bit slower but this fact was more than compensated by the improvement in the development speed.

Later, in Chapter 9, the contributions will be explained in more detail along with the work done during the thesis.

1.4 Contributed Publications

This thesis has been developed through the publication of the following materials:

[13] **X. Sáez, A. Soba, E. Sánchez, R. Kleiber, R. Hatzky, F. Castejón and J. M. Cela**, Improvements of the particle-in-cell code EUTERPE for Petascaling machines, *Poster presented at Conference on Computational Physics (CCP)*, June 23–26, Norges Teknisk-Naturvitenskapelige Universitet, Trondheim, Norway (2010)

[14] **E. Sánchez, R. Kleiber, R. Hatzky, A. Soba, X. Sáez, F. Castejón, and J. M. Cela**, Linear and Nonlinear Simulations Using the EUTERPE Gyrokinetic Code, *IEEE Transactions on Plasma Science* **38**, 2119 (2010)

[15] **X. Sáez, A. Soba, E. Sánchez, R. Kleiber, F. Castejón and J. M. Cela**, Improvements of the particle-in-cell code EUTERPE for Petascaling machines, *Computer Physics Communications* **182**, 2047 (2011)

[16] **X. Sáez, A. Soba, E. Sánchez, J. M. Cela, and F. Castejón**, Particle-in-cell algorithms for plasma simulations on heterogeneous architectures, in *Proceedings of the 19th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 385-389, IEEE Computer Society (2011)

[17] **X. Sáez, T. Akgün, and E. Sánchez**, Optimizing EUTERPE for Petascaling, *Whitepaper of PRACE-1IP project*, Partnership for Advanced Computing in Europe (2012)

[18] **X. Sáez, A. Soba, E. Sánchez, M. Mantsinen, S. Mateo, J. M. Cela and F. Castejón**, First experience with particle-in-cell plasma physics code on ARM-based HPC systems, *Journal of Physics: Conference Series* **640**, 012064 (2015)

[19] **X. Sáez, A. Soba, and M. Mantsinen**, Plasma physics code contribution to the Mont-Blanc project, in *Book of abstracts of the 2nd BSC International Doctoral Symposium*, pp. 83–84, Barcelona, Spain (2015)

[20] **X. Sáez, A. Soba, E. Sánchez, M. Mantsinen, and J. M. Cela**, Performance analysis of a particle-in-cell plasma physics code on homogeneous and heterogeneous HPC systems, in *Proceedings of Congress on Numerical Methods in Engineering (CNM)*, pp. 1–18, APMTAC, Lisbon, Portugal (2015)

1.5 Other Publications

Although the following works are not linked to the thesis, I list them for information purposes because they were published during the same period of this thesis.

1.5.1 Journals

- **E. Sánchez, R. Kleiber, R. Hatzky, M. Borchardt, P. Monreal, F. Castejón, A. López-Fraguas, X. Sáez, J. L. Velasco, I. Calvo, A. Alonso and D. López-Bruna**, Collisionless damping of flows in the TJ-II stellarator, *Plasma Physics and Controlled Fusion* **55**, 014015 (2013)

1.5.2 Papers

- **E. Sánchez, X. Sáez, A. Soba, I. Calvo, R. Kleiber, R. Hatzky, F. Castejón and J.M. Cela**, Modification of turbulent transport by magnetic shear in cylindrical gyrokinetic simulations, in *Europhysics Conference Abstracts*, Vol. 35G, p. 4, European Physical Society, (2011)
- **A. Jackson, F. Reid, J. Hein, A. Soba, and X. Sáez**, High performance I/O, in *Proceedings of the 19th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 349-356, IEEE Computer Society (2011)

1.5.3 Technical Reports

- **X. Sáez and J. M. Cela**, ELSA: Performance Analysis, `http://www.bsc.es/sites/default/files/public/about/publications/1802.pdf`, *Barcelona Supercomputing Center*, 08-1 (2008)
- **X. Sáez and A. Soba**, GENE: Installation guide and performance analysis, *EUFORIA project*, `http://www.euforia-project.eu/EUFORIA/portal.do?TR=A&IDR=1&identificador=12`, TR-CASE-08-003 (2008)
- **J. C. Meyer, J. Amundsen and X. Sáez**, Evaluating Application I/O Optimization by I/O Forwarding Layers, *Whitepaper of PRACE-1IP project*, Partnership for Advanced Computing in Europe (2011)

1.5.4 Posters

- **E. Sánchez, R. Kleiber, R. Hatzky, M. Borchardt, P. Monreal, F. Castejón, A. Soba, X. Sáez and J. M. Cela**, Simulaciones girocinéticas de turbulencia en plasmas de fusión con geometría tridimensional, *Poster presented at XXXIII Biennial Meeting of the Real Spanish Society of Physics*, September 19–23, Santander, Spain (2011)

1.6 Structure of the Thesis

The way in which this work has been written, follows the same logical approach that I followed to perform this thesis.

At the beginning, I needed to answer a couple of simple questions: what the plasma is? and what do I need to learn about this subject?. That's why **Chapter 2** is dedicated to initiate in this topic a possible reader who is not familiar with fusion plasma physics. The chapter introduces the basics concepts that any scientist interested in contributing to the numerical plasma physics need to know, from the definition of plasma as the fourth state of the matter until the simulation in plasma physics.

Once the basic concepts are assimilated, the next step in the logic evolution to understand this work is to present a general description of the PIC algorithm, which is the numerical method applied in EUTERPE. On that basis, throughout this work I present an overview of a PIC code. In fact, many PIC codes follow basically the same schema described in **Chapter 3**. There are no major differences between this general description and the scheme followed by the original supplied version of EUTERPE, which is the reference code used to develop the entire thesis.

Perhaps the next question is why we chose this code and did not start with one simpler code, i.e. why we did not choose an usual skeleton of a PIC code as many publications do, in where some computer science students try an improvement in a PIC code and take this skeleton as their starting point. As explained in **Chapter 1**, Barcelona Supercomputing Center (BSC) has a close contact through several projects with the plasma community. For example, EUFORIA project started a close relation with Centro Investigaciones Energéticas, Medioambientales y Tecnológicas (CIEMAT), an institution that collaborates with Ralf Kleiber's Garching Max-Planck-Institut für Plasmaphysik (IPP) group, which is the owner of EUTERPE code. CIEMAT suggested us the use and improvement of this production code, with all the challenges that this work implied.

The process to port (or adapt) a skeleton code to a multi-core architecture by introducing thread level parallelism or programing several heterogeneous devices would have been an easier road. A highway compared to the same task in the EUTERPE code due to the bigger issues to be solved, such as hundreds of variables, big data structures and a complex organization of the code. As a consequence, we spent a great deal of time to learn the EUTERPE code and **Chapter 5** summarizes briefly the knowledge acquired through a battery of tests and performance analysis over this code, which were necessary for the in-depth knowledge of EUTERPE before starting to change some aspect of it. Apart from this knowledge, the parametric tests also gave us a good opportunity to work in the optimization of the simulations.

Most of the PIC codes used to simulate the real conditions of a fusion reactor have been written with a big emphasis on physics ignoring the latest technologies available in computer science. Therefore, when these codes were parallelized, new supercomputers based on multi-core architectures (described in **Chapter 4**) were not taken into account and these codes only took advantage of the process level parallelism, basically using MPI, as stated in **Chapter 6**.

However, the inclusion of hybrid parallelization techniques in fusion codes allows us to exploit the latest multiprocessor supercomputers as discussed in **Chapter 7** and **Chapter 8**. There are several codes written in this way, but in general they are simplified versions of production codes like those mentioned in this work. These simplified versions are used by scientific computing groups to implement performance improvements. However they can not be used by any physicist that needs to simulate some aspect of a real reactor fusion.

Still, in spite of what we have said so far, we decided to use EUTERPE from the beginning, knowing all the troubles that this decision implied, but taking on this challenge as an opportunity that our work would be useful for the plasma community, specially for those interested in the simulation of a fusion reactor, as ITER.

And finally, **Chapter 9** summarizes all the work done, describes the contributions made and gives some possible perspectives to continue the research initiated with this thesis.

Chapter 2

Plasma Physics

"The way we'll make energy in the future is not from resource, it's really from knowledge."

Head of the CCFE
STEVEN CHARLES COWLEY

Much of the knowledge of plasmas has come from the pursuit of controlled nuclear fusion and fusion power, for which plasma physics provides the scientific basis. This chapter introduces some basic concepts of the plasma physics that will appear further on in the thesis.

2.1 Matter

The common definition of *matter* is anything that has mass and takes up space (volume). We can say that matter is made up of atoms.

The *atom* is the smallest unit of matter that has the properties of a chemical element. It consists of a *nucleus* made up of *protons* and *neutrons* and a surrounding cloud of *electrons* that turn around the nucleus (Fig. 2.1). Protons have positive charge, electrons negative one and neutrons have no charge.

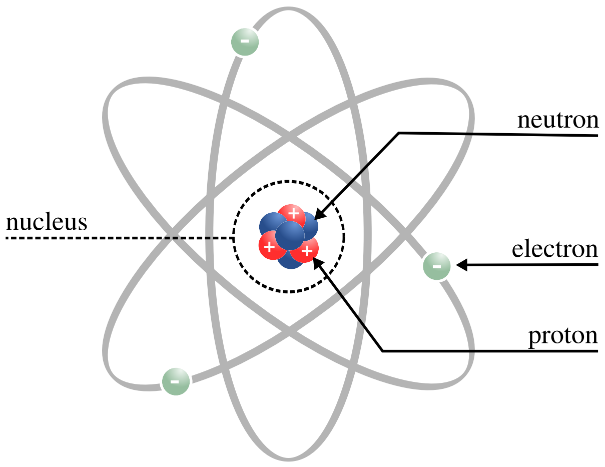


Figure 2.1: Atom model (Lithium).

Most atoms have a neutral charge because they have the same number of protons and electrons. By contrast, when they have a different number of them, the atom has an electrical charge and it is called *ion*.

Isotopes are atoms of elements with different numbers of neutrons, either by loss of neutrons or by getting extra neutrons.

Electrons orbit around the nucleus since they are attracted towards it by the *electromagnetic force* (Fig. 2.2). This force appears due to the attraction between electrons with a negative charge and the protons in the nucleus with a positive charge.

Despite the electrostatic repulsion between positively charged protons, the nucleus holds together by the *strong nuclear force* (Fig. 2.2). This attractive force is stronger than the electromagnetic force but its range is very short (protons and neutrons must be extremely close).

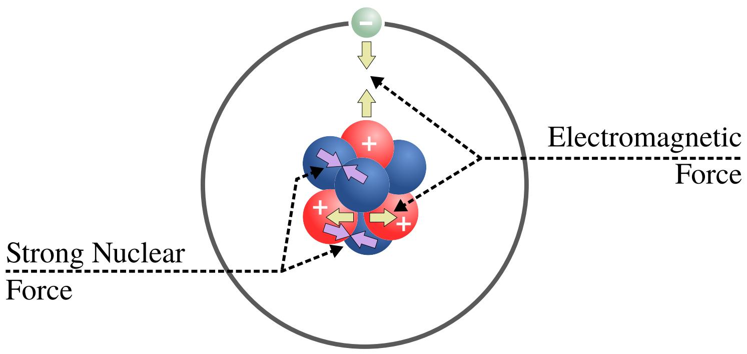


Figure 2.2: Forces inside an atom.

2.1.1 States of Matter

There are four common states of matter (also known as *phases*) observable in everyday life (Fig. 2.3):

- **solid**: matter maintains a fixed volume and shape. The atoms stay tightly close together so they can only vibrate and stay fixed into place.
- **liquid**: matter has a fixed volume but has a variable shape that adapts to its container. The atoms are still close enough to be in contact, but not enough to fix a particular structure. So, they are free to move.
- **gas**: matter has variable volume and shape that expand to fill in the entire container. The atoms are separated and they can move freely and fast.
- **plasma**: matter has variable volume and shape. Plasmas are gases that have so much energy that electrons leave the atoms, conforming a "soup" of neutral atoms, electrons and ions. These charged particles move around freely and make plasma electrically conductive.

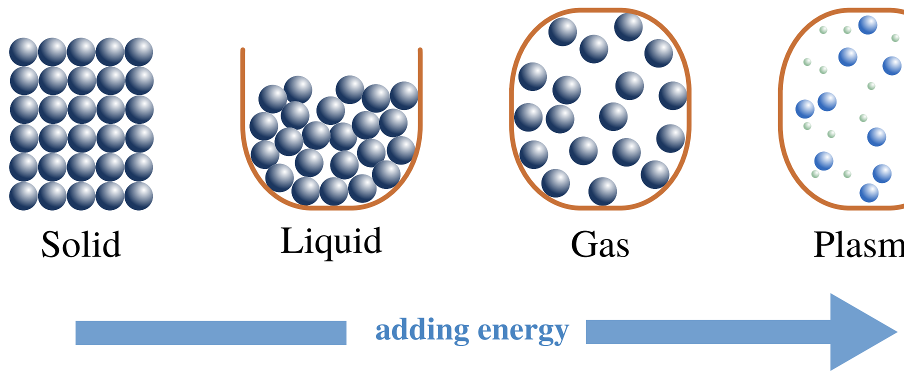


Figure 2.3: States of matter.

The way that matter moves from one state to another is by adding energy through temperature or pressure among others. When matter changes from one state to another affects its properties (such as density, viscosity and conductivity), but it is still the same element (for instance, water).

2.1.2 What is Plasma?

Plasma is a gaseous media in the nature which contains a great number of free charged particles (electrons and ions), but which is approximately neutral on average. In 1928, Langmuir was the first scientist to use the word plasma when he was experimenting with gas ionization [21].

This great amount of free charges provides high electric conductivities and the possibility of establishing electric currents which interact with electromagnetic external fields and with the own electromagnetic fields generated by these currents.

Plasma is the most common form of visible matter in the universe, e.g. the stellar medium (interplanetary and interstellar) and the high atmospheres planetariums. Nevertheless, in the cold media where life is developed, the plasma state is more difficult to find (Fig. 2.4), because of the tendency of free charges to recombine.

In the laboratory, it is necessary to apply great amounts of energy to generate a plasma from a gas. It is usually performed from a huge voltage difference between two points, or by exposing the gas to extremely high temperatures (as in the stars). As a result of this energy increment, the electrons of the gas are so energized that can leave their atoms (*ionization*) and move freely together with the ions. If the plasma becomes cooler, the ions and electrons will bind again into a gas (*deionization*).

Figure 2.4: Plasma examples: lightning bolts (*left*) and auroras (*right*) [www.wikipedia.org].

Plasmas are described by many characteristics that enable to classify them in different types. As an example, Table 2.1 [22] lists some representative plasmas with a few parameters that characterize them (density, temperature, magnetic field and Debye length) which may vary by many orders of magnitude. The *Debye length* describes a screening distance beyond which charges are unaffected by other charges.

Plasma	**Density** n (m^{-3})	**Temperature** T (keV)	**Magnetic field** B (T)	**Debye length** λ_D (m)
Interstellar	10^6	10^{-5}	10^{-9}	0.7
Solar wind	10^7	10^{-2}	10^{-8}	7
Ionosphere	10^{12}	10^{-4}	10^{-5}	$2 \cdot 10^3$
Solar corona	10^{12}	0.1	10^{-3}	0.07
Ion thruster	10^{15}	10^{-3}	–	$4 \cdot 10^{-4}$
Arc discharge	10^{20}	10^{-3}	0.1	$7 \cdot 10^{-7}$
Tokamak	10^{20}	1	10	$7 \cdot 10^{-5}$
Inertial Confinement Fusion	10^{28}	10	–	$7 \cdot 10^{-9}$

Table 2.1: Characteristic parameters of different types of plasmas.

2.2 Nuclear Reactions

A nuclear reaction is a process involving a nucleus or several nuclei. The following nuclear reaction types are related with the plasma fusion:

- **Nuclear fusion**, when two or more small nucleus collide to form new larger nuclei.
- **Nuclear fission**, when a large nucleus splits into smaller nuclei.
- **Radioactive decay**, when a nucleus spontaneously changes itself into a different kind of nucleus.

2.2.1 Nuclear Fusion

Nuclear fusion is the process of combining light nuclei to create a heavier nucleus, such as the Deuterium-Tritium (D-T) reaction shown in Figure 2.5 and explained in detail in next section. The reaction is usually accompanied by the emission of particles (e.g. neutrons). As the total mass of the fusion products is lower than the total mass of the starting nuclei, there is a loss of mass in the process. However, matter has to be conserved following Einstein's formula ($E = mc^2$) so the loss mass is transformed into kinetic energy of the products. This large amount of emitted energy transforms matter to a plasma state.

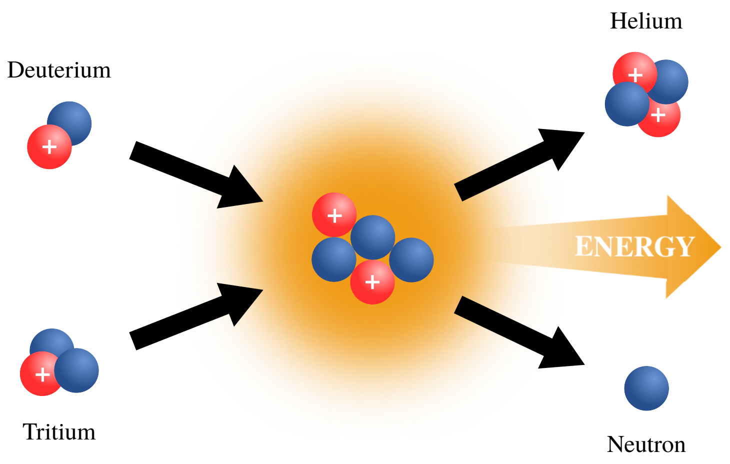


Figure 2.5: Schematic representation of a fusion reaction.

The emission of energy only occurs when initial nuclei have lower mass than iron, otherwise when they are heavier than iron then the reaction absorbs energy instead of emitting it.

To achieve a fusion reaction, it is necessary bringing the nuclei so close together that nuclear forces counteract the electromagnetic force that repels the nuclei (with positive charge due to their protons). This condition easily occurs in a high density and high temperature environment as in the stars where nuclei have enough kinetic energy that they can approach each other despite the electrostatic repulsion (Fig. 2.6).

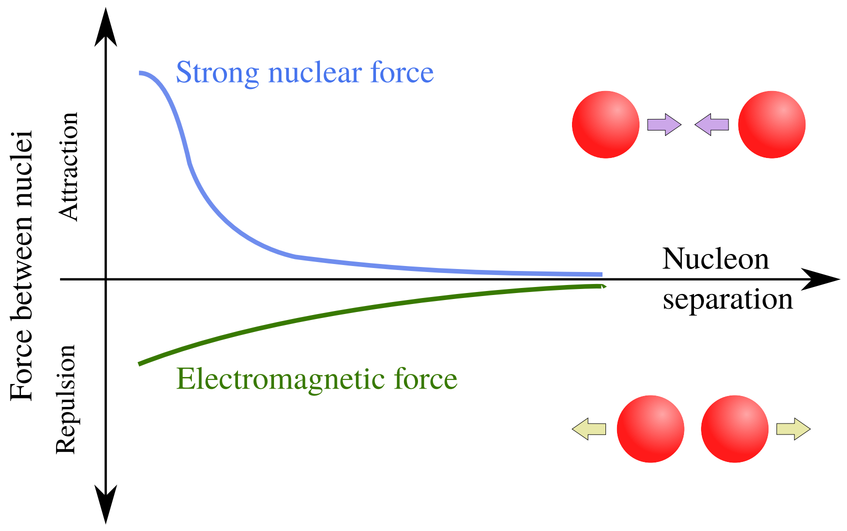


Figure 2.6: Forces between two nuclei according to the distance between them. At short range the nuclear force is sufficiently attractive as to overcome the electromagnetic repulsive force.

On Earth it is very difficult to start and maintain nuclear fusion reactions. Sadly, the only successful approach so far has been in nuclear weapons as the hydrogen bomb which uses an atomic (*fission*) bomb to start the fusion reactions. On the other side, in a pacific way, for over 50 years scientists and engineers have been trying to find a safe and working way of controlling and containing fusion reactions that can generate electricity on-demand. They still have many challenges to overcome before it can be used as a clean source of energy.

2.2.2 Deuterium-Tritium (D-T) Fusion

The idea of a controlled fusion reaction starts in the 1950's, when the reaction D-T was studied (Fig. 2.7). This reaction is produced between a deuterium nucleus (D) with a tritium nucleus (T) to generate a helium nucleus (He), a neutron (n) and a certain amount of energy (around 17.6 mega electron volts):

$$D + T \rightarrow\ ^4He\ (3.5MeV) + n\ (14.1MeV) \tag{2.1}$$

where 1 electron volt (eV) $\approx 1.6 \cdot 10^{-19}$ joules (J).

Deuterium (one proton and neutron) and Tritium (one proton and two neutrons) are isotopes of hydrogen and they are promising ingredients because hydrogen is a key part of water, which covers the Earth.

However, Tritium is a radioactive isotope and is more challenging to locate in large quantities naturally due to its radioactive decay. It has a half-life of a little over 12 years, i.e. half of the quantity decays every twelve years. However, it is relatively easy to produce it from Lithium, a common element in Earth's crust and in seawater (e.g. in sea salt) with enormous reserves, using the following reaction (Fig. 2.7):

$$^6Li + n \rightarrow T +\ ^4He + 4.8MeV \tag{2.2}$$

The fusion reaction rate increases rapidly with temperature up to a maximum point and then gradually drops off. The optimum range for the D-T reactions is at temperatures close to 100 keV.

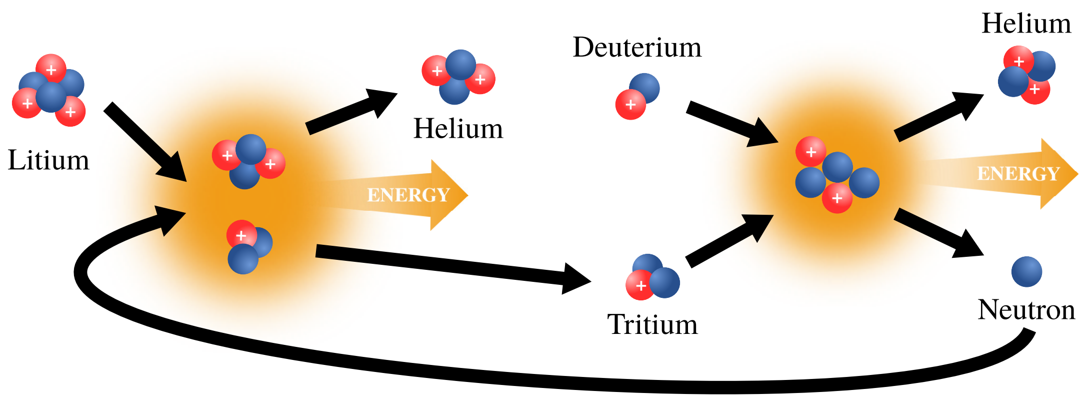


Figure 2.7: Complete D-T reaction.

2.3 Fusion Reactors

Fusion reactor is a device used to initiate and control a sustained nuclear fusion reaction. To achieve its goal, a fusion reactor has to satisfy the following requirements:

- **High temperature**: provide hundreds of millions degrees Celsius (as in the center of stars) to separate electrons from nucleus (plasma state) and give nuclei enough kinetic energy to overcome the electrostatic repulsion and get sufficiently close for nuclear forces come into play.
- **Enough Density**: the electromagnetic repulsion makes head-on collisions very unlikely, so most particles will not crash with each other. It is necessary to increase the density so that nuclei are more close to each other and the probability of collisions increases sufficiently to generate the nuclear fusion reactions.
- **Confinement**: plasma tends to expand because the nuclei move away among them at high speed and this significantly reduces the probability of collisions and kills the fusion reaction. In order to keep the two previous requirements to extend the fusion reactions during a prolonged time, it is necessary to confine the plasma in a closed space (the *vessel*) where it cannot escape.

Given that extremely high temperatures are required to sustain the fusion reaction, the plasma confinement is not possible using material walls only. Two different types of confinement are investigated to confine the plasma for a sufficiently long time to achieve fusion reactions:

- **Inertial confinement**: seeks to fuse nuclei so fast that they do not have time to move apart. High-energy beams of laser are used to heat outer layer of a fuel target, typically in the form of a micro-balloon (Fig. 2.8). This layer explodes outward, producing a reaction force against the remainder of the target, accelerating it inwards and compressing the target. This compression can heat the fuel at the center so much that fusion reactions occur.
- **Magnetic confinement**: seeks to extend the time that ions spend close to each other in order to facilitate fusion. Magnetic fields are used to confine the hot fusion fuel in a plasma state.

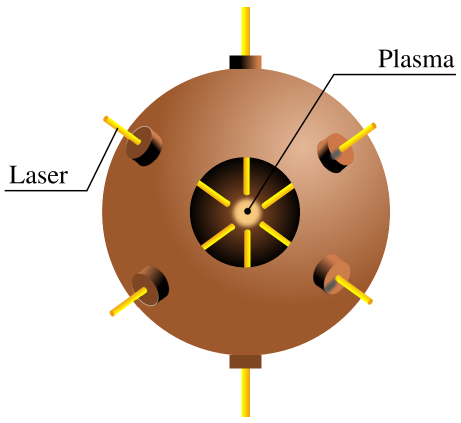


Figure 2.8: Inertial Confinement Fusion (ICF).

In the following, a more detailed explanation of magnetic confinement reactors is given because the plasma simulations covered in this thesis are focused on this kind of fusion reactors.

2.3.1 Magnetic Confinement

To confine the particles, magnetic fields are used taking benefit from the conductivity of the particles in the plasma. This is called *magnetic confinement*. The magnetic field prevents particles reaching the walls while it maintains the high pressure and the density required for fusion reactions.

Charged particles into the mentioned electromagnetic configurations describe helical orbits around an axis parallel to the direction of the magnetic field, as shown in Figure 2.9. We say then that these particles *"gyrate"* around the field line while moving freely along it [23].

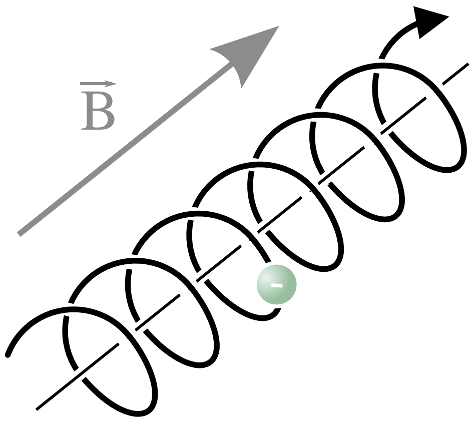


Figure 2.9: The gyrating motion of a charged particle.

The angular frequency of this circular motion is known as the *gyrofrequency*, or *cyclotron frequency*. When this charged particle is an ion, its angular frequency is named *ion cyclotron frequency* and can be expressed as

$$\Omega_i = \frac{q_i B}{2\pi m} \tag{2.3}$$

where q_i is the charge of the ion, m is the mass of the ion and B is the magnetic field strength.

The simplest magnetic confinement fusion reactor configuration is a cylinder solenoid, which is a long cylinder wound with magnetic coils producing an electromagnetic force parallel to the axis of the cylinder. This field prevents the escape of ions and electrons radially, but does not prevent its loss at the ends of the solenoid.

Mainly, there are two magnetic confinement systems (Fig. 2.10) for solving this problem [24–26]:

- **Open configurations**: tries to stop up the plasma at the ends of the device by increasing the magnetic field strength at each end of the solenoid. This compresses the magnetic field lines and causes that most of the charged particles (within a limited range of velocity and approach angle) to be reflected to the low density magnetic field. Due to this behaviour, this configuration receives the name of *magnetic mirrors*. However the confinement time proves to be too short, because although the loss at the ends is slowed, it remains too large.

- **Close configuration** : To cancel out this drift, the toroidal magnetic field is supplemented by a poloidal field component which adds a helical twist to the total magnetic field configuration. This poloidal component is produced in one of two ways: *tokamak* (by the magnetic field produced by an electric current flowing toroidally around the plasma configuration) and *stellarator* (by modifying the geometry of the torus, twisting one end by 180 degrees to produce a figure-8 shaped device).

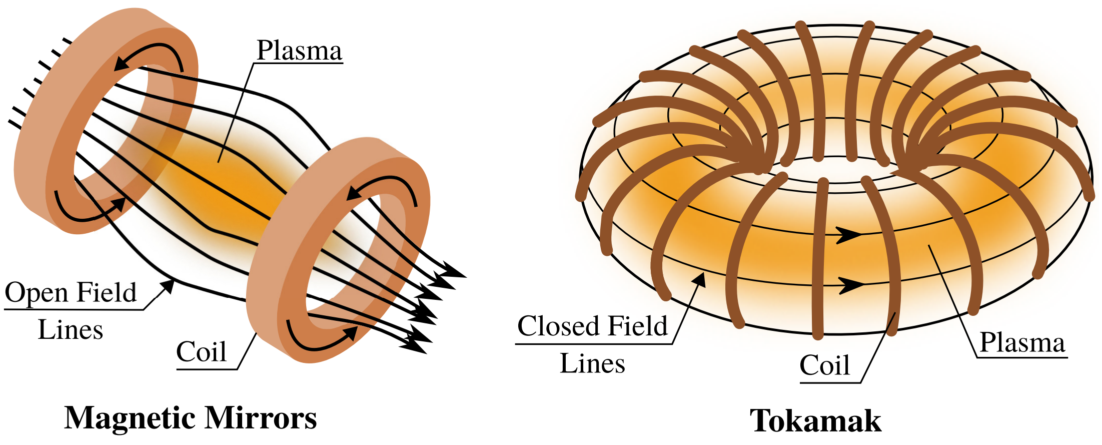


Figure 2.10: Magnetic confinements and field lines.

Once plasma is confined, energy has to be provided to reach ignition temperature for starting the fusion reaction. This is done using radiofrequency techniques and neutral beam injection.

The motion of a single charged particle in a external magnetic field is well-understood, but when a large number of charged particles are put together, complex interactions appear among them and electromagnetic fields are generated by the charged particles. The simplest approach to study is modeling the entire plasma as a magnetized fluid by following the Magnetohydrodynamics (MHD) methodology.

While the MHD methodology explains reasonably successfully the macroscopic state and the stability of fusion plasmas under actual operating conditions, some key phenomena demand a *full kinetic description* of the plasma. In particular, the micro-turbulences generated by the above mentioned interactions have been studied during the past decade to analyze the transport of energy and particles in tokamaks reactors [27].

2.3.2 Tokamak

Tokamak (*TOroidal KAmera (chamber) MAgnet and Katuschka (coil)*) is a magnetic confinement device with the shape of a torus. The initial design was first theorized by Oleg Lavrentiev in the 1950s, but it was not until a few years later when Igor Tamm and Andrei Sakharov invented what we know now as a Tokamak.

The main parts of a tokamak reactor (Fig. 2.11) are:

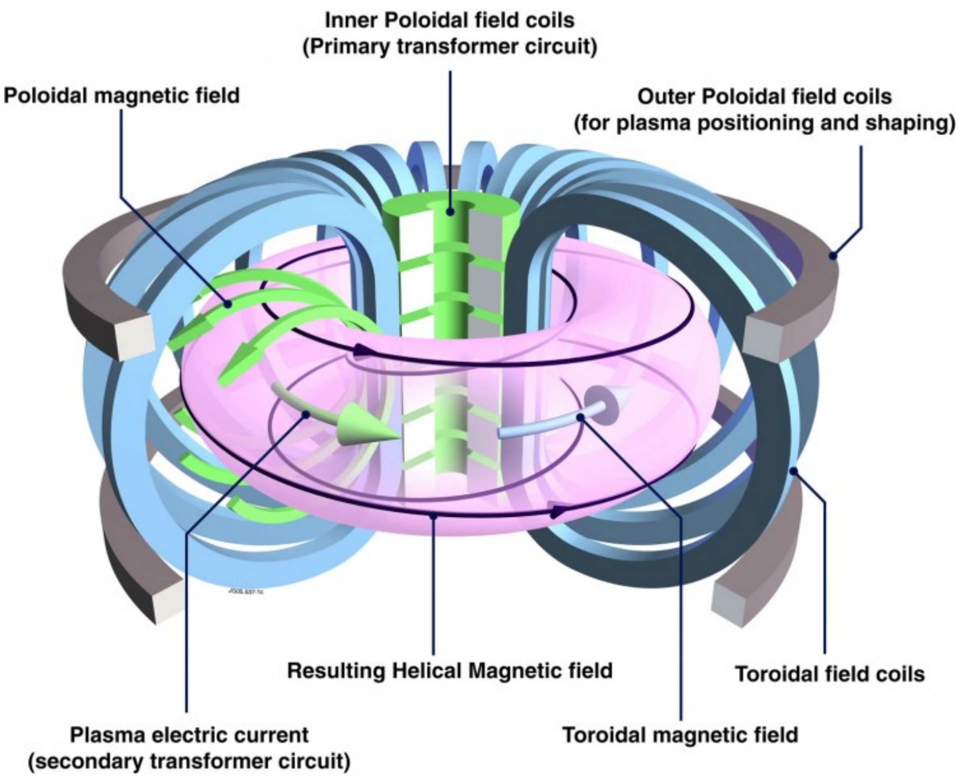

Inner Poloidal field coils
(Primary transformer circuit)
Poloidal magnetic field
Outer Poloidal field coils
(for plasma positioning and shaping)
Resulting Helical Magnetic field
Toroidal field coils
Plasma electric current
(secondary transformer circuit)
Toroidal magnetic field

Figure 2.11: Diagram illustrating a tokamak reactor. [www.euro-fusion.org]

- **Vessel** or **Chamber**: the container where plasma is contained in vacuum.
- **Toroidal** and **Outer poloidal field coils**: the combination of both coils creates a field in both vertical and horizontal directions, respectively, acting as a magnetic "cage" to hold and shape the plasma.
- **Inner poloidal field coil**: it induces the plasma current acting as the primary transformer.

- **Plasma electric current**: it is the secondary transformer circuit. The heating provided by it (known as Ohmic heating) supplies up to a third of the temperature required to achieve the fusion.
- **Neutral beam injectors**: neutral atoms are injected at high speed into the plasma to provide additional plasma heating. Here they are ionized and they transfer their energy to the plasma as they slow down in collisions with the plasma particles and heat it.
- **Radiofrequency emitters**: High-frequency oscillating currents are induced in the plasma by external coils or waveguides. The frequencies are chosen to match regions where the energy absorption is very high (resonances). In this way, large amounts of power may be transferred to the plasma.

ITER

Several fusion reactors have been built, but only recently reactors have been able to release more energy than the amount of energy used in the process. Although research started in the 1950s, no commercial fusion reactor is expected before 2050. The ITER project is currently leading the effort to harness fusion power.

The *International Thermonuclear Experimental Reactor (ITER)*[7] is an international tokamak research project that could help us to make the transition from today's studies of plasma physics to future fusion power plants. Its building is based on research done with previous devices such as DIII-D, EAST, KSTAR, TFTR, ASDEX Upgrade, JET, JT-60, Tore Supra and T-15.

Seven participants are involved formally in the creation of this nuclear fusion reactor: the European Union, India, Japan, People's Republic of China, Russia, South Korea and USA. The site preparation began in Cadarache (France) and the construction of its large components has started.

ITER (Fig. 2.12) was designed to produce approximately 500 MW (500,000,000 watts) of fusion power sustained up to 1,000 seconds by fusing of about 0.5 g of deuterium/tritium mixture in its approximately 840 m^3 reactor chamber. Although the energy produced in the form of heat is expected to be $5 - 10$ times greater than the energy required to heat up the plasma, it will not be used to generate any electricity [26]. The main goal of ITER is to prove the viability of fusion as an energy source and collect the data necessary for the design and subsequent operation of the first nuclear fusion power plant named DEMOnstration Power Plant (DEMO).

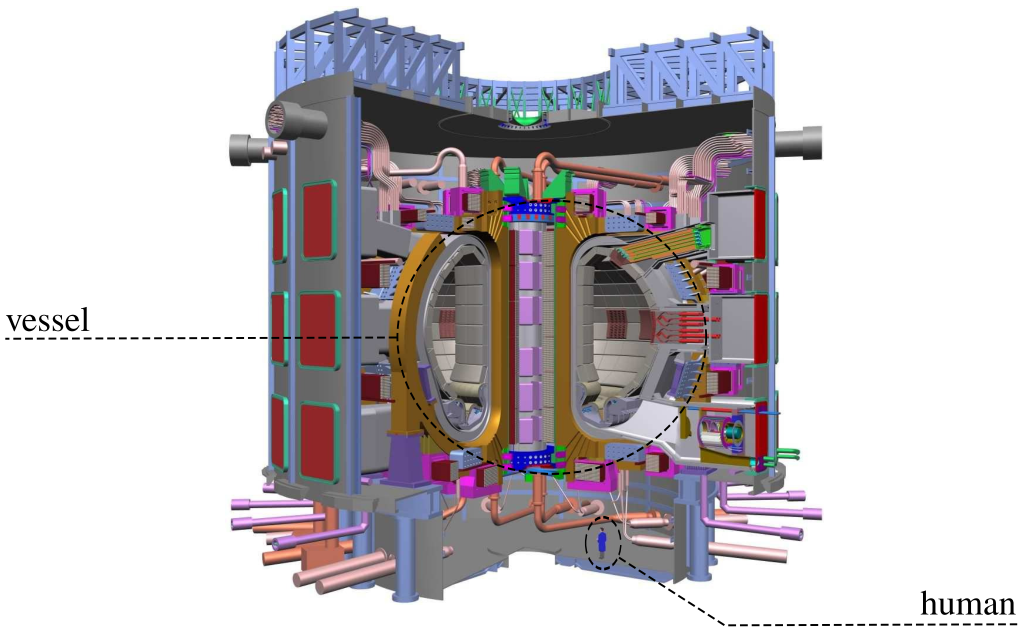


Figure 2.12: The projected design for the ITER tokamak. [www.iter.org]

2.4 Mathematical Models in Plasma Physics

The dynamics of a plasma has a great complexity and can be described using different approximations depending on the physical aspect to analyze. The most complete description is provided by the *kinetic* theory, which considers the microscopic velocities of the particles over the distribution function of each species.

The *distribution function* (f_α) indicates the number of particles of the species α per volume in the phase space. A *phase space* is a mathematical space in which all possible states of a system are represented, and where each possible state corresponds to one unique point in this space. Every axis of the phase space is a parameter or a degree of freedom of the system, i.e. position and velocity.

When the ionisation of the atoms balances the recombination of the electrons and ions, the average number of plasma particle species is constant in time. In these conditions, *full kinetic description* of a plasma can be expressed with the equation of conservation of the particle distribution function (*Boltzmann equation*):

$$\frac{\partial f_\alpha}{\partial t} + \vec{v} \cdot \frac{\partial f_\alpha}{\partial \vec{x}} + \frac{\vec{F}^{tot}}{m_\alpha} \cdot \frac{\partial f_\alpha}{\partial \vec{v}} = C_\alpha \tag{2.4}$$

where m_α is the particle mass, t is the time, v is the speed vector of the particle, x is the position vector of the particle and C_α represents collisions with all species. The force ($\vec{F}^{tot}$) exerts on plasma particles is the addition of two kind of forces: the external ones ($\vec{F}^{ext}$) provided by external fields and the internal ones ($\vec{F}^{int}$) provided by the fields generated from the particles far away of the Debye sphere. These mentioned fields are macroscopic.

The *Debye sphere* is a characteristic plasma parameter which divides the space in two regions: one with the short-range interaction inside the sphere and the other region with the long-range interaction outside it. The number of particles inside the Debye sphere will determine much of the physics involved in plasma [24, 25]. The radius of this sphere is the *Debye length* which may be expressed as:

$$\lambda_D = \sqrt{\frac{\varepsilon_0 k_B T}{n_e^2}} \tag{2.5}$$

here ε_0 is the permittivity of free space, k_B is Boltzmann's constant, T is the absolute temperature and n_e is the density of electrons.

The $\vec{F}^{ext}$ is modelled by the *Lorentz equation* (Eq. 2.6), which calculates the movement of a particle of mass m, velocity v and charge q moving in an electric field E and a magnetic field B.

$$\vec{F}^{ext} = m\frac{d\vec{v}}{dt} = q\left[\vec{E} + \vec{v} \times \vec{B}\right] \tag{2.6}$$

Using the conservation of number of particles (Eq. 2.4) (without particle recombination or escape from the system), the full kinetic description of a plasma can be expressed using the *Vlasov equation* (Eq. 2.7) where particles are assumed to be uncorrelated.

$$\frac{df}{dt} \equiv \frac{\partial f}{\partial t} + \vec{v} \cdot \frac{\partial f}{\partial \vec{x}} + \left(\frac{\vec{F}^{ext}}{m_\alpha} + \int \frac{\vec{F}^{int}}{m} f(\vec{x}, \vec{v}, t) d\vec{x} d\vec{v}\right) \cdot \frac{\partial f}{\partial \vec{v}} = 0 \tag{2.7}$$

The *Vlasov equation* is a differential equation that describes the time evolution of the distribution function of plasma consisting of charged particles with long-range interaction. It is said that particles are *uncorrelated* when there are no collisions among them, so C_i is considered to be equal to 0. This situation is accomplished when the number of particles into the Debye sphere is large, typically $n\lambda_D^3 \approx 10^9$.

In addition, the electromagnetic fields (external and internally) are modeled by the *Maxwell's equations* (Eq. 2.8).

$$\begin{aligned}
\nabla \cdot \vec{E} &= \frac{1}{\varepsilon_0}\sum_\alpha q_\alpha \int f_\alpha d^3v \\
\nabla \cdot \vec{B} &= 0 \\
\nabla \times \vec{E} &= -\frac{\partial \vec{B}}{\partial t} \\
\nabla \times \vec{B} &= \mu_0 \sum_\alpha q_\alpha \int \vec{v} f_\alpha d^3v + \frac{1}{c^2}\frac{\partial \vec{E}}{\partial t}
\end{aligned} \tag{2.8}$$

where $\vec{E}$ is the electric field, $\vec{B}$ is the magnetic field, μ_0 is the permeability of free space and c is the speed of light in free space.

To compute the field $\vec{E}$, or equivalently $-\nabla\phi$, we solve the *Poisson's equation* (Eq. 2.9),

$$\nabla^2 \phi = -\frac{q}{\varepsilon_0}\int f\left(\vec{x}, \vec{v}, t\right) d\vec{v} \tag{2.9}$$

where ϕ is the electrostatic potential arising from ensemble-averaged charge density distribution of all the species in plasma, i.e.

$$\rho = \sum_\alpha q_\alpha \int f_\alpha \left(\vec{x}, \vec{v}, t\right) d\vec{v} \tag{2.10}$$

The high electric conductivity of plasmas, due to electrons are very mobile, implies that any charge that is developed is readily neutralized, and thus plasmas can be considered as electrically neutral. *The quasi-neutrality equation* (Eq. 2.11) describes the apparent charge neutrality of a plasma overall, or in other words it describes that the densities of positive and negative charges in any sizeable region are equal. When the region is less than the Debye sphere, the quasi-neutrality is not guaranteed.

$$\sum_{\alpha} q_\alpha n_\alpha = 0 \tag{2.11}$$

Finally, it is interesting to note an approximation used for investigating low frequency turbulence in fusion plasmas. As it was explained in Section 2.3.1, the trajectory of charged particles in a magnetic field is a helix that winds around the field line. This trajectory can be decomposed into a fast gyration about the magnetic field lines (called *gyromotion*) and a slow drift of the guiding center along the field line (Fig. 2.13).

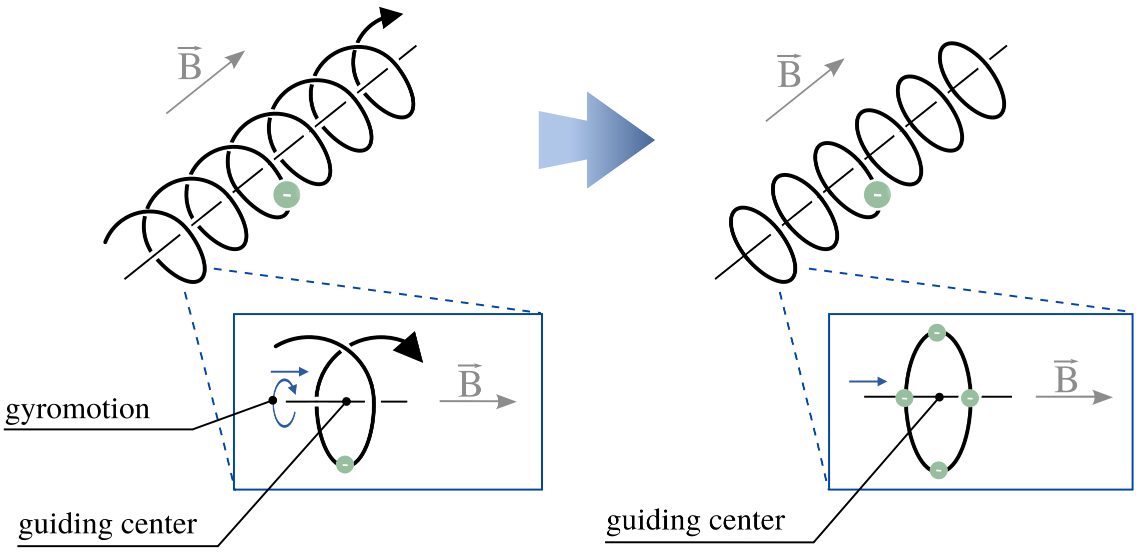


Figure 2.13: Gyrokinetic model (using 4-point average)

For most plasma behavior, this gyromotion is irrelevant. So the gyromotion of a particle is approximated by a charged ring moving along the field line and the gyro-angle dependence is removed in the equations. From *Vlasov equation*, by averaging over this gyromotion, one arrives at the *gyrokinetic equation* which describes the evolution of the guiding center in a phase space with one less dimension (3 spatial, 2 velocity and time) than the full Vlasov equation (3 spatial, 3 velocity and time):

$$\frac{\partial f}{\partial t} + \frac{\mathrm{d}v_{\parallel}}{\mathrm{dt}}\frac{\partial f}{\partial v_{\parallel}} + \frac{\mathrm{d}\vec{R}}{\mathrm{dt}}\frac{\partial f}{\partial \vec{R}} = 0, \tag{2.12}$$

where $v_{\parallel}$ is the velocity parallel to the magnetic field line and $\vec{R}$ is the position vector of the guiding center of the charged rings.

2.5 Simulating Plasma Behaviour

The great growth in the computational power of computers has introduced a new approach to science: *computational physics*. It is the study and the implementation of numerical algorithms to solve problems in physics. Often, there is a mathematical theory that explains a system's behaviour, but solving its theory's equations in order to get an useful prediction is not practical due to its complexity.

In particular, the impact of computational physics in the study of plasma physics has been very significant. The complexity of fusion-grade plasmas and the increased computational power in recent years have made of the simulation of plasmas a prime object of study in the field of fusion research. The computer simulation enables the study of the behaviour of a physical device model when the experimentation with the physical device is prohibitively expensive, or is very difficult to change quickly device parameters or is not possible to control the external influences on the device.

The development of codes that numerically simulate plasma behaviour under realistic conditions, can improve our understanding and control of plasma dynamics in fusion devices such as tokamak. This better knowledge significantly increases the probability that ITER will be successful. For example, energy and the particle motion are responsible for the confinement degradation, so their comprehension is vital because the sustaining of fusion reactions depend on the confinement of plasma and consequently the commercial feasibility too.

There are so many non-linear phenomena in fusion plasma with wide ranges of time scale and spatial scale interacting each other (Fig. 2.14) that one simulation can not cover all range.

As a consequence, there is a great number of codes that model different aspects of a fusion reactor's operation. Some examples are given below:

- *Plasma turbulence*: It is the dominant form of the transport in fusion plasma and limits the maximum reachable temperature and density. It generates loss of particles and heat. Some codes in this area: GENE [28], GYSELA [29], EUTERPE [30] and ELMFIRE [31].
- *Fusion neutronics*: Fusion plasma is a source of neutrons which impinge on all surrounding the plasma. Through interaction with neutrons, materials become activated and subsequently, emit decay photons. The resulting shut-down dose rate

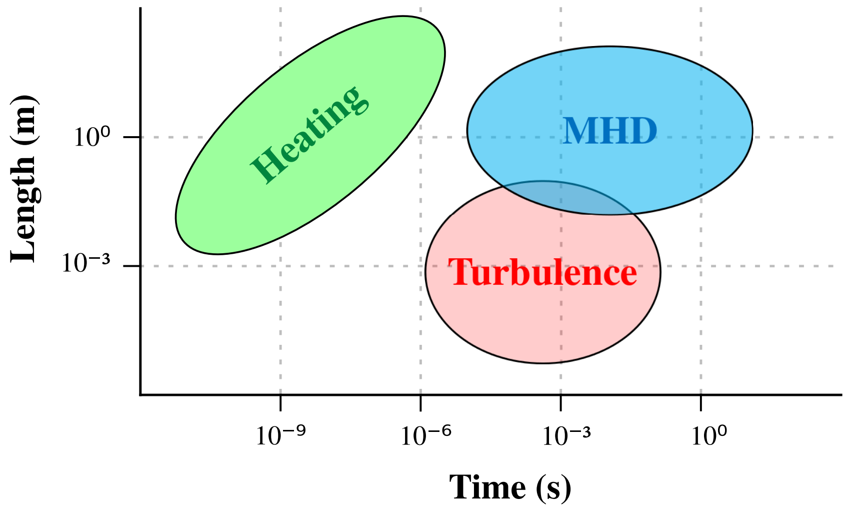


Figure 2.14: Physical processes take place over several length and time scales in a fusion plasma reactor.

is an occupational health issue that needs to be assessed carefully. There are two families of neutronics codes: deterministic (e.g. PARTISN [32] and ATTILA [33]) and Monte Carlo (e.g. MCNP [34] and TRIPOLI [35]).

- *Fast particle physics*: These particles will collide with other plasma particles causing instabilities and losses of fast particles that can damage the surrounding materials and thus need to be kept at a tolerable level. Two examples of codes in this area are SPIRAL[36] and ASCOT [37].

- *Magnetohydrodynamics (MHD) in fusion plasma.* Fusion plasma must have an edge pressure gradient which is limited by an MHD instability, called Edge Localised Mode (ELM). ELMs lead to a large erosion of surrounding materials and a limited lifetime of the fusion reactor. A pair of codes in this area are JOREK [38] and M3D [39].

The latest challenge is integrating these codes modeling different aspects in a consistent manner to get a complete simulation of a fusion reactor.

Chapter 3

Particle-in-Cell

"The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work."

John von Neumann

This chapter describes in detail the Particle-In-Cell (PIC) method and lists some existing codes that implement this method as example. Finally, it presents the selected code to perform this thesis.

3.1 Description

The Particle-In-Cell (PIC) is a technique used to model physical systems whose behaviour varies on a large range of spatial scales. On a macroscopic level, the dynamics is described by a continuum model (a system of partial differential equations), whereas on a microscopic level it is modeled by a collection of discrete particles. Dealing with both levels separately can be problematic, thus PIC methods attempt to bridge the gap between the microscopic dynamics of each particle and the macroscopic behaviour of the system.

In other words, in a PIC method, the individual particles (or fluid elements) are tracked in a continuous phase space, whereas moments of the distribution (such as densities and currents) are computed concurrently on stationary mesh points (Fig. 3.1). The particles have one or more properties assigned (e.g. mass, velocity, charge, material, etc.) and they are free to move and follow the dynamics of a flow regardless of its complexity. The particles are responsible for moving mass, momentum and energy through the grid.

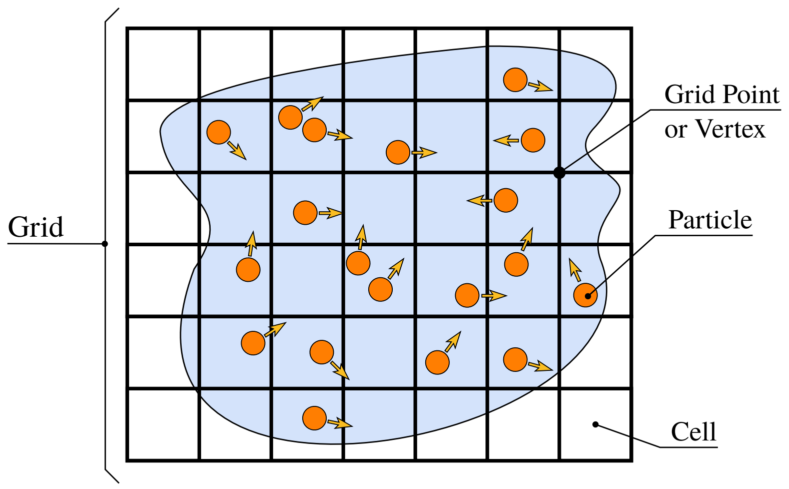


Figure 3.1: The PIC model.

3.1.1 Schema

The PIC method consists of four operations [40] that are repeated in the same order at each time step (Fig. 3.2):

- **scatter**: the particle attributes are interpolated to nearby points of a regular computational mesh that discretizes the problem domain.
- **solve**: the appropriated moment equations are solved on the computational mesh.
- **gather**: the momentum on each particle is found by interpolation on the mesh.
- **push**: the particle are moved under the influence of the momentum and the particle attributes are updated.

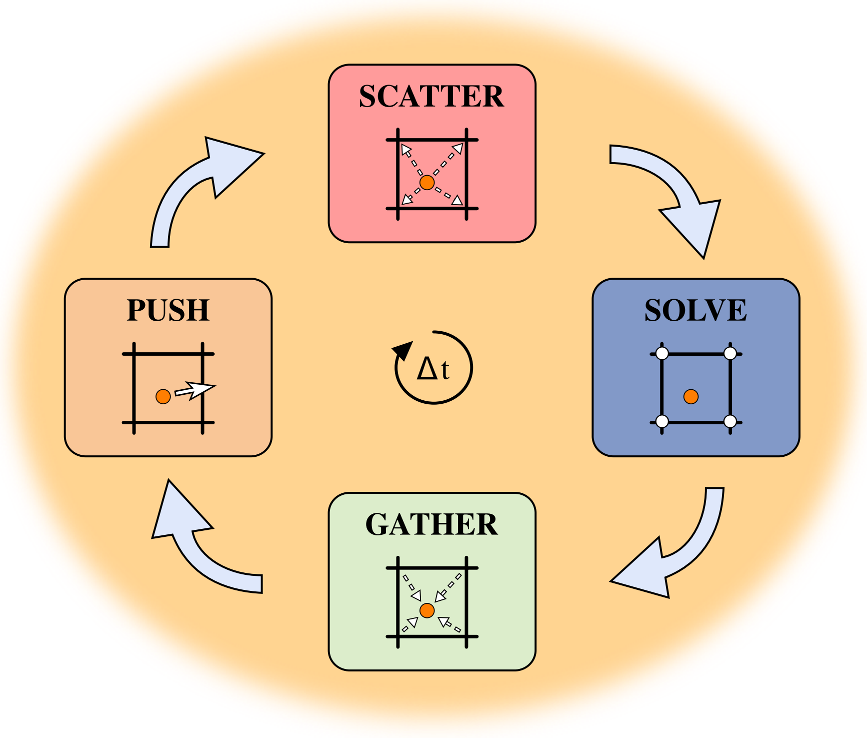


Figure 3.2: Schema of the PIC method.

3.2 PIC Plasma Simulations

Historically, PIC methods were already in use as early as the 1950s [41]. They gained popularity for plasma simulation after the works of Buneman, Dawson, Hockney [25], Birdsall [3], Morse and others. Nowadays, PIC methods are used in several research areas such as astrophysics, plasma physics and semiconductor device physics [25].

We can consider a fusion plasma as a huge N-body problem since plasma can contain the order of 10^{20} particles per cubic meter (Table 2.1). If we use a *particle-particle method* for the simulations, then the amount of work will be on the order of $\mathcal{O}(N^2)$ because all pairwise interactions among particles will be evaluated. Therefore, the computing cost for this method is prohibitive for simulations on big geometries, such as International Thermonuclear Experimental Reactor (ITER) which will have a plasma volume of 840 cubic meters.

In plasma physics, the PIC method is a numerical approach that simulates a collection of charged particles that interact via external and self-induced electromagnetic fields. A spatial grid is used to describe the field while the particles move in the continuous space.

The field and the particle motion are solved concurrently. In this case the simulation requires less amount of work, since each particle only interacts with the grid points of the cell where it is located.

Another way to reduce this computational complexity is to use a representative subset of the particles and then extrapolate their behaviour to the whole plasma. This approximation is applied to the *full kinetic description* (Section 2.4) in the PIC method by replacing the distribution function f_α by a number of *macroparticles* (or *markers*), where each macroparticle is a cloud of particles. The charge and current densities of macroparticles are accumulated on the spatial grid using an interpolation method and then the field equations (Eq. 2.8) are solved on the grid. Finally, the forces ($\vec{F}^{tot}$) acting on the macroparticles are obtained by interpolating the fields back at the macroparticles positions [2].

In short, PIC is one of the most used methods in plasma physics simulations. As the quality of results achieved by this method relies on tracking a very large number of particles, PIC codes require intensive computation and need to be adapted to new computing platforms constantly.

3.3 PIC Algorithm

Algorithm 3.1 illustrates a basic PIC algorithm for plasma simulations [40]. After the initialization, the four phases introduced in Section 3.1.1 are repeated at each time step. Essentially we can say that the process involves moving charged particles due to forces from the fields. At the end, the results are recorded.

3.3.1 Main Data Structures

The particle data structure stores the velocity and the position for each particle. The charge and the mass can be stored elsewhere since all the particles belonging to the same species (such as ions, electrons...) have the same value.

The grid (usually uniform and rectilinear) is composed by discrete points (vertices) in space so it can be stored as an array of grid points. The number of cells or points depends on the physics to be studied. The fields are continuous quantities that are discretized on the grid, so they are stored in the grid points array.

Algorithm 3.1 PIC schema for plasma simulations

```
 1: Initialize particle data                              ▷ INITIAL
 2: Initialize grid (field) data
 3: while $t < t_{max}$ do
 4:     Compute particle contributions to the grid        ▷ SCATTER
 5:     Calculate the fields                              ▷ SOLVE
 6:     Update forces on particle positions               ▷ GATHER
 7:     Move particles to new positions                   ▷ PUSH
 8:     $t \leftarrow t + t_{step}$
 9: end while
10: Print particles and grid data                         ▷ RESULTS
11: Print statistics
```

3.3.2 Initial Phase

The simulation can start from the beginning (reading the input parameters and initializing to $t_{ini} = 0$) or from a state of a previous simulation (by reading restart files). In both cases, the particle and field structures are filled in with appropriate initial conditions (Algorithm 3.2). Once done, the loop starts with the computation of the charge density on the grid.

Algorithm 3.2 Initialization phase (when $t_{ini} = 0$).

```
1: build Domain from the input
2: for all particle $p \in Domain$ do
3:     $position(p) \leftarrow random()$
4:     $velocity(p) \leftarrow 0$
5: end for
6: for all vertex $v \in Grid$ do
7:     $field(v) \leftarrow 0$
8: end for
```

3.3.3 Scatter Phase

This step consists in calculating the charge density (ρ) at grid points from nearby charged particles. The charge of each particle is distributed among the grid points which form the enclosing cell using a *weighting algorithm*.

The first PIC codes implemented the Nearest Grid Point (NGP) approximation as weighting procedure, but this method caused excessive fluctuations when the particles moved across the grid. Nowadays most codes use a *linear interpolation* procedure (Fig. 3.3), which involves 4 points in a 2D grid (called bilinear) and 8 points in a 3D grid (called trilinear) [2].

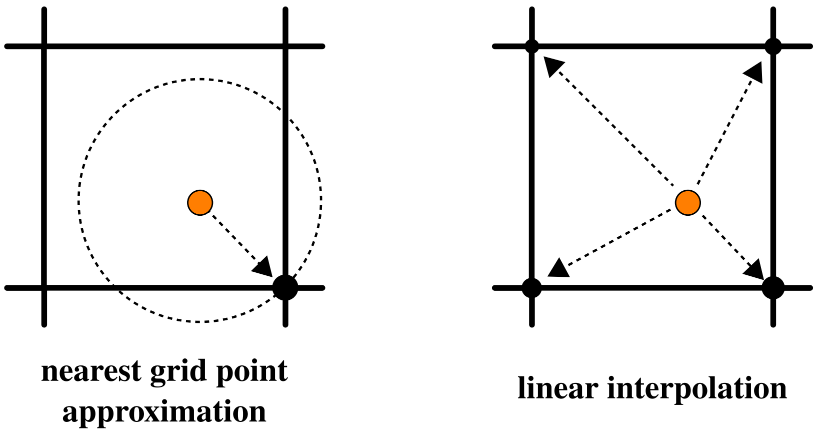


Figure 3.3: Weighting methods (in 2D).

To illustrate the weighting method in detail, the algorithm based on a bilinear interpolation is considered in Figure 3.4. The particle divides the two-dimensional (2D) grid cell where it is located into four smaller areas. The total area of the grid cell is normalized to 1, so each subarea is the weighting coefficient to the opposite vertex. It means that the weights (areas) are inversely related to the distance from the grid points to the particle, so the closest grid point receives more charge of the particle than the rest.

Finally, the charge density at the grid point is the charge of the particle multiplied by the calculated weighting coefficient (subarea with the same color). Algorithm 3.3 formalizes the explanation given for this phase.

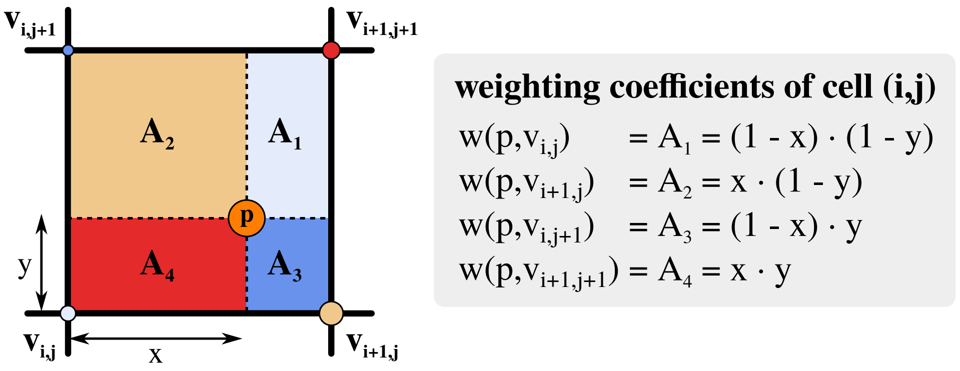

$v_{i,j+1}$
$v_{i+1,j+1}$
A_2
A_1
p
y
A_4
A_3
$v_{i,j}$
x
$v_{i+1,j}$
weighting coefficients of cell (i,j)
$w(p,v_{i,j}) = A_1 = (1 - x) \cdot (1 - y)$
$w(p,v_{i+1,j}) = A_2 = x \cdot (1 - y)$
$w(p,v_{i,j+1}) = A_3 = (1 - x) \cdot y$
$w(p,v_{i+1,j+1}) = A_4 = x \cdot y$

Figure 3.4: Bilinear interpolation. Where $w(p, v)$ is the weighting coefficient for the particle p at the vertex v of a grid cell.

Algorithm 3.3 Scatter phase.

```
   $\rho(v_{i,j})$: charge density at the vertex $v_{i,j}$
   $q_p$: charge of the particle $p$
1: for all particle $p \in Domain$ do
2:     $cell(i,j) \leftarrow$ find grid cell where $p$ is located
3:     $\{v_{i,j}, v_{i+1,j}, v_{i,j+1}, v_{i+1,j+1}\} \leftarrow vertexs(cell(i,j))$
4:     $\rho(v_{i,j}) \leftarrow \rho(v_{i,j}) + q_p \cdot w(p, v_{i,j})$
5:     $\rho(v_{i+1,j}) \leftarrow \rho(v_{i+1,j}) + q_p \cdot w(p, v_{i+1,j})$
6:     $\rho(v_{i,j+1}) \leftarrow \rho(v_{i,j+1}) + q_p \cdot w(p, v_{i,j+1})$
7:     $\rho(v_{i+1,j+1}) \leftarrow \rho(v_{i+1,j+1}) + q_p \cdot w(p, v_{i+1,j+1})$
8: end for
```

3.3.4 Solve Phase

Once charge densities have been deposited on the grid, the electric and magnetic field equations are solved at each grid point using *Poisson's equation* (Eq. 2.9).

For this purpose, several solvers are available depending on the properties of the equations (linearity, dimensionality, variation of coefficients...) and the boundary conditions (periodic, simple, isolated...). Simple equations can be solved using *direct solvers* (e.g. Fast Fourier Transform (FFT)), whereas more complicated equations and boundaries need *iterative solvers* which go through iterations improving the initial guess (e.g. conjugate gradient).

The Poisson's equation is a common Partial Differential Equation (PDE) which can be solved using a direct solver based on the FFT when periodic boundary conditions are involved [42]. In this case, the time spent on this phase is usually not large [23].

3.3.5 Gather Phase

In this step, the fields from the grid are interpolated at the particle locations to compute later the electrostatic force. The value of the fields on the grid points that delimit each cell are weighted at the positions of all interior particles (Algorithm 3.4). The weighting algorithm is similar to the scatter phase but in the opposite direction. This time the field contributions from the grid points are accumulated at the particle position.

Algorithm 3.4 Gather phase.

```
1: for all particle $p \in Domain$ do
2:     $(i, j) \leftarrow$ find grid cell where $p$ is located
3:     $\{v_{i,j}, v_{i+1,j}, v_{i,j+1}, v_{i+1,j+1}\} \leftarrow vertexs(i, j)$
4:     $E(p) \leftarrow E(v_{i,j}) \cdot w(p, v_{i,j}) + E(v_{i+1,j}) \cdot w(p, v_{i+1,j}) +$
5:                $E(v_{i,j+1}) \cdot w(p, v_{i,j+1}) + E(v_{i+1,j+1}) \cdot w(p, v_{i+1,j+1})$
6: end for
```

3.3.6 Push Phase

The force which displaces each particle is calculated by the *Lorenz equation* (Eq. 2.6) in terms of the fields computed at the position of the particle. Particles modify its velocity according to this force and advance to their new positions.

To find the particle trajectories, we integrate particle motion through the time step. The *Euler* method is the most basic of numerical integration techniques. It is accurate if the rate at which the function changes is constant over the time step. But when we integrate the particle velocity to get the position in a plasma, it is not a good approximation because particle velocity is not constant since it is changing over time due to the acceleration caused by forces. In this case, given the same time interval, there are others methods more accurate such as the time-centered Leapfrog method and the Runge-Kutta method (Fig. 3.5).

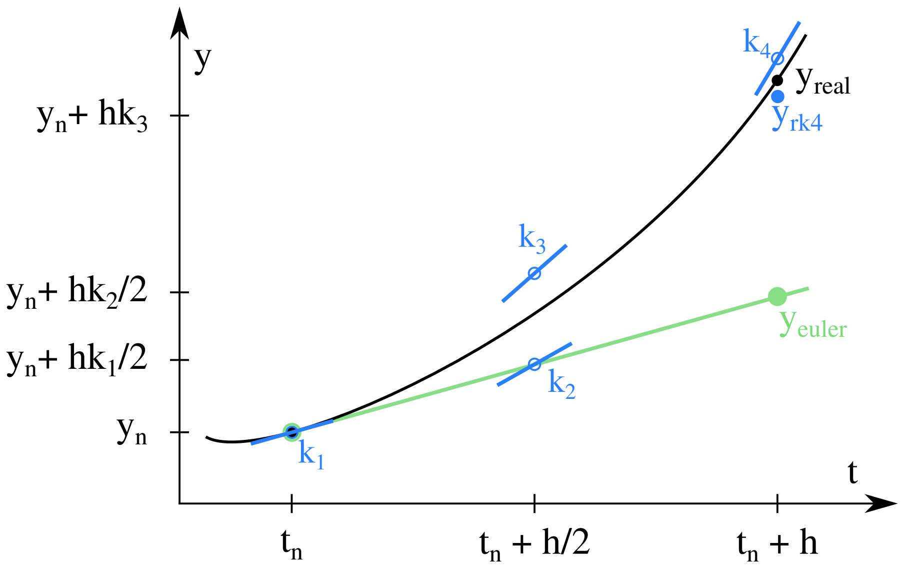


Figure 3.5: Euler vs RK4.

The *Runge-Kutta methods (RKs)* are an important family used in temporal discretization for the approximation of solutions of ordinary differential equations. Essentially, the RKs evaluate the derivative at multiple points along a time step interval to converge to a more accurate solution. Each successive evaluation of the derivative uses the previous derivative as input.

To explain more in detail this last method, we choose a Runge-Kutta fourth order method (RK4) which is stable at large time steps, accurate and relatively fast. It is used to integrate a function with respect to time using its derivative, $y' = f(t, y)$, and with a known initial value $y(t_0) = y_0$

This method calculates four derivatives between the current step and the next:

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \frac{h}{2}, y_n + k_1\frac{h}{2}) \\ k_3 &= f(t_n + \frac{h}{2}, y_n + k_2\frac{h}{2}) \\ k_4 &= f(t_n + \frac{h}{2}, y_n + k_3 h) \end{aligned} \tag{3.1}$$

where h is the time step, t_n is time at the beginning of the last time step and the k-coefficients indicate the slope of the function at three instants in the time step: at the beginning, at the mid-point and at the end. The slope at the midpoint is estimated twice, first using the value of k_1 to calculate k_2 and then using k_2 to compute k_3 .

The next time step value is obtained from the previous time step value as a weighted average of the computed derivatives:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{3.2}$$

Finally, at this phase, RK4 is applied over the following function (coming from Equation 2.6) to find the new particle velocity:

$$\begin{aligned} y' = f(t, y) \quad &\Rightarrow \quad \frac{d\vec{v}}{dt} = \frac{\vec{F}}{m} \\ \vec{v}_{n+1} &= \vec{v}_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{3.3}$$

And as particles move according to Newton's laws of motion, the new position is computed using the resulting velocity:

$$\frac{d\vec{x}}{dt} = \vec{v} \quad \Rightarrow \quad \vec{x}_{n+1} = \vec{x}_n + \vec{v}_{n+1} \cdot h \tag{3.4}$$

Once particles have been moved to its new locations, it should be checked that they are still inside the domain. When a particle leaves the domain, three situations are possible depending on the boundary condition:

- *opened boundary*, the particle is deleted.
- *reflective boundary*, the particle returns to the domain by a elastic rebound.
- *periodic boundary*, the particle returns to the domain by the opposite side

3.3.7 Results Phase

Once the simulation finished, several diagnostics are printed out. Some of them are snapshots at particular times (e.g. charge densities, electron temperature or fields distributions) and others are histories (e.g. energy over time). These results are the data from which physicists deduce the physics of the simulation.

3.4 PIC Codes

As noted in Section 2.5, within the fusion community, a large number of codes are used to simulate various aspects of the plasma behaviour. Among them many codes implement the PIC method and for this reason are called PIC codes. When a PIC code models all the species (ions and electrons) in a plasma then it is called a *full PIC code*.

Some examples of PIC codes are described below.

ELMFIRE

ELMFIRE [31] is a gyrokinetic PIC code for plasmas in toroidal geometry developed at Aalto University (Finland). The code has been built to study the development of instabilities in a quasi-neutral plasma, and its influence on the transport coefficients. It is a single-threaded code and it is mainly written in FORTRAN 90 with some auxiliary C functions.

ELMFIRE has also been applied to numerically investigate transport phenomena in the tokamak, such as ion temperature gradient and trapped electron modes, zonal flows, and neoclassical feedback to turbulence.

GENE

GENE (*Gyrokinetic Electromagnetic Numerical Experiment*) [28] is an open source plasma microturbulence code which can be used to efficiently compute gyroradius-scale fluctuations and the resulting transport coefficients in magnetized fusion/astrophysical plasmas. To this aim, it solves the nonlinear gyrokinetic equations on a fixed grid in five-dimensional phase space (plus time). It was originally developed at Max-Planck-Institut für Plasmaphysik (IPP) in Garching (Germany) and has subsequently extended thanks to several collaborations, such as Centre de Recherches en Physique des Plasmas (CRPP).

GENE is coupled to the geometry interface code GIST and it can deal with arbitrary toroidal geometries (tokamaks or stellarators).

GTC

GTC (*Gyrokinetic Toroidal Code*) [27] is a three-dimensional (3D) parallel PIC code that was originally develop by Zhihong Lin at the Princeton Plasma Physics Laboratory (PPPL) to simulate turbulent transport in fusion plasmas on toroidal magnetic devices. In other words, the aim of GTC is to simulate global influence of microturbulence on the particle and energy confinement in tokamak fusion reactors. It is a relatively small code of about 6,000 lines written in FORTRAN language, parallelized using Message Passing Interface (MPI) and incorporates a full 3D toroidal geometry.

PPPL has a long history of expertise in PIC methods, pioneered by John Dawson in the 1960s and improved over the years by researchers such as H. Okuda and W.W. Lee.

ORB5

ORB5 [4] is a non-linear gyrokinetic global code which solves the Vlasov-Poisson system in the electrostatic and collisionless limit, and has the unique capability of handling true Magnetohydrodynamics (MHD) equilibria. The main purpose of ORB5 is the study of micro-instabilities (e.g. Ion Temperature Gradient (ITG), Trapped Electron Mode (TEM)) and the resulting effects on transport phenomena in a tokamak plasma. It can also run in a neoclassical mode. It was originally written by Scott E. Parker and was further developed by Trach-Minh Tran.

PAR-T

Par-T (*PARallel-Tristan*) [43] is a parallel relativistic fully 3D electromagnetic PIC code designed for rod-shaped geometries, which often occur in astrophysical problems. It is based on the sequential code Tristan and has been parallelized using the idea of domain decomposition and MPI. The code is written in FORTRAN 90 and uses a leapfrog scheme to push particles and a Finite Difference Method (FDM) to update the electromagnetic field.

VORPAL

VORPAL (*Versatile Object-oRiented PlAsma simuLation Code*) [44] is a code originally implemented at the University of Colorado Center and currently developed and marketed by Tech-X Corporation. It is used for modeling vacuum electronics, photonic devices, laser wake-field acceleration, plasma thrusters and fusion plasmas. The application was parallelized using message passing and domain decomposition.

Besides VORPAL includes a fully dynamic runtime load balancer that adjusts load distribution during the simulation run.

VPIC

VPIC [45] is a fully kinetic PIC code developed at Los Alamos National Laboratory (LANL) to simulate Laser-Plasma Instabilities (LPI). It solves the relativistic Vlasov-Maxwell system of equations using an explicit charge-conserving algorithm. It has also been applied to a wide range of problems including laser-plasma interactions, magnetic reconnection in space and laboratory plasmas, particle acceleration and turbulence.

VPIC code was migrated to the Roadrunner supercomputer (a hybrid multi-core platform build by IBM for LANL) where achieved a simulation with over one trillion particles.

EUTERPE

EUTERPE [30] is a parallel gyrokinetic PIC code for global simulations (Fig. 3.6). Its main target is to simulate the microturbulences in fusion plasmas on several 3D geometries and configurations as stellarators (Wendelstein 7-X), tokamaks (ITER) or cylindrical simple ones [46]. It was created at Centre de Recherches en Physique des Plasmas (CRPP) in Switzerland, and has subsequently been further developed at Max-Planck-Institut für Plasmaphysik (IPP) in Germany. It has been parallelized using MPI and it has been ported to several platforms. The Fusion Theory unit of Centro Investigaciones Energéticas, Medioambientales y Tecnológicas (CIEMAT) and Barcelona Supercomputing Center (BSC) in Spain have collaborated with IPP on the development and the exploitation of this code.

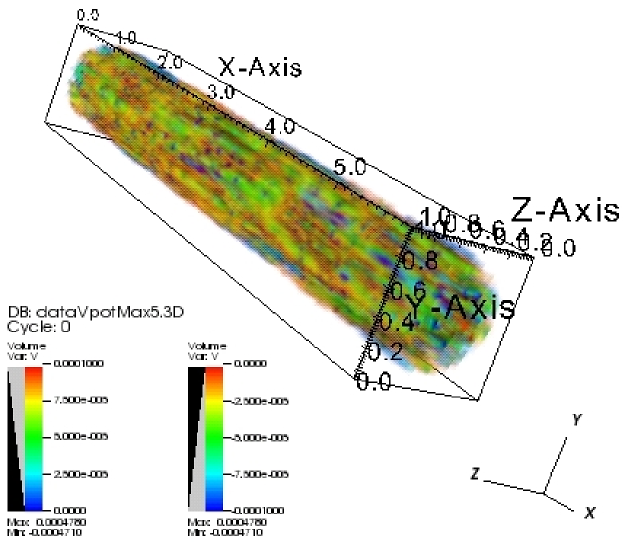


Figure 3.6: Example of a simulation using EUTERPE (3D potential).

This code was our selection at the outset of the thesis because it was in the forefront of plasma simulations and it was a free and open-source software. Additionally, it also provided good results both in linear and non-linear simulations of ITG instabilities [14, 47–50]. In those simulations, especially for the non-linear ones, it became clear that the amount of computational resources that a global three-dimensional PIC code required for typical simulations was huge, and this made EUTERPE a strong candidate for our work of running efficiently on multi-core architectures.

Unfortunately, the conditions of the use of EUTERPE changed some years ago. The authors requested the right to control explicitly who could have access to the code. As a consequence, EUTERPE stopped being open-source, which limited our collaboration.

Chapter 4

Computer Architecture

"Good afternoon, gentlemen. I am a HAL 9000 computer. I became operational at the H.A.L. plant in Urbana, Illinois on the 12th of January 1992. My instructor was Mr. Langley, and he taught me to sing a song. If you'd like to hear it I can sing it for you."

2001: A Space Odyssey
HAL'S SHUTDOWN

This chapter groups the computer architecture concepts that appear in the thesis: multi-core systems (homogeneous and heterogeneous), the list of machines and software tools used to develop the work, and finally the metrics to measure and analyze the performance of an application.

4.1 Introduction

Traditionally, microprocessors have evolved increasing transistor count and clock frequency (Fig. 4.1). The technology scaling trend has been following **Moore's Law**: "*increasing in* *transistor integration capacity, improving transistor performance and reducing transistor integration cost by half every generation*" [51].

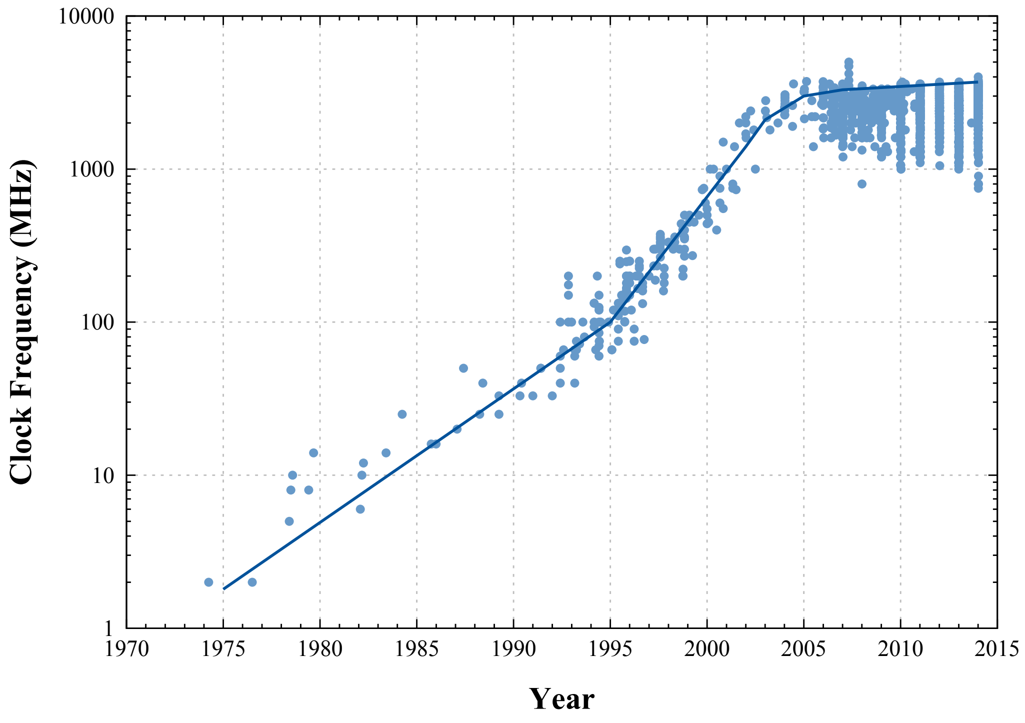


Figure 4.1: Evolution of the CPU clock frequency.

In the past, a great amount of resources were used to improve the performance of single core processors extracting instruction level parallelism: out-of-order execution, register renaming, branch prediction, reorder buffers, etc. As a result, each generation of processors grew faster but dissipated more heat and consumed more power.

Power became the most important barrier to take advantage of technology scaling [52]. The High-Performance Computing (HPC) community began searching an alternative architecture to reduce the increasing power dissipation.

Recently, the designers turned their attention to higher-level parallelism to offer a more efficient use of the power. Adding multiple cores within a processor allows us to run at lower frequencies (as is shown in Figure 4.1 by the year 2004) and get better performance than a single core processor, but adding new problems such as the difficulty in programming to exploit all the performance.

4.2 Multi-core Architecture

The most important trend in contemporary computer architecture is the movement towards processors with multiple cores per socket. This approach is a straightforward technique to solve the problems related to physical constraints (e.g., power, thermal and signalling) and the limited instruction level parallelism.

Thus, it is easy to understand that multi-core technologies are having a deep impact on the HPC world, where supercomputers are strongly limited by power consumption and heat dissipation.

Multi-core processor is a processing system composed of two or more independent cores (or Central Processing Units (CPUs)). The smaller cores are, more power efficient they are [53]. There are a great variety of multi-core implementations which differ depending on the sharing of caches, inter-core communication methods, number of cores and functions of each core.

Multi-core systems can be divided into two groups as shown in Figure 4.2: *homogeneous multi-core* systems where all cores are identical and *heterogeneous multi-core* systems where the cores are not identical.

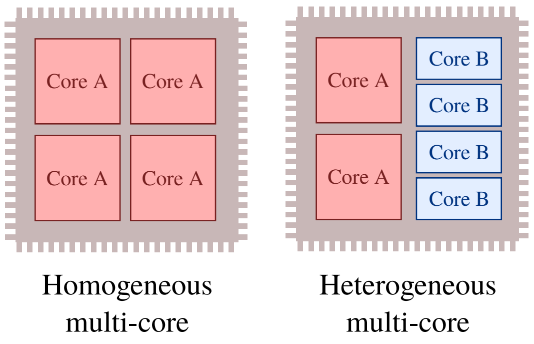


Figure 4.2: Multi-core systems.

When we develop applications in these architectures, it is important to have the following points in mind to reach its full potential, otherwise there will be no gain or even there will be loss:

- Applications need to be written in order that different parts can be run concurrently (without dependencies).

- The cores are associated with relatively small local memories (either caches or explicitly managed memories). To achieve high degree of parallelism, tasks need to be split because they operate on small portions of data in order to reduce bus traffic and improve data locality.
- As the granularity of operations becomes smaller, the presence of synchronization points affects the efficiency of a parallel algorithm. Using asynchronous execution models would be possible to hide the latency of memory accesses.

Examples of this processor trend are AMD Opteron, IBM Power6, Sun Niagara, Intel Montecito, Tilera TILE64 and STI Cell BE.

4.2.1 Homogeneous vs Heterogeneous Architectures

As mentioned before, from the design point of view, homogeneous cores include only identical cores, whereas heterogeneous cores have cores that are not identical since each core may have a different function or a different frequency or a different memory model, etc.

From the complexity point of view, homogeneous architecture seems easier to produce than heterogeneous one, since each core contains the same hardware and instruction set.

On the other hand, heterogeneous architecture is more efficient than homogeneous one, since each core could have a specific function and run its own specialized instruction set. For example, running serial portions of an algorithm on a large, fast and relatively power-inefficient core while executing parallel portions of the algorithm on multiple small power efficient cores [54].

Consequently, the benefits of heterogeneous architecture (heat dissipation, power saving and efficiency) outweigh its complexity.

From the developers point of view, if it is hard to extract reasonable performance from homogeneous multi-core architectures, the situation is even worse with the heterogeneous multi-core architectures. We need to extract explicit parallelism, deal with communications and take care of the way in which computational threads are mapped on the cores.

4.3 Heterogeneous Architectures

Below is a short description of the two heterogeneous chips that we have had access to perform the work for the thesis: Cell BE and Samsung Exynos 5 Dual.

4.3.1 Cell BE

The Cell Broadband Engine [55], or *Cell* as it is more commonly known, was released in 2005 and is an example of a heterogeneous chip multiprocessor. Its processing power of more than 200 GFLOPS and power energy efficiency, which outperformed by a factor of 3-4 times any other HPC platform at that time, demonstrated its tremendous potential [56].

Cell was designed by a partnership of Sony, Toshiba and IBM (STI) to be the heart of Sony's PlayStation 3 video game console. But before long it became an example of a heterogeneous processor employed in scientific computing, e.g the *RoadRunner* project at the Los Alamos National Laboratory (LANL) with a peak performance of 1.4 PFLOPS.

Cell consists of one Power Processing Element (PPE) that manages eight independent Synergistic Processing Elements (SPEs) for the computationally intensive work. The PPE is a conventional dual-threaded 64-bit PowerPC core. In contrast, each SPE is a lightweight core with a simple Single Instruction Multiple Data (SIMD) enabled instruction set, a dual-issue pipeline, no branch prediction, an uniform 128-bit 128 entry register file, a Local Store (LS) and a Memory Flow Controller (MFC). The PPE and SPEs are connected via the Element Interconnect Bus (EIB) providing internal communication. An overview of Cell is shown in Figure 4.3.

The LS is a small local memory (256KB) where each SPE stores its code and data. The transfer data between LS and main memory is supported by a high bandwidth Direct Memory Access (DMA) engine (25 Gbytes/s). This approach allows more efficient use of memory bandwidth than standard prefetch schemes on conventional cache hierarchies (for example, programming double buffering to overlap data movement with computation on the SPEs), but makes the programming model more complex.

Other interesting components of the Cell processor are the Power Management Unit (PMU) and the Thermal Management Unit (TMU). The PMU enables power reduction slowing and pausing the unit and the TMU uses digital thermal sensors to

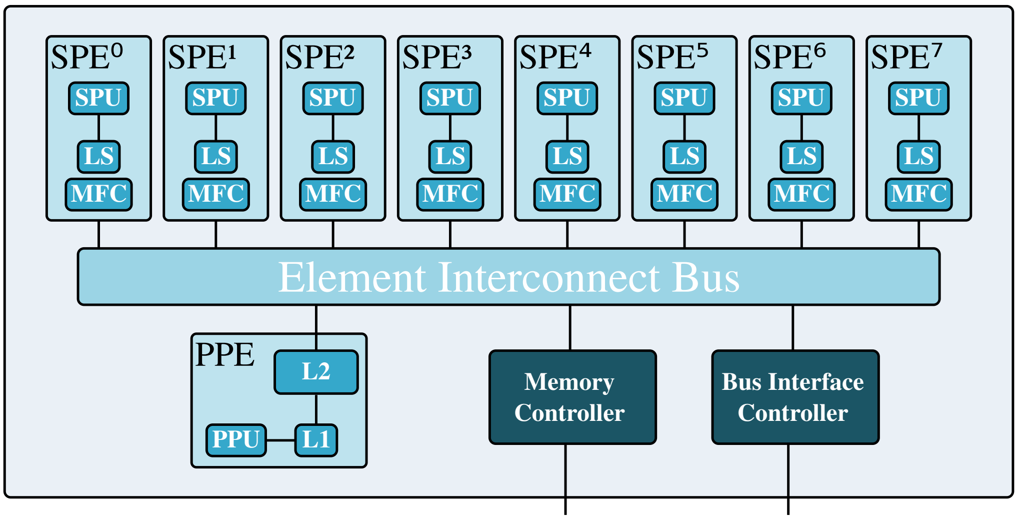


Figure 4.3: Block diagram of a Cell chip.

monitor temperatures throughout the chip. Power and temperature are key points in the design of the Cell processor.

When an application is developed, two different codes have to be programmed: one for the PPE, which handles thread allocation and resource management, and other for the SPE, which computes intensive work.

There are 3 programming models for programming parallelism on Cell [56]:

- **task parallelism** with independent tasks scheduled on each SPE;
- **pipelined parallelism**, where large data blocks are passed from one SPE to the next;
- **data parallelism**, where the processors perform identical computations on distinct data. It is quite similar to parallelization afforded by Open Multi-Processing (OpenMP).

In summary, from the developer point of view, *the impressive computational efficiency of Cell comes with a high price: the difficult programming environment to achieve this efficiency.*

4.3.2 Samsung Exynos 5 Dual

Samsung Exynos 5 Dual (Fig. 4.4) is a System on Chip (SoC) which contains an ARM Cortex-A15 dual core at 1.7 GHz and a quad-core ARM Mali T604 Graphic Processing Unit (GPU). It was released in 2012 and was the first embedded SoC that had enough potential for HPC, since it supports 64-bit floating point arithmetic and provides support for parallel programming languages (such as Open Computing Language (OpenCL) v1.1).

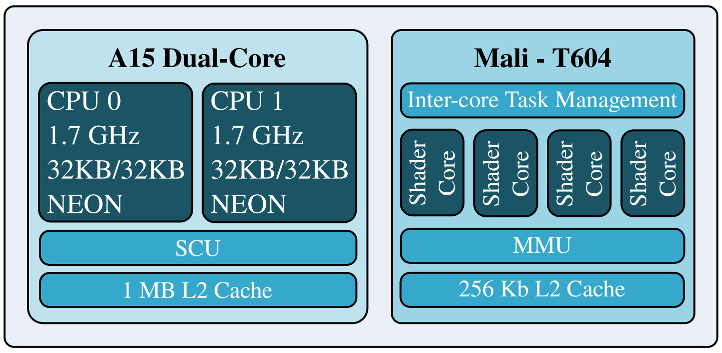


Figure 4.4: Samsung Exynos architecture.

ARM (Acorn RISC Machines) is a British company which designs a Reduced Instruction Set Computer (RISC) architecture. This kind of architectures reduces costs, heat and power use, which is desirable for embedded systems, such as smartphones and tablets. Moreover, its simple design (Fig. 4.5) provides efficient multi-core CPUs with large number of cores at low cost and an improved energy efficiency for HPC. Although ARM processors do not provide a sufficient level of performance for HPC yet, it is worthwhile to explore their potential considering ARM has a promising roadmap ahead.

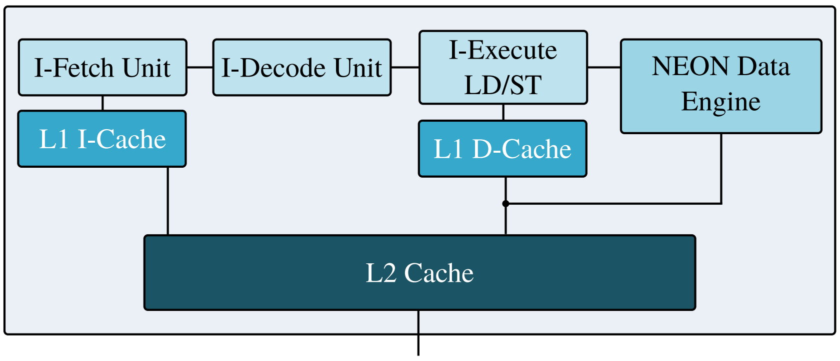


Figure 4.5: Block diagram of an ARM Cortex chip.

4.4 Mont-Blanc Project

Transitions in the HPC world are not casual facts. The highest-volume commodity market (desktop computers) tends to drive lower-volume HPC market, because the design of an HPC chip has to be amortized selling the maximum number of units to minimize its final price.

For example, Figure 4.6 shows how systems based on special-purpose HPC systems were replaced by RISC microprocessors in the TOP500 list [57] in the 1990s. This transition took place due to the cheapness of the RISC microprocessors although they were slower.

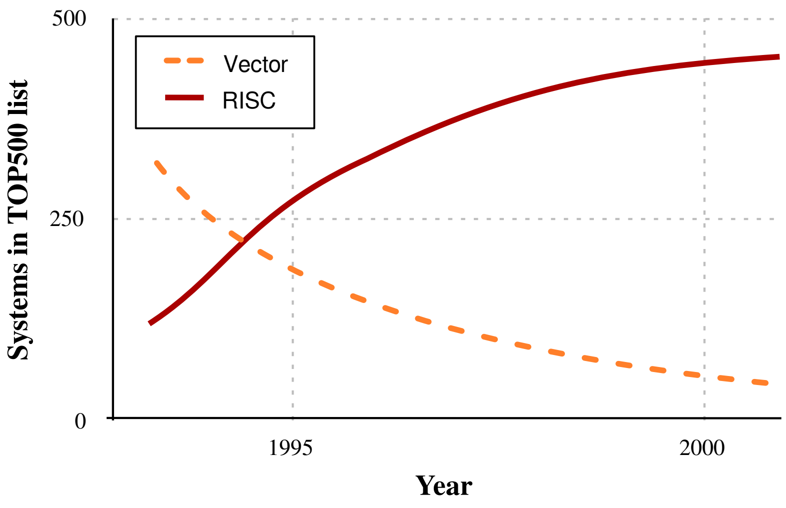


Figure 4.6: Number of systems in TOP500.

Currently, as shown in Figure 4.7, we observe a similar trend: low-power microprocessors (used in mobile devices) are improving their performance and are including features needed to run HPC applications (as an on-chip floating-point unit). It is reasonable to consider that the same market forces that replaced vector processors by RISC processors will be able to replace present HPC processors with mobile processors [58].

Mont-Blanc is an European Exascale computing approach to develop a full energy-efficient HPC prototype. The aim of this project is to develop a prototype using low-power commercially available embedded technology to exploit the large volume of these platforms and their high accessibility in the commodity market. The project is coordinated by

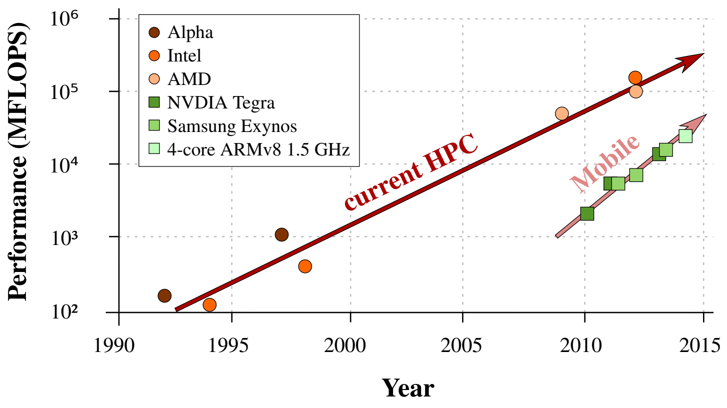


Figure 4.7: Peak performance in floating double-precision calculations.

Barcelona Supercomputing Center (BSC) since October 2011 and it is an European consortium comprised of:

- Industrial technology providers: Allinea (UK), ARM (UK) and BULL (France).
- Research and supercomputing centres: BSC (Spain), CINECA (Italy), CEA (France), CNRS (France), GENCI (France), HLRS (Germany), INRIA (France), JUELICH (Germany), LRZ (Germany), University of Bristol (UK) and Universidad de Cantabria (Spain).

4.4.1 Prototypes

The Mont-Blanc project has deployed several ARM-based prototypes, as Figure 4.8 shows, with the aim of integrating mobile technology in a fully functional HPC prototype. The knowledge gained in their developments contributed towards the deployment of the final prototype.

The first prototype from the Mont-Blanc project was called *Tibidabo* and came out in November 2011. It was based on NVIDIA Fermi GPUs and NVIDIA Tegra 3 processors and it was a proof of concept to demonstrate that building such energy-efficient clusters with mobile processors was possible.

In 2012, a new prototype called *Carma* was built to explore the combination of ARM processor with external mobile NVIDIA GPU. It was a cluster with 16 nodes and each node was powered by a NVIDIA Tegra 3 quad-core ARM Cortex-A9 processor plus a NVIDIA Quadro 1000M mobile GPU (with 96 CUDA cores). The amount of memory per

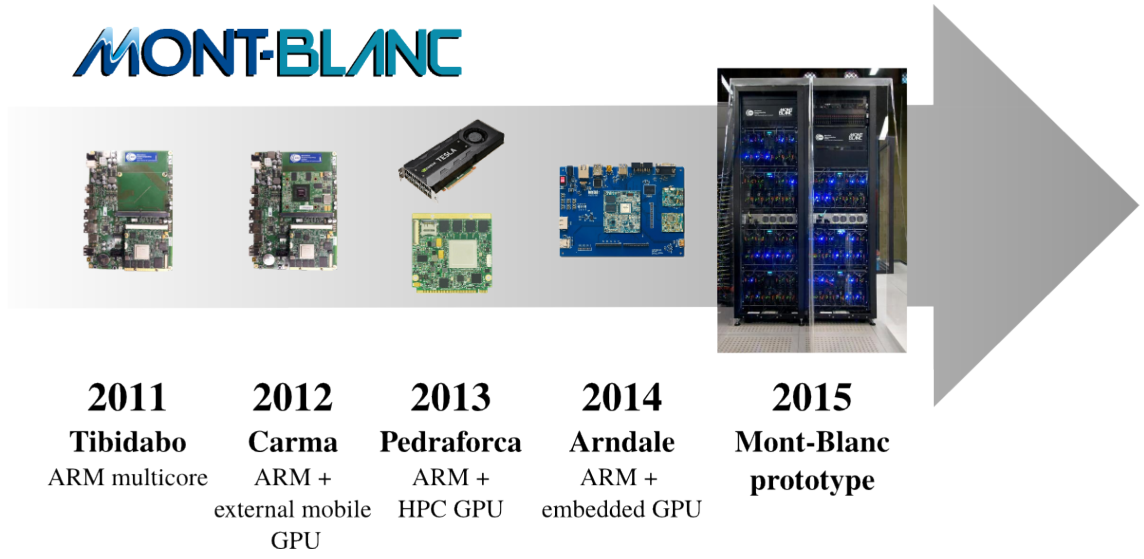


Figure 4.8: Roadmap of the Mont-Blanc prototypes.

node was 4 GBytes (2GBytes (CPU) + 2GBytes (GPU)). The prototype could achieve 270 GFLOPS in Single Precision computation, in other words 5 GFLOPS/Watt.

The next prototype was *Pedraforca*. It was developed in 2013 as an experiment to test the performance, scalability, and energy efficiency of ARM multi-core processors using high-end GPU accelerators. It consisted of 3 BullX-1200 racks with 70 compute nodes with a quad-core CPU Cortex-A9 at 1.4 GHz, a GPU Nvidia Tesla K20 card and 4 GBytes DDR3 RAM. We can consider that *Pedraforca* was the first large HPC system based on ARM architecture.

The next step was to test the building block for powering the final prototype using a small cluster. It was the stepping-stone towards a full system. *Arndale* came out in 2014 and was based on the Samsung Exynos 5 Dual SoC. It had three nodes, each one equipped with a dual-core ARM Cortex-A15 at 1.7 GHz and an integrated ARM Mali-T604 GPU.

The final *Mont-Blanc prototype* has come out in 2015 and consists of 2 racks of blade servers that host 8 BullX B505 blade server carriers. Each carrier contains 9 blades and each blade has 15 compute nodes and a microSD slot of local flash storage. Each node integrates a Samsung Exynos 5 Dual chip and 4 GBytes of DDR3 memory running at 1.6 GHz. In short, the prototype has 1,080 compute nodes that can deliver 34.7 TFLOPS of computing with a 24 kW power draw. This corresponds to about 1.5 GFLOPS per watt, which is a factor of ten better than the first prototype.

4.5 Computers

This section gives an overview of the different machines used for this thesis.

The thesis has been developed in several spaced steps over a long period of time, which implies that a wide range of platforms have been used. But, on the other hand, several machines used in some works either have been retired or the authorization to access them has ended before we could continue the work on them. For this reason, in the following chapters there will not be a comparison among all listed machines since they have been used at different times.

Arndale

As mentioned above, *Arndale* is a prototype formed by three ARM-based nodes. Each of them contains a Samsung Exynos 5 with 2 GBytes of RAM. The nodes are interconnected using a 100 Mb Ethernet network. This machine is at BSC.

Huygens

Huygens is an IBM pSeries 575 system. It consists of 104 nodes, 16 dual core processors (IBM Power6, 4.7 GHz) per node and either 128 GBytes or 256 GBytes of memory per node. The total peak performance is 60 TFLOPS. It is located at Stichting Academisch Rekencentrum Amsterdam (SARA).

Jugene

Jugene is a Blue Gene/P system. It consists of 73,728 nodes, 4 core processors (32-bit PowerPC 450 at 850 MHz) per node and 2 GBytes of memory per node. The total peak performance is about 1 PFLOPS. The supercomputer is situated at Forschungszentrum Jülich (FZJ).

MareNostrum II

MareNostrum II was a cluster of IBM JS21 server blades. Each node had 2 PowerPC 970MP processors (2-Core) at 2.3 GHz with 8 GBytes of RAM. The nodes were connected using a Myrinet network. The peak performance was 94 TFLOPS. This supercomputer was situated at BSC.

MareNostrum III

MareNostrum III consists of 3,056 JS21 computing nodes. Each node contains 2 Intel SandyBridg-EP E5-2670 processors (8-Core) at 2.6 GHz with 32 GBytes of RAM. The nodes are connected using an InfiniBand network. The peak performance is 1.1 PFLOPS. It is located at BSC.

MariCel

MariCel was a prototype composed of 72 IBM Blade-Center QS22 nodes. Each node contained 2 PowerXCell 8i processors at 3.2 GHz with 32 GBytes of RAM. The nodes were connected using an InfiniBand network. The peak performance was 15.6 TFLOPS and the energy efficiency was 500 MFLOPS/Watt.

MinoTauro

MinoTauro is a NVIDIA GPU cluster with 128 Bull B505 blades. Each node contains 2 Intel Xeon E5649 (6-Core) at 2.53 GHz and 2 M2090 NVIDIA GPU Cards with 24 GBytes of RAM. The peak performance is 185.78 TFLOPS.

4.6 Software Tools

The following tools developed at BSC have been used in this thesis for the performance analysis:

EXTRAE

Extrae [59] is a performance extraction tool to trace programs. It gathers data like hardware performance counters and source code references to generate trace files, which are as a log of application activity.

Extrae supports the following programing models among others: OpenMP, Message Passing Interface (MPI), OpenCL, Compute Unified Device Architecture (CUDA), OpenMP Super-scalar (OmpSs) and their corresponding combinations with MPI.

To instrument an execution, it is only necessary to submit a script which defines some environment variables and calls the user application. The instrumentation is configured by an XML file where one can specify the hardware counters, level of detail in the tracing, how to handle intermediate files, etc. After the execution, one file with the collected performance metrics is generated by each process. A merger tool matches and merges the different files into a single one.

The basic mechanism to get traces relies on preloading a library into an application before it is loaded. This library provides the same symbols as the shared library but it injects code in the calls to generate the trace. In Linux systems this technique is based on the `LD_PRELOAD` environment variable.

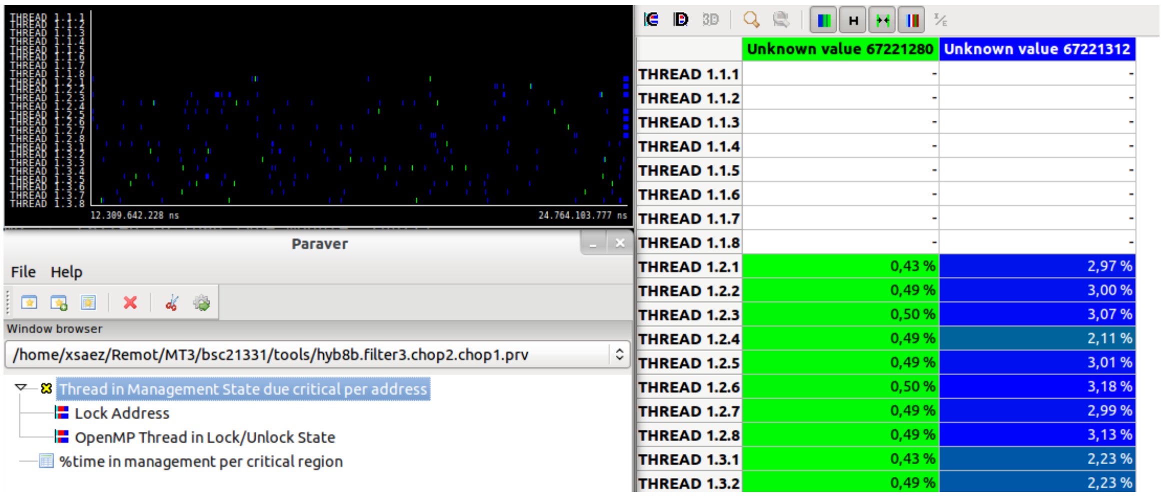
However, if this mechanism does not fulfill the user's needs it is possible manually instrumenting the application with own Extrae events.

PARAVER

Paraver [60] (PARAllel Visualization and Events Representation) is a tool to visualize and analyze trace files. From the events and counters collected in the trace file, Paraver (Fig. 4.9) can display timelines of a wide range of metrics (such as duration, FLoating-Point Operations Per Second (FLOPS), Instructions Per Second (IPS), Instructions Per Cycle (IPC), bandwidth, MPI calls, user functions...) and statistics of these metrics for any interval of the trace (duration, hardware counts, derived metrics, . . .).

A performance analysis using Paraver consists of the following steps:

- Generate a summarized trace that allows us to watch all the trace. Normally, all communications and the smallest computation bursts are discarded.
- Identify repetitive structures in the trace and extract a couple of repetitions of these structures from the original trace to get a new subtrace.
- This representative subtrace will be analysed carefully to identify regions with specific behaviour and understand the performance problems in the application.



/home/xsaez/Remot/MT3/bsc21331/tools/hyb8b.filter3.chop2.chop1.prv

- Thread in Management State due critical per address
  - Lock Address
  - OpenMP Thread in Lock/Unlock State
- %time in management per critical region

	Unknown value 67221280	Unknown value 67221312
THREAD 1.1.1	-	-
THREAD 1.1.2	-	-
THREAD 1.1.3	-	-
THREAD 1.1.4	-	-
THREAD 1.1.5	-	-
THREAD 1.1.6	-	-
THREAD 1.1.7	-	-
THREAD 1.1.8	-	-
THREAD 1.2.1	0,43 %	2,97 %
THREAD 1.2.2	0,49 %	3,00 %
THREAD 1.2.3	0,50 %	3,07 %
THREAD 1.2.4	0,49 %	2,11 %
THREAD 1.2.5	0,49 %	3,01 %
THREAD 1.2.6	0,50 %	3,18 %
THREAD 1.2.7	0,49 %	2,99 %
THREAD 1.2.8	0,49 %	3,13 %
THREAD 1.3.1	0,43 %	2,23 %
THREAD 1.3.2	0,49 %	2,23 %

Figure 4.9: A sample of the Paraver interface.

The main advantages of this tool are the simple interface for exploring the collected data and the great amount of metrics and profiles that are available to use. The main disadvantages are the long learning curve needed to exploit the tool (sometimes user can get confused) and the tedious process to get traces and select the correct regions.

GPROF

Gprof [61] is a performance analysis tool for applications which uses a hybrid of instrumentation and sampling. It was created as an extended version of the older *prof* tool.

The compiler inserts instrumentation code into the application when it is called with the suitable flag (e.g. `-pg` for GCC compiler). This instrumentation code gather caller-function data that is saved in the profile file `gmon.out` just before the application finishes.

Once the program finishes the run, we can analyze the profile file calling `gprof {binary} gmount.out >profile.txt`. It produces a flat profile (Fig. 4.10) on the standard output that can be divided in two parts:

- The first part displays how much time the program spent in each function, and how many times that function was called. This list is sorted by the execution time, so the first lines point out the candidate functions to be hot spots.

- The second part is the call graph, which shows for each function who called it (parent) and who it called (child subroutines). There is an external tool called *gprof2dot* that is capable of converting this information to a graphical form.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
31.50   1286.46  1286.46  800000000   0.00     0.00  fields_mp_getfield_
25.45   2325.87  1039.42 1604000000   0.00     0.00  equil_mp_equil_xy2sc_ra_
14.63   2923.31   597.44        804   0.74     1.48  part_ompccassign_mp_quad_ass_
11.74   3402.83   479.52 8716363512   0.00     0.00  equil_mp_equil_grad_
 6.24   3657.75   254.92        800   0.32     3.17  part_omp_mp_push_
 2.19   3747.03    89.28 1314735184   0.00     0.00  equil_mp_equil_xy2sc_r_
 1.07   3790.56    43.53                             floor.N
 0.92   3828.03    37.47        404   0.09     0.16  part_mp_cache_sort_
 0.85   3862.91    34.88       2000   0.02     0.02  control_all_mp_pic_copy_
 0.58   3886.44    23.53                             exp.L
 0.54   3908.42    21.98                             MatSolve_SeqSBAIJ_1_NaturalOrdering
 0.53   3930.10    21.68          4   5.42     8.70  loadtor_mp_load_r_z_
```

Figure 4.10: A sample of Gprof tool.

4.7 Measuring Performance

The purpose to introduce levels of parallelism in a program is to reduce its execution time. But we find that not all programs are alike when one wants to parallelize, so it is important to know which parts of the code are better to parallelize by understanding its costs and benefits.

For this reason, we need a way of measuring the quality of the parallelization. One of the most common metrics for parallel programing is the *speedup*.

4.7.1 Speedup

Speedup is the ratio between the serial execution time, $T(1)$, and the parallel execution time on n processors, $T(n)$:

$$S(n) = \frac{T(1)}{T(n)} \tag{4.1}$$

The speedup shows us the scalability of a program, i.e. the ability of a parallel program to scale its performance as more processors or Processing Elements (PEs) are added.

When analysing the scalability of a program, one may find the following types of speedup curve (Fig. 4.11):

- **Linear (or ideal) speedup** is obtained when $S(n) = n$. It means that the program runs n times faster on n processors than on a single processor. It is the theoretical limit and thus is considered very good scalability.
- **Super-linear speedup** is when $S(n) > n$. It means that the parallel program can achieve a speedup greater than linear. It happens rarely and one possible reason that this occurs may be the cache effect resulting from the different memory hierarchies which can significantly reduce data access times.
- **Sub-linear speedup** is when $S(n) < n$, or in other words, when adding more processors improves performance, but not as fast as linear. This is the common speedup observed in many parallel programs.
- **Retrograde speedup** is when $S(n)$ drops after it has reached the maximum speedup. It happens when adding processors does not improve the execution time since the sequential code dominates the execution time.

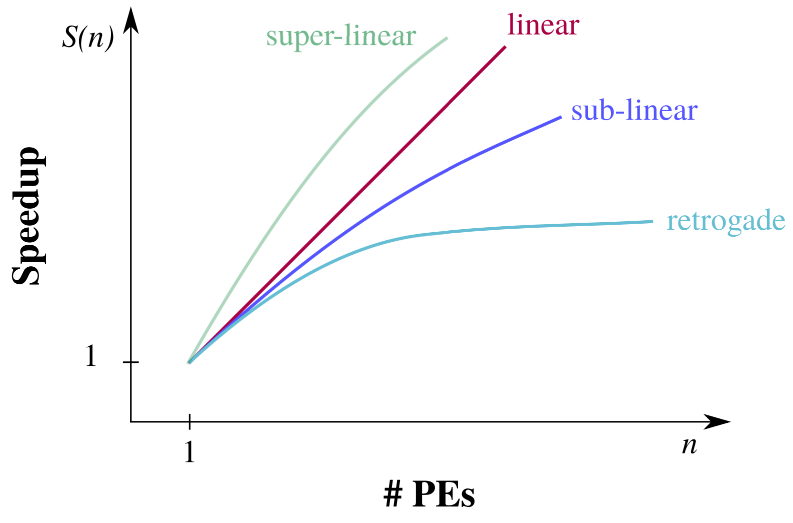


Figure 4.11: Types of speedup.

4.7.2 Strong and Weak Scalability

In the field of HPC, there are two basic ways to approach the scalability of a program:

- **Strong scaling**, which is defined as how the solution time varies with the number of processors for a fixed total problem size.
- **Weak scaling**, which is defined as how the solution time varies with the number of processors for a fixed problem size per processor.

It is important to mention that *in this work our choice has been to study the strong scalability*. It means that hereafter the speedups are computed based on simulations where the number of the particles and the grid size stay fixed while the number of processors increases.

In general, it is difficult to achieve a linear speedup with a large number of processors when one studies the strong-scaling, because usually the communication overheads increase in proportion to the number of processors used.

4.7.3 Amdahl's Law

Amdahl's Law was formulated by Gene Amdahl in 1960 and originally it was focused towards processor architectures, but it can be perfectly applied to parallel programming. Basically, it states that the performance improvement gained from parallelizing a program is limited by the portion of the program which can not be parallelized.

In other words, *Amdahl's Law* computes the theoretical maximum performance improvement (or speedup) that a parallel program can achieve. Therefore, the theoretical maximum speedup $S_{max}(n)$ that can be achieved by executing a given algorithm on n processors is:

$$S_{max}(n) = \frac{T\,(1)}{T\,(n)} = \frac{T\,(1)}{T\,(1)\left((1-P)+\frac{1}{n}P\right)} = \frac{1}{(1-P)+\frac{P}{n}} \tag{4.2}$$

where $P \in [0,1]$ is the proportion of the program that has been parallelized and $(1-P)$ is the proportion that cannot be parallelized (i.e. it remains sequential).

Figure 4.12 illustrates the Amdahl's law. For example, if P is 0.75, then $(1 - P)$ is 0.25, and the program can be speed up by a maximum of a factor of 4, no matter how many processors are used. But, if we increase the proportion of parallelization to 0.9 then the program can speed up by a maximum of a factor of 10. For this reason, our focus in the parallel programming consists in identifying the maximum portion of code that can be parallelized to improve its speedup and thus fully exploit the hardware.

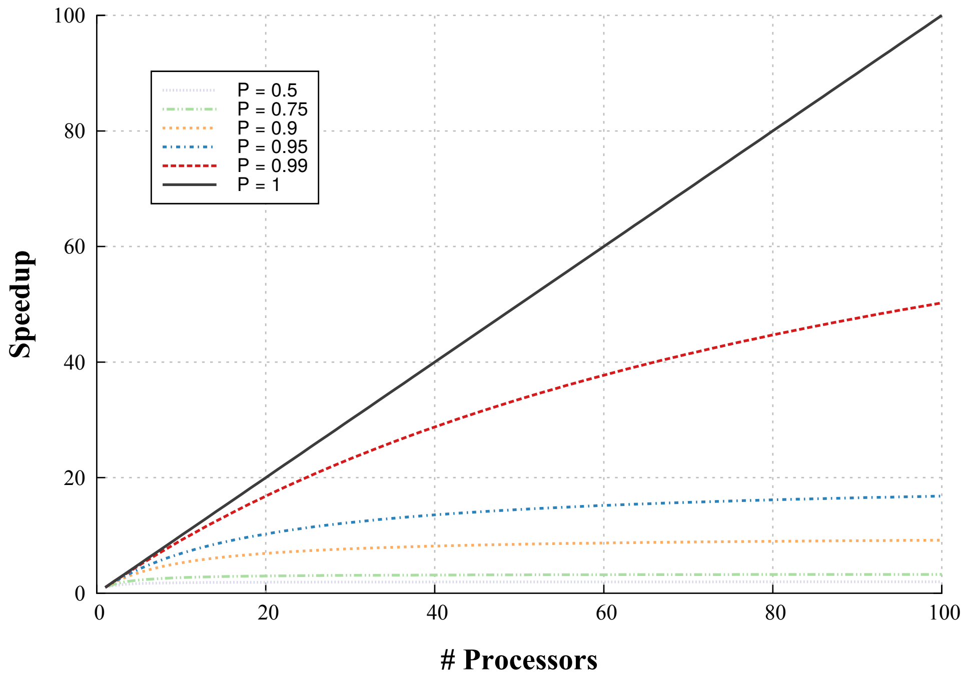


Figure 4.12: Theoretical maximum speedups depending on the fraction of the program that can be parallelized (P).

Chapter 5

EUTERPE

"Euterpe, the muse of music and lyric poetry. She loved flute playing, and some even say she invented the double flute. Euterpe had a son named Rhesus, who was killed in the battle at Troy, according to Homer's Iliad."

WIKIPEDIA.ORG

This chapter gives an overview of the Particle-In-Cell (PIC) code called EUTERPE and presents a parametric study developed in this code to evaluate the influence of the parameters on the quality of the simulation. The optimization task may take advantage of the modification of simulation parameters in order to reduce the execution time without affecting significantly the quality of the results.

The work done in this chapter resulted in the following publication:

[15] **X. Sáez, A. Soba, E. Sánchez, R. Kleiber, F. Castejón and J. M. Cela**, Improvements of the particle-in-cell code EUTERPE for Petascaling machines, *Computer Physics Communications* **182**, 2,047 (2011)

5.1 Description

EUTERPE is a gyrokinetic PIC code that can simulate up to three kinetic species: ions, electrons and a third species with any charge and mass.

The evolution of the distribution function of each kinetic species (f_α) is given by the gyrokinetic equation (Eq. 2.12) over an electrostatic fixed equilibrium, so it does not consider collisions among particles ($C(f_\alpha) = 0$):

$$\frac{\partial f_\alpha}{\partial t} + \frac{\mathrm{d}v_\parallel}{\mathrm{d}t}\frac{\partial f_\alpha}{\partial v_\parallel} + \frac{\mathrm{d}\vec{R}}{\mathrm{d}t}\frac{\partial f_\alpha}{\partial \vec{R}} = C(f_\alpha) = 0 \tag{5.1}$$

where $v_\parallel$ is the velocity parallel to the magnetic field line and $\vec{R}$ is the position vector of the guiding center of the charged rings. The evolution in time of $v_\parallel$ and $\vec{R}$ is given by the non linear equations in the electrostatic approximation used in this work:

$$\frac{\mathrm{d}\vec{R}}{\mathrm{d}t} = v_\parallel \vec{b} + \frac{\mu B + v_\parallel^2}{B\Omega_i}\vec{b} \times \nabla B + \frac{v_\parallel^2}{B\Omega_i}(\nabla \times B)_\perp - \frac{\nabla\langle\phi\rangle}{B} \times \vec{b} \tag{5.2}$$

$$\frac{\mathrm{d}v_\parallel}{\mathrm{d}t} = -\mu\left[\vec{b} + \frac{v_\parallel}{B\Omega_i}(\nabla \times B)_\perp\right]\nabla B - \frac{q_i}{m_i}\left(\vec{b} + \frac{v_\parallel}{B\Omega_i}[\vec{b} \times \nabla B + (\nabla \times B)_\perp]\right)\nabla\langle\phi\rangle \tag{5.3}$$

$$\frac{\mathrm{d}\mu}{\mathrm{d}t} = 0, \tag{5.4}$$

where μ is the magnetic moment per unit mass (which is a constant of motion), q_i and m_i are the ion charge and mass respectively, $\Omega_i = \frac{q_i B}{m_i}$ is the ion cyclotron frequency, $\frac{\vec{b}\nabla B}{B}$ is the unit vector in the magnetic field B direction and $\langle\phi\rangle$ is the renormalized potential introduced in [62].

The distribution function f_α is discretized using *macroparticles* (henceforth *markers* or *particles*) and the electrostatic potential is represented on a spatial grid. At first, particles are randomly distributed in the full grid. Each particle contributes in a part of the distribution function whose evolution is given by the above gyrokinetic Vlasov equation (Eq. 2.12).

The δf approximation is used in the code to reduce the discretization noise: the distribution function is separated into an equilibrium part (Maxwellian) and a time-dependent perturbation. Only the evolution of the perturbation is followed, which allows us to reduce the noise and the required resources, in comparison with the alternative of simulating the evolution of the full distribution function.

In the electrostatic approximation, the external magnetic field is considered fixed during the simulation and it is generated using the Variational Moments Equilibrium Code (VMEC) [63] at the beginning.

Moreover, the system of equations (5.2-5.4) is also complemented with the Poisson equation from [62], which can be simplified by neglecting high order terms and by making the long-wavelength approximation, resulting in the following quasi-neutrality equation:

$$\langle n_i \rangle - n_0 = \frac{e n_0 (\phi - \bar{\phi})}{T_e} - \frac{m_i}{q_i} \nabla \left(\frac{n_0}{B^2} \nabla_\perp \phi \right) \tag{5.5}$$

where n_0 is the equilibrium density, e is the elementary charge, T_e is the electron temperature, $\langle n_i \rangle$ is the gyroaveraged ion density and $\bar{\phi}$ is the flux surface averaged potential.

The evolution of electric and magnetic internal fields is given by the Poisson equation and the Ampere's law (in the electromagnetic version) where the charge and current densities are calculated from the gyro-ring average using a charge assignment procedure. The charge density is discretized in the grid, and finite elements techniques like B-splines [64] are used to estimate it at the grid points from the charge in the vicinity.

The electric potential at each grid point is obtained by solving the quasi-neutrality equation using the Portable, Extensible Toolkit for Scientific Computation (PETSc) library [65]. The electric field is obtained from the potential values at the grid points.

Once fields have been calculated, particles interact with these fields and are moved according to the gyrokinetic equations. The equations of motion for particles are integrated in time using the explicit Runge-Kutta fourth order method (RK4).

The code uses two coordinate systems: a system of magnetic coordinates (s, θ, ϕ) for the potential, and a system of cylindrical coordinates (r, z, ϕ) for pushing the particles (where $s = \Psi/\Psi_0$ is the normalized toroidal flux). The coordinate transformation between both coordinate systems is facilitated by the existence of the common coordinate (ϕ).

In general, global non-linear PIC simulations of turbulent field structures are very demanding with respect to numerical methods and computational effort. This is due to the fact that the five-dimensional distribution function (three in space coordinates and two in velocities) of the ion guiding centers has to be sampled. In addition, the unavoidable statistical noise at the charge-assignment has to be reduced to a minimum. One strategy against this inherent problem of noise production in PIC codes is to use Fourier filters (FFT) to suppress the statistical noise.

5.2 Implementation Details

The first version of EUTERPE that became available to us was 2.51. It consisted of 21 files written in FORTRAN 90 and it also contained the VMEC code to generate the electrostatic fixed equilibrium. EUTERPE is rather well documented and this has allowed us to understand it without complications. The main part of the thesis work was performed with version 2.61 which arrived some time later.

EUTERPE requires the following external libraries in order to compile: a library with Message Passing Interface (MPI) implementation such as MPICH [66], a FFT library such as FFTW [67] and a library for solving sparse linear systems of equation such as PETSc.

EUTERPE performs simulations in two executions. The finite element matrix, which follows from the discretization of the potential equation, is time-independent, so it is assembled in the first run and is stored in files for later use. The next executions load these files to continue the simulation. As long as the equilibrium and grid do not change it can be used for several simulations. This system facilitates the reutilization of the matrix and avoids repeating the same computations.

The parameters needed for the simulation are introduced by an input file called `input`. Unfortunately, EUTERPE is not backward compatible, i.e. an input file of a previous version may not work in a modern version of the program. This fact has become a major complication and we have had to adapt the input for rerunning some tests.

Listing 5.1 Example of an input file for EUTERPE version 2.54

```
&BASIC
 job_time=240000., extra_time=1500.,
 nrun=10,
 dt=20.,
  clones=4,
  nlres=f,
  timescheme=1,
/
&SOLVER
  iquafor = 2,
  model = 2,
  errbnd=1.e-5, lmat=2,
/
&FIELDS
  ns=32, nchi=512 nphi=512,
  mfilt=128, nfilt=16,
  nidbas =2,
/
&EQUIL
  lx=537.313,
/
&IONS
  nptot=1000000000, loadv=2,
  pertw = 1.e-3,
  kappan0=0.0, kappat0=3.5, speak=0.5, widths=0.2,
/
&DIAG
  nfreq=5,
/
```

Moving on, EUTERPE mixes serial and parallel accesses to its data files. The initial electrostatic equilibrium is distributed into files which are read by the corresponding parallel processes. The finite element matrix is also divided into a file per process. Finally, the files with the results are sequential and the master process is the only one that writes into them.

So the Input/Output (I/O) activity of EUTERPE can be summarized in: an initialization phase, periodic updates of histogram and "restart" files, periodic storing of partial diagnostic information and the writing of final result files.

Finally, the first received version of EUTERPE had already been optimized at the single processor level, for example: by adding a cache sort algorithm to sort particles according to the grid cell they belong to, or by changing the order of dimensions in arrays to help their access inside the loops. After some analysis, we decided to trust the outcome of this work and look for other possible improvements of performance through the parallelization.

5.3 Reference Test

In order to evaluate the developed versions we defined a cylindrical geometry as the point of reference because it is a typical scenario of Ion Temperature Gradients (ITGs). The studied test (Fig. 5.1) corresponds to a θ-pinch with radius $a = 0.55$ m, length $l = 2\pi R_0$ with $R_0 = 5.55$ m and a fixed homogeneous magnetic field along the axis of the cylinder.

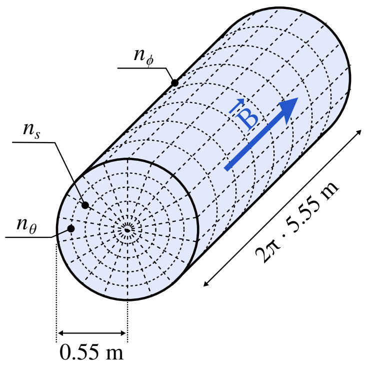


Figure 5.1: Geometry of the reference test.

The resolution of the grid used in the simulations is $n_s \times n_\theta \times n_\phi$ and their dimensions will vary according to the study to do. Normally, the more processors are used, the more cells are required. Similarly, the number of particles will depend on the amount of work required to perform the test.

5.4 Parametric Analysis

Another way to optimize the simulations is to analyze the relation between input parameters and the quality of the results given by the code. This study, besides helping us to understand better the code, can give us an idea about how to modify the input parameters to reduce the amount of work to perform by the code without worsening the quality of the results. Therefore, it is a valid option for reducing the time of a simulation.

Obviously, there is a limit in the modification of input parameters (such as time step size, number of particles and grid size) since the increase of numerical noise in the results can cause the results diverge too much from reality. One of this critical points is related to the amount of particles and the time step used in the integration of the equations of motion.

In this way, we decided to perform a parameter analysis for time step, grid size and number of particles to clarify their close relation and to help in the planning of the best way to reduce the simulation needs.

In a previous work [14], EUTERPE provided good results both in linear and non-linear simulations of ITG instabilities. The comparison with results obtained with TORB (a variant of the ORB5 code) in screw-pinch geometry [68] showed that the time step could be increased from 1 to 20 times the ion cyclotron frequency without altering the results. The saturation of energies, heat flux and the structures that appeared in the potential during the non-linear phase were quite similar in both cases. This is very important because it means that similar results can be obtained with less CPU time.

5.4.1 Dependence on Time Step Size

Firstly we made a deeper study of the dependence of the simulation quality on the time step used in the integration of the equations of motion. A RK4 was used in our simulations. Here we addressed the question about how the time step used for time integration depends on the size of the problem and on the grid used to discretize the domain.

In order to study its influence on the results, several non-linear electrostatic simulations of ITG in a screw-pinch with $\iota/2\pi = 0.8$ were performed on *MareNostrum II* using different combinations of time steps, numbers of particles and grids. The time steps were $\Delta t/\Omega_i = 5, 10, 20, 40, 80, 120, 160, 200$ and 300, where $\Omega_i = 1.2 \cdot 10^8 s^{-1}$ is the ion cyclotron frequency. The number of particles were $N = 16, 32, 64$ and 256 million particles and the spatial grids were $100 \times 64 \times 64$, $100 \times 128 \times 128$ and $100 \times 256 \times 256$.

The Signal-to-Noise ratio (S/N) is used as a measure of the quality of the simulation. It is defined as the ratio of the spectral power kept inside the filter to the spectral power filtered out. In EUTERPE, a filter on the density in Fourier space is always used to reduce particle noise. For the simulations we used a diagonal filter along the n/m $= \iota/2\pi$ line with a width of $\Delta m = 4$. The squared filter limit is always set to $\frac{2}{3}$ of the Nyquist frequency. The *Nyquist frequency*, named after electronic engineer Harry Nyquist, is half of the sampling rate of a discrete signal processing system, that is to say, it is the highest frequency that the sampled signal at a given sampling rate can unambiguously represent.

In Figure 5.2 one can see the S/N for several test cases with different number of particles and time steps. In all the cases the grid had $100 \times 64 \times 64$ grid points. The

S/N does not show any dependency on the time step and the numerical noise decreases when the number of particles increases for a fixed grid.

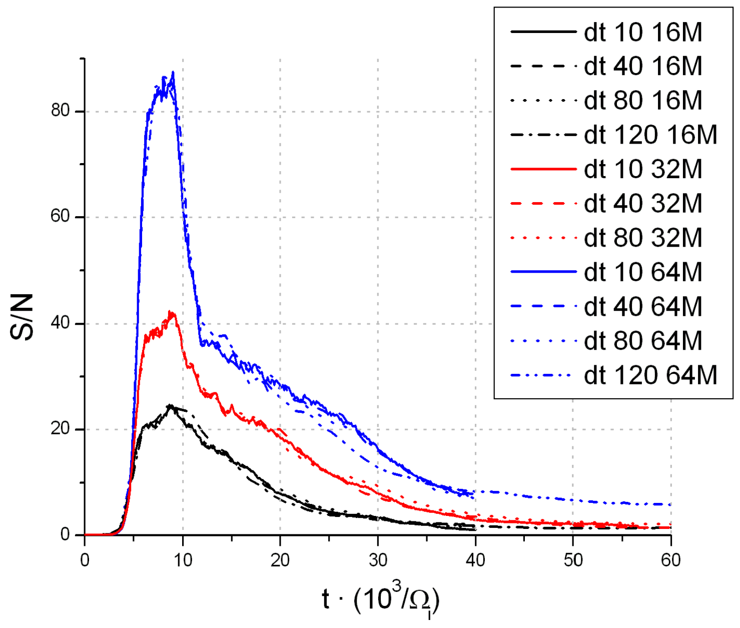


Figure 5.2: Comparison between time step size and particle numbers in a fixed grid.

Furthermore, Figure 5.3 shows that the numerical noise increases with the size of the grid, in agreement with results obtained by other authors [69, 70] because the increase in the mesh elevates the numerical error in all the calculations.

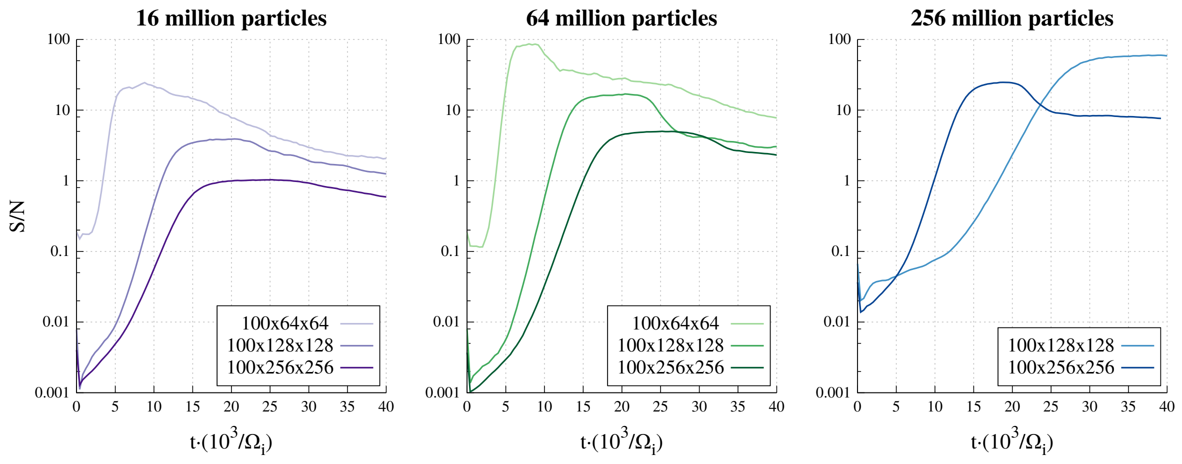


Figure 5.3: Comparison among three grid sizes for a time step ($dt = 40$).

5.4.2 Energy Conservation

The difference between the electrostatic and the kinetic energy of the particles constitutes a measure of the energy conservation, and consequently, another measure of the quality of the simulation.

In the previous section, the time step size did not show any influence on the signal, so now we are going to look into if there is an influence on the energy conservation. There is a linear phase where both energies grow exponentially and, after this phase, the non-linear interactions between modes become important. In order to study how the energy conservation depends on the time step, we define an average measure of energy conservation as

$$q = \left\langle \frac{|E_f + E_k|}{|E_f|} \right\rangle \tag{5.6}$$

where E_f and E_k are the electrostatic and kinetic energies and $<>$ means ensemble average. This average is computed omitting the initial phase of the simulation, before the linear growth.

In Figure 5.4, the average quality measure is computed in a time step of $10^4/\Omega_i$ for a fixed grid size of $100 \times 64 \times 64$. Just like the initial perturbation is different in simulations with different number of particles, we have to take into account that at a given simulation time, the energy E_f (and E_k) can also reach distinct levels in different simulations. Moreover, the levels can be very different for times corresponding to the exponential growth phase. To prevent distortions due to this effect, the beginning of the averaging window is not located at a fixed time, but when the E_f reaches a minimum level (10^{-4}) in normalized units instead.

Given the above, Figure 5.4 depicts how the time step influences the energy conservation. For small time steps the energy conservation is almost the same for different number of particles ($N = 16, 32$ and 64 millions) since these lines nearly superimpose. On the other side, when the time step increases the energy conservation gets worse.

Looking at the simulations, we noted that the field energy changes only slightly when the time step is increased, whereas the kinetic energy of the particles increases significantly. The increase in the kinetic energy grows with the size of time step, as shown in Figure 5.5. This figure displays the increase in energy over the initial value for different time steps using 64 million particles.

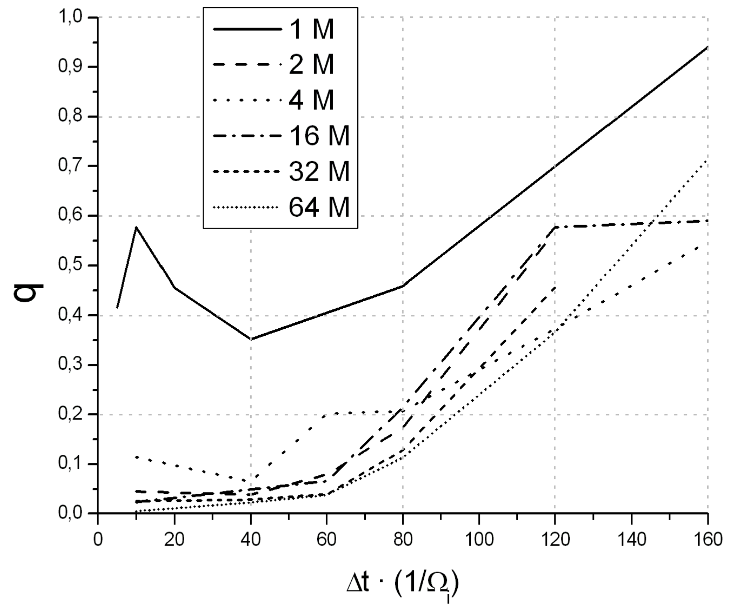


Figure 5.4: Average energy conservation ($<(|E_f + E_k|)/|E_f|>$) as a function of the time step for a fixed grid size of $100 \times 64 \times 64$ cells.

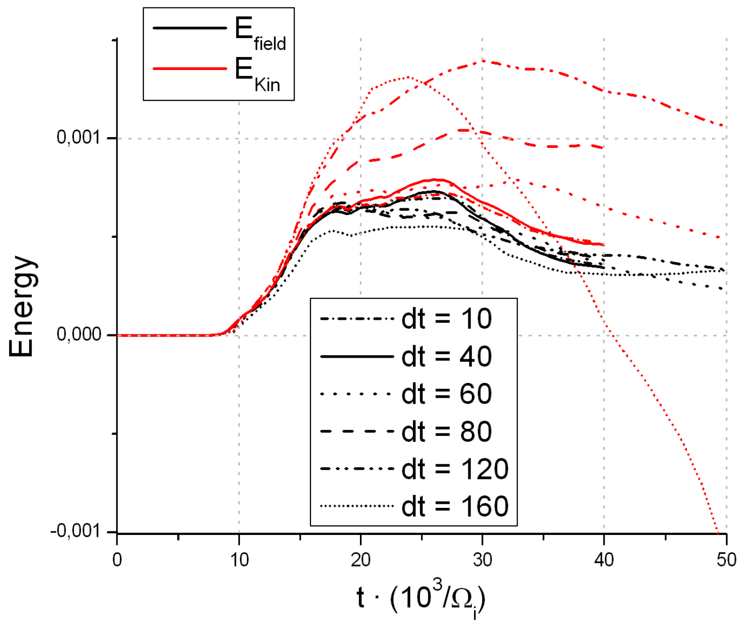


Figure 5.5: Potential and kinetic energies for different time steps in a $100 \times 64 \times 64$ grid.

Figure 5.5 also shows that the kinetic energy is responsible for the degradation in the conservation of energy. As the time step size grows in the figure, the lines move away since energy is not conserved due to the loss of information. As a result, for a time step of 80 the kinetic energy is 40% larger (in absolute value) than the potential.

5.4.3 Conclusion

In tests carried out with varying step sizes, number of markers and grid sizes, it turned out that S/N does not depend on the step sizes. These tests also showed that the noise decreases (i.e. the quality of the simulation improves) with increasing number of markers, and the noise increases (i.e. the quality worsens) with increasing grid size.

From the results obtained, we can suggest the following window of parameters to perform simulations with the least amount of work possible and get a reliable result. The simulations should use a minimum of 32 million particles (Fig. 5.4) and a maximum time step of $40 \cdot 10^3/\Omega_i$ (Fig. 5.5). Regarding the grid size, the suitable size depends on the number of particles selected and it would require a deeper study to find an optimal value. Even so with the previous data, a good value could be $100 \times 128 \times 128$ cells (Fig. 5.3).

With regard to energy conservation, it is observed that increasing the time step and decreasing the number of markers both degrade the quality of the solution. The potential energy is less affected by a change in time step than the kinetic energy which grows very quickly with increasing time step. This could be an indication of numerical noise introduced by the temporal integrator (RK4), particularly for large time steps. In further work one should consider the use of a new time integrator (for example a *symplectic integrator*) for the equations of motion of the particles. This could allow a better conservation of energy or the use of a larger integration time step with acceptable results.

Chapter 6

Process Level Parallelism

"For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers."

Gene Amdahl

This chapter discusses the way to parallelize a code by Message Passing Model. The chapter explains the data decompositions applied to any Particle-In-Cell (PIC) code and which strategies are implemented in EUTERPE. Finally, several tests are presented to analyze the performance of EUTERPE up to a large number of processes.

The work done in this chapter resulted in the following publications:

[16] **X. Sáez, A. Soba, E. Sánchez, J. M. Cela, and F. Castejón**, Particle-in-cell algorithms for plasma simulations on heterogeneous architectures, in *Proceedings of the 19th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 385-389, IEEE Computer Society (2011)

[13] **X. Sáez, A. Soba, E. Sánchez, R. Kleiber, R. Hatzky, F. Castejón and J. M. Cela**, Improvements of the particle-in-cell code EUTERPE for Petascaling machines, *Poster presented at Conference on Computational Physics (CCP)*,

June 23–26, Norges Teknisk-Naturvitenskapelige Universitet, Trondheim, Norway (2010)

[14] **E. Sánchez, R. Kleiber, R. Hatzky, A. Soba, X. Sáez, F. Castejón, and J. M. Cela**, Linear and Nonlinear Simulations Using the EUTERPE Gyrokinetic Code, *IEEE Transactions on Plasma Science* **38**, 2,119 (2010)

6.1 Introduction

The large number of particles and spatial grid cells used in PIC simulations causes the corresponding runs to be extremely time consuming. Some simulations can take more than 10^4 CPU hours. As a result, the parallelization of PIC codes is a vital work in order to increase the capabilities of PIC simulations. However, it is a hard mission, since PIC codes have data accesses with multiple levels of indirection and complicated message passing patterns.

In the literature at the time of starting this thesis, the algorithms used to parallelize PIC codes using data decomposition were explained in several works as [71, 72], achieving very good parallel efficiency when the computation was well distributed on processors.

The key step in the parallelization of a program is called *decomposition* and consists in breaking the problem into discrete parts that can be solved simultaneously. In this way, each part of the problem can be executed concurrently on different processors.

A program is a set of data and an algorithm, so there are two main ways of approaching the problem: to partition the data and then work out how to associate the computation with divided data (*data decomposition*) or to partition the computation into disjoint processes and then dealing with the data requirements of these processes (*functional decomposition*).

6.2 Message Passing Model

The *Message Passing Model* is used when parallel processes do not share the memory space, so the only way to move data from the address space of one process to that of another process is via messages (Fig. 6.1).

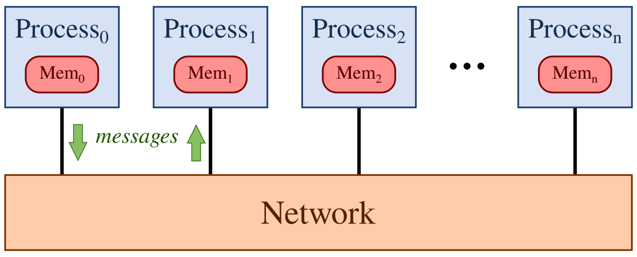


Figure 6.1: Message passing model.

The programmer is responsible for writing all the extra code to: create processes, manage processes, synchronize processes and pass messages among processes. Also, the programmer has to divide and distribute the program data among the different processes. So, the parallelization of a sequential code using this model requires a considerable programming effort, but on the other hand it is easier to get a good performance.

Message Passing Interface (MPI) [73] is a standard based on the consensus of the MPI Forum, which has over 40 participating organizations. Its first version was presented in 1994. It provides the specification of a set of functions for the programmer to distribute data, synchronize processes and manage communication among processes. The advantages of MPI over other message passing libraries are portability (because MPI has been implemented for almost every distributed memory architecture) and speed (because each implementation is in principle optimized for the hardware on which it runs).

Originally, MPI was designed for distributed memory architectures, which were popular at that time. Nevertheless, it has been adapted to new platforms such as shared memory systems, although the programming model clearly remains a distributed memory model.

All processes have the same source code but an identifier enables that each process executes a different part of the code. Information can be exchange between two specific processes (point-to-point communication) or among all processes (collective communication). The communications can be synchronous (blocking a process until the operation completes) or asynchronous (the process continues without waiting the operation completes).

EUTERPE implements this programming model for parallel computing using the interface MPI to define processes and communicate amongst themselves.

6.3 Data Decomposition

The first step to parallelize a code is to divide the domain among the processors (Fig. 6.2). The performance of PIC codes will depend critically on the decomposition strategy chosen for distributing the domain made up of grid points and particles. In fact, a significant part of PIC codes perform interpolations between particles and grids, and as a consequence it is interesting that these two data structures reside on the same processor. Among all the strategies, it is worth highlighting the *domain decomposition*, the *particle decomposition* and the *domain cloning*.

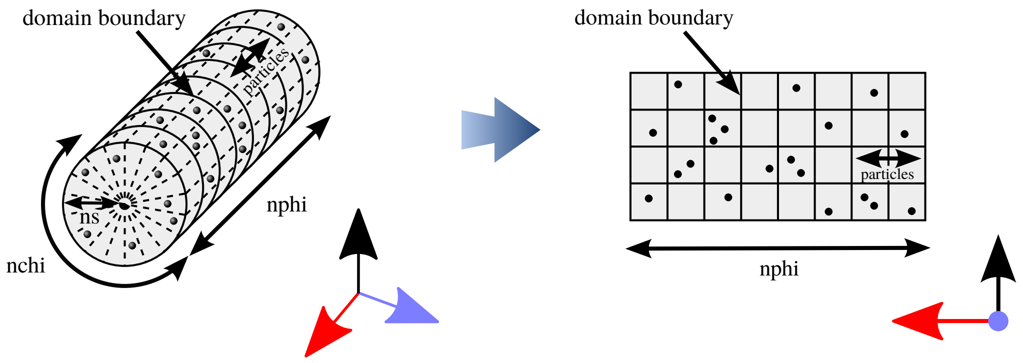


Figure 6.2: Cylindrical domain composed of a grid and a set of particles.

6.3.1 Domain Decomposition

The most usual technique used to parallelize a PIC code is *domain decomposition* [71], where the grid is divided into portions. Each portion of grid with the particles located in its interior is called partition and it is assigned to a processor. Since particles are assigned to processors according to their coordinates, as long as the particles are evenly distributed, this technique divides efficiently the computation among processors (Fig. 6.3).

At the end of each time step, the particles whose new positions are in other partitions are transferred to their new processors. It is considered that the particles only can traverse one partition at each time step, since the elapsed time is small enough. So, at the end of the loop, each processor sends buffers with the particles that leave its partition and receives buffers with the particles that arrive [40].

In the boundary cells, the grid points (vertices) belonging to two processors have a non-definitive values because the adjacent cells from neighbours processors contribute to

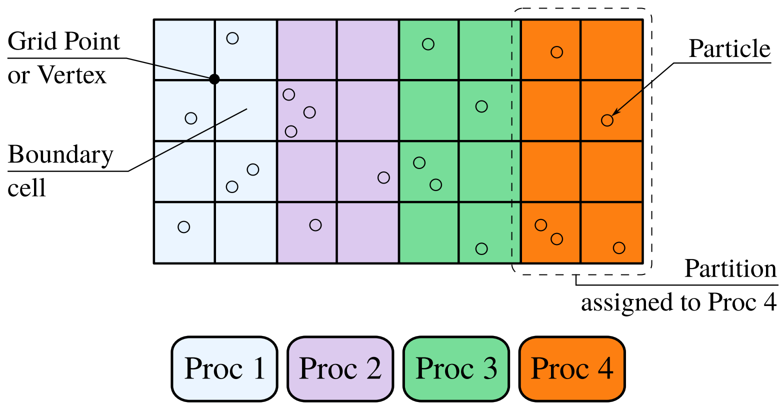


Figure 6.3: Domain decomposition.

the computed value. To update them, electromagnetic values from boundary vertices are copied into a buffer and are sent to the neighbour processors.

Some of the contributions on this strategy are: Decyk [72] compares the results obtained by the parallelized code on different architectures; Norton [74] proposes the benefits of object-oriented approach to the parallel PIC code; and Akarsu [40] presents the results of the implementation of parallel PIC in High Performance Fortran.

The main advantages of this technique are: the *data locality* for the gather and scatter phases because the needed data always reside inside the same partition, and the *intrinsic scalability* of the physical-space resolution when increasing the number of processors, i.e. adding further processors allows adding new cells to the physical domain and performing more partitions.

Nevertheless, there are also disadvantages: the *parallelization is limited* by the resolution of the grid for fixed domains; and the likely appearance of *load unbalance* coming from particle migration between partitions, since inflow and outflow of particles in a partition may be different. It means that this technique requires a *dynamic load balancing*, such as a dynamical redefinition of the partitions, that can be very complicated to implement and it introduces an extra computational load.

Additionally, Tskhakaya [75] points out a problem with the dimension of the array, since the particle array size has to be large enough to store incoming particles to the partition. If there is a good load balancing, there will be no problem since all partitions will have nearly the same population of particles. Nevertheless, if there is not a good load balancing, then the array will not have enough space and more time will be required to generate a larger array and move the data to it.

Some authors defend this technique because they say that there is a good load balancing, as Messmer [43]. Others, like Lin [76], propose that fluctuations in electromagnetic forces cause particles are distributed non-uniformly over the grid and consequently over the processors. As a result, *since the number of particles on each processor does not remain constant over time, load imbalance may appear between processors.*

Przebinda [44] points out that the computing time per processor depends on the changing concentration of plasma during the simulation. Due to the movement of particles, the load balancing varies during the simulation. So, it would be necessary a *dynamic load balancing*, as for example, his method based on adjusting the decomposition boundaries at each time step. In this method, all the processors are ranked by the update time from greatest to least. The first ranked processor adjusts its decomposition boundaries by shrinking all boundaries around it. Next the second ranked processor shrinks all its boundaries except those who have already been adjusted by previous processors, and so on. Unfortunately, in practice this method is extremely costly due to communications involved in resynchronizing the data values after the decomposition adjustment, such as particles that have to move to another processor.

6.3.2 Particle Decomposition

Di Martino [77] proposes a different technique called *particle decomposition*, where the particle population is divided into subsets whereas the grid is not split up. Concretely, each processor contains a partition, which is formed by a copy of the whole grid and one particle subset, as shown in Figure 6.4.

This strategy presents the advantages of an *intrinsic load balancing*, since the particle population remains constant in all partitions during the simulation, and the lack of extra communications due to the absence of particle migrations between partitions.

However, to update the electromagnetic field in each partition, the partial contributions at grid points have to be sent to the rest of partitions to get the correct charge density. It means that there is an expensive all-reduce communication among all processors to update the field data on each processor. In addition to this, there is the drawback that the full grid has to fit in the memory of each processor. For this reason, any increase in the number of processors, unfortunately only enables to increase the number of particles but not the spatial resolution.

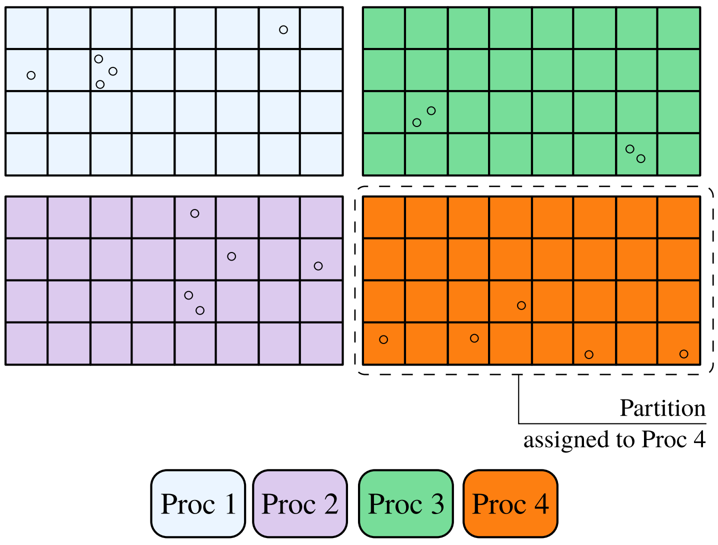


Figure 6.4: Particle decomposition.

Finally, it is for all these reasons that Di Martino explains that this method is suitable for moderated parallel architectures, up to a few tens of processors, as long as each processor can store the entire spatial grid. Otherwise, if the number of processors is larger, the communication of the field among all processors will be more expensive than particle movement among neighbours processors (domain decomposition method). In other words, *particle decomposition is not suitable for highly parallel architectures*.

6.3.3 Domain Cloning

There is a third alternative proposed by Hatzky [78], it is a mixing between *domain decomposition* and *particle decomposition*. It is called *domain cloning* and introduces the notion of *clone* (Fig. 6.5). A *clone* is a copy of the full grid that only contains a part of the particles. The processors are subdivided into as many groups as clones and each group is responsible for one clone. Finally, a domain decomposition is performed inside each clone for distributing the work among processors of the clone.

This strategy increases the parallelization afforded by the domain decomposition because adds a particle subdivision to the domain subdivision at the same time. As a result, the number of partitions is decoupled of the grid resolution. Besides, this method is also less restrictive than particle decomposition, because field data are only distributed among the processors of a group. As a result, if we assign only a clone to each node, *domain cloning seems a good idea to implement in a multi-core architecture*. In this way

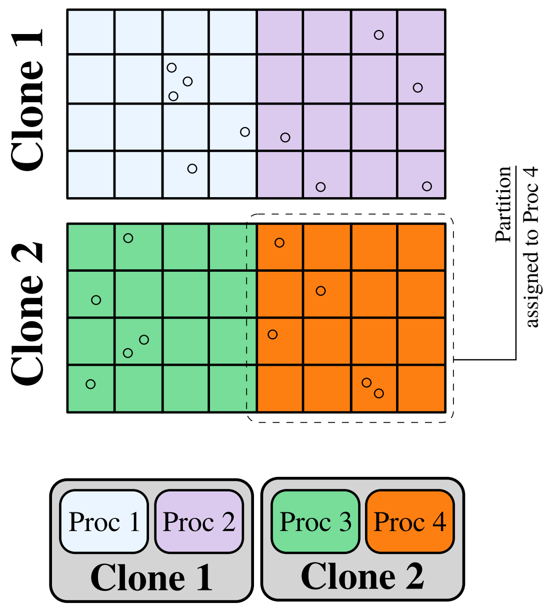


Figure 6.5: Domain cloning.

the intensive communication by particle movement among partitions will be restricted to the memory inside nodes (more bandwidth), without boosting the inter-processor communications to prohibitive levels, and the smaller communication among nodes due to the field accumulation will take place via network.

6.4 Euterpe Parallelization

The EUTERPE code offered two ways to divide the domain in one-dimension among the processors for the process level parallelization :

- the *domain decomposition*, where the distribution of work between processors is based on the division of the physical domain into portions in the toroidal (ϕ) direction, as shown in Figure 6.6.
- The *domain cloning*, which is an alternative proposed by Hatzky for enabling the use of many more processors without increasing the inter-processor communications to prohibitive levels [78].

Once the problem has been divided, the simulation begins to move particles in the domain. The phase of moving the particles can be parallelized because of their independent behaviour.

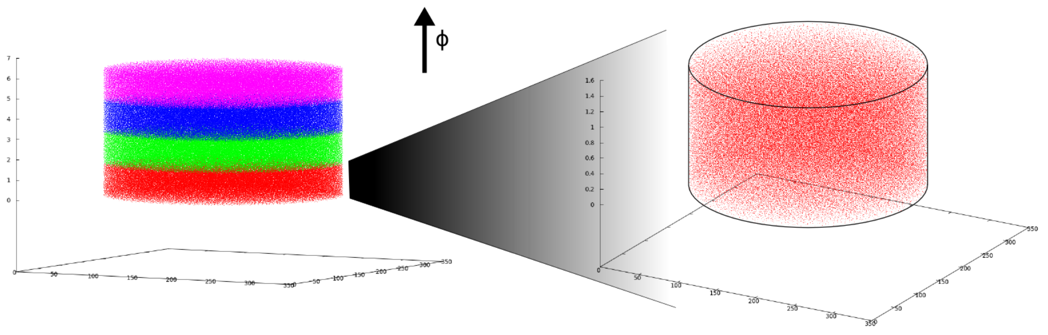


Figure 6.6: Domain decomposition of a real cylindrical grid among 4 processes (with zoom in the process 0).

6.5 Evaluation

One fact to note is that EUTERPE has two pre-processing steps: the generation of the electrostatic fixed equilibrium as initial condition done by an external application and the computation of the finite element matrix contained in the quasi-neutrality equation. Their computational load is not significant because they are only executed once, and therefore they are omitted in the performed analyses. The analyses will be based on the duration of one iteration of the main loop (or *time step*) as a reference.

6.5.1 Load Balancing

In order to verify the quality of the domain distribution performed by EUTERPE, we executed a small test on *MareNostrum II* to get a trace and analyse it with Paraver to see if the workload was balanced among the processes. The resolution of the spatial grid was $n_s \times n_\theta \times n_\phi = 32 \times 64 \times 64$ and the number of particles was 1 million.

Figure 6.7 shows a detailed view of the temporal sequence of 32 MPI processes. This trace confirms the good behaviour of the MPI version and the well balanced work between the processes, as all the processes describe a very similar behaviour. The names of the routines are specified on the top of the graphic. On the right, there is the legend to know how to interpret the state of the processes. Additionally, the peak performance of push routine, 650 MFLOPS, gives us signals that we are in front of a memory bound issue, because few floating point operations are computed per memory access.

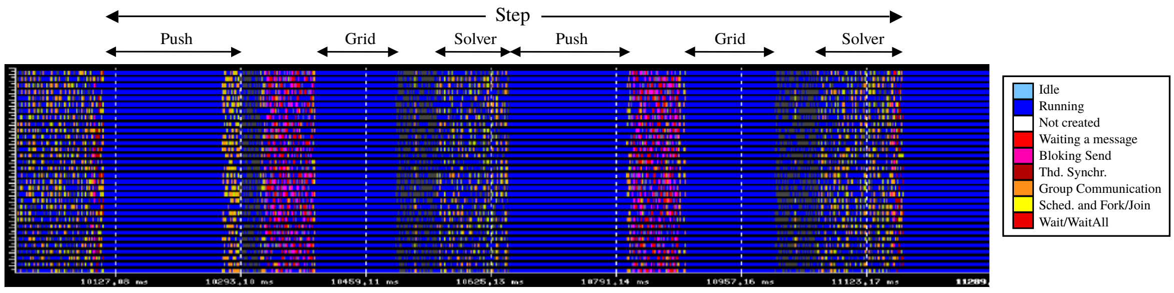


Figure 6.7: Timeline showing the state of MPI processes during one time step.

6.5.2 Scaling

When we started to work on this topic, the published scaling studies in this area reached up to 512 processors [12, 43, 75] and, as discussed in Chapter 4, the trend was to increase greatly the number of cores in the computer systems. Therefore, our first decision was to do a scaling study with a large number of processes to evaluate if the program was well prepared to face the future requirements of a large number of processes.

Our purpose was to analyse the strong-scalability of the original version of EUTERPE. The methodology followed consisted in maintaining fixed the size of the problem while the number of processors used in each simulation was increased. The first test ran up to 1,024 processors on *MareNostrum II*, thereupon we increased the number of processors up to 2,560 processors on *Huygens* and finally we ran up to 23,552 processors on *Jugene*.

To enable running the simulations for these numbers of processes, the grid resolution was increased to $32 \times 512 \times 512$ and the number of particles to 10^9. The use of clones allowed us to reach more than the initial 512 processes that this grid allows in the toroidal direction.

Without the domain cloning, the number of grid partitions should increase to provide one partition per process. It implies that the partition size of the grid would be so small, that a big number of particles would leave their slice after a time step. This fact would increase the communications among neighbouring processes up to ruining the performance.

Figure 6.8 shows the speedup of one time step on *MareNostrum II* and *Huygens*. The figures depict a very good scalability since the speedup curves are close to linear. This confirms a good performance on several thousands of processors using few clones (2 in the case of *MareNostrum II* and 5 in the case of *Huygens*).

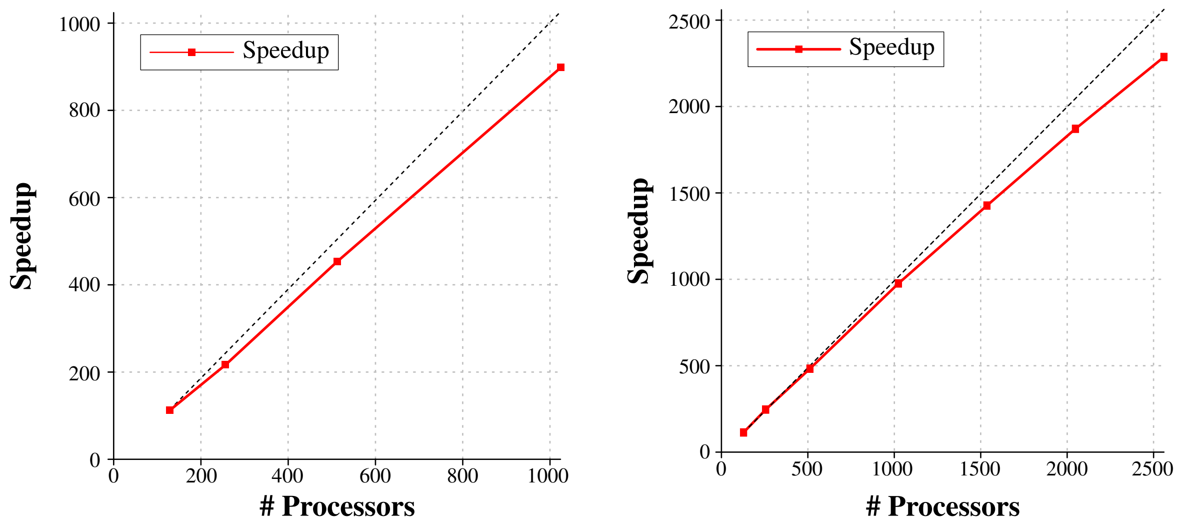


Figure 6.8: Scalability of EUTERPE on *MareNostrum II* (left) and *Huygens* (right).

However, Figure 6.9 illustrates a worse scalability on *Jugene*, where a degradation of the speedup appears from 12,288 processors. The reason for this fact is that as the number of processors (i.e. processes) increases, the number of particles assigned to each process decreases, and also the communications to update the electromagnetic field among clones increase. Therefore there comes a time when the amount of work per processor is not enough to hide the increasing time for the inter-processors communications and the code stops scaling.

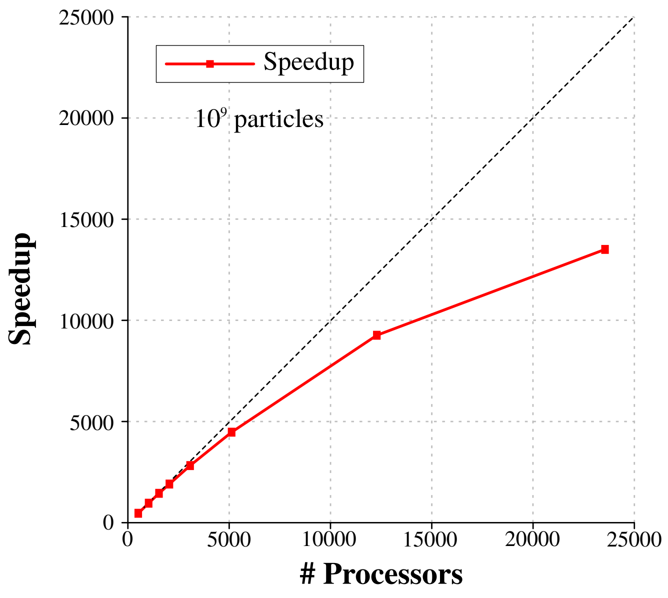


Figure 6.9: Scalability of EUTERPE on *Jugene*.

If our interpretation is correct, when the number of particles is increased, we should get a better scalability. The result of moving $5 \cdot 10^9$ particles is plotted in Figure 6.10, which shows an improvement that allows us to achieve the number of 61,440 processors (or processes). But the performance does not reach the ones shown in Figure 6.8, because the increased number of clones (that reaches 60) involves more inter-clones communications and its cost can not be hidden sufficiently by the increment of particles.

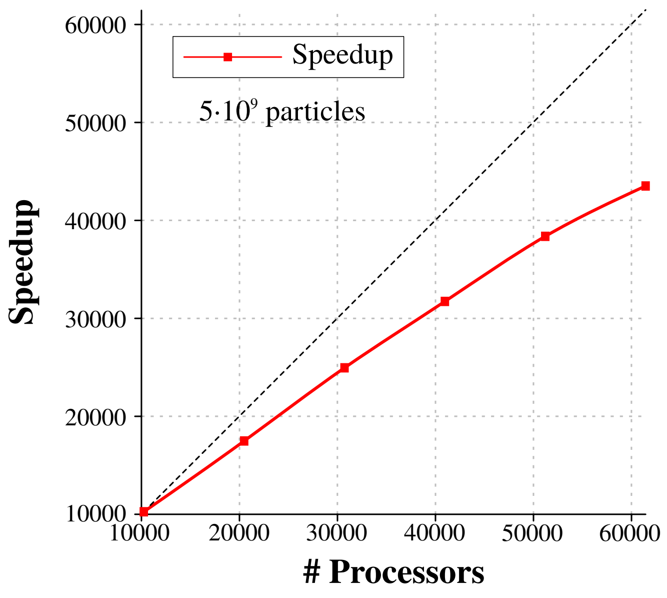


Figure 6.10: Scalability of EUTERPE on *Jugene* with $5 \cdot 10^9$ particles.

6.6 Conclusion

In a nutshell, to assess the readiness of the EUTERPE code to run on multi-core systems with a large number of cores, we have studied its scalability up to 60,000 processors with a very good performance. Therefore, we could say in general terms that the original version of this code scales correctly for a large number of processors.

The reason to start the simulation from 10,240 processors in Figure 6.10 was that there was not enough memory to store $5 \cdot 10^9$ particles into less processes, since more than $5 \cdot 10^5$ particles did not fit into the portion of the node memory corresponding to a process.

Jugene is a Blue Gene system that contains nodes with 4 cores and 2 GBytes of memory per node. It means that if a MPI process is assigned to each core on a node, then each process has only access to 500 MBytes, which is a really small amount of memory. And as this situation can only get worse with more cores per node, we think that it will be very interesting that any process can access all the memory of a node without loosing the workforce of the rest of cores. So, there is a need to implement a thread level parallelism in this code to exploit multi-core architecture as we are going to see in the next chapter.

Chapter 7

Thread Level Parallelism

"The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it."

Apple Computer co-founder
STEVE JOBS

This chapter introduces the parallelization at thread level by Shared Memory Model. The main focus of the chapter is the implementation of the hybrid model (MPI/OpenMP), which is chosen in order to take advantage of all the levels of parallelism that a multi-core architecture offers. The implementation done is described in detail, including a thread-safe solver. Lastly, a scaling analysis is performed to demonstrate the suitability of this new hybrid version.

The work done in this chapter resulted in the following publications:

[16] **X. Sáez, A. Soba, E. Sánchez, J. M. Cela, and F. Castejón**, Particle-in-cell algorithms for plasma simulations on heterogeneous architectures, in *Proceedings of the 19th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 385-389, IEEE Computer Society (2011)

[17] **X. Sáez, T. Akgün, and E. Sánchez**, Optimizing EUTERPE for Petascaling, *Whitepaper of PRACE-1IP project*, Partnership for Advanced Computing in Europe (2012)

[20] **X. Sáez, A. Soba, E. Sánchez, M. Mantsinen, and J. M. Cela**, Performance analysis of a particle-in-cell plasma physics code on homogeneous and heterogeneous HPC systems, in *Proceedings of Congress on Numerical Methods in Engineering (CNM)*, pp. 1–18, APMTAC, Lisbon, Portugal (2015)

7.1 Introduction

One of the most important trends in contemporary computer architecture is to build processors with multiple cores that have access to a shared memory.

Within the fusion community, many codes designed to simulate various aspects of the plasma behavior [4, 6] were written by physicists with emphasis on the physics without using the latest technologies available in computer science. As a result, these codes were not thought for current supercomputers based on multi-core architectures and only take advantage of the process level parallelism, basically using Message Passing Interface (MPI). In this chapter we discuss ways to apply hybrid parallelization techniques in a Particle-In-Cell (PIC) code for exploiting new multiprocessor supercomputers.

In general, computer scientists perform their studies with very simplified versions of production codes used in the simulation of real plasma physics environments. In our case, the aim was to apply the thread level parallelism in a real production code (EUTERPE) with all the involved complications (e.g. big data structures).

7.2 Shared Memory Model

The *Shared Memory Model* is used when parallel processes share the memory space, so several processes can operate on the same data (Fig. 7.1). This opens up a lot of concurrency issues that are common in parallel programming (such as *race conditions*) and require protection mechanisms such as locks, semaphores and monitors to control concurrent access.

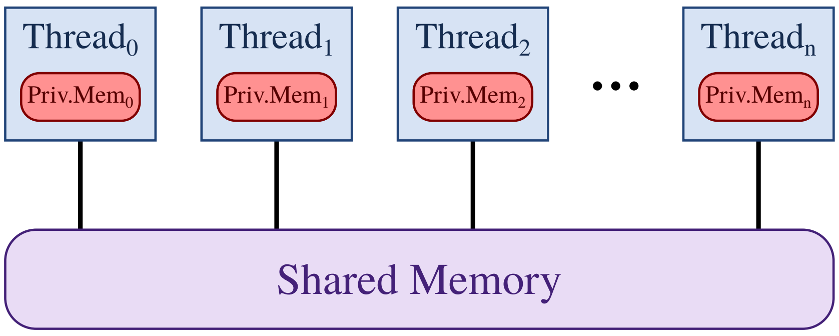


Figure 7.1: Shared memory model.

Here, the parallel processes are named *threads*. Each thread has a identifier. All threads have access to the shared memory, but they can also have a private memory. A *race condition* is when several threads can write data simultaneously into the same shared memory position. The result obtained depend on the order in which these threads access to this memory position, so the result is not predictable and can be different among executions. Therefore, this situation has to be prevented or controlled (if it is unavoidable) to get right result.

Open Multi-Processing (OpenMP)[79] is an Application Programming Interface (API), jointly defined by a group of major computer hardware and software vendors. Its first version was presented in 1997. It is supported on many architectures, specially on Symmetric Multiprocessing (SMP) architectures. One advantage of this model is the simplicity and promptness when it is applied to a sequential code, since it is only necessary to introduce some pragmas to indicate the candidate regions to parallelize and the visibility of the variables (private or shared) respect to other threads. For all these reasons, OpenMP has been our choice although not always it is easy to get a good performance.

OpenMP uses the fork-join model of parallel execution, as shown in Figure 7.2. The program begins with a unique thread (*master thread*) which executes the code sequentially until it finds a parallel region. Then the master thread creates a collection of parallel threads (*slave threads*) and the work is distributed among all the threads. The master and slave threads run concurrently the parallel region. When the threads complete the parallel region, they synchronize and terminate, leaving only the master thread which continues sequentially the execution until the following parallel region.

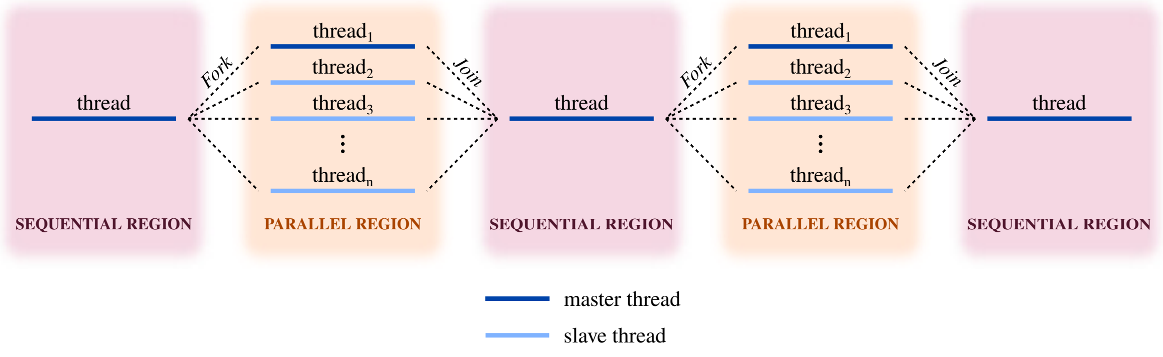


Figure 7.2: Fork-Join model.

The programmer identifies the regions of the code that wants to execute in parallel through special compiler directives or *pragmas*. Next, the compiler will be in charge of generating automatically the parallel code for this marked regions.

In order to introduce thread parallelism in a loop, in addition to using the pragma `PARALLEL DO`, we probably would require other pragmas to indicate the compiler how to parallelize appropriately:

- **Loop scheduling**: The iterations of a parallel loop are assigned to threads according to a scheduling method:
    - **static**: If all the iterations have the same workload, then the iterations are divided among threads equally before the loop starts. The thread management overload is minimum.
    - **dynamic**: All iterations have not the same workload, so the distribution of work can not be done before the loop starts. Iterations are grouped in identical sets of few iterations (blocks) and they are assigned to threads during the loop execution. When a thread finishes the iterations of its assigned block, it returns to obtain another one from the remaining blocks. The load balancing is achieved at the expenses of a high thread management overload.
    - **guided**: In contrast to dynamic scheduling, the block size starts with a large number of iterations and decreases progressively at each block assigned. In this way, a better relation between the thread management overload (because needs less blocks) and the balanced workload (because final blocks are thinner) is achieved.

- **Data scope**: the `SHARED` variables are shared among all iterations and they are visible and accessible to all threads simultaneously; whereas the `PRIVATE` variables have an independent value among iterations due to having a different private copy per thread.
- **Synchronize threads**: to protect shared data from data race conditions, sometimes there is no other way than to force the execution of a region of code by only one thread at a time (`CRITICAL SECTION`). However, there is another option called `ATOMIC` instructions, where the memory update (write, or read-modify-write) in the next instruction is performed atomically. Normally, atomic instructions are better than critical sections because the compiler may use special hardware instructions to implement them.

7.3 Hybrid Version: MPI-OpenMP

A *hybrid model* is the combination of different programming models in a complementary manner in order to take advantage of the good points of each one.

The combination of a shared memory model (OpenMP) and a message passing model (MPI) allows us to exploit the shared memory among the threads (cores) on a node and reduce communications to only among processes (nodes).

In our case, we decided to introduce OpenMP in EUTERPE because we thought it could make more efficient use of the shared memory on SMP node, and in this way mitigating the need for explicit intra-node communications by MPI.

The way to proceed is to launch a single process on each SMP node in the cluster, and then each process creates a thread per core, as for example shown in Figure 7.3. This distribution implies another advantage, each MPI process has access to all memory of a node, whereas previously it only had access to a part of it because several MPI processes were sharing the node, hence the hybrid version is able to face bigger problems than before.

To sum up, introducing OpenMP into MPI is not complicated, but often the resulting hybrid application is slower than the original MPI application. This is because benefits depend on how many loops can be parallelized, since slave threads do not do anything outside the parallelized loops. As a result, the processor utilization on each node will become an issue to deal with.

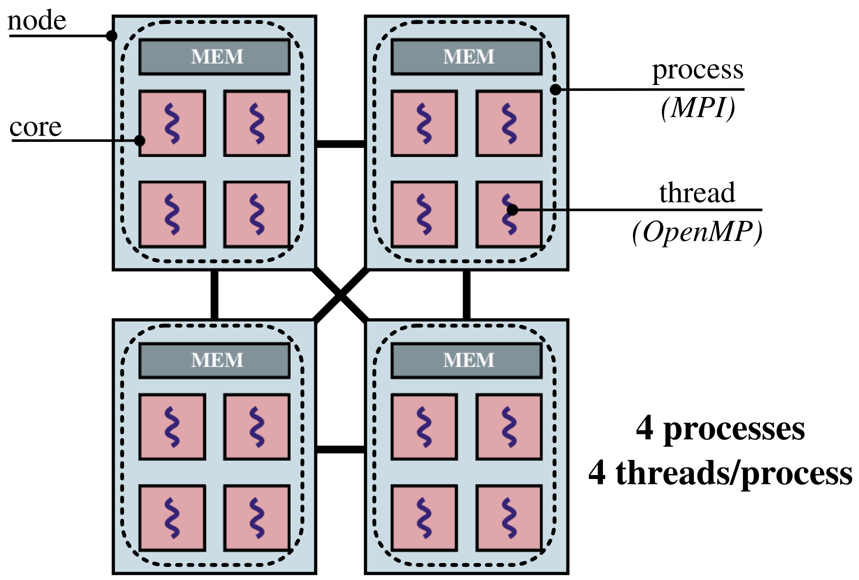


Figure 7.3: Hybrid model over a multi-core architecture (4 nodes with 4 cores/nodes).

7.4 Introducing OpenMP in the Code

The EUTERPE code came to our hands using only MPI, or in other words, process level parallelism. Its performance was really great at this level for several thousands of processors, but as seen in Section 6.5.2, for a larger number of processors the scalability was worse due to the increase of inter-clones communications and the reduction of the work per processor. Besides, another issue appeared in Section 6.6, EUTERPE had low memory available for each process in a multi-core system, because all the shared memory on the node was distributed separately among the processes.

For those reasons, our idea was to develop a hybrid code by introducing OpenMP pragmas to exploit the thread level parallelism. This approach would enable to take advantage of all the levels of parallelism that a multi-core architecture offers, and also would enable one MPI process to have access to all the memory of a node.

As *Amdahl's law* teaches us in Section 4.7.3, only the most time-consuming phases of an application are worth parallelizing. Accordingly, the algorithm has been profiled to find the most time-consuming routines. Four different populations of particles (1, 10, 50 and 100 millions) have been simulated into a cylinder divided in a grid of $64 \times 1{,}024 \times 1{,}024$ cells on *MareNostrum II*.

From the results obtained in Figure 7.4, we identified three functions that covered jointly more than 80 per cent of the execution time: the routine that moves the particles (`push`), the one that determines the charge density (`setrho`) and the one that solves the field equation (`poisson`). Consequently, these routines were the candidates to incorporate threads using OpenMP.

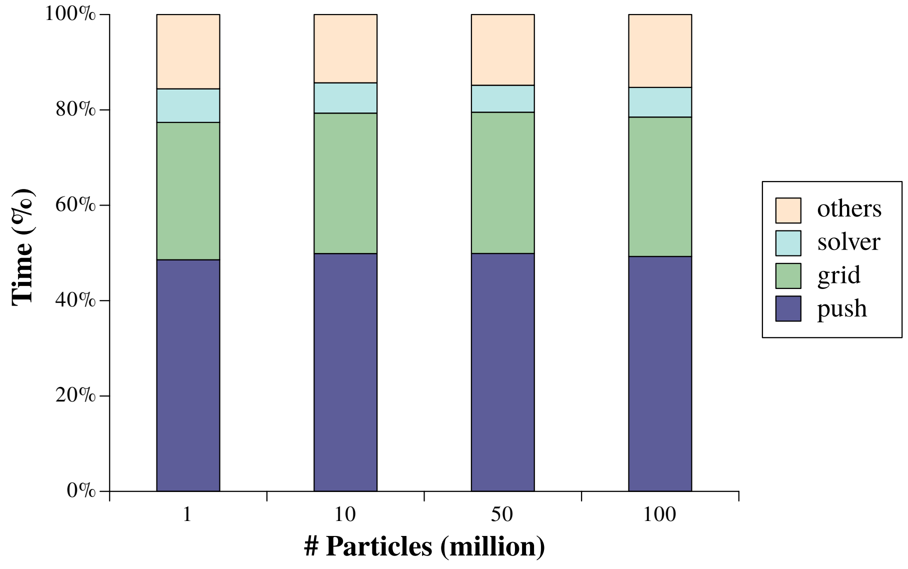


Figure 7.4: Time distribution in the EUTERPE code.

After analysing the source code and its time distribution, we believed that the most appropriate way to work was to divide the time loop of EUTERPE into three steps in place of the four mentioned in Section 3.1.1, since the phases *gather* and *push* are intermingled in the code. Thus, hereinafter the three phases to summarize the code are (Fig. 7.5):

- **pull**: *scatter phase*, where the particle properties are interpolated to neighboring points in the computational mesh.
- **solve**: *solve phase*, where the moment equations are solved on the mesh.
- **push**: *gather* and *push phase*, where the momentum of each particle is calculated by interpolation on the mesh. The particles are repositioned under the influence of the momentum and the particle properties are updated.

All routines are not susceptible to be parallelized at thread level, because they must contain a loop structure preferably with independent iterations. Moreover, to introduce OpenMP is interesting to look for large loops enclosing a whole computation. This

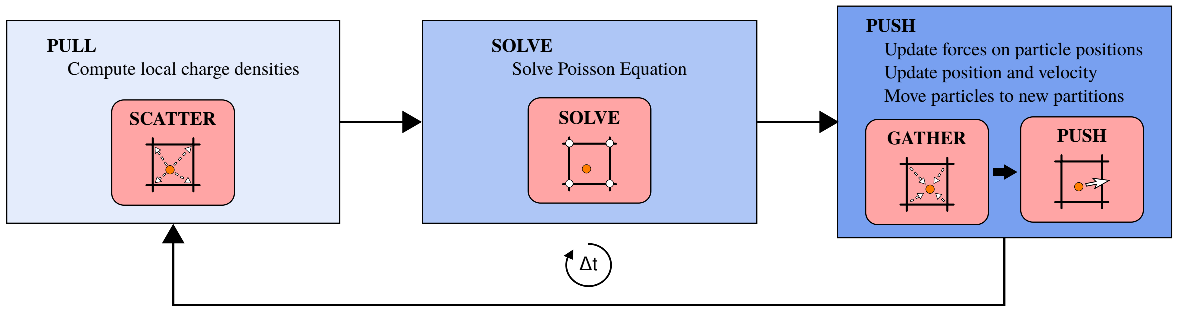

PULL
Compute local charge densities
SCATTER
SOLVE
Solve Poisson Equation
SOLVE
PUSH
Update forces on particle positions
Update position and velocity
Move particles to new partitions
GATHER
PUSH
Δt

Figure 7.5: Steps of the EUTERPE code.

approach involves more programming effort because there are more data structures with dependencies to control. However, it can provide a higher level of scalability since it hides the overhead of thread management behind the computing time, offers more chances to reuse data in cache and provides a bigger context for compiler optimizations.

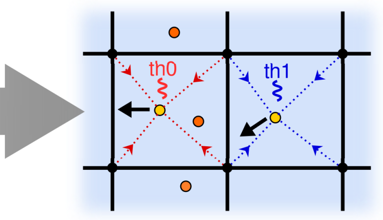
Special care has to be taken to find the right place where to put the OpenMP directives, since in the worst case some data dependencies among iterations can cause the so called *race condition*. Basically, race conditions arise when different threads execute a section of the code that tries to update the same memory position. This region of the code which accesses a shared resource is known as *critical region* and may lead to non-deterministic results, therefore it requires mutual exclusion to avoid this issue. The abuse of this solution could serialize the execution, since only one thread can execute the critical region meanwhile the rest of threads must wait to enter, resulting in significant decline of the performance. In order to minimize this drawback, one has to avoid critical regions or, if not possible, one has to break them down into different smaller critical regions or, even better, replace them with atomic clauses instead.

7.4.1 Push Phase

The `push` routine (Algorithm 7.1) moves the particles inside a domain. It contains a loop on particles with a coarse granularity. The computation of each iteration is independent from the rest, since the computation of the movement of any particle does not depend on the rest of particles and several threads can read simultaneously the electric field on the same grid point without any conflict. Therefore, the `parallel do` directive was placed in the loop (surrounded by the frame in the algorithm) and no critical regions were needed.

Algorithm 7.1 Introducing OpenMP in the **push** routine.

```
...
do ip = 1, npart_loc
   (rt,zt,phit) = pic_loc(ip)
   ...
   call getfield (rt,zt,phit...)
   ...
   if (diag_flag) then
      ...
   endif
enddo                          PARALLEL DO
...
```



Listing 7.1 shows the `PARALLEL DO` pragma introduced to parallelize the loop in the push routine. A `GUIDED` scheduling is specified to balance the workload.

Listing 7.1 Schematic of the **push** routine

```
!$OMP PARALLEL DO                                                                        &
!$OMP SCHEDULE(GUIDED)                                                                   &
!$OMP DEFAULT (NONE)                                                                     &
!$OMP PRIVATE (ip , rt , zt , phit , vpart , wghtt , mut0, cosalp , pvol ,              &
!$OMP          f0pvolt0 , vpar2 , b_abs , hx, hy, hz, bgrbx , bgrby , bgrbz ,           &
!$OMP          divh , grdsx , grdsy , grdsz , brotb , rotbpx , rotbpy , rotbpz ,        &
!$OMP          gbx , gby , gbz , b_abs_inv , bstar , bstar_inv , vperp ,                &
!$OMP          vperp2 , ztemp2 , vgbx , vgby , vgbz , vparcurvx , vparcurvy ,          &
!$OMP          vparcurvz , vpar2curvx , vpar2curvy , vpar2curvz , vparx ,              &
!$OMP          vpary , vparz , rhog , st , chit , Phi , mgPsi_xp , mgPsi_yp ,          &
!$OMP          mgPsi_zp , Apar , mgApar_xp , mgApar_yp , mgApar_zp , vebx ,            &
!$OMP          veby , vebz , phi_s_eq , E_xp_eq , E_yp_eq , E_zp_eq , vebx_eq ,        &
!$OMP          veby_eq , vebz_eq , vgcx_0 , vgcy_0 , vgcz_0 , hxvparcurvx ,            &
!$OMP          hyvparcurvy , hzvparcurvz , vgcx , vgcy , vgcz , vpar_0 ,               &
!$OMP          vpar_1 , zenergy , zvth2_inv , u_bulk , kappa_ubulk , ze_maxw ,         &
!$OMP          grads , gradchi , vdr , kappa , ctemp_w1 , ctemp_w2 , zf0 ,             &
!$OMP          f0pvolt , zsource , ctemp1 , ctemp2 , ctemp3 , ztemp1 , ctemp4 ,        &
!$OMP          ctemp5 , sloc , absgrads_inv , fges)                                     &
!$OMP SHARED  (npart_loc , pic1_loc , pic2_loc , twopinfp , species , msdqs ,          &
!$OMP          Phi_bspl , E_s_bspl , E_chi_bspl , E_phi_bspl , elmag ,                  &
!$OMP          Apar_bspl , mgApar_s_bspl , mgApar_chi_bspl ,                            &
!$OMP          mgApar_phi_bspl , qsdms , nlin_r , torflag_r , rhfs , dtloc ,           &
!$OMP          work1_loc , smax0 , msdmp , nsel_ubulk , scheme , diag_flag ,           &
!$OMP          jdote , numb0 , numb1 , pentr , ekin0 , ekin1 , ekin_dot0 ,             &
!$OMP          ds_inv , eflux0_s , eflux1_s , pflux0_s , pflux1_s , f_av ,             &
!$OMP          v_par_av , v_par2_av , v_perp2_av , ger_min , ger_max , gez_min,&
!$OMP          gez_max , phi_edge , rphi_map , rgedr , rgedz , nphip_min ,             &
!$OMP          iphipl , iphipr , nequ_r , nequ_z , nequ_rm1 , nequ_zm1 , b_equ ,      &
!$OMP         islw , islw1 , jclw , jclw1 , kplw , kplw1 , nidbas , m0 , n0 ,          &
!$OMP         sgrid, chigrid, phigrid, dsgrid_inv, dchigrid_inv,     &
!$OMP         dphigrid_inv, k_offset, zcos2, zsin2, phieq,           &
!$OMP         ds_profiles_inv, t_s, ubulk_s, nsn_s, tst_s, n_s )
  DO ip = 1, npart_loc
     rt     = pic1_loc(R_PIC, ip)
     ...
  END DO
!$OMP END PARALLEL DO
```

In order to test the performance of this implementation, we executed a small test on one node without MPI to avoid interferences. The test is the cylindrical geometry that we have studied in the previous chapters but downscaled to fit into one node. The resolution of the spatial grid was reduced to $16 \times 16 \times 16$ and the number of particles was reduced up to 10^6. Table 7.1 gives the time of the *push phase* after 10 time steps on a node (4 cores) of *MareNostrum II*. The table shows the possible combinations of loop scheduling methods (`STATIC`, `DYNAMIC` and `GUIDED`) and number of threads (1,2,3 and 4). The results provide grounds for optimism since show a quite good scalability and reaffirm our choice of `GUIDED` scheduling.

# Threads	Time (s)			Speedup		
	static	**dynamic**	**guided**	**static**	**dynamic**	**guided**
1	100.38	100.38	**100.31**	1.00	1.00	**1.00**
2	54.77	63.97	**54.63**	1.83	1.57	**1.84**
3	37.31	44.62	**37.23**	2.69	2.25	**2.69**
4	29.19	34.65	**28.61**	3.44	2.90	**3.51**

Table 7.1: OpenMP version of the push phase depending on loop scheduling.

7.4.2 Pull Phase

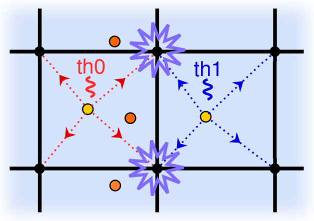
The `setrho` routine distributes the charge density on the grid points, meaning that it carries out the *scatter phase*.

This time the outer loop for introducing OpenMP is within a routine called by `setrho`. The loop is inside `grid` routine (Algorithm 7.2) and loops through all particles in the domain. Unfortunately, although its granularity is coarse, the iterations are not fully independent, because some particles can deposit charge at the same grid point simultaneously. As a result, several threads can write data into the same memory position at the same time (*race condition*), and consequently it has to be controlled to get right results.

Algorithm 7.2 Introducing OpenMP in the `grid` routine.

```
rho = 0.0
...
do ip = 1, npart_loc
   navg = gyroring_points(ip)
   ...
   do l = 1, navg
      (s,chi,phi) = gridcell(ip,l)
      ...
      do p = 1, 4
         (i,j,k) = gridvertex(s,chi,phi)
         rho(i,j,k) = rho(i,j,k) + ...
      enddo
   enddo
   ...
enddo
...
```

PARALLEL DO



Since the number of threads per node was small and the use of critical regions could serialize the execution, our decision was to avoid critical regions by creating private copies of the problematic data structure (`rho` array), concretely as many copies as threads per node. Once the loop is over, the private copies are merged into one structure. Obviously, if the memory amount is not enough to hold the copies, we should consider other options such as atomic operations or the split into different smaller critical regions.

Figure 7.6 depicts the time of one call to grid routine in a simulation with 50,000 particles on a $32 \times 32 \times 16$ grid using the three mentioned strategies to avoid the race conditions in the `rho` array. The test ran on a node (12 cores) of *MinoTauro*. The critical sections ensure that only one thread is able to read/write the same memory position at any time but suffer a huge overhead due to the synchronization cost as shown in figure. The atomic clauses avoid the synchronization overhead. The private copies of the problematic variables deliver the best time in the figure because they do not need any special treatment, and as a consequence threads do less work. Therefore, this figure demonstrates the superiority of our choice to use private copies to prevent race conditions.

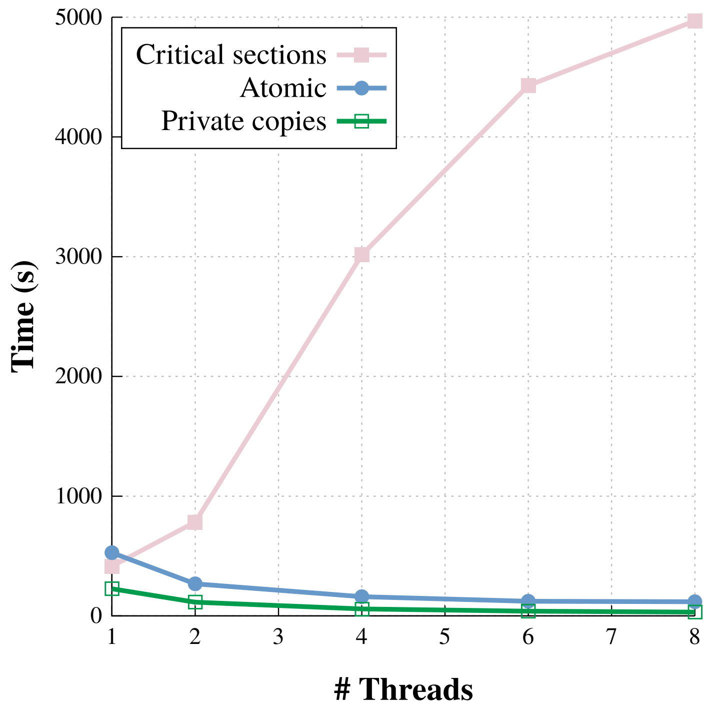


Figure 7.6: Comparison of strategies to prevent race conditions in the pull phase.

In order to test the performance of this implementation, we repeat the test executed in the push phase. Table 7.2 gives the time of the *pull phase* applied over all particles during 10 time steps on a node (4 cores) of *MareNostrum II*. Again the times provide a quite good scalability and this time `STATIC` is the best scheduling option.

# Threads	Time (s)			Speedup		
	static	dynamic	guided	static	dynamic	guided
1	**48.56**	48.53	48.58	**1.00**	1.00	1.00
2	**24.42**	38.40	26.38	**1.99**	1.26	1.84
3	**16.79**	31.00	19.15	**2.89**	1.57	2.54
4	**12.74**	28.52	15.77	**3.81**	1.70	3.08

Table 7.2: OpenMP version of the pull phase depending on the loop scheduling.

Finally, Listing 7.2 shows the implemented code with the private copy solution to prevent race conditions and the loop scheduling selected.

Listing 7.2 Schematic of the `grid` routine

```
   numthrds = OMP_GET_MAX_THREADS()
   ALLOCATE (rho_OMP(islw:isup, jclw:jcup, kplw:kpup, numthrds))
   rho_OMP(:,:,:,:) = 0.0
   rho(:,:,:) = 0.0
   ...
!$OMP PARALLEL DO                                                           &
!$OMP SCHEDULE (STATIC)                                                     &
!$OMP DEFAULT  (NONE)                                                       &
!$OMP PRIVATE  (ip, rt, zt, phit, vpart, wt, mut0, cosalp, pvol, s_gc,      &
!$OMP            chi_gc, b_abs, vperp2, u_bulk, ze_maxw, ztval_inv,         &
!$OMP            zvth2_inv, zf0, zphi_adi_val, za_par_adi_val, rhog, navg,  &
!$OMP            wght, wght_cur, zcos1, zsin1, l, rt_x, zt_x, st_x,         &
!$OMP            chit_x, k, k_loc, wphi, phase, i, ws, j, wchi, kc, kkc,    &
!$OMP            temp1, jc, jjc, temp2, zksi, xksi, yksi, thrid )           &
!$OMP SHARED   (npart_loc, pic1_loc, pic2_loc, twopinfp, zreduce_flag,      &
!$OMP            sgrid, ns, nsel_ubulk, msdmp, turbo_adi, Phi_bspl, qsde,   &
!$OMP            Apar_bspl, zcurr_flag, species, msdqs, zcos2, zsin2,       &
!$OMP            dphigrid_inv, phigrid, k_offset, dsgrid_inv, chigrid,      &
!$OMP            dchigrid_inv, rho_OMP, b_equ, nequ_zm1,                    &
!$OMP            nequ_rm1, nequ_z, nequ_r, iphipr, iphipl, nphip_min,       &
!$OMP            rgedz, rgedr, rphi_map, phi_edge, gez_max, gez_min,        &
!$OMP            ger_max, ger_min, nphip, kplw, jclw, islw, smax0,          &
!$OMP            me_cart, nidbas, ds_profiles_inv, t_s, ubulk_s, n_s, elmag)
   DO ip = 1, npart_loc
      thrid = OMP_GET_THREAD_NUM() + 1
      rt    = pic1_loc(R_PIC,ip)
      zt    = pic1_loc(Z_PIC,ip)
      phit  = pic1_loc(PHI_PIC,ip)
      ...
      rho_OMP(i,jjc,kkc,thrid) = rho_OMP(i,jjc,kkc,thrid) + temp2*ws(j)
      ...
   END DO
!$OMP END PARALLEL DO
   ...
   DO ii = 1, numthrds
      rho(:,:,:) = rho(:,:,:) + rho_OMP(:,:,:,ii)
   ENDDO
   DEALLOCATE (rho_OMP)
```

7.4.3 Solver Phase

Most of the time spent in the `poisson` routine is consumed in a call to the `petsc_solve` function. This function solves the Poisson equation and belongs to the Portable, Extensible Toolkit for Scientific Computation (PETSc) library, which is a well-known package for solving Partial Differential Equations on parallel machines.

Unfortunately, at that time PETSc v3.0.0 was not thread-safe, which means its routines could not guarantee safe execution by multiple threads at the same time.

As we wanted to develop a hybrid version of EUTERPE by introducing threads into the processes, this involved having a hybrid version of the solver that was able to take advantage of threads. Therefore, we decided to develop our own version of the solver and this is what we are going to describe it in next section.

7.5 A Thread-safe Solver

A *solver* is a computing routine that “solves” a mathematical problem. In our case, the problem to solve is a linear system of equations:

$$Ax = b \tag{7.1}$$

consisting of discretized partial differential equations, where A is the coefficient matrix $(n \times n)$, b is the right-hand side vector and x is the vector of unknowns.

Direct methods for solving the Equation 7.1 can be prohibitively expensive when the matrix A is large and sparse because the result of factorization can be dense, and as a result, a considerable amount of memory and computing time would be required.

In contrast, *iterative methods* begin with an initial guess for the solution (x_0) and successively improve it (x_k) until the solution is as accurate as desired. The evaluation of an iterative method focuses on how quickly the x_k values converge to the exact solution and what is the cost of each iteration.

7.5.1 Matrix A

A *sparse matrix* is a matrix in which most of the elements are zero. The *density* of a matrix is the fraction of nonzero elements over the total number of elements. In our case, the matrix A is large and sparse with a sparsity in the range of 0.05% to 1%, as shown in Figure 7.7.

The storage model and the operations used in dense matrices are inefficient and too slow in a sparse matrix because of the large amount of zero elements. In order to take

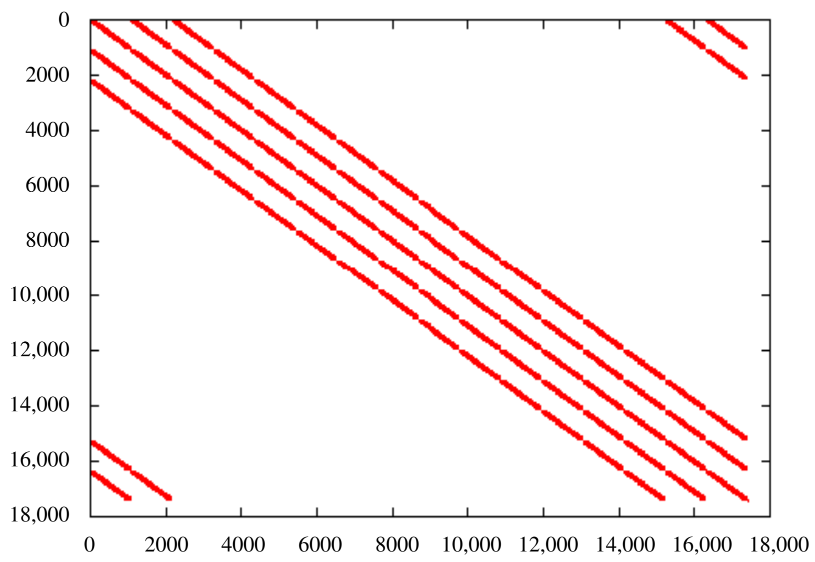


Figure 7.7: A sparse matrix (*density* $= 0.7$ %) provided by EUTERPE in a test with a small grid $32 \times 32 \times 16$. The red dots are nonzero elements.

advantage of this large number of zero elements, our main goal was to represent only the nonzero elements and be able to perform the common matrix operations.

For this reason, we chose probably one of the most popular schemes to store sparse matrices, the *Compressed Sparse Row* (CSR) format. The data structure of this format to store a matrix A of size $n \times n$ consists of three arrays (Fig. 7.8):

- AA, a real array containing all the values of the nonzero elements a_{ij} of A in the order by row and then ordered by column. The length of AA is N_z, which is the number of nonzero elements.
- JA, an integer array containing the column indices of the elements a_{ij} as stored in the array AA. The length of JA is N_z.
- IA, a second integer array containing the pointers to the beginning of each row in the arrays AA and JA. Therefore, the content of $IA(i)$ is the position where the i-th row starts in the arrays AA and JA. The length of IA is $n + 1$.

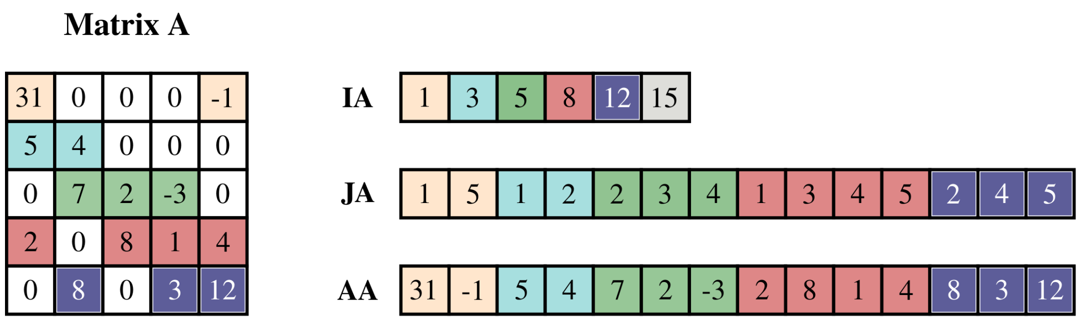


Figure 7.8: Example of a matrix A stored in the Compressed Sparse Row (CSR) format.

7.5.2 Jacobi Preconditioned Conjugate Gradient

Among the iterative methods, the *Conjugate Gradient* is an algorithm for the numerical solution of systems of n linear equations whose matrix (A) is symmetric and positive-definite. Each iteration of the algorithm requires only a single Sparse Matrix-Vector Multiplication (SpMV) and a small number of dot products. The storage requirements are also very modest, since vectors can be overwritten. In practice, this method often converges in far fewer than n iterations.

However, the conjugate gradient can still converge very slowly if the matrix A is ill-conditioned. Then, the convergence can often be accelerated by *preconditioning*, which consists in including a new matrix P (named *preconditioner*) and rewriting the system (Eq. 7.1) as follows:

$$P^{-1}Ax = P^{-1}b \tag{7.2}$$

where P is a non-singular matrix so that systems of the form $Pz = y$ are easily solved, and the inverse of P approximates that of A, so that $P^{-1}A$ is better conditioned. This means the choice of P is crucial in order to obtain a fast converging iterative method.

The *Jacobi preconditioner* is one of the most popular forms of preconditioning, in which P is a diagonal matrix with diagonal entries equal to those of A. The advantages of this preconditioner are the facility of its implementation and the low amount of memory that it needs.

Therefore, the Conjugate Gradient with the Jacobi preconditioner, called *Jacobi Preconditioned Conjugate Gradient (JPCG)* method, was our point of departure and Algorithm 7.3 lists the steps of this method.

Algorithm 7.3 Jacobi Preconditioned Conjugate Gradient Algorithm.

```
1: $x_0 \leftarrow 0; k \leftarrow 0$
2: $P \leftarrow diag(A)$
3: $r_0 \leftarrow b$
4: $z_0 \leftarrow P^{-1} r_0$
5: $p_0 \leftarrow z_0$
6: **while** $\frac{\|r_k\|}{\|b\|} > tol$ **do**
7:     $\alpha_k \leftarrow \frac{r_k^t z_k}{p_k^t A p_k}$
8:     $x_{k+1} \leftarrow x_k + \alpha_k p_k$
9:     $r_{k+1} \leftarrow r_k - \alpha_k A p_k$
10:    $z_{k+1} \leftarrow P^{-1} r_{k+1}$
11:    $\beta_{k+1} \leftarrow \frac{z_{k+1}^t r_{k+1}}{z_k^t r_k}$
12:    $p_{k+1} \leftarrow z_{k+1} + \beta_{k+1} p_k$
13:    $k \leftarrow k + 1$
14: **end while**
```

The first step was to implement a sequential version of Algorithm 7.3 and check that it worked well. Some improvements were applied. For example, some steps of the algorithm (from 3 to 5 and from 8 to 10) were grouped together into loops to take advantage of the data reusing and to facilitate the introduction of OpenMP in the code later. Also, as a division is more expensive than a multiplication, the division of step 6 (inside the loop) was substituted for a multiplication of the inverse of the norm of b (`inormB`, which is computed outside the loop). By last, as P^{-1} is used inside the loop, the inverse of P is computed outside the loop and is stored in `invDiag` to be used in steps 4 and 10.

Algorithm 7.4 depicts this intermediate sequential version. The numbers on the left side indicate what step of Algorithm 7.3 produces the result of that line. The Sparse Matrix-Vector Multiplication (SpMV) operation is implemented as a subroutine, and the matrix A is accessed considering the CSR format, so the `ia`, `ja` and `aa` vectors are benefited from being accessed contiguously.

Algorithm 7.4 Sequential version of the new solver.

```
     do ii = 1, lcRows
 3:       r(ii) = b(ii)
 4:       z(ii) = r(ii) * invDiag(ii)
 5:       p(ii) = z(ii)
          inormB = inormB + b(ii)*b(ii)
          c = c + z(ii)*r(ii)
      enddo
     inormB = 1.0 / sqrt (inormB)

 6:  do while ((err > tolerance) .and. (iter < iterMax))
        call spmv (p, z)
        pz = 0.0
        do ii = 1, lcRows
           pz = pz + p(ii) * z(ii)
        enddo
 7:     alpha = c / pz
        d = 0.0
        normR = 0.0
        do ii = 1, lcRows
 8:        x(ii) = x(ii) + alpha*p(ii)
 9:        r(ii) = r(ii) - alpha*z(ii)
10:        z(ii) = r(ii) * invDiag(ii)
           normR = normR + r(ii)*r(ii)
           d = d + z(ii)*r(ii)
        enddo
        normR = sqrt (normR)
11:     beta = d/c
        do ii = 1, lcRows
12:        p(ii) = z(ii) + beta*p(ii)
        enddo
        c = d
        iter = iter + 1
 6:     err = normR*inormB
      enddo
```

```
SUBROUTINE spmv (in b(:), out c(:))

  c = 0.0

  do ii = 1, lcRows
     c(ii) = 0.0
     do jj = ia(ii), ia(ii+1)-1
        c(ii) = c(ii) + aa(jj) * b(ja(jj))
     enddo
  enddo
```

In order to get parallelism at thread level in the solver, OpenMP was introduced into all the internal loops of the algorithm as Algorithm 7.5 below illustrates. The only dependencies among iterations came from the dot products (variables `inormB`, `c`, `normR` and `d`) and they were solved with reductions.

Algorithm 7.5 OpenMP version of the new solver.

```
     !$OMP PARALLEL DO                                  &
     !$OMP SCHEDULE (STATIC)                            &
     !$OMP PRIVATE  (ii)                                &
     !$OMP SHARED   (b, invDiag, lcRows, p, r, z)       &
     !$OMP REDUCTION (+: c, inormB)
     do ii = 1, lcRows
 3:       r(ii) = b(ii)
 4:       z(ii) = r(ii) * invDiag(ii)
 5:       p(ii) = z(ii)
          inormB = inormB + b(ii)*b(ii)
          c = c + z(ii)*r(ii)
      enddo                                   PARALLEL DO

     inormB = 1.0 / sqrt (inormB)

 6:  do while ((err > tolerance) .and. (iter < iterMax))
        call spmv (p, z)
        pz = 0.0

     !$OMP PARALLEL DO                &
     !$OMP SCHEDULE (STATIC)          &
     !$OMP PRIVATE  (ii)              &
     !$OMP SHARED   (lcRows, p, z)    &
     !$OMP REDUCTION (+: pz)
        do ii = 1, lcRows
          pz = pz + p(ii) * z(ii)
        enddo                         PARALLEL DO

 7:     alpha = c / pz
        d = 0.0
        normR = 0.0

     !$OMP   PARALLEL DO                                         &
     !$OMP   SCHEDULE (STATIC)                                   &
     !$OMP   PRIVATE  (ii)                                       &
     !$OMP   SHARED   (alpha, invDiag, lcRows, p, r, x, z)       &
     !$OMP   REDUCTION(+: d, normR)
        do ii = 1, lcRows
 8:        x(ii) = x(ii) + alpha*p(ii)
 9:        r(ii) = r(ii) - alpha*z(ii)
10:        z(ii) = r(ii) * invDiag(ii)
           normR = normR + r(ii)*r(ii)
           d = d + z(ii)*r(ii)
        enddo                                       PARALLEL DO

        normR = sqrt (normR)
11:     beta = d/c

     !$OMP   PARALLEL DO                     &
     !$OMP   SCHEDULE (STATIC)               &
     !$OMP   PRIVATE  (ii)                   &
     !$OMP   SHARED   (beta, lcRows, p, z)
        do ii = 1, lcRows
12:        p(ii) = z(ii) + beta*p(ii)
        enddo                         PARALLEL DO

        c = d
        iter = iter + 1
 6:     err = normR*inormB
      enddo
```

```
SUBROUTINE spmv (in b(:), out c(:))

  c = 0.0

 !$OMP PARALLEL DO                                &
 !$OMP SCHEDULE (STATIC)                          &
 !$OMP DEFAULT  (NONE)                            &
 !$OMP PRIVATE  (ii, jj)                          &
 !$OMP SHARED   (aa, b, c, ia, ja, lcRows)
 do ii = 1, lcRows
    c(ii) = 0.0
    do jj = ia(ii), ia(ii+1)-1
       c(ii) = c(ii) + aa(jj) * b(ja(jj))
    enddo
 enddo                               PARALLEL DO
```

The loop scheduling was set to `STATIC` for all the loops, because the code inside each loop was the same for all iterations, and thus the workload per iteration was identical. But there is an exception, the Sparse Matrix-Vector Multiplication (SpMV), because the internal loop may do a different number of iterations since each row can have a different number of nonzero elements.

If we take a look at the appearance of the matrix provided by EUTERPE, we will see that it is a sparse matrix with a distribution similar to a band matrix (Fig. 7.9). This appearance is due to the domain decomposition strategy applied in EUTERPE, which in this test is equivalent to slice a cylinder in the toroidal direction. Well then, this distribution involves that each row has identical number of nonzero elements.

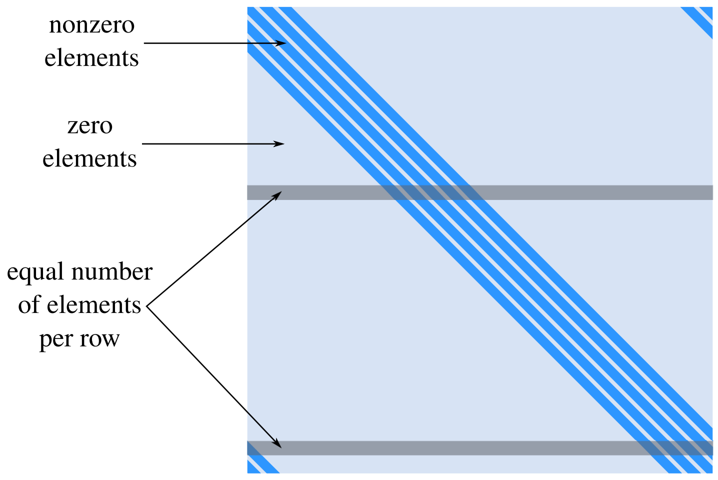
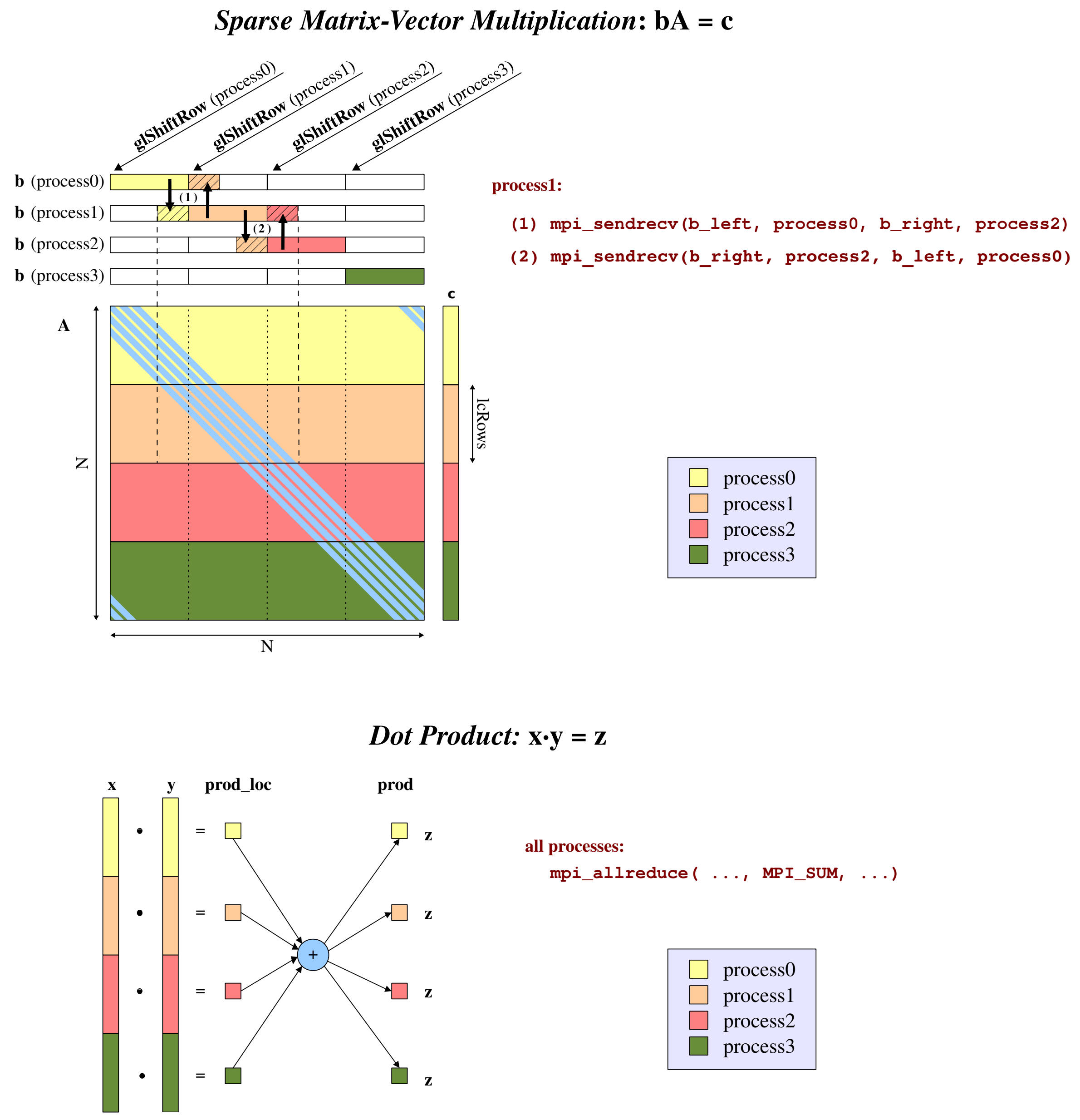


Figure 7.9: Aspect of the matrix to solve provided by EUTERPE.

As a consequence, if all rows have the same number of nonzero elements, then the workload per iteration is identical, and therefore the best scheduling for the external loop of the SpMV routine is `STATIC`.

The next step was to parallelize the solver through *processes* using MPI to get a hybrid version. The vectors were divided in as many pieces of consecutive positions as processes. Each process stored the same respective piece of all vectors. Concerning the matrix, this was divided in blocks of consecutive rows and these blocks were distributed among processes. Figure 7.10 shows graphically how matrix and vectors are distributed among 4 processes (the colour indicates in which process is stored each piece of data).



process1:

```
(1) mpi_sendrecv(b_left, process0, b_right, process2)
(2) mpi_sendrecv(b_right, process2, b_left, process0)
```

all processes:

```
mpi_allreduce( ..., MPI_SUM, ...)
```

Figure 7.10: Domain decomposition in MPI.

If the size of the global field matrix is $(N \times N)$ as in Figure 7.10, each process holds a local matrix of size $(lcRows \cdot N)$, where $lcRows$ is N divided by the number of processes. Vectors are also distributed in pieces of $lcRows$ elements and each one belongs to a process. Of particular note is the vector p whose content is distributed among all processes although each process allocates N positions for it. In this manner, the elements received from adjacent processes for the SpMV computation can be stored into vector p and its access does not add any conversion cost. To exploit all shared memory on a node, each process is assigned to a node.

Moreover, Figure 7.10 also shows the communication patterns for each iteration of the solver. In the SpMV, as the matrix has bands of elements (the blue lines are the positions with nonzero values), each process asks to the adjacent processes the needed data (e.g. `process1` will receive the striped fragments of vector `b` from `process0` and `process2`) via point-to-point communications (`mpi_sendrecv`) to accomplish the computation.

The communication protocol to prevent deadlocks is as follows: firstly each process sends data to the left process and receives data from the right process, and secondly each process sends data to the right process and receives data from the left process.

In the dot products, as vectors are distributed into the processes, each process computes the local dot product and sends the result to all other processes via a broadcast communication (`mpi_allreduce`). Then, each process computes the total dot product with the dot products from the other processes.

The final result of mixing the two programming models in one code is shown in Algorithm 7.6, where the new MPI calls are in bold and the OpenMP pragmas only include the new modifications from Algorithm 7.5.

Table 7.3 contains a comparison between our version (JPCG) and the PETSc version used by EUTERPE. For its elaboration we executed a test on *MareNostrum II* with a grid of $32 \times 64 \times 64$ cells and 1 million particles, and measured only the time of the `solver` routine. When the solver used only MPI, we assigned one MPI process per core (or processor), whereas when the solver used the hybrid version, we assigned one MPI process per node and one OpenMP thread per core (i.e. 4 threads/process). In comparing the results, we can check that the two implementations have a similar performance, but our solver is also thread-safe.

Algorithm 7.6 Hybrid version (OpenMP+MPI) of the new solver.

```
     !$OMP PARALLEL DO                  &
     !$OMP PRIVATE (jj, ...)            &
     !$OMP SHARED  (glShiftRow, ... )   &
     !$OMP ...
     do ii = 1, lcRows
          jj = glShiftRow + ii
 3:       r(ii) = b(ii)
 4:       z(ii) = r(ii) * invDiag(ii)
 5:       p(jj) = z(ii)
          inormB = inormB + b(ii)*b(ii)
          c = c + z(ii)*r(ii)
      enddo                        PARALLEL DO

     mpi_sendrecv (p_left, process-1, p_right, process+1)
     mpi_sendrecv (p_right, process+1, p_left, process-1)
     mpi_allreduce (inormB, c, SUM)
     inormB = 1.0 / sqrt (inormB)

 6:  do while ((err > tolerance) .and. (iter < iterMax))
        call spmv (p, z)
        pz = 0.0

     !$OMP PARALLEL DO                  &
     !$OMP PRIVATE (jj, ...)            &
     !$OMP SHARED  (glShiftRow, ... )   &
     !$OMP ...
       do ii = 1, lcRows
          jj = glShiftRow + ii
          pz = pz + p(jj) * z(ii)
       enddo                       PARALLEL DO

        mpi_allreduce (pz, SUM)
 7:     alpha = c / pz
        d = 0.0
        normR = 0.0

     !$OMP PARALLEL DO                  &
     !$OMP PRIVATE (jj, ...)            &
     !$OMP SHARED  (glShiftRow, ...)    &
     !$OMP ...
        do ii = 1, lcRows
           jj    = glShiftRow + ii
 8:        x(ii) = x(ii) + alpha*p(jj)
 9:        r(ii) = r(ii) - alpha*z(ii)
10:        z(ii) = r(ii) * invDiag(ii)
           normR = normR + r(ii)*r(ii)
           d = d + z(ii)*r(ii)
        enddo                          PARALLEL DO

        mpi_allreduce (normR,d, SUM)
        normR = sqrt (normR)
11:     beta = d/c

     !$OMP PARALLEL DO                  &
     !$OMP PRIVATE (jj, ...)            &
     !$OMP SHARED  (glShiftRow, ...)
        do ii = 1, lcRows
           jj    = glShiftRow + ii
12:        p(jj) = z(ii) + beta*p(jj)
        enddo                          PARALLEL DO

        mpi_sendrecv (p_left, process-1, p_right, process+1)
        mpi_sendrecv (p_right, process+1, p_left, process-1)
        c = d
        iter = iter + 1
 6:     err = normR*inormB
      enddo
```

```
SUBROUTINE spmv (in b(:), out c(:))

 c = 0.0

!$OMP PARALLEL DO                              &
!$OMP ...
do ii = 1, lcRows
   c(ii) = 0.0
   do jj = ia(ii), ia(ii+1)-1
      c(ii) = c(ii) + aa(jj) * b(ja(jj))
   enddo
enddo                              PARALLEL DO
```

One data that may attract your attention is the modest speedup achieved by our version and the PETSc version (about 7 on 16 processors). The most time-consuming part in the `solver` routine is the SpMV, where the memory access cost exceeds the computation cost. This is due to a low data reuse and a lack of floating-point operations to hide the elapsed time in memory operations. Therefore, we are in front of a *memory bound problem*, or in other words, the limiting factor is the memory access speed. This explains the performance shown by PETSc and by our new solver.

Solver	**# Cores**	**# Processes**	**# Threads/Process**	**Time (s)**	**Speedup**
PETSc	1	1	—	237.60	1.00
(MPI)	2	1	—	204.77	1.16
	4	1	—	131.07	1.81
	8	1	—	72.56	3.27
	16	**1**	—	**34.68**	**6.85**
JPCG	1	1	—	287.42	1.00
(MPI)	2	1	—	171.53	1.68
	4	1	—	131.40	2.19
	8	1	—	68.32	4.21
	16	**1**	—	**35.64**	**8.06**
JPCG	1	1	1	284.24	1.00
(Hybrid)	1	1	2	157.44	1.81
	1	1	3	139.51	2.04
	4	1	4	131.69	2.16
	8	2	4	87.27	3.26
	16	**4**	**4**	**37.28**	**7.62**

Table 7.3: Comparison between the JPCG and PETSc solvers.

Later, we saw that there were improvement opportunities in the convergence, in particular by reducing the number of iterations required to achieve a result close enough to the exact solution. In this manner, fewer iterations would imply less work and a shorter execution time, and therefore we decided to implement and evaluate two variants of conjugate gradient method as described below.

7.5.3 Block Jacobi Preconditioned Conjugate Gradient

The *Block Jacobi Preconditioned Conjugate Gradient (BJPCG)* is a method that uses a *Block Jacobi matrix* as a preconditioner. When the matrix A results from the discretization of the Poisson equation, this preconditioner has the advantage that the condition number of $P^{-1}A$ is smaller than for the *Jacobi preconditioner*. The smaller condition number almost always means faster convergence of the method [80].

A *Block Jacobi matrix* is a matrix whose entries are partitioned into sub-matrices or blocks. All blocks are zero except those on the main block diagonal which are equal to those of the matrix A, as depicted in Figure 7.11.

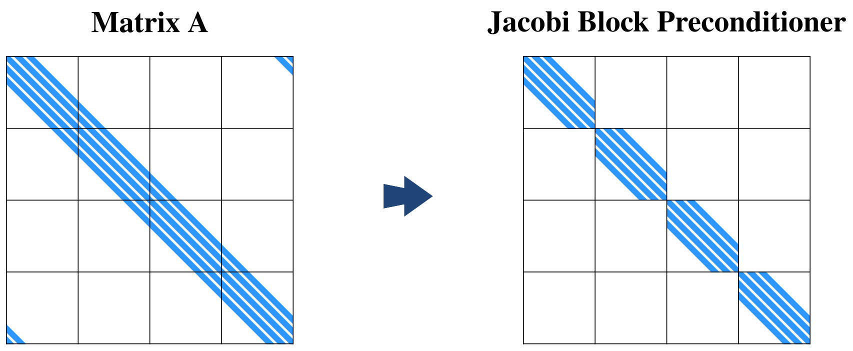


Figure 7.11: Example of a Block Jacobi preconditioner.

We implemented the BJPCG, because although this method implies more workload per iteration, on the other hand it reduces the number of iterations to find the solution, and as a result, the convergence is much faster than using JPCG.

One example that more workload exists is the computation of $P^{-1}r_0$ and $P^{-1}r_{k+1}$ in steps 4 and 10 of Algorithm 7.3, respectively. When P was the Jacobi preconditioner, the multiplication between P and a vector was especially simple since P was a diagonal matrix. There was only one scalar multiplication per row, as one can see in Algorithm 7.4. Now, in Algorithm 7.7, as P is a Block Jacobi preconditioner, there is a multiplication between a small sub-matrix and a vector which implies more operations than above.

In order to confirm that the convergence was much faster with a Block Jacobi preconditioner than with a Jacobi preconditioner, we built a synthetic matrix similar to one provided by EUTERPE and we solved the system with a tolerance of 10^{-5} using JPCG and BJPCG with different block sizes. The size of the matrix was 640×640 with

Algorithm 7.7 Hybrid version (OpenMP+MPI) of the BJPCG solver.

```
 4:  call jacobian_mv (b, z)

     !$OMP PARALLEL DO                &
     !$OMP ...
     do ii = 1, lcRows
          jj = glShiftRow + ii
 3:       r(ii) = b(ii)
 5:       p(jj) = z(ii)
          inormB = inormB + b(ii)*b(ii)
          c = c + z(ii)*r(ii)
      enddo                                PARALLEL DO

     mpi_sendrecv (p_left, process-1, p_right, process+1)
     mpi_sendrecv (p_right, process+1, p_left, process-1)
     mpi_allreduce (inormB, c, SUM)
     inormB = 1.0 / sqrt (inormB)

 6:  do while ((err > tolerance) .and. (iter < iterMax))
        call spmv (p, z)
        pz = 0.0

     !$OMP PARALLEL DO                &
     !$OMP ...
       do ii = 1, lcRows
          jj = glShiftRow + ii
          pz = pz + p(jj) * z(ii)
       enddo                               PARALLEL DO

        mpi_allreduce (pz, SUM)
 7:     alpha = c / pz
        d = 0.0
        normR = 0.0

     !$OMP PARALLEL DO                &
     !$OMP ...
        do ii = 1, lcRows
           jj    = glShiftRow + ii
 8:        x(ii) = x(ii) + alpha*p(jj)
 9:        r(ii) = r(ii) - alpha*z(ii)
        enddo                              PARALLEL DO

10:     call mult_jacobian_matrix (r, z)

     !$OMP   PARALLEL DO              &
     !$OMP   SCHEDULE  (STATIC)       &
     !$OMP   PRIVATE   (ii)           &
     !$OMP   SHARED    (lcRows, r, z) &
     !$OMP   REDUCTION (+: d, normR)
        do ii = 1, lcRows
           normR = normR + r(ii)*r(ii)
           d = d + z(ii)*r(ii)
        enddo                              PARALLEL DO

        mpi_allreduce (normR,d,SUM)
        mpi_allreduce (d, SUM)
        normR = sqrt (normR)
11:     beta = d/c

     !$OMP PARALLEL DO                &
     !$OMP ...
        do ii = 1, lcRows
           jj    = glShiftRow + ii
12:        p(jj) = z(ii) + beta*p(jj)
        enddo                              PARALLEL DO

        mpi_sendrecv (p_left, process-1, p_right, process+1)
        mpi_sendrecv (p_right, process+1, p_left, process-1)
        c = d
        iter = iter + 1
 6:     err = normR*inormB
      enddo
```

```
SUBROUTINE jacobian_mv (in b(:), out c(:))

  !$OMP PARALLEL DO                                &
  !$OMP SCHEDULE(STATIC)                           &
  !$OMP PRIVATE (ii, jj, kk)                       &
  !$OMP SHARED  (lcRows, c, rowsBlock, b, bj)
     do ii = 1, lcRows
        c(ii) = 0
        kk = ((ii-1)/rowsBlock)*rowsBlock
        do jj = 1, rowsBlock
           c(ii) = c(ii) + b(kk+jj)*bj(ii,jj)
        enddo
     enddo                                 PARALLEL DO
```

71,120 nonzero elements. The synthetic matrix and the involved preconditioners are plotted in Figure 7.12.

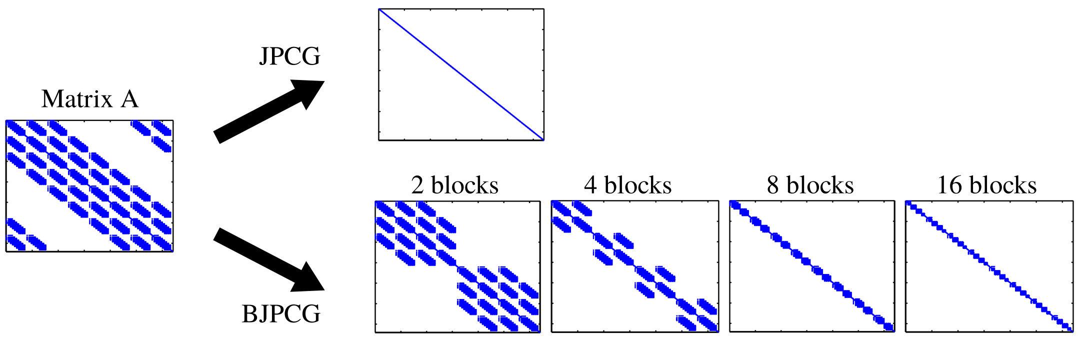


Figure 7.12: Examples of preconditioner matrices.

The results of the experiment are shown in Figure 7.13. The graphic gives the tolerance of all iterations for each preconditioning method, and also shows the number of iterations required to achieve the initial desired tolerance. As can be seen, the convergence with BJPCG is faster than with JPCG at all variants. In addition, the BJPCG with 8 blocks is the best option, because from there the number of iterations increases as the number of blocks grows.

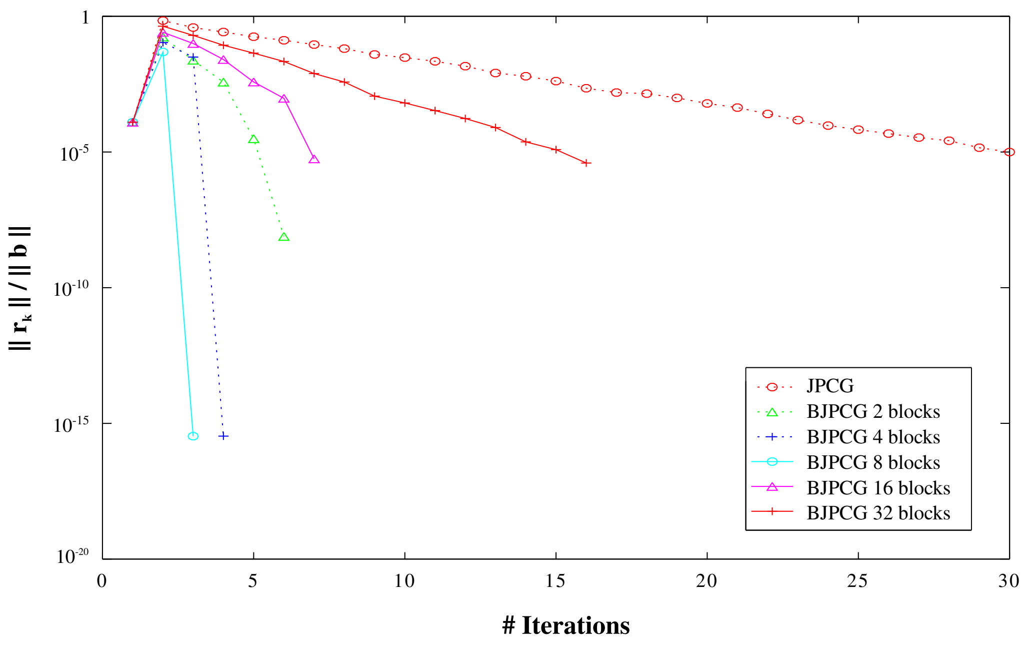


Figure 7.13: Comparison of the convergence between JPCG and BJPCG.

Finally, to test the scalability intra-node of this version, we executed the hybrid version of EUTERPE with a BJPCG of 64 blocks using 128 processes on *Jugene*. The simulation moved 10^9 particles on a grid of $16 \times 256 \times 256$ cells. Table 7.4 give us an idea of the good scalability of the new solver, which will be studied in the next section.

# Threads	Time (s)	Speedup
1	215.15	1.00
2	112.30	1.92
4	66.70	3.23

Table 7.4: Execution times of EUTERPE using the BJPCG solver.

7.5.4 Deflated Preconditioned Conjugate Gradient

The *Deflated Preconditioned Conjugate Gradient* was implemented in order to reduce the number of iterations [81]. A deflation method is used to accelerate the convergence of the Preconditioned Conjugate Gradient (PCG) and it is done using a region-based decomposition of the unknowns.

The main concept is to remove components of the initial residual which may improve convergence. This technique has shown a considerable reduction in the number of iterations, but it depends on the kind of problem. Although it was a step forward in the reduction of the number of iterations, the reduction did not compensate the extra work of a more complex preconditioner. As a result, it was not integrated into our code.

7.6 Evaluation of the Hybrid Version (MPI-OpenMP)

In this chapter, we have seen that the separate results in the push, pull and solver phases are encouraging: firstly, the performance with one thread is comparable to the sequential version and, secondly, the thread version is speeding up as expected. Therefore, this section is going to focus on an analysis of the hybrid version as a whole.

We begin with a small case to perform a detailed analysis using the tracing tools. This test was run on *MareNostrum II* using 4 nodes as follows: one MPI process per node and four threads per MPI process, so in total 16 cores were employed, as shown in Figure 7.3.

Figure 7.14 shows graphically the behavior of the hybrid version of EUTERPE (MPI + OpenMP) on *MareNostrum II* using the Paraver tool. The names of the routines are specified on the top of the graphic: *push* (computes the motion of the particles), *solver* (solves the field equation) and *grid* (determines the charge density). The dark blue colour means that the thread is running and the light blue colour means that the thread is idle. According to this figure, we deduce that the work is well balanced and the threads work most of the time. Only a few sections of the code (indicated in the figure) do not yet have OpenMP, hence they are executed by a single thread in each process while the rest of threads are shown as idle. As a consequence, such regions of code are candidates to expand the introduction of OpenMP in a future work.

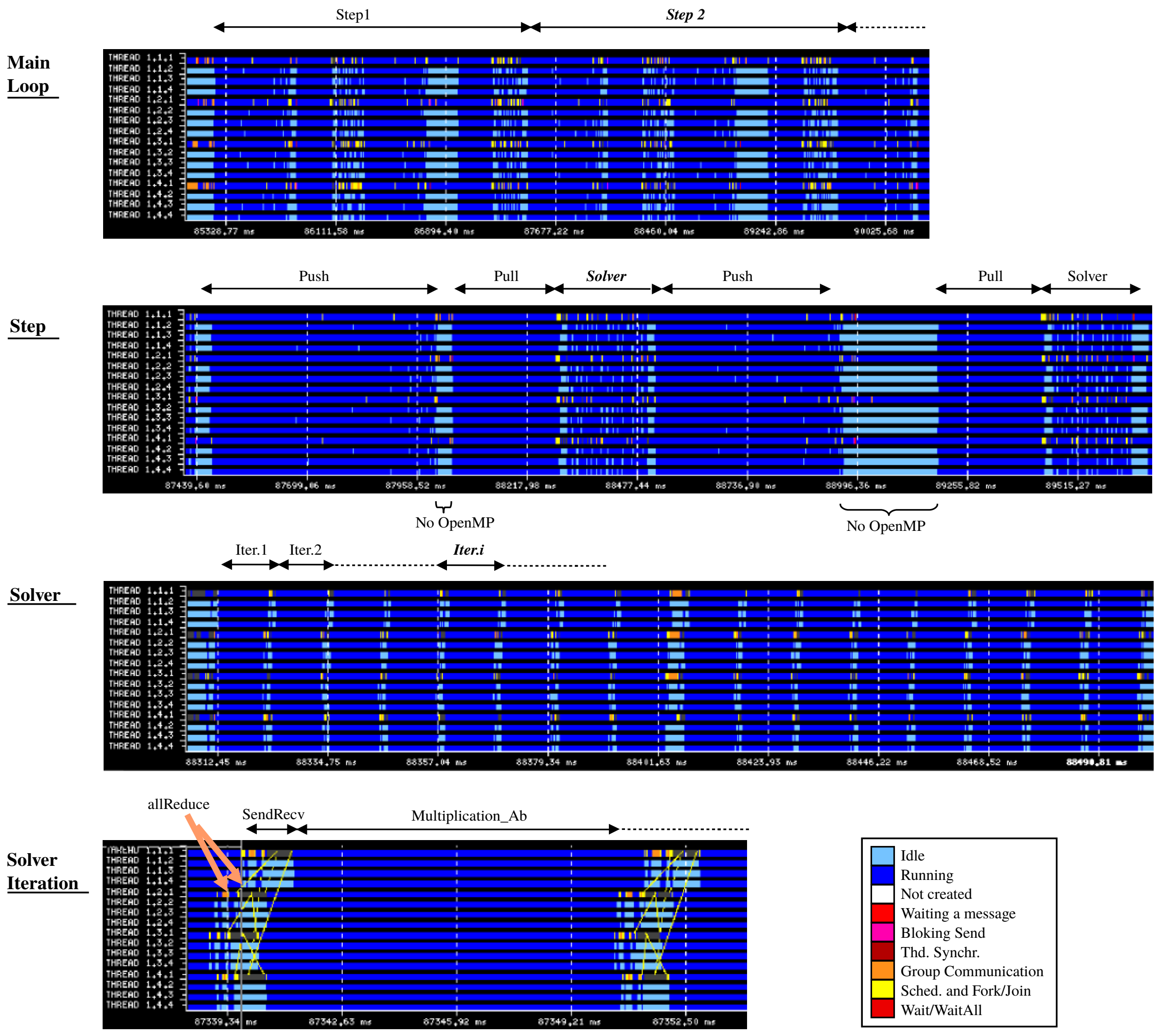


Figure 7.14: Trace of one time iteration of the hybrid version on *MareNostrum II*.

Moreover, we have observed that this scheme of execution repeats across similar simulations on different machines. As example, Figure 7.15 shows an execution of the hybrid version on *Huygens* with a different work distribution: 4 processes and 32 threads per each process. The result is equivalent and we can reach the same conclusions.

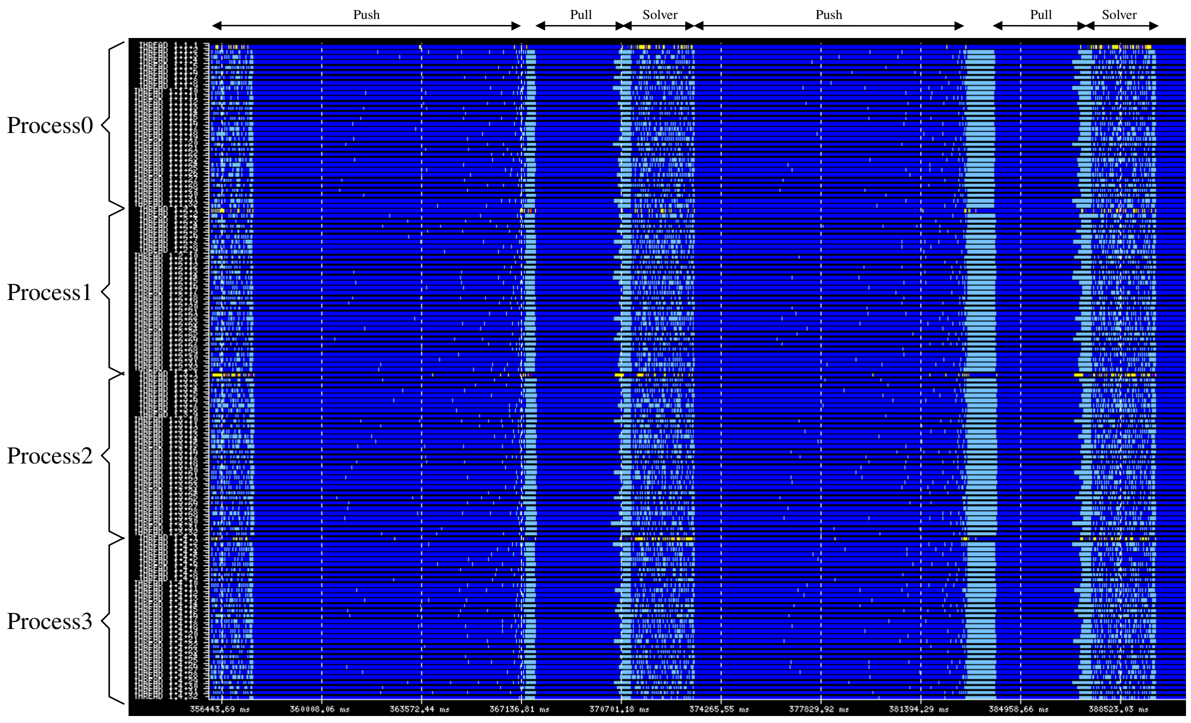


Figure 7.15: Trace of the hybrid version on *Huygens*.

Our next goal was to check the behaviour of the application together with one of the distinctive features of EUTERPE. The domain cloning, as explained in Section 6.3.3, allows us to run on more processors than other PIC codes using domain decomposition.

To evaluate the performance of our hybrid version in a more comprehensive way, several tests were performed varying: the block size of BJPCG from 8 to 9,216, the number of clones from 1 to 8, and the number of threads from 1 to 4. The simulations ran on *Jugene* and consisted of 10^9 particles and a cylinder with a grid of $16 \times 256 \times 256$ cells that provided a matrix with 1,179,648 rows to solve. The measured times for one time step are presented in Table 7.5.

Looking at this table, one can observe that clones improve the execution time of the hybrid version for any number of threads and blocks. Also, when one increases the number of blocks, the execution time improves considerably up to 128 blocks. After that number of blocks, the improvement is really minor.

Node mode	**# Blocks**	**Step time (s)**			
		128 processes 1 clone	**256 processes 2 clones**	**512 processes 4 clones**	**1,024 processes 8 clones**
VN	8	101.60	24.23	11.15	5.50
1 thrd/process	16	77.13	22.14	10.75	5.39
	32	50.97	21.31	10.52	5.30
	64	43.06	20.85	10.34	5.29
	128	**41.81**	**20.48**	**10.31**	**5.28**
	256	40.80	20.44	10.30	—
	512	40.71	20.41	—	—
	1,024	40.63	—	—	—
	9,216	40.62	—	—	—
DUAL	8	36.72	11.86	5.90	2.91
2 thrd/process	16	27.81	11.72	5.70	2.84
	32	23.30	11.32	5.58	2.80
	64	22.47	11.08	5.49	2.79
	128	**21.97**	**10.89**	**5.47**	**2.78**
	256	21.59	10.85	5.46	—
	512	21.53	10.84	—	—
	1,024	21.52	—	—	—
	9,216	21.49	—	—	—
SMP	8	20.28	7.28	3.62	1.79
4 thrd/process	16	14.50	7.14	3.42	1.71
	32	14.20	6.76	3.31	1.67
	64	13.39	6.58	3.24	1.67
	128	**12.93**	**6.37**	**3.22**	**1.66**
	256	12.62	6.35	3.22	—
	512	12.55	6.33	—	—
	1,024	12.51	—	—	—
	9,216	12.51	—	—	—

Table 7.5: Hybrid version using domain cloning strategy.

There are two more things in this table that merit our attention: the entries with a dash and the maximum number of processors achieved.

Our algorithm requires that all blocks have the same number of rows. When an entry in the table does not satisfy this condition, then this entry shows a dash. For example, in the case of 256 blocks and 1,024 processes, there are $1,179,648\ rows/1,024\ processes = 1,152\ rows/process$, and these rows have to be distributed in 256 blocks, so $1,152\ rows/$

256 $blocks = 4.5\ rows/block$. As the result is not an integer number, the number of rows per block will not be the same at all blocks, and therefore the entry in the table will have a dash.

Regarding the number of processors, we made use up to 1,024 processes with 4 threads/process, i.e. we made use up to 4,096 processors. To evaluate this number, an execution of the original EUTERPE version is needed in the same conditions (Table 7.6). The same test is repeated using only processes. The node mode selected on *Jugene* was `SMP` because all the memory of a node was required to store the data of a process. All other modes produced insufficient memory errors with this test.

Node mode	**Step time (s)**			
	128 processes 1 clone	**256 processes 2 clones**	**512 processes 4 clones**	**1024 processes 8 clones**
SMP (4 thrd/process)	38.02	18.45	9.62	4.93

Table 7.6: Original EUTERPE version using PETSc solver.

Under such conditions, the original version could achieve 1,024 processes using 1,024 nodes, i.e. although only 1,024 processors were working, in fact 4,096 processors were occupied. So, even though our times with the single thread execution are a little worse (but in the same line as original), when the original EUTERPE is compared against the 2 and 4 threads executions, our results are better because we are exploiting all the available cores.

Nevertheless, a better way to show the contribution made by this new hybrid version is comparing the Figure 7.16 and Figure 7.17. Figure 7.16 shows that if we take into account the number of working processors, the hybrid version has a similar performance to the original EUTERPE version. The performance achieved is almost the original. The reason is because the scalability of MPI version is excellent, and there are parts of the code that are not parallelized with OpenMP, so only one thread per process is working in those parts. Also, there is a small overhead related to the extra work due to the thread management.

In addition to this, Figure 7.17 shows that the hybrid version exploits all the hardware when the use of all the memory on a node is required. The original version of EUTERPE only could use processes and this limited its performance. Now, with the hybrid version, a process can use all the memory of a node and can create threads to use all the cores of

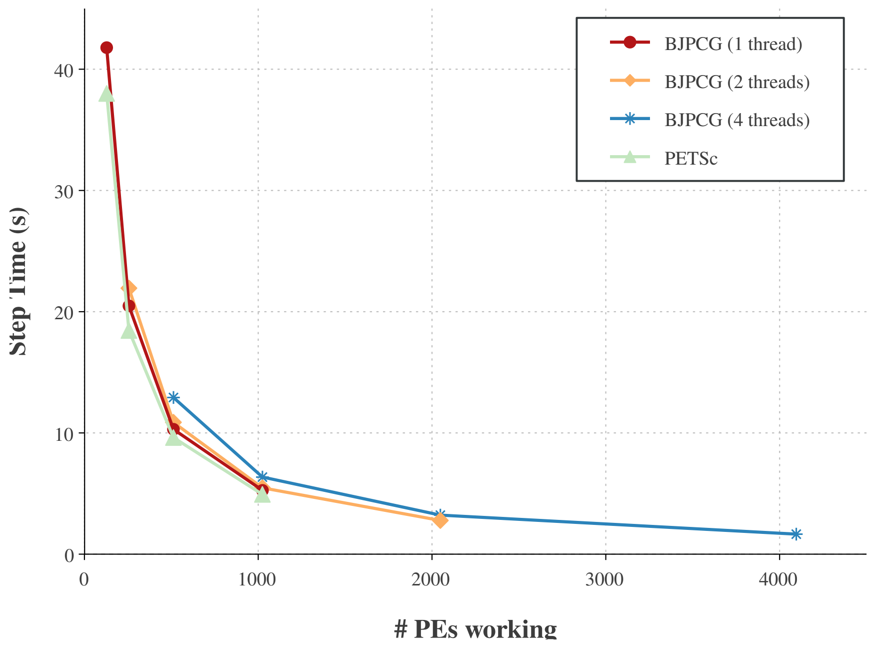


Figure 7.16: Comparison between the hybrid version (with BJPCG of 128 blocks) and the original version of EUTERPE in terms of the number of PEs running.

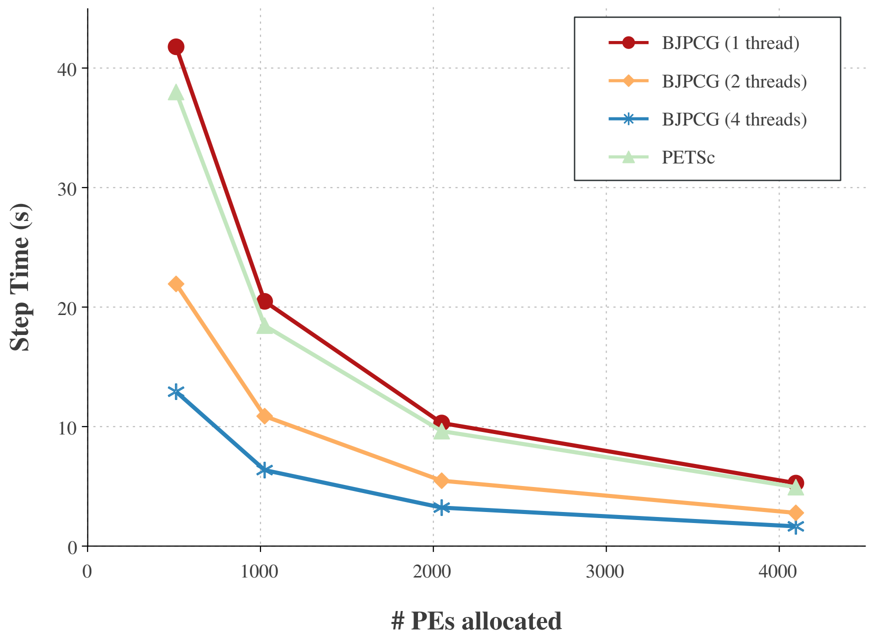


Figure 7.17: Comparison between the hybrid version (with BJPCG of 128 blocks) and the original version of EUTERPE in terms of the number of PEs occupied.

a node. Therefore, the figure shows that the hybrid version improves the time compared to the original version.

Finally, Figure 7.18 confirms that the scalability of the hybrid version is good because its curves are nearly linear.

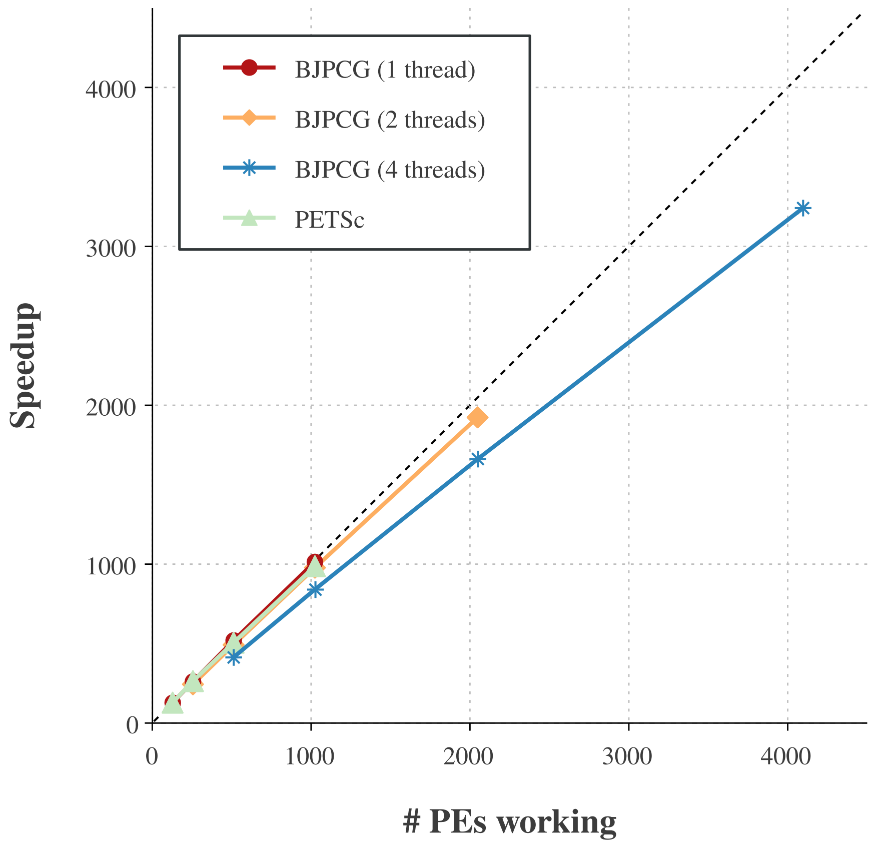


Figure 7.18: Speedups of the hybrid version (with BJPCG of 128 blocks) and the original version of EUTERPE (with PETSc) in terms of the number of PEs used.

7.7 Conclusion

Nowadays the new challenges in fusion simulations have encouraged developers to adapt their codes for massively parallel architectures introducing OpenMP in their codes, but when this work was developed this trend was still unusual. The hybrid version presented in this chapter represented in those days a significant improvement of the code with a similar behavior of the original EUTERPE code.

One example offering such a benefit with respect the original version is that some simulations, which can not use all cores on a machine because memory per core is too small to fit a MPI process with its data, now can be executed with the hybrid version since each MPI process is assigned to a node with access to all the memory of the node and all its cores running threads.

Since most PIC codes follows basically the same schema, described in Algorithm 3.1, the implementation of the methodology followed in the studied code (EUTERPE) is straightforward to apply to others similar codes. Afterwards, I also have successfully applied this methodology outside the thesis to another code which is not related with fusion area.

Although OpenMP is the easiest way to parallelize a code, because we simply add pragmas to the loops, really it is hard to parallelize all the code because there are regions without loops. Furthermore, it is difficult to hide the communications between the processors because the communication is implicit, not like MPI where the communication is explicitly.

Even though the scalability of EUTERPE has been proven to be good, more work will be needed to maintain this performance level in the next High-Performance Computing (HPC) platforms with multi-core nodes and million of cores in total. For example, extending the introduction of OpenMP inside more routines with less weight in the list of most-time consuming routines.

Chapter 8

Heterogeneous Parallelism

"We stand at the threshold of a many core world. The hardware community is ready to cross this threshold. The parallel software community is not."

Principal engineer at INTEL
TIM MATTSON (2011)

In this chapter, we will explore the feasibility of porting a Particle-In-Cell (PIC) code to a heterogeneous system based on two specific experiences. The first case is the adaptation of a solver used in EUTERPE to a Cell platform. The second case is more extensive and explains the porting of basic kernels of a PIC code to a more modern heterogeneous platform which contains ARM processors.

The work done in this chapter resulted in the following publications:

[18] **X. Sáez, A. Soba, E. Sánchez, M. Mantsinen, S. Mateo, J.M. Cela and F. Castejón**, First experience with particle-in-cell plasma physics code on ARM-based HPC systems, *Journal of Physics: Conference Series* **640**, 012064 (2015)

[19] **X. Sáez, A. Soba, and M. Mantsinen**, Plasma physics code contribution to the Mont-Blanc project, in *Book of abstracts of the 2nd BSC International Doctoral Symposium*, pp. 83–84, Barcelona, Spain (2015)

[20] **X. Sáez, A. Soba, E. Sánchez, M. Mantsinen, and J. M. Cela**, Performance analysis of a particle-in-cell plasma physics code on homogeneous and heterogeneous HPC systems, in *Proceedings of Congress on Numerical Methods in Engineering (CNM)*, pp. 1–18, APMTAC, Lisbon, Portugal (2015)

8.1 Introduction

During the last two decades, supercomputers have grown rapidly in performance to provide scientists the required computing power at the cost of a similar growth in power consumption. The most used metric for evaluating supercomputers performance has been the speed of running benchmark programs, such as LINPACK [82].

But nowadays, the performance of the computer is limited by power consumption and power density. Energy has become one of the most expensive resources and, as a consequence, the energy expenses in a High-Performance Computing (HPC) center can easily exceed the cost of infrastructure after a few years in operation. Therefore, energy efficiency is already a primary concern for the design of any computer system and will define the maximum achievable performance. This change of the point of view is reflected in the increasing popularity of the Green500 list [83], in which supercomputers are ranked in terms of power efficiency (Fig. 8.1).

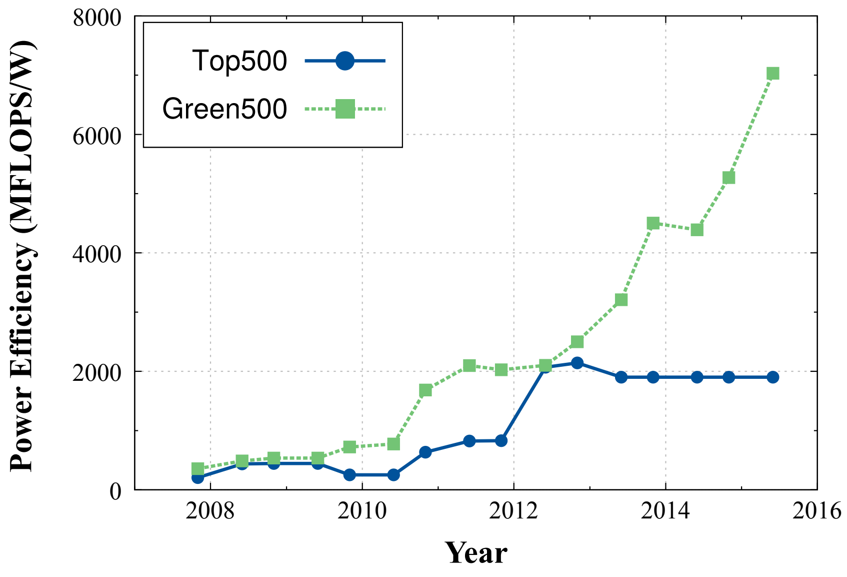


Figure 8.1: Comparison of power efficiency between the leaders of the TOP500 and Green500 lists.

Looking to the future, new platforms designs for a sustainable exaflop supercomputer will have to be based on the power efficiency. The *Mont-Blanc project* (as explained in Section 4.4) has the aim to design a computer architecture capable of delivering an exascale performance using 15 to 30 time less energy than the present architectures. The reduction of energy consumption is achieved by developing an HPC prototype based on the energy-efficient technology originally designed for mobile and embedded systems.

In order to obtain the best possible results on the prototype, all the available resources have to be used by the application: the Central Processing Unit (CPU) multi-core, using Open Multi-Processing (OpenMP) version developed previously, and the Graphic Processing Unit (GPU), using a new Open Computing Language (OpenCL) version.

8.2 Implementation of a Solver on Cell

To the best of our knowledge, neither sparse kernel nor sparse library had been ported to the Cell processor (described in Section 4.3.1) when we developed this work. Nevertheless, there were some studies in the area [56] exploring different possibilities for efficient methods in multi-core architectures.

8.2.1 Sparse Matrix-Vector Multiplication (SpMV)

The most demanding part of the solvers developed in Section 7.5 is the SpMV. Therefore, we decided to start the porting to Cell with this routine, which is shown in Listing 8.1. As we explained in Section 7.5, we worked with the Compressed Sparse Row (CSR) format, which is a common way to store the sparse data.

Listing 8.1 The SpMV routine.

```
for (i = 0 ; i < n ; i++)
{
   c[i] = 0;
   for (j = ia[i] ; j < ia[i+1] ; j++)
   {
      c[i] = c[i] + aa[j] * b[ja[j]];
   }
}
```

In our implementation, Power Processing Element (PPE) acted as the execution coordinator and intended to take part as less as possible in the computation. The responsibility of Synergistic Processing Elements (SPEs) was to catch coordination messages and launch computation routines while informing the PPE of what they were doing.

The first step was getting the amount of nonzero elements from the sparse matrix and the number of SPE threads that Cell was able to launch. Next, PPE split the sparse matrix data as equally as possible among the available SPE threads.

Data was transferred between main memory and each Local Store (LS) by Direct Memory Access (DMA) instructions. The Cell architecture imposed multiple alignment and size constraints on data transfers. SPEs could transfer only from/to 16 byte boundary. We found that it was more efficient to split the matrix elements not only when a new matrix row began, but also when it did in an aligned memory position. This condition made the code less complex, but it required extra code to check the alignments and assure that transfers would not corrupt any memory position.

Access to the right-hand vector (`b` array) with double indexation (`b[ja[i]]`) had direct implications on the splitting method because DMA transfers from main memory were easier to control when data was in consecutive positions since only first position and size were required. We developed two different approaches to face the double indexation issue depending on who prepared the elements of the right-hand vector (`b` array).

PPE Gatherer

The PPE was responsible for preparing a working array with the elements of the right-hand vector (`b` array) in a way that they matched with the nonzero matrix elements for the multiplication. The working array was split in the same way as the matrix. The DMA transfers of this working array supplied the elements from the right-hand vector in the manner that they corresponded one by one with the matrix elements, and as a result SPEs could start computation as soon as the transfers were completed.

Figure 8.2 shows full execution time of three calls to the SpMV routine. The top line is the PPE thread and the lines below correspond to the eight SPE threads launched after matrix has been loaded. It is obvious that SPEs suffer from starvation because light blue color means idle and therefore the expected performance is poor. We can summarize the major bottleneck in this implementation is the PPE.

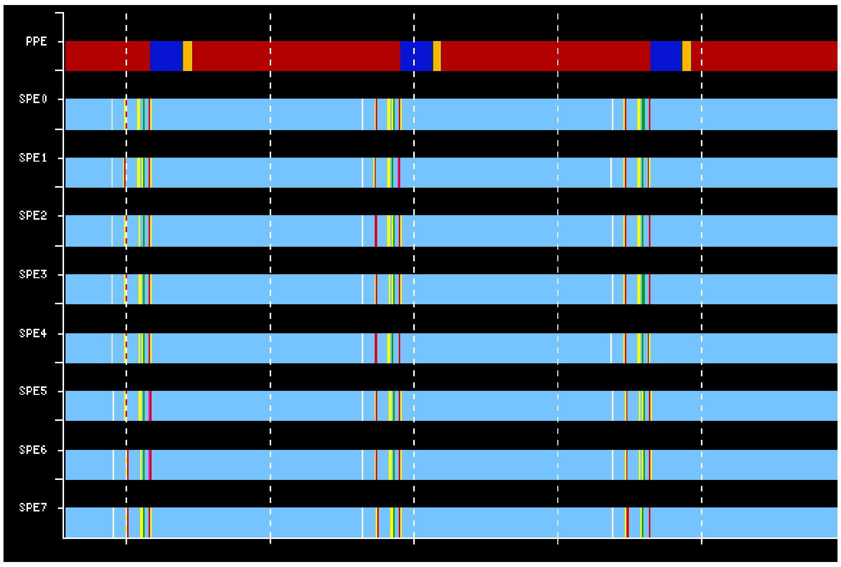


Figure 8.2: Trace view for three calls to the SpMV routine using the "PPE gatherer".

SPE Gatherer

The goal was to remove the PPE scalar loop that prepared the working array before any SpMV call. The solution kept the right-hand vector as it was in memory and the SPE was in charge of transferring its assigned chunk (element by element). We took into consideration several methods to reach the best efficiency on multiple small DMA transfers, but finally we decided to implement a variation of the method described in [84].

This time SPEs got elements of the right-hand vector through a DMAlist. A new data structure was required in the main memory to contain the addresses of the right-hand vector and their memory alignments (0 or 8). This new structure was due to a memory constraint in SPE when less than 16 bytes were moved via DMA transfers, in this case the data kept the same alignment on main memory and on LS. The structure was computed once by the PPE and SPEs were free to check it before each transfer.

The procedure was as follows. Firstly, SPEs transferred addresses and alignments. Secondly, each SPE configured the DMAlist with the addresses and launched it to get the element values. Since each element had its own main memory alignment, it was impossible to get them consecutive from main memory. In our implementation, SPEs allocated double the amount of memory than the transfer represented and each element occupied bytes 0-7 or 8-15 without conflicts. Next, a compression method took the alignments and removed the gaps between elements.

Figure 8.3 contains three iterations of this implementation and confirms that now PPE thread has not gather time and SPE are now performing the gather step and the SpMV. The white color shows that SPE is working most of time due to this new approach.

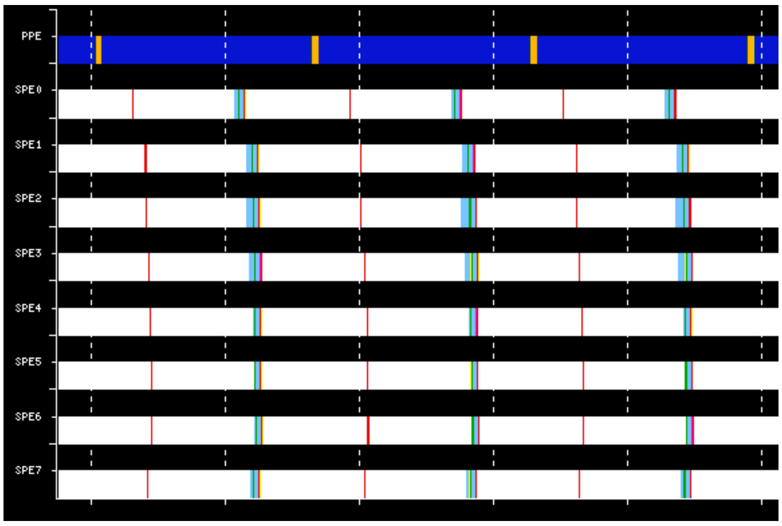


Figure 8.3: Trace view for three calls to the SpMV routine using the "SPE gatherer".

8.2.2 Evaluation

We have introduced our implementation of the SpMV routine into a conjugate gradient solver to test its performance.

Figure 8.4 and Figure 8.5 present time and speedup of this solver with two different matrix sizes (0.47M and 3.5M of nonzero elements) on one Cell processor of *MariCel*. Both graphics show that the implementation with SPE gatherer performs better although it increases significantly the complexity of the SPE and PPE codes.

Figure 8.6 shows the time computing per iteration with a matrix of 4.5M nonzero elements. After comparing the results, the Cell processor (*Maricel*) is unable to reach the performance of the PowerPC processor (*Marenostrum II*).

To sum up, although we made real efforts to exploit all the resources that the Cell processor brought to us, a good performance was not reached. Our humble impression is that this architecture was conceived for dense computing, such as video-games, but scientific computing requires a good performance with sparse data. Therefore, the constraints of the Cell processor did not allow to provide a good performance with sparse data and also made the programming a tedious task.

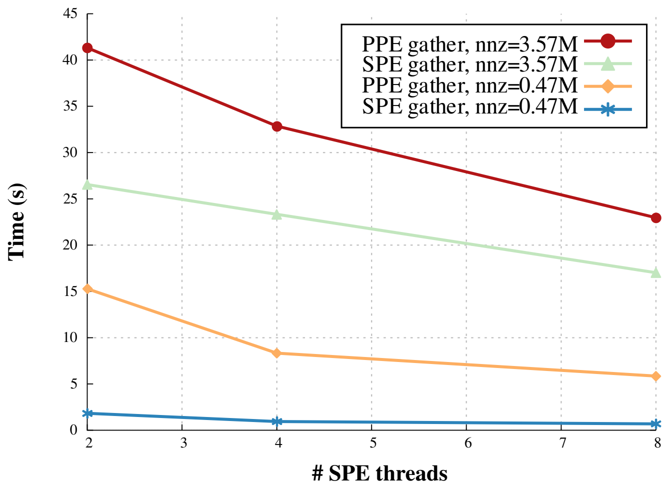


Figure 8.4: Time for the two gather implementations on *MariCel*.

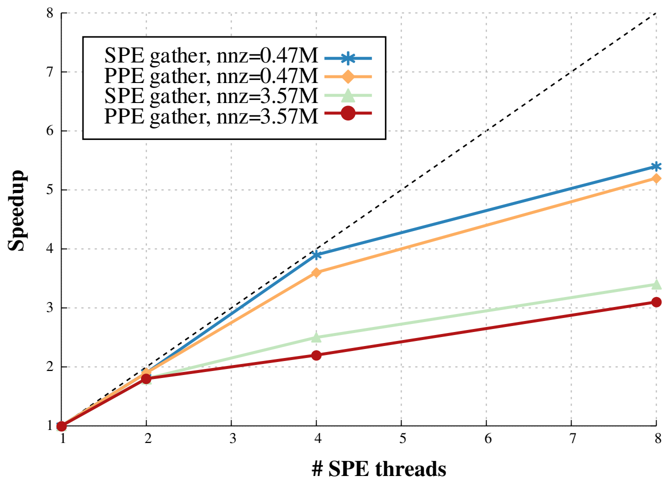


Figure 8.5: Scalability for the two gather implementations on *MariCel*.

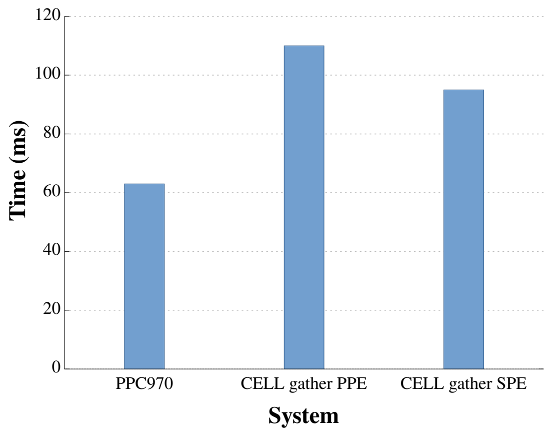


Figure 8.6: Time per iteration on single processor between *MariCel* and *MareNostrum II*.

8.3 OpenCL

Open Computing Language (OpenCL) is an open royalty-free standard for general purpose parallel programming across heterogeneous processing platforms. It gives software developers a portable and efficient access to the power of these platforms. This standard has been developed by Khronos Group [85].

It defines an Application Programming Interface (API) that provides a homogeneous view of all computational resources through the following abstractions: the platform model, the memory model, the execution model and the programming model. It also supports data-based and task-based parallel programming models.

OpenCL divides the computing system into a host (CPU) and a set of compute devices (CPU, GPU or another accelerator). Moreover, OpenCL also divides the application in two parts:

- **host code**, which sets up the environment, sends kernels and arguments to the compute devices and receives the results from kernels.
- **kernels**, which compute portions of the application.

As Figure 8.7 shows, the *host code* runs on the host and, after setting up the environment, it dispatches the *kernels* and data to the compute devices. Each kernel is compiled and is built at runtime to adapt the executable to the compute device. Next, each compute device executes many instances of the same kernel (called *work-items*). The work-items are grouped by *work-groups* and the work-items inside a group share resources such as memory.

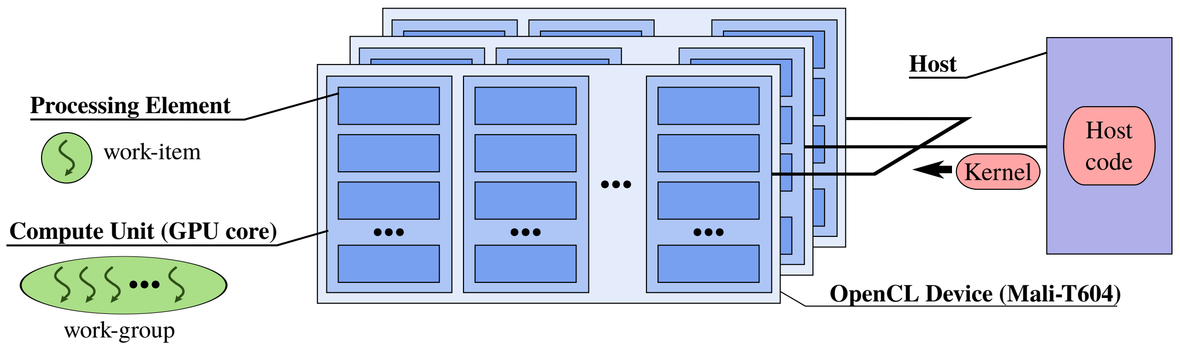


Figure 8.7: The platform and execution models of OpenCL.

8.4 Developing OpenCL Kernels

We will study here the Mali GPU contained in the Samsung Exynos 5 Dual (explained in Section 4.3.2) which is used in the *Arndale* prototype of the Mont-Blanc project. It has two special features which affect the common OpenCL programming in our case:

- Firstly, the Mali GPU has a unified memory system different from the usual physically separated memory components. The local and private memory are physically the global memory, so moving data from global memory to local or private memory typically does not improve performance.
- Secondly, all GPU threads have individual program counters. This means that all the threads are independent and can diverge without any performance impact. As a result, branch divergence is not a major issue.

In order to exploit the accelerator (Mali GPU) of the prototype, the routines that will run on the accelerator have to be written in OpenCL. This mission involves some new tasks to do: the translation of the code from FORTRAN to C, the distribution of work between work-items and the creation and initialization of the OpenCL components (context, kernels, command queues and memory objects).

As a result of the previous chapter, we know which sections in the code are the most compute intensive: pushing the particles (*push*) and depositing their charge (*pull*).

Our first purpose was to introduce OpenCL in these routines by minimizing the changes in comparison with the previous version. This strategy attempted to increase the productivity and decrease the maintenance cost, because scientists add modifications and upgrades inside the code with certain regularity, e.g. including more details into the simulations. This happens to EUTERPE which requires a prompt and efficient response to keep up-to-date the code.

8.4.1 Push Phase

In the *push* phase, the planning of the work distribution was simple: one work-item was assigned to each particle (Fig. 8.8). Since each work-item works on a different particle, all work-items write in different memory locations (like what is illustrated in the figure inside Algorithm 7.1 but replacing threads with work-items). Therefore, the adaptation of the routine was reduced to a straightforward translation.

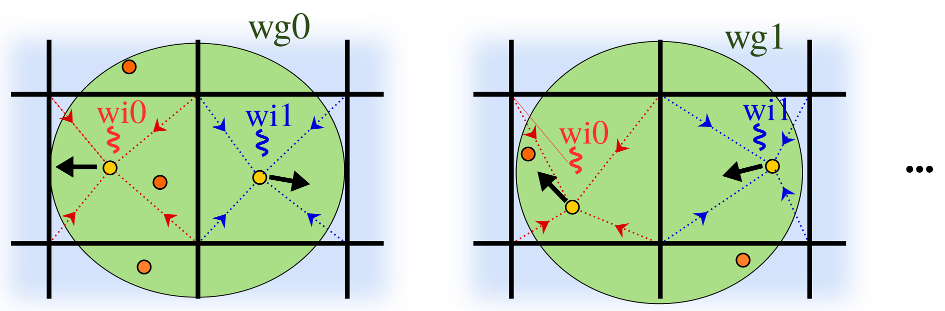


Figure 8.8: Representation of the Push kernel in OpenCL. Where wi_n are the work-items (in charge of particles), and wg_n represent the work-groups and their common data.

In order to see the behaviour of the `push` kernel, this time the test ran on the *Arndale* prototype. The simulation moved 1 million of particles in a grid with a resolution of $n_s \times n_\theta \times n_\phi = 32 \times 32 \times 16$. The work-group size was set to 32 work-items (it means that each work-group was in charge of 32 particles). It should also be noted that the data set was smaller than previous tests because of the lower amount of memory available in this prototype.

Figure 8.9 provides a timeline to show OpenCL calls on the host and on the compute device (all compute devices are grouped in just one line). The figure makes evident the significant additional work needed to configure the OpenCL environment (left of the figures), although luckily it only has to be paid once at the beginning. The data movement between the host and the compute devices is also included inside the configuration time, so it is really important to be careful with this data movement if we do not want to ruin the improvement achieved with the GPU.

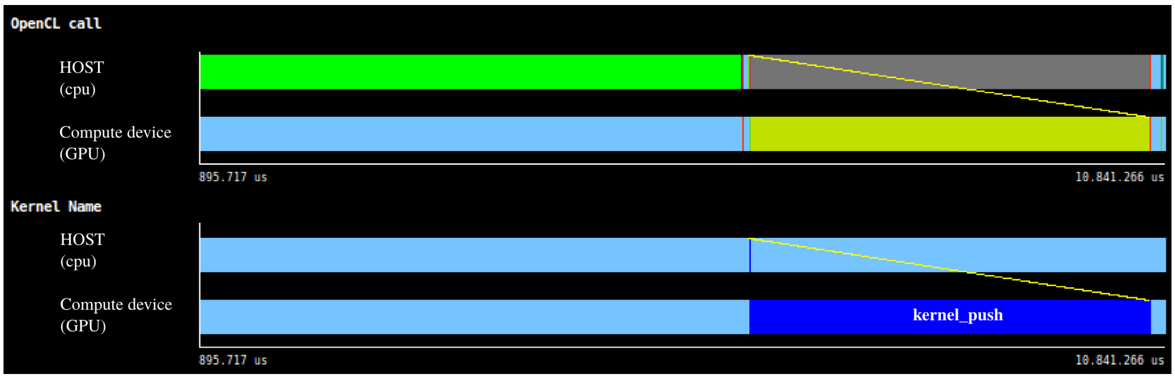


Figure 8.9: Timeline showing OpenCL calls and the kernel push running.

8.4.2 Pull Phase

The *pull* phase was more challenging to implement, since different particles can contribute to the charge density on the same grid point (Fig. 8.10). As we maintained the same work distribution as in the *push* phase, several threads could update the same memory location (similarly to what is illustrated in the figure inside Algorithm 7.2 but replacing the threads with work-items). To avoid these memory conflicts, the two previously mentioned solutions used for the OpenMP implementation (Section 7.4.2) did not work properly due to the high number of threads on a GPU: *atomic operations* became inefficient because of lock contention that serialized the execution, and *grid copies* required too much memory.

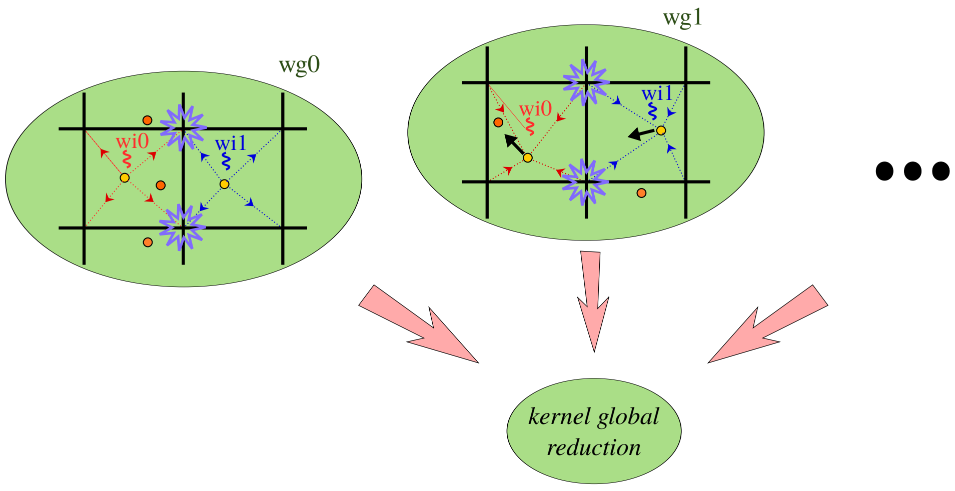


Figure 8.10: Representation of the two OpenCL kernels in Pull phase. Where wi_n are the work-items (in charge of particles), and wg_n represent the work-groups and their common data.

The strategy to solve this issue was inspired by the way in which OpenCL distributes data processing. A grid copy is created per work-group, so only the work-items in the same work-group accumulate contributions on the same grid copy. As a consequence, the lock contention is far minor and the reduction of memory conflicts makes feasible the use of atomic operations.

Nevertheless, a new step is required to reduce all these copies to the final grid. The final global reduction is minor and a single kernel is enough to perform it since the number of grid copies is limited by the number of work-groups. This time each work-item will collect the result of the same point in all the copies of the grid.

Finally, one last aspect to mention was an issue that happened during the elaboration of this OpenCL version. Some double-precision arithmetic operations were not properly defined in the OpenCL driver on Mali GPU, and this fact delayed the work until ARM managed to solve it.

In order to see the behaviour of the pull kernel, we performed again a test on *Arndale* prototype. The dataset for the simulation was the same as for the push routine.

Figure 8.11 provides a timeline of OpenCL calls on the host and on the compute devices. It also shows the calls to the `pull` and `global_reduct` kernels. Apart from what was commented above in Figure 8.9, we can add that the `global_reduct` kernel computes the global reduction of the charge density, and its execution time is much smaller than that of the `pull` kernel.

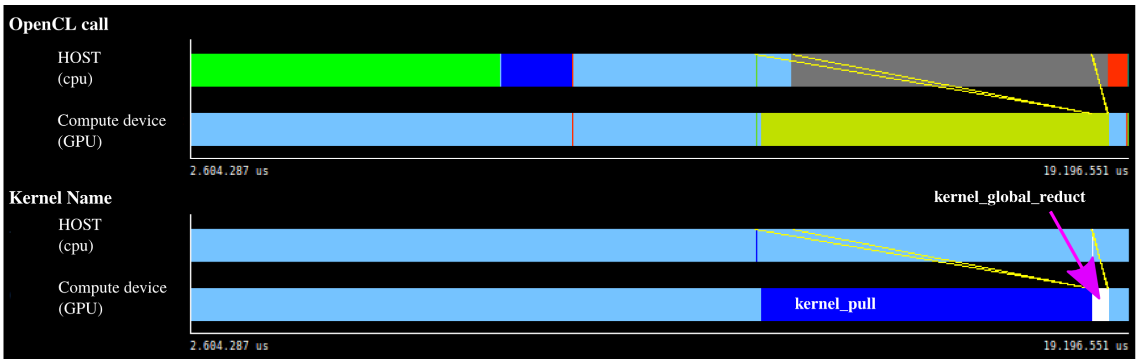


Figure 8.11: Timeline showing OpenCL calls and the pull and global_reduction kernels running.

8.5 Hybrid Version + OpenCL

The next step was to take advantage of all resources, so a convenient decision was to distribute the particles between CPU and GPU. Since the two devices shared the memory, all the particles were distributed between them without any additional data transfer. The particles on the CPU were computed using the OpenMP version, whereas the particles on the GPU were computed using the OpenCL version.

Figure 8.12 and Figure 8.13 show the results of the ARM hybrid version (OpenMP + OpenCL) on *Arndale*. The test contains 1 million particles. We can see that as more particles are sent to GPU, the execution time of kernels decreases. Being the optimal configuration when the particles sent to the GPU are 40 % in the *push* kernel and 60 %

in the *pull* kernel. At these minimum points, the combination of the CPU and GPU devices yields a modest but significant improvement over the CPU time only: about 30 % in the *push* kernel and near 50 % in the *pull* kernel.

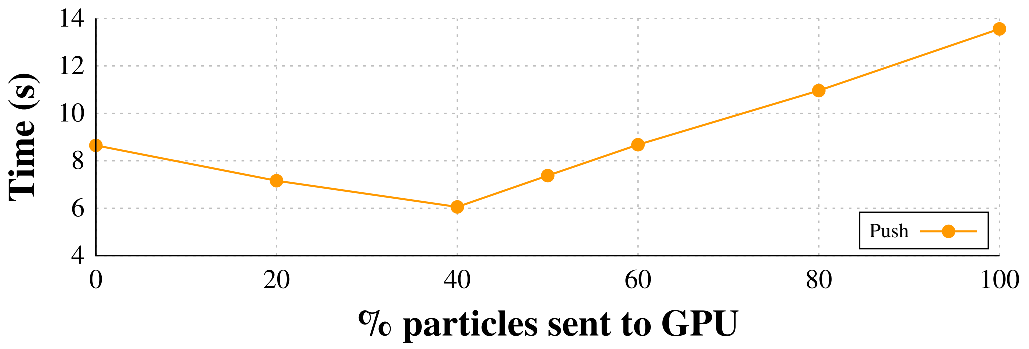


Figure 8.12: Execution time of the *push* kernel for different combinations of particles sent to the GPU and the CPU.

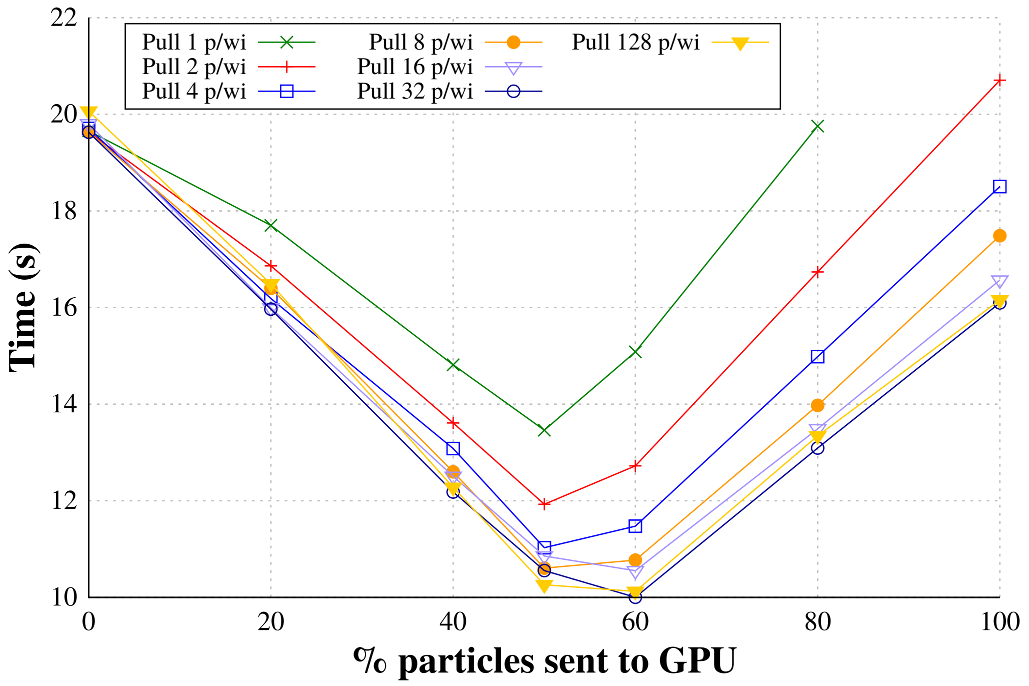


Figure 8.13: Execution time of the *pull* kernel for different combinations of particles sent to the GPU and particles per work-item (*p/wi*)

It would be interesting to analyse a trace of this hybrid version as we already did previously for other cases, but at the time of developing this work Extrae was not enable to create a trace for a hybrid version with OpenMP and OpenCL. Therefore the trace analysis has been left for a future work.

8.6 OmpSs Programing Model

When an application is ported to a certain platform, there are two issues of high importance: the possibility to reach the maximum performance of the platform and the easiness of programming it. The main drawback of OpenCL is the low programmability because it is a low-level programming language. Therefore, we will have to invest a lot of time to develop a code in OpenCL with a good performance, or in other words we can say that OpenCL offers a low productivity.

To address this shortcoming, we ported the code to *OpenMP Super-scalar (OmpSs)*[86] that is a task-based programming model developed at Barcelona Supercomputing Center (BSC). It provides an abstraction to the user which reduces programmer effort and unifies the Symmetric Multiprocessing (SMP), heterogeneous and cluster programming in one model.

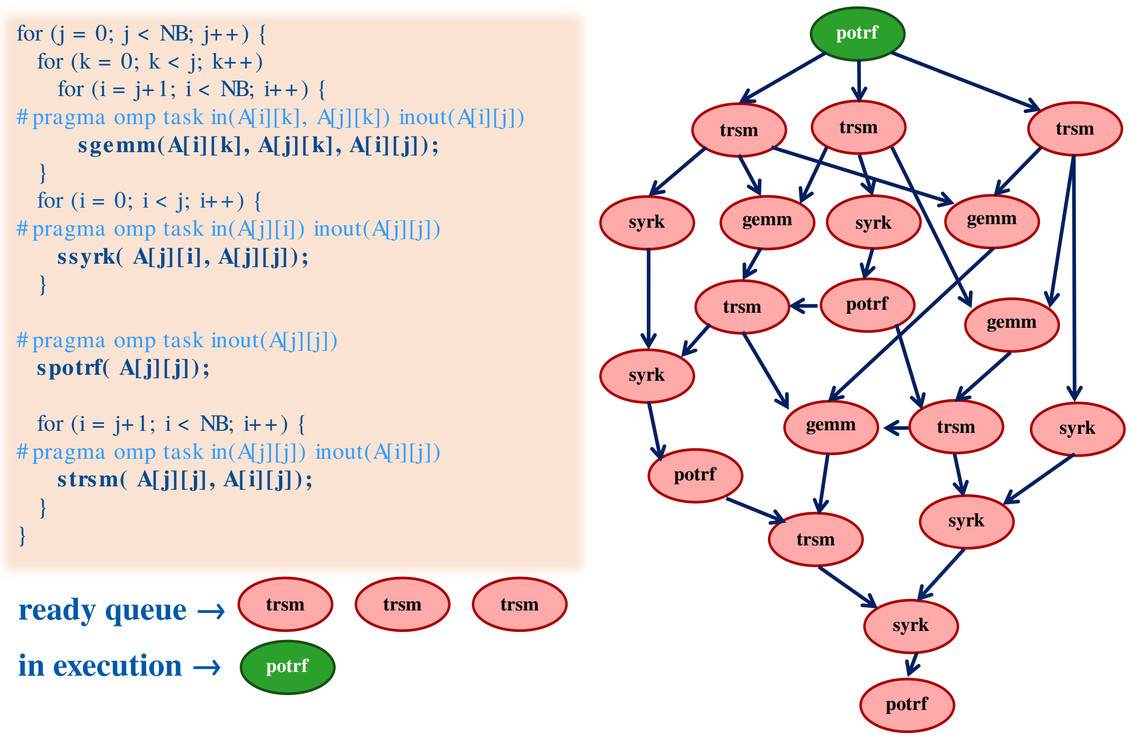
The programmer annotates the functions of the program to parallelize with the `TASK` pragma. This pragma can be complemented with the definition of data dependencies (what data are needed (`IN`), what data are produced (`OUT`) or both situations (`INOUT`)) and the device where the task has to be executed.

When the program runs, each call to a function with the `TASK` pragma creates a task and its in/out/inout dependencies are matched against of those of existing tasks. If one of the following data dependencies is found, then the task becomes a successor of the corresponding tasks:

- an input parameter is generated by a existing task (RAW),
- an output parameter is already produced by a existing task (WAW),
- an existing task has an input parameter that is the same as the output parameter of this task, so it can be overwritten (WAR).

This analysis process creates a task dependency graph at runtime (Fig. 8.14). Tasks are scheduled automatically for execution as soon as all their predecessors in the graph have finished. When tasks have no predecessors, then they are executed immediately.

There is one more point to explain, the `TARGET` pragma allows us to indicate on which device the task should be run (e.g. GPU, SMP, etc.). In this way, OmpSs will perform the data transfers among the tasks and the low level work required by OpenCL when the task is going to be executed on a GPU. This low level work can be summarized in



```
for (j = 0; j < NB; j++) {
  for (k = 0; k < j; k++)
    for (i = j+1; i < NB; i++) {
#pragma omp task in(A[i][k], A[j][k]) inout(A[i][j])
      sgemm(A[i][k], A[j][k], A[i][j]);
  }
  for (i = 0; i < j; i++) {
#pragma omp task in(A[j][i]) inout(A[j][j])
    ssyrk( A[j][i], A[j][j]);
  }

#pragma omp task inout(A[j][j])
  spotrf( A[j][j]);

  for (i = j+1; i < NB; i++) {
#pragma omp task in(A[j][j]) inout(A[i][j])
    strsm( A[j][j], A[i][j]);
  }
}
```

Figure 8.14: Example of an OmpSs code and the task scheduling [www.bsc.es].

creating the context and command queues, copying data to devices, compiling the kernel code at runtime and getting results from devices.

In addition, there is a modifier of the `TARGET` pragma called `IMPLEMENTS (orig)`, which specifies that the function is an alternative implementation for the target device of the function `orig`. As a result, a function can dispose of an implementation for GPU and an alternative for CPU, so runtime will decide how to distribute the tasks (work) among these devices to take advantage of both resources.

8.7 OmpSs+OpenCL Version

As we said above, some work has to be done to use OmpSs in the code. This process is called *taskification* and consists in reorganizing some parts of the code to introduce tasks.

The easiest way to do it is splitting any parallelizable loop in two nested loops. The internal loop repeats a block of iterations from the original loop, and the external loop performs as many iterations as blocks of iterations. The next step is to pack the internal loop into a function which is defined as `TASK`. Each time that the function is called by the external loop, a task is generated and is enqueued to run when its dependencies are complied.

8.7.1 Push Phase

The functioning of OmpSs to distribute the *push phase* among the compute units was tested with a simulation of 125,000 particles in a grid of $32 \times 32 \times 16$ cells. Each task was in charge of 128 particles, so the total number of tasks in this test was 977. The number of particles per task is a trade off: too few particles imply consuming too much time in managing the tasks, while too many particles imply insufficient tasks to distribute the work and achieve a load balance among compute devices.

Figure 8.15 shows the results obtained when executing the push phase. The top graph indicates the distribution of the OpenMP and OpenCL tasks within the CPUs and GPU, respectively. The red (or darker) colour means running tasks. The middle graph shows the evolution of the tasks in the execution and how the OmpSs runtime distributes the tasks. The green (or lighter) colour shows the first created tasks and the blue (or darker) colour shows the last ones. And the bottom graph depicts the number of enqueued tasks.

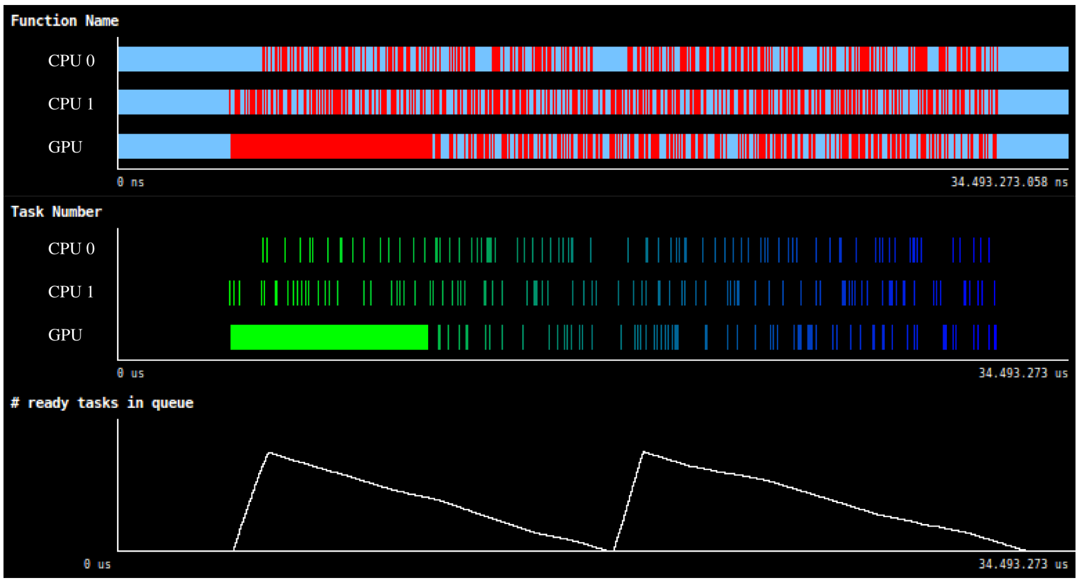


Figure 8.15: Timelines showing the functioning of OmpSs in the push phase.

The figure allows us to verify that the tasks of this simulation are distributed relatively balanced between the hardware resources (CPUs and GPU). These resources are working most of the time and it is probable that the application does not achieve the maximum performance, but this fact is compensated by the improvement on productivity thanks to the reduction in the programming complexity. The short length of the tasks suggests us that it could be possible to increase the number of particles per task reducing the management of task. Moreover, the steady pace at which tasks are dequeued gives us the idea that their lengths are similar.

8.7.2 Pull Phase

We repeated the previous test to verify the functioning of OmpSs with the *pull phase*. This time the number of particles was increased up to 1 million to increase the chances of conflicts when charge density was deposited at grid points. The number of tasks was 25 to reduce the amount of memory needed because each task had a grid copy. The number of tasks is also a trade-off: too many tasks imply many grid copies that may drain the memory, while too few tasks imply insufficient tasks to distribute the work and achieve a load balance among compute devices.

Figure 8.16 shows the results obtained when executing the pull kernel. The graphics are equivalent as those explained in Figure 8.15. This time, the different lengths of the tasks and the changeable pace at which tasks are dequeued reveal us the presence of lock contention. This contention comes from memory conflicts because different particles contribute to the charge density at the same grid point.

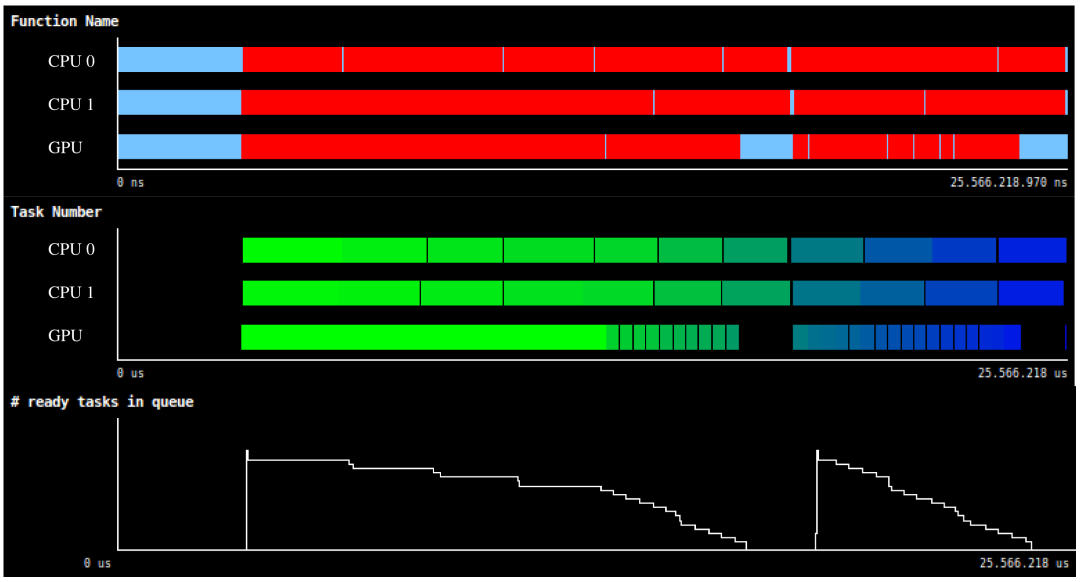


Figure 8.16: Timelines showing the functioning of OmpSs in the pull phase.

8.7.3 Evaluation

Figure 8.17 and Figure 8.18 show the results of the ARM OmpSs version. We can see that the performance is very stable and few tasks are enough to get the best performance for both kernels. In this experiment, we reused the same OpenCL kernels developed previously and we added the OmpSs pragmas to specify the tasks and its data dependencies.

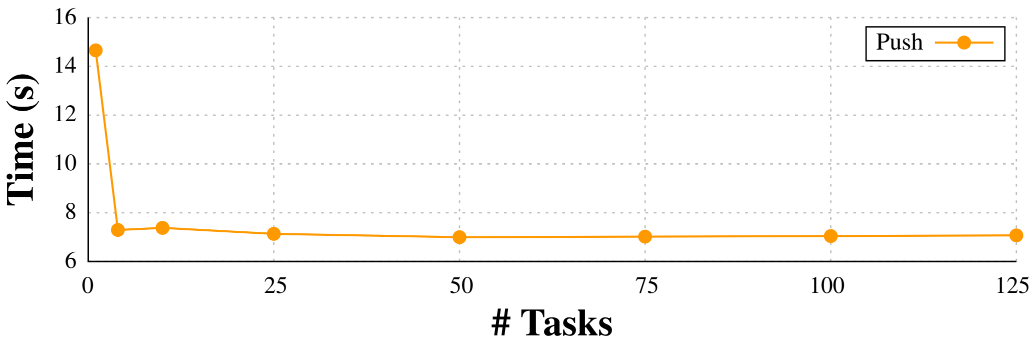


Figure 8.17: Execution time of the *push* kernel for different combinations of number of tasks.

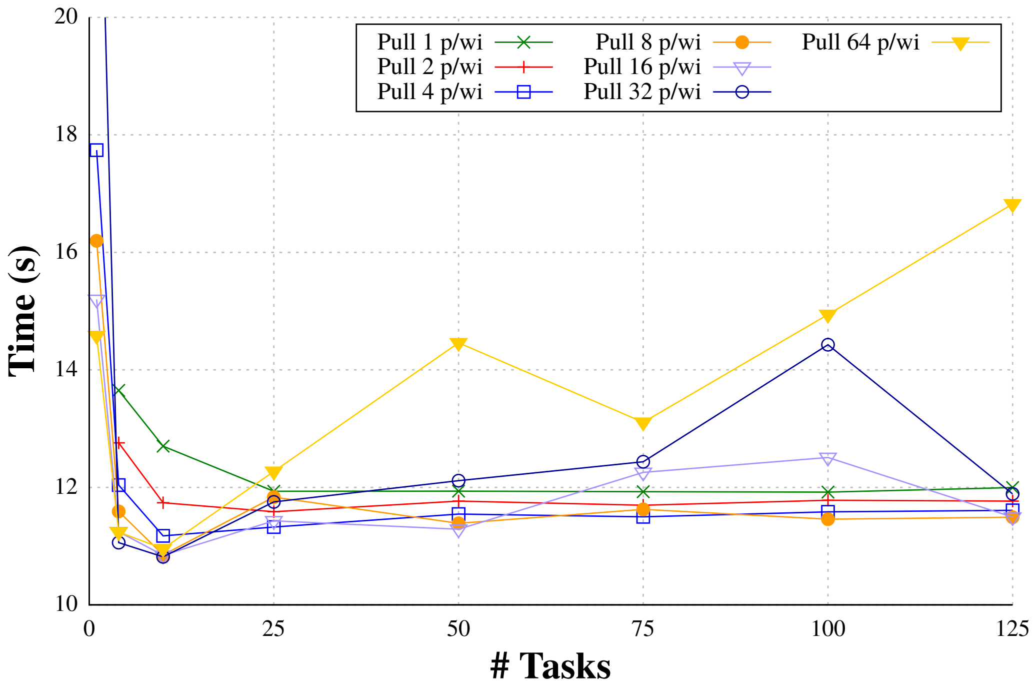


Figure 8.18: Execution time of the *pull* kernel for different combinations of number of tasks and particles per work-item (p/wi).

Moreover, OmpSs offers a reduction in programming complexity, since the number of OmpSs directives included is far lower than the OpenCL calls included in the hybrid version (table 8.1).

Kernel	Performance - Best time (s)		Programmability - Productivity	
	OpenMP+OpenCL	**OmpSs**	**OpenCL calls**	**OmpSs Directives**
Push	6.02	6.92	161	12
Pull	10.31	10.83	167	18

Table 8.1: Comparison between the hybrid version (OpenMP+OpenCL) and the OmpSs version.

The results shown in Figure 8.19 allow us to have an idea about the performance of *Mont-Blanc* prototype with respect to *MareNostrum III*. The data used for this comparison is the cylindrical geometry with $32 \times 32 \times 16$ cells and two different distributions of particles: a small test with 125,000 particles and a second bigger one with near 1 million particles. The figure shows the execution times of the OmpSs+OpenCL version on *Mont-Blanc* (using one node), which obviously are worse than the OpenMP version on *MareNostrum III* (using only 2 cores), because the computing power gap between them is really big. But the figure also shows that the increase of time between the two tests (small and big) is proportional between both platforms. Therefore, we can say that the new version for the prototype is working reasonably well.

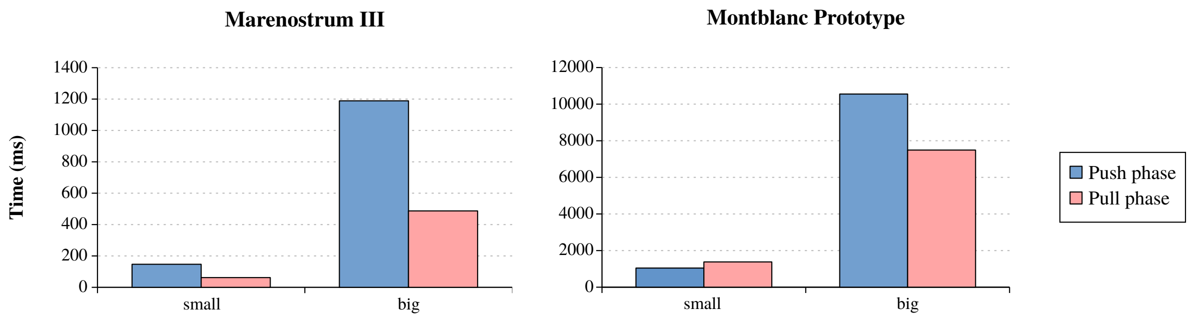


Figure 8.19: Times of the kernels running a small and a big test.

8.8 Conclusions

The work developed in this chapter confirmed the feasibility of porting a PIC code to an ARM-based platform. We have developed a hand-tuned hybrid version (OpenMP + OpenCL) and an OmpSs version of the most time-consuming kernels. Although OmpSs version is a bit slower than the hybrid version, it is a simpler version and its productivity has improved considerably. In our view, the easiness of OmpSs to port a code to a heterogeneous platform compensates the slight decrease in the observed performance.

A lot of research is being made to port scientific applications to new platforms, but the finish line of this efforts is still far from sight because in each case an exhaustive programming effort is needed to achieve the maximum performance. In our opinion, porting applications to new platforms following the classical models is largely limited and, as a result, the trend to use new programming models that gives priority to productivity (such as OmpSs) is here to stay.

Finally, another contribution of the developed work has been the aid to the development of OmpSs by detecting more than twenty issues in the compilation of the codes and by sending them to the developer team. For example, some of the bugs found and already solved are: incorrect detection of variables and operators in reduction clauses, some variables not detected inside a parallel loop, optional arguments considered as necessary, loop labels did not work inside parallel loops, etc.

Chapter 9

Conclusions and Future Work

"I am turned into a sort of machine for observing facts and grinding out conclusions."

CHARLES DARWIN

The main objective of this thesis was porting the Particle-In-Cell (PIC) algorithms to a heterogeneous architecture. In other words, the purpose was to explain the strategies applied to adapt a PIC code to an heterogeneous architecture.

At this point, I can affirm that this main objective has been reached. This last chapter will serve to sum up all the work done, describe the contributions made and give some possible lines to continue this research.

9.1 Work Done and Conclusions

We started working with a very good code from the performance point of view. EUTERPE had been optimized prior to this work. For the first time, as part of this thesis, its scalability was studied on several thousands of processors, concretely up to nearly 60,000 processors in the framework of the project PRACE. We tested the code in several platforms and used the Paraver and Extrae tools for the purpose of understanding the behaviour of the code and analyze the more time-consuming code sections.

EUTERPE was originally employing a Jacobi Preconditioned Conjugate Gradient (JPCG) solver provided by the external package PETSc to solve the Poisson equation. In the hybridization, we found an issue on this point because PETSc was not thread-safe and we decided to implement a new solver completely adapted to EUTERPE possibilities and based on MPI and OpenMP. This solver could be used in place of the solvers from PETSc with a similar behavior to the original version of the code. Chapter 7 contains a summary of the work done in this field and the solvers programmed and tested to reach better results in a hybrid version of the EUTERPE code. It is true that the results did not shown a significant improvement from the performance point of view, but we were able to use these solvers in machines where PETSc or similar packets could not achieve any result, at least at the time when they were developed.

Even though the scalability of EUTERPE has been proven to be good, more work will be needed to maintain this performance level in the next High-Performance Computing (HPC) platforms with multi-core nodes and million of cores in total. If the HPC systems had followed the same trend since 2004, the only thing that users of PIC codes would need to do in order to extend their analyses is to add more machines to their superclusters. But unfortunately for scientists, a new HPC trend usually involves new programming paradigms and as a consequence the code has to be rewritten and revalidated.

Nowadays the energy puts limits to this way of expansion and new heterogeneous platforms require us to rewrite our codes in order to get better results. The power consumption and power density limitations are promoting new platforms designs with the aim to build a future sustainable exaflop supercomputer based on the power efficiency. For example, the Mont-Blanc project (mentioned in Chapter 4) appeared some years ago with the aim to design a computer architecture capable of delivering an Exascale performance using 15 to 30 time less energy than present architectures.

This leads us to the main motivation for this thesis that is the study of porting a production code, with a very good performance, to the next generation of HPC architectures. This work had not been reported until then by any other group in the plasma community. Thus we started with the hybridization by introducing OpenMP thinking about the Petaflop supercomputers which were going to be developed in the following years.

We considered an ARM-based platform with a multi-core processor and Graphic Processing Unit (GPU). Given that maintaining a code in constant evolution on different platforms with different programming models is a challenge, our selection was therefore based on the programmability. Although OmpSs version is a bit slower than the hybrid version, it is a simpler version that enables to exploit heterogeneous systems and its productivity has improved considerably.

We can affirm that OmpSs simplifies and accelerates the porting of codes to new platforms in a reasonable time and our work contributes to confirm the viability of this approach to real production codes. The traditional way requires an amount of time that does not allow one to follow the development of complex codes normally coded by a small set of scientists, or simply scientists do not want to spend more time recoding part of their codes and revalidating the results since they only wish to add more physics to their code.

It is necessary to clarify that part of the work was performed with prototypes and software under development as seen in Chapter 4. As a consequence, we faced the challenge of executing a production code on test machines. During the thesis, many times our development were blocked for malfunction or incomplete parts in the hardware and software. In a certain way, our work has contributed to help their developments by showing these problems to the developers and also asking for remedying them. For example, in the development of OpenMP Super-scalar (OmpSs) we detected issues (more than twenty) in the compilation of the codes, such as incorrect detection of variables and operators in reduction clauses, some variables and labels not detected inside a parallel loops, etc.

9.2 Contributions

This last section intends to sum up all the discussion of this chapter in a few sentences.

In the initial learning phase of the PIC codes and in particular the EUTERPE code, we found interesting to perform a parametric analysis to evaluate the possibility to optimize the simulations through a careful selection of the input data. We published this work in [15] and we exposed our conclusions over the effects in the quality of the results when one reduced the data resolution (less particles, less grid cells or a bigger time step) in a simulation.

Among all the tests performed during this phase, it is worth emphasising two tests performed on *Jugene* to study the scalability of the application, and in which we achieved firstly up to 23,552 processors and secondly up to 61,440 processors. Up to that moment, the scaling of EUTERPE had been studied up to 512 processors in the framework of the project DEISA [12], but our test with 23,552 processors was a turning point and was included into a journal article [14]. Subsequently, we achieved the number of 61,440 processors that we published in [13, 15] too.

These tests brought to light a degradation in scalability for a great number of processors due to the increase of communications among processors and the reduction of the work per processor. In addition to this, we observed another issue due to the small amount of memory available per MPI process, since all shared memory on the node was distributed among all the MPI processes assigned to that node.

To solve this last issue that could limit the size of the simulation in a multi-core architecture, we developed a hybrid code by introducing OpenMP to take advantage of all levels of parallelism that a multi-core architecture provides and also would enable to one MPI process accessing all the memory of a node. At that time, this was innovative in a fusion production code and was published in [16].

During the development of the hybrid version, we had to face the issue of a solver which was non thread-safe in EUTERPE. This challenge was overcome by implementing our own solvers based on the JPCG [16] and BJPCG [17, 18] methods. They showed a similar behavior to the original version of the code and also added the ability to enable all cores work in the same MPI process simultaneously, which was not possible at that time.

Along the development of this thesis, we have seen a transition in the majority trend in HPC. Energy has become one of the most expensive resources and, as a consequence, new supercomputers are based on the power efficiency provided by new homogeneous and heterogeneous architectures. New heterogeneous designs have arisen in recent years such as IBM Cell and ARM-based processors. Unfortunately, to exploit all their potential, scientists have to rewrite their codes and they are normally not willing to do it. Therefore, we focus on the programmability and we coded two more versions of EUTERPE in the framework of the Mont-Blanc project: one with OpenMP and OpenCL and other with OmpSs and OpenCL. They were analyzed in [18–20] and although OmpSs version was a bit slower than the other version, the porting was faster and simpler, and as a result this opened the door to real production codes to exploit heterogeneous systems at a reasonably time cost.

To sum up, in this work we have developed several versions of the EUTERPE code using different programming models to take advantage of the different HPC systems that one can find currently or in the near future. A key in achieving this end has been the participation in scientific projects in the forefront of HPC world, such as EUFORIA, PRACE and the Mont-Blanc project. The work developed in these projects, summarized in [17, 20], has allowed us to prove that our versions of EUTERPE were completely functional through several tests. Therefore, this thesis has not to be seen as the end, but rather as the beginning, of a work to extend the physics simulated in fusion codes through exploiting available HPC resources.

9.3 Suggestions for Future Work

The work done in this thesis has an innovative value in two different fields. On one hand, porting a PIC code in a completely new architecture (ARM-based), and on the other hand, the attempt to adapt a production code in the field of plasma physics to a new programming paradigm. With that in mind, the work done is the initiator of a new research in both fields.

The next step for the immediate future is to merge our version of EUTERPE with the latest version from Max-Planck-Institut für Plasmaphysik (IPP) Garching. In this way, we make accessible our developments in the main branch of EUTERPE and add the possibility to install EUTERPE automatically on these kind of machines.

Concerning ARM-based systems, some extra work is needed to complete the OmpSs version. Specifically, we will extend the implementation to several nodes using the parallelization with Message Passing Interface (MPI) on a new prototype. Once that is done, we have other changes in mind to improve our version, such as sorting particles in order to improve access to the particle data in the GPU, provided that compensates for the extra cost of sorting.

A more ambitious plan for the future is to extend the heterogeneous version to new platforms such as Many Integrated Core (MIC), which is a new coprocessor computer architecture developed by Intel. Also, we plan to study new profiling tools (as clustering) to examine in more detail the structure of parallel applications by characterizing its computation regions.

The result of the parametric analysis done in EUTERPE opens us another option to continue the research. Our opinion is that one should consider the use of a new time integrator (for example a symplectic integrator) for the equations of motion of the particles. This could allow for a better conservation of energy or the use of a larger integration time step with still acceptable results.

Finally, some years ago we collaborated with a Norwegian team to study Input/Output (I/O) performance in real scientific applications and we saw that this was a possible way to pursue. A PIC code produces diagnostic data periodically that could potentially lead to a bottleneck in the future depending on the grade of precision required in the results. Specially when a great physical domain and a large number of particles are involved in the simulation. For that reason, it could be interesting to devote some time in the future to study the way how EUTERPE accesses hard disks and evaluate if it needs any improvement in a production case.

Bibliography

[1] **Thijssen, J. M.**, Computational Physics, Cambridge University Press, 1999.

[2] **Pritchett, P. L.**, *Particle-in-Cell Simulation of Plasmas - A Tutorial*, *Space Plasma Simulation*, edited by J.Büchner, C. and M.Scholer, Vol. 615 of *Lecture Notes in Physics, Berlin Springer Verlag*, pp. 1–24, 2003.

[3] **Birdsall, C. K.** and **Langdon, A. B.**, Plasma physics via computer simulation, Series in Plasma Physics and Fluid Dynamics, Taylor & Francis Group, 2005.

[4] **Jolliet, S., Bottino, A., Angelino, P., Hatzky, R., Tran, T. M., Mcmillan, B. F., Sauter, O., Appert, K., Idomura, Y.** and **Villard, L.**, *A global collisionless PIC code in magnetic coordinates*, Computer Physics Communications, Vol. 177, No. 5, pp. 409 – 425, 2007.

[5] **Görler, T., Lapillonne, X., Brunner, S., Dannert, T., Jenko, F., Merz, F.** and **Told, D.**, *The global version of the gyrokinetic turbulence code GENE*, Journal of Computational Physics, Vol. 230, No. 18, pp. 7053 – 7071, 2011.

[6] **Heikkinen, J. A., Janhunen, S. J., Kiviniemi, T. P.** and **Ogando, F.**, *Full f gyrokinetic method for particle simulation of tokamak transport*, Journal of Computational Physics, Vol. 227, No. 11, pp. 5582, 2008.

[7] **ITER Organization**, *ITER: The way to new energy*, http://www.iter.org/.

[8] **Kemp, R., Ward, D. J., Federici, G., Wenninger, R.** and **Morris, J.**, *DEMO design point studies*, *25th IAEA International Conference on Fusion Energy*, 2014.

[9] **Mont-blanc**, *European approach towards energy efficient high performance*, http://www.montblanc-project.eu.

[10] **EUFORIA**, *Combination of Collaborative projects & Coordination and support actions. Annex I - Description of Work*, Grant agreement 211804, Seventh Framework Programme, 2007.

[11] **Jost, G., Tran, T. M. et al.**, *First global linear gyrokinetic simulations in 3D magnetic configurations*, *26th EPS Conference on Controlled Fusion and Plasma Physics*, 1999.

[12] **DEISA**, *JRA3: Activity report and roadmap evaluation*, Tech. Rep. DEISA-D-JRA3-5, Distributed European Infrastructure for Supercomputing Applications, Oct 2006.

[13] **Sáez, X., Soba, A., Sánchez, E., Kleiber, R., Hatzky, R., Castejón, F.** and **Cela, J. M.**, *Improvements of the particle-in-cell code EUTERPE for petascaling machines*, *Poster presented at Conference on Computational Physics (CCP), June 23–26, Trondheim, Norway*, 2010.

[14] **Sánchez, E., Kleiber, R., Hatzky, R., Soba, A., Sáez, X., Castejón, F.** and **Cela, J. M.**, *Linear and Nonlinear Simulations Using the EUTERPE Gyrokinetic Code*, IEEE Transactions on Plasma Science, Vol. 38, pp. 2119–2128, 2010.

[15] **Sáez, X., Soba, A., Sánchez, E., Kleiber, R., Castejón, F.** and **Cela, J. M.**, *Improvements of the particle-in-cell code EUTERPE for petascaling machines*, Computer Physics Communications, Vol. 182, No. 9, pp. 2047 – 2051, 2011.

[16] **Sáez, X., Soba, A., Sánchez, E., Cela, J. M.** and **Castejón, F.**, *Particle-In-Cell algorithms for Plasma simulations on heterogeneous architectures*, *Proceedings of the 19th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 385–389, IEEE Computer Society, 2011.

[17] **Sáez, X., Akgün, T.** and **Sánchez, E.**, *Optimizing EUTERPE for Petascaling*, Tech. rep., Whitepaper of PRACE-1IP project, 2012.

[18] **Sáez, X., Soba, A., Sánchez, E., Mantsinen, M., Mateo, S., Cela, J. M.** and **Castejón, F.**, *First experience with particle-in-cell plasma physics code on ARM-based HPC systems*, Journal of Physics: Conference Series, Vol. 640, No. 1, pp. 012064, 2015.

[19] **Sáez, X., Soba, A.** and **Mantsinen, M.**, *Plasma Physics Code Contribution to the Mont-Blanc Project*, *Book of abstracts of the 2nd BSC International Doctoral Symposium*, pp. 83–84, Barcelona, Spain, May 2015.

[20] **Sáez, X., Soba, A., Sánchez, E., Mantsinen, M.** and **Cela, J. M.**, *Performance analysis of a particle-in-cell plasma physics code on homogeneous and heterogeneous HPC systems*, *Proceedings of Congress on Numerical Methods in Engineering*, pp. 1–18, APMTAC, Lisbon, Portugal, 2015.

[21] **Langmuir, I.**, *Oscillations in Ionized Gases*, Proceedings of the National Academy of Science, Vol. 14, pp. 627–637, Aug. 1928.

[22] **Boyd, T. J. M.** and **Sanderson, J. J.**, The Physics of Plasmas, Cambridge University Press, 2003, cambridge Books Online.

[23] **Ethier, S., Tang, W. M.** and **Lin, Z.**, *Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms*, *Journal of Physics: Conference Series*, Vol. 16, pp. 1–15, Institute of Physics Publishing, 2005.

[24] **Tajima, T.**, Computational Plasma Physics: with applications to fusion and astrophysics, Westview Press, 2004.

[25] **Hockney, R. W.** and **Eastwood, J. W.**, Computer simulation using particles, Taylor & Francis, Inc., Bristol, PA, USA, 1988.

[26] **Wesson, J.**, Tokamaks, Clarendon Press - Oxford, Bristol, PA, USA, 2004.

[27] **Lin, Z., Hahm, T. S., Lee, W. W., Tang, W. M.** and **White, R. B.**, *Turbulent Transport Reduction by Zonal Flows: Massively Parallel Simulations*, Science, Vol. 281, pp. 1835–1837, Sep. 1998.

[28] **Jenko, F., Dorland, W., Kotschenreuther, M.** and **Rogers, B. N.**, *Electron temperature gradient driven turbulence*, Physics of Plasmas (1994-present), Vol. 7, No. 5, pp. 1904–1910, 2000.

[29] **Commissariat à l'énergie atomique et aux énergies alternatives (CEA)**, *GYSELA5D: a 5D GYrokinetic SEmi-LAgrangian code*, http://gyseladoc.gforge.inria.fr.

[30] **Hatzky, R., Tran, T. M., Könies, A., Kleiber, R.** and **Allfrey, S.**, *Energy conservation in a nonlinear gyrokinetic particle-in-cell code for ion-temperature-gradient-driven modes in θ-pinch geometry*, Physics of Plasmas, Vol. 9, pp. 898–912, Mar. 2002.

[31] **Aalto University**, *Department of Applied Physics. ELMFIRE*, http://physics.aalto.fi/groups/fusion/research/elmfire.

[32] **Alcouffe, R. E., Baker, R. S., Dahl, J. A., Turner, S. A.** and **Wart, R.**, *PARTISN: A Time-Dependent, Parallel Neutral Particle Transport Code System*, Tech. Rep. LA-UR-05-3925, Los Alamos National Lab, 2005.

[33] **Varian Medical Systems**, *Attila Radiation Transport Software*, http://www.transpireinc.com/html/attila.

[34] **Los Alamos National Laboratory**, *A General Monte Carlo N-Particle (MCNP) Transport Code*, https://mcnp.lanl.gov.

[35] **Brun, E., Damian, F., Diop, C. M., Dumonteil, E., Hugot, F. X., Jouanne, C., Lee, Y. K., Malvagi, F., Mazzolo, A., Petit, O., Trama, J. C., Visonneau, T.** and **Zoia, A.**, *TRIPOLI-4®, CEA, EDF and AREVA reference Monte Carlo code*, Annals of Nuclear Energy, Vol. 82, pp. 151 – 160, 2015.

[36] **Kramer, G. J., Budny, R. V., Bortolon, A., Fredrickson, E. D., Fu, G. Y., Heidbrink, W. W., Nazikian, R., Valeo, E.** and **Van Zeeland, M. A.**, *A description of the full-particle-orbit-following SPIRAL code for simulating fast-ion experiments in tokamaks*, Plasma Physics and Controlled Fusion, Vol. 55, No. 2, pp. 025013, 2013.

[37] **Hirvijoki, E., Asunta, O., Koskela, T., Kurki-Suonio, T., Miettunen, J., Sipilä, S., Snicker, A.** and **Äkäslompolo, S.**, *ASCOT: Solving the kinetic equation of minority particle species in tokamak plasmas*, Computer Physics Communications, Vol. 185, pp. 1310–1321, Apr. 2014.

[38] **Commissariat à l'énergie atomique et aux énergies alternatives (CEA)**, *JOREK non-linear MHD Code*, http://jorek.eu.

[39] **Princeton Plasma Physics Laboratory**, *M3D HOME*, http://w3.pppl.gov/m3d/index.php.

[40] **Akarsu, E., Dincer, K., Haupt, T.** and **Fox, G. C.**, *Particle-in-cell simulation codes in High Performance Fortran*, *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, p. 38, IEEE Computer Society, Washington, DC, USA, 1996.

[41] **Harlow, F. H.**, *A Machine Calculation Method for Hydrodynamic Problems*, Tech. Rep. LAMS-1956, Los Alamos Scientific Laboratory, 1955.

[42] **Paes, A. C., Abe, N. M., Serrao, V. A.** and **Passaro, A.**, *Simulations of*

plasmas with electrostatic PIC models using the finite element method, Brazillian Journal of Physics, Vol. 33, No. 2, pp. 411–417, 2003.

[43] **Messmer, P.**, *Par-T: A Parallel Relativistic Fully 3D Electromagnetic Particle-in-Cell Code*, *PARA '00: Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, pp. 350–355, Springer-Verlag, London, UK, 2001.

[44] **Przebinda, V.** and **Cary, J.**, *Some improvements in PIC performance through sorting, caching, and dynamic load balancing*, 2005.

[45] **Bowers, K. J., Albright, B. J., Bergen, B., Yin, L., Barker, K. J.** and **Kerbyson, D. J.**, *0.374 Pflop/s trillion-particle kinetic modeling of laser plasma interaction on Roadrunner*, *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–11, IEEE Press, Piscataway, NJ, USA, 2008.

[46] **Grieger, G. et al.**, *Physics and engineering studies for Wendelstein 7-X*, *Proceedings of the 13th International Conference on Plasma Physics and Controlled Nuclear Fusion Research*, Vol. 3, pp. 525–532, Washington, DC, USA, 1991.

[47] **Jost, G., Tran, T. M., Cooper, W. A., Villard, L.** and **Appert, K.**, *Global linear gyrokinetic simulations in quasi-symmetric configurations*, Physics of Plasmas, Vol. 8, No. 7, 2001.

[48] **Kornilov, V., Kleiber, R., Hatzky, R., Villard, L.** and **Jost, G.**, *Gyrokinetic global three-dimensional simulations of linear ion-temperature-gradient modes in Wendelstein 7-X*, Physics of Plasmas, Vol. 11, No. 6, 2004.

[49] **Kornilov, V., Kleiber, R.** and **Hatzky, R.**, *Gyrokinetic global electrostatic ion-temperature-gradient modes in finite β equilibria of Wendelstein 7-X*, Nuclear Fusion, Vol. 45, No. 4, pp. 238, 2005.

[50] **Kleiber, R.**, *Global linear gyrokinetic simulations for stellarator and axisymmetric equilibria*, *AIP Conference Proceedings*, Vol. 871, p. 136, IOP INSTITUTE OF PHYSICS PUBLISHING LTD, 2006.

[51] **Moore, G. E.**, *Progress in Digital Integrated Electronics*, International Electron Devices Meeting, Vol. 21, pp. 11–13, 1975.

[52] **Borkar, S., Jouppi, N. P.** and **Stenstrom, P.**, *Microprocessors in the era of terascale integration*, *DATE '07: Proceedings of the conference on Design, automation*

and test in Europe, pp. 237–242, EDA Consortium, San Jose, CA, USA, 2007.

[53] **Marcuello, P., González, A.** and **Tubella, J.**, *Speculative multithreaded processors*, *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pp. 77–84, ACM, New York, NY, USA, 1998.

[54] **Annavaram, M., Grochowski, E.** and **Shen, J.**, *Mitigating Amdahl's Law through EPI Throttling*, SIGARCH Comput. Archit. News, Vol. 33, No. 2, pp. 298–309, 2005.

[55] **Hofstee, H. P.**, *Power Efficient Processor Architecture and The Cell Processor*, *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pp. 258–262, IEEE Computer Society, Washington, DC, USA, 2005.

[56] **Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P.** and **Yelick, K.**, *Scientific computing Kernels on the cell processor*, Int. J. Parallel Program., Vol. 35, No. 3, pp. 263–298, 2007.

[57] **TOP500.org**, *TOP500 Supercomputer sites*, http://www.top500.org.

[58] **Rajovic, N., Carpenter, P. M., Gelado, I., Puzovic, N., Ramirez, A.** and **Valero, M.**, *Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC?*, *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pp. 40:1–40:12, ACM, New York, NY, USA, 2013.

[59] **Barcelona Supercomputing Center**, *Extrae user guide*, http://www.bsc.es/computer-sciences/extrae.

[60] **Pillet, V., Labarta, J., Cortes, T.** and **Girona, S.**, *PARAVER: A Tool to Visualize and Analyze Parallel Code*, *Proceedings of WoTUG-18: Transputer and occam Developments*, edited by Nixon, P., pp. 17–31, mar 1995.

[61] **Graham, S. L., Kessler, P. B.** and **Mckusick, M. K.**, *Gprof: A Call Graph Execution Profiler*, *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pp. 120–126, ACM, New York, NY, USA, 1982.

[62] **Hahm, T. S.**, *Nonlinear gyrokinetic equations for tokamak microturbulence*, Physics of Fluids (1958-1988), Vol. 31, No. 9, pp. 2670–2673, 1988.

[63] **Hirshman, S. P.** and **Whitson, J. C.**, *Steepest-descent moment method for three-*

dimensional magnetohydrodynamic equilibria, Physics of Fluids (1958-1988), Vol. 26, No. 12, pp. 3553–3568, 1983.

[64] **Hollig, K.**, Finite Element Methods with B-Splines, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[65] **Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F., Zampini, S.** and **Zhang, H.**, *PETSc Users Manual*, Tech. Rep. ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

[66] **MPICH**, *High-Performance Portable MPI*, https://www.mpich.org.

[67] **MIT**, *FFTW Home Page*, http://www.fftw.org.

[68] **Nührenberg, C., Hatzky, R.** and **Sorge, S.**, *Global ITG turbulence in screw-pinch geometry*, *IAEA Technical Meeting on Innovative Concepts and Theory of Stellarators*, Madrid, Spain, 2005.

[69] **Bottino, A., Peeters, A. G., Hatzky, R., Jolliet, S., McMillan, B. F., Tran, T. M.** and **Villard, L.**, *Nonlinear low noise particle-in-cell simulations of electron temperature gradient driven turbulence*, Physics of Plasmas, Vol. 14, No. 1, pp. –, 2007.

[70] **Nevins, W. M., Hammett, G. W., Dimits, A. M., Dorland, W.** and **Shumaker, D. E.**, *Discrete particle noise in particle-in-cell simulations of plasma microturbulence*, Physics of Plasmas, Vol. 12, No. 12, pp. –, 2005.

[71] **Liewer, P. C., Decyk, V. K., Dawson, J. M.** and **Fox, G. C.**, *A universal concurrent algorithm for plasma particle-in-cell simulation codes*, *Proceedings of the third conference on Hypercube concurrent computers and applications*, pp. 1101–1107, ACM, New York, NY, USA, 1988.

[72] **Decyk, V. K.**, *Skeleton PIC codes for parallel computers*, Computer Physics Communications, Vol. 87, No. 1-2, pp. 87 – 94, 1995.

[73] **Message Passing Interface Forum**, *MPI: A Message-Passing Interface Standard version 3.1*, http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[74] **Norton, C. D., Szymanski, B. K.** and **Decyk, V. K.**, *Object-oriented parallel computation for plasma simulation*, Commun. ACM, Vol. 38, No. 10, pp. 88–100,

1995.

[75] **Tskhakaya, D.** and **Schneider, R.**, *Optimization of PIC codes by improved memory management*, J. Comput. Phys., Vol. 225, No. 1, pp. 829–839, 2007.

[76] **Lin, C. S., Thring, A. L., Koga, J.** and **Seiler, E. J.**, *A parallel particle-in-cell model for the massively parallel processor*, J. Parallel Distrib. Comput., Vol. 8, No. 2, pp. 196–199, 1990.

[77] **Martino, B. D., Briguglio, S., Vlad, G.** and **Sguazzero, P.**, *Parallel PIC plasma simulation through particle decomposition techniques*, Parallel Computing, Vol. 27, No. 3, pp. 295–314, 2001.

[78] **Hatzky, R.**, *Domain cloning for a particle-in-cell (PIC) code on a cluster of symmetric-multiprocessor (SMP) computers*, Parallel Computing, Vol. 32, No. 4, pp. 325–330, 2006.

[79] **Dagum, L.** and **Menon, R.**, *OpenMP: An Industry-Standard API for Shared-Memory Programming*, Computing in Science and Engineering, Vol. 5, No. 1, pp. 46–55, 1998.

[80] **Hegland, M.** and **Saylor, P. E.**, *Block jacobi preconditioning of the conjugate gradient method on a vector processor*, International journal of computer mathematics, Vol. 44, No. 1-4, pp. 71–89, 1992.

[81] **Aubry, R., Mut, F., Löhner, R.** and **Cebral, J. R.**, *Deflated preconditioned conjugate gradient solvers for the Pressure-Poisson equation*, Journal of Computational Physics, Vol. 227, No. 24, pp. 10196 – 10208, 2008.

[82] **Dongarra, J., Luszczek, P. et al.**, *The LINPACK Benchmark: past, present and future.*, Concurrency and Computation: Practice and Experience, Vol. 15, No. 9, pp. 803–820, 2003.

[83] **The Green 500**, *Ranking the world's most energy-efficient supercomputers*, `http://www.green500.org`.

[84] **Jowkar, M., de la Cruz, R.** and **Cela, J. M.**, *Exploring a Novel Gathering Method for Finite Element Codes on the Cell/B.E. Architecture*, *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–11, IEEE Computer Society, Washington, DC, USA, 2010.

[85] **OpenCL**, *OpenCL 1.1 Reference Pages*, https://www.khronos.org/opencl.

[86] **Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., et al.**, *OmpSs: A proposal for programming heterogeneous multi-core architectures*, Parallel Processing Letters, Vol. 21, No. 02, pp. 173–193, 2011.

List of Acronyms

3D	three-dimensional
2D	two-dimensional
API	Application Programming Interface
BJPCG	Block Jacobi Preconditioned Conjugate Gradient
BSC	Barcelona Supercomputing Center
CIEMAT	Centro Investigaciones Energéticas, Medioambientales y Tecnológicas
CPU	Central Processing Unit
CRPP	Centre de Recherches en Physique des Plasmas
CSR	Compressed Sparse Row
CUDA	Compute Unified Device Architecture
DDR3	Double Data Rate type 3
DEMO	DEMOnstration Power Plant
DMA	Direct Memory Access
D-T	Deuterium-Tritium
EIB	Element Interconnect Bus
ELM	Edge Localised Mode
FDM	Finite Difference Method
FFT	Fast Fourier Transform
FLOPS	FLoating-Point Operations Per Second

GPU	Graphic Processing Unit
HPC	High-Performance Computing
ICF	Inertial Confinement Fusion
I/O	Input/Output
IPC	Instructions Per Cycle
IPP	Max-Planck-Institut für Plasmaphysik
IPS	Instructions Per Second
ITER	International Thermonuclear Experimental Reactor
ITG	Ion Temperature Gradient
JPCG	Jacobi Preconditioned Conjugate Gradient
LPI	Laser-Plasma Instabilities
LS	Local Store
MFC	Memory Flow Controller
MHD	Magnetohydrodynamics
MIC	Many Integrated Core
MPI	Message Passing Interface
NGP	Nearest Grid Point
OmpSs	OpenMP Super-scalar
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PCG	Preconditioned Conjugate Gradient
PDE	Partial Differential Equation
PE	Processing Element
PETSc	Portable, Extensible Toolkit for Scientific Computation
PIC	Particle-In-Cell
PMU	Power Management Unit
PPE	Power Processing Element
RAM	Random Access Memory
RAW	Read-after-Write

RISC	Reduced Instruction Set Computer
RK	Runge-Kutta method
RK4	Runge-Kutta fourth order method
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multiprocessing
S/N	Signal-to-Noise ratio
SoC	System on Chip
SPE	Synergistic Processing Element
SpMV	Sparse Matrix-Vector Multiplication
TEM	Trapped Electron Mode
TMU	Thermal Management Unit
VMEC	Variational Moments Equilibrium Code
WAR	Write-after-Read
WAW	Write-after-Write

List of Figures

List of Tables

List of Algorithms

Listings