Pic-Vert : a particle-in-cell implementation for multi-core architectures

Yann Barsamian

UNIVERSITÉ DE STRASBOURG

École Doctorale Mathématiques, Sciences de l'Information et de l'Ingénieur

Laboratoire des sciences de l'Ingénieur, de l'Informatique et de l'Imagerie (ICube), UMR 7357

THÈSE présentée par :

Yann BARSAMIAN

soutenue le : **31 octobre 2018**

pour obtenir le grade de : **Docteur de l'université de Strasbourg**

Discipline/ Spécialité : Informatique

Pic-Vert : Une implémentation de la méthode particulaire pour architectures multi-cœurs

THÈSE dirigée par :

M. VIOLARD Éric	Maître de conférences à l'université de Strasbourg, Strasbourg, France

RAPPORTEURS :

M. JEANNOT Emmanuel	Directeur de recherche à INRIA Bordeaux Sud-Ouest, Bordeaux, France
M. SONNENDRÜCKER Eric	Professeur au Max Planck Institute for Plasma Physics, Garching, Allemagne

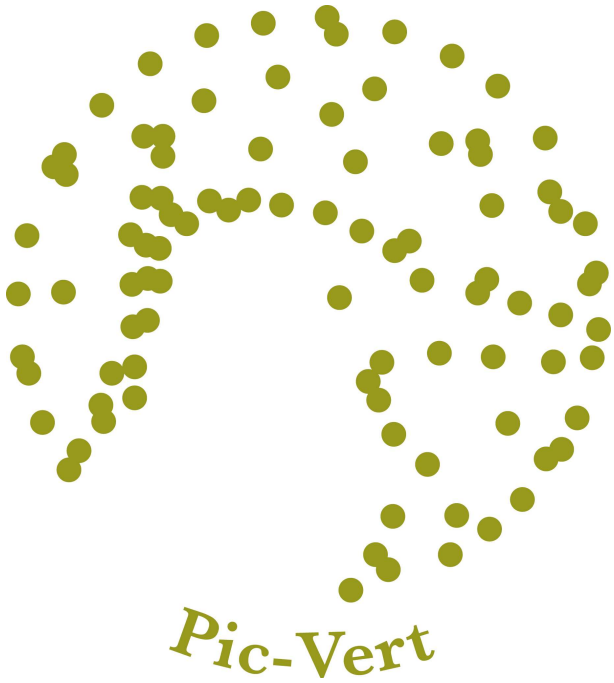
---

AUTRES MEMBRES DU JURY :

M. COTTET Georges-Henri	Professeur à l'université de Grenoble, Grenoble, France
M. HIRSTOAGA Sever	Chargé de recherches à INRIA Nancy Grand-Est, Strasbourg, France
M. MEHRENBERGER Michel	Professeur à l'université d'Aix-Marseille, Marseille, France

Pic-Vert: A Particle-in-Cell Implementation for Multi-Core Architectures

Yann Barsamian
yann.barsamian@gmail.com



Contents

Remerciements

J'aimerais exprimer ma gratitude à mes directeurs de thèse Sever Hirstoaga, Michel Mehrenberger et Éric Violard. Ils m'ont offert tous les conseils indispensables à l'entrée dans le monde de la recherche, tout en me considérant comme un collègue et non comme un étudiant. Avoir trois directeurs de recherche est une expérience extrêmement enrichissante, puisque j'ai ainsi pu voir des points de vue différents. Je tiens à chaleureusement les remercier d'avoir passé tant de temps à me former, et espère avoir le bonheur de travailler avec eux dans le futur.

J'aimerais ensuite remercier mes rapporteurs de thèse Emmanuel Jeannot et Éric Sonnendrücker pour leurs relectures attentives. Je remercie également Georges-Henri Cottet d'avoir accepté de faire partie de mon jury.

On ne devient pas chercheur du jour au lendemain, et je dois bien sûr presque tout aux enseignants qui m'ont formé aux mathématiques et à l'informatique. Je tiens à remercier en particulier Jean Voedts et Alain Juhel pour m'avoir supporté deux ans de suite, khûbe oblige. Merci également à Silvano Dal Zilio qui m'a le premier fait rentrer dans le bain de la recherche. Je n'aurais jamais eu un tel parcours sans eux.

J'aimerais maintenant remercier toute l'équipe ICPS. Ils m'ont appris que nulle recherche digne de ce nom ne peut se faire sans parties endiablées de backgammon autour d'un bon café. Plus précisément… Merci à Alain pour toutes ses anecdotes et son précieux savoir sur tous les sujets que j'ai pu aborder avec lui. Merci à Arthur pour ses idées qui ont été bien utiles en fin de thèse. Merci à Artyom de nous avoir montré que nous étions des petits joueurs avec nos techniques de backgammon, et qu'en Arménie, le nardi, c'est bien plus sérieux ! Merci à Esteban de m'avoir laissé gagner un nombre incalculable de parties de backgammon en jouant toujours plus pour s'amuser que pour gagner. Merci à Harenome de m'avoir initié aux joies de la musique en groupe. Merci à Imen et Mariem d'avoir fourni une présence féminine dans ce monde informatique bien trop masculin. Merci à Jens pour ses précieux conseils sur le langage `C`. Merci à Juan et Maxime d'avoir été des colocataires fabuleux. Merci à Luke d'avoir été un cobureau génial pendant 3 ans. Merci à Philippe d'avoir géré l'administratif pour l'équipe. Merci à Stéphane pour tous ses conseils sur l'après-thèse. Merci à Vincent pour l'organisation de la salle café : grâce à lui, nos dose de drogues étaient toujours disponibles à temps. Merci enfin à tous les autres avec qui j'ai passé moins de temps mais qui ont également participé à faire de l'équipe ICPS ce qu'elle est.

> “ Dicebat Bernardus Carnotensis nos esse quasi nanos, gigantium humeris incidentes, ut possimus plura eis et remotiora videre, non utique proprii visus acumine, aut eminentia corporis, sed quia in altum subvehimur et extollimur magnitudine gigantea.[1]
>
> J. de Salisbury [200] ”

J'aimerais aussi remercier toutes les personnes qui font vivre les différents outils que j'ai utilisés pendant ma thèse. Merci tout d'abord aux innombrables chercheurs dont j'ai utilisé,

[1] Bernard de Chartres disait que nous sommes comme des nains juchés sur les épaules de géants, de sorte que nous pouvons voir davantage de choses qu'eux et plus loin, non certes à cause de l'acuité de notre vue ou de notre plus grande taille, mais parce que nous sommes soulevés en hauteur et élevés à la taille d'un géant. J. de Salisbury (traduction par F. Lejeune)

directement ou indirectement, les résultats. Si ma thèse peut avoir un quelconque intérêt, c'est bien parce qu'il y a eu une quantité astronomique de personnes qui ont contribué à la recherche avant moi. Merci ensuite aux équipes qui construisent et qui assurent la maintenance des supercalculateurs sur lesquels j'ai travaillé : Curie, Marconi et Occigen. Sans eux, aucune simulation de grande envergure n'aurait été possible pendant cette thèse. Merci également à toutes les personnes qui développent et maintiennent les langages de programmation, les logiciels ou les bibliothèques que j'ai utilisés pendant cette thèse. Sans eux, il faudrait sans arrêt réinventer la roue et perdre un temps précieux. Merci enfin aux personnes qui font vivre les forums d'entraide sur lesquels j'ai trouvé réponse à la quasi-totalité des questions d'ordre technique que je me suis posées pendant ma thèse : Stack Overflow et LaTeX Stack Exchange. Sans elles, ce manuscrit n'aurait pas pu être réalisé avec autant de soin dans les détails.

J'aimerais également remercier mes amis, et plus particulièrement André. Ses innombrables conseils prodigués tout au long de la thèse, y compris sur mon manuscrit, m'ont été d'un secours indispensable.

Malgré toutes les qualités des personnes que je viens de citer, je dois tout de même souligner que ces études n'auraient jamais été possibles sans un soutien continu et inconditionnel de mes parents. Je crois qu'on ne dit jamais assez à ses parents combien on les aime et combien ils sont importants : papa, maman, merci énormément pour tout ce que vous m'avez apporté.

J'aimerais évidemment remercier celle sans qui ces années de thèse n'auraient pas été possibles, celle qui a su s'occuper de notre fils quand j'étais à Strasbourg pour la thèse, celle qui m'a toujours soutenu pendant ces années, celle grâce à qui j'ai pu finir d'écrire ce manuscrit dans les meilleures conditions. Merci du fond du cœur.

Enfin, j'aimerais remercier mes enfants pour leur bonne humeur perpétuelle et leurs sourires à tout instant. C'est aussi un peu grâce à eux si cette thèse a pu aboutir, et ce sont eux qui me rappellent à chaque instant qu'il faut continuer de regarder même les sujets les plus complexes avec des yeux d'enfant pour mieux les apprivoiser.

Acknowledgements

I would like to thank my thesis advisors Sever Hirstoaga, Michel Mehrenberger and Éric Violard. They gave me all the necessary pieces of advice to work in the research world, while treating me as a colleague and not a student. Having three thesis advisors was a very rich experience, because I could have different points of view on my work. I want to warmly thank them for their time, and hope I will work again with them in the future.

I would then like to thank my thesis referees Emmanuel Jeannot and Éric Sonnendrücker for their careful reading of my manuscript. I also thank Georges-Henri Cottet who accepted to be in my defense committee.

It takes more than one day to become a researcher, and I am in my mathematics and computer science teachers' debt. They were the ones who gave me the first necessary skills in those domains. I wish in particular to thank Jean Voedts and Alain Juhel for enduring me two years in a row — because I was a khûbe. Thanks also to Silvano Dal Zilio who first introduced me to the research world. I would never have accomplished this without them.

I would now like to thank the whole ICPS team. They taught me that no real research can be performed without some good backgammon games around coffee cups. More precisely… Thanks to Alain for all his anecdotes and for his precious knowledge on all the subjects we discussed together. Thanks to Arthur for his ideas which were useful in the end of my thesis. Thanks to Artyom for showing us real backgammon techniques; in Armenia, nard is more serious than our little games! Thanks to Esteban for letting me win an uncountable number of backgammon games by playing for fun more than for winning. Thanks to Harenome for allowing me to play music in the band he is part of. Thanks to Imen and Mariem for bringing a feminine touch into the computer science world, which is far too much masculine. Thanks to Jens for his precious pieces of advice on the `C` language. Thanks to Juan and Maxime for being awesome roommates. Thanks to Luke for being an awesome office mate during those 3 years. Thanks to Philippe for handling the administrative stuff for the team. Thanks to Stéphane for all his pieces of advice to prepare what comes after the PhD. Thanks to Vincent for the organization of the coffee room: thanks to him, our daily drug was available on time. Thanks then to all the others with whom I spent less time but who are also part of the great ICPS spirit.

> “ Dicebat Bernardus Carnotensis nos esse quasi nanos, gigantium humeris incidentes, ut possimus plura eis et remotiora videre, non utique proprii visus acumine, aut eminentia corporis, sed quia in altum subvehimur et extollimur magnitudine gigantea.[2]
>
> J. de Salisbury [200] ”

I would like to also thank all the people whose work was very useful for me during my thesis. First, thanks a lot to the countless researchers from whom I borrowed, directly or indirectly, some results. If my thesis can have any interest at all, it is because it is based on an astronomical

[2] Bernard of Chartres used to compare us to [puny] dwarfs perched on the shoulders of giants. He pointed out that we see more and farther than our predecessors, not because we have keener vision or greater height, but because we are lifted up and borne aloft on their gigantic stature. J. de Salisbury (translation by D. D. MacGarry)

number of previous results. Thanks then to the teams that build and maintain the supercomputers on which I worked: Curie, Marconi et Occigen. Without them, no big simulation could have been run during this thesis. Thanks also to all the people who develop and maintain the programming languages, the softwares and the libraries that I used during this thesis. Without them, we would have to constantly reinvent the wheel and lose precious time. Finally thanks to the people who run Internet help forums, where I found the answers to most of my technical questions during my thesis: Stack Overflow and LaTeX Stack Exchange. Without them, this manuscript could not have been written with so much care in the details.

I would now like to thank all my friends, and more particularly André. His countless pieces of advice during my thesis, even on my manuscript, were of great help.

Even though the persons I mentioned have great qualities, I have to say that my studies would never have been possible without a continuous and unconditional support from my parents. I think we never tell enough our parents how much we love them and how much we care for them: dad, mum, thanks a lot for everything.

I would like, of course, to thank the one without whom all those thesis years would not have been possible, the one who took care of our son when I was working in Strasbourg, the one who supported me during those years, the one thanks to whom I was given the best conditions to finish writing this manuscript. Thanks from the bottom of my heart.

Last but not least, I would like to thank my children for their constant joy and their numerous smiles. It is also thanks to them if this thesis could come to an end, and they are the ones who remind me that we should look at all subjects, even the most complex ones, with children's eyes to better understand them.

Foreword

In this thesis, we are interested in solving systems of equations useful in the domain of plasma physics. Our main focus is on the efficiency of implementations that solve those equations, on multi-core architectures. Even though the main application of our implementation considered in this manuscript is plasma physics, it is possible to apply it in other contexts, *e.g.*, astrophysics.

In many cases, when reading scientific articles or manuscripts, several questions remain unanswered for the reader. Among those unanswered questions, one of them appears in the vast majority of articles that present an optimization. This question is: "How can the optimization(s) presented in a given article be brought into another implementation?". We identified several reasons why this occurs:

- a first reason is one's lack of expertise in the subject. Most of the times, everything seems straightforward for an expert, and the details needed for the non-experts are not present;
- a second reason is the fact that the field of plasma physics is multi-disciplinary. It contains high-level mathematics, high-level physics, and high-level computer science. If the focus of a particular paper is not on the computer science aspect, there is very little chance that any technical detail will be given in that direction;
- a third reason is the fact that most computer science publications are in conference proceedings, where there are page limits. In this space, you have to present the background, convince (mostly the reviewers, but also the future readers) that the work is new and good, and provide results. There is not so much room for technical details;
- a fourth reason lies in the code itself. Explaining the technical details often means that you have to show some part(s) of the code. Most of the time, the code is not shown, so it cannot really be explained. When it is shown, it is mostly with a link to a repository, then "good luck have fun" to dive into it and understand what is inside.

In this manuscript, our goal is to get rid of the problems identified in the previous items, and to provide to the non-expert ways to appropriate themselves the work that will be presented. The focus will thus be given on the computer science part of the work, and we will try to explain all the parts of the code that need explanations. The implementation has been peer-reviewed and earned a "Best Artifact Award" at the Euro-Par 2018 conference [206]. A more recent repository is available at `http://www.barsamian.am/Pic-Vert/`, that contains some comments in the different files, in addition to the ones provided in this manuscript. The organization of this repository is explained in Section 3.4.2. We argue in Chapter 3 that our implementation is 3 times faster than other recent and comparable implementations on the same architecture.

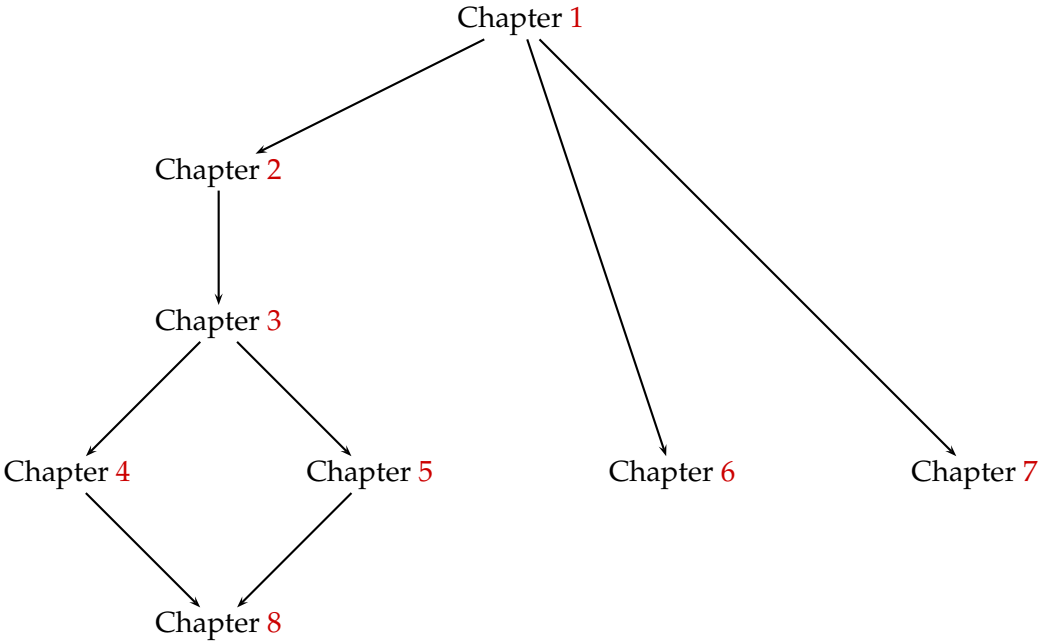
The general structure of this manuscript is the following:

- First, Chapter 1 gives the general context of the present study. Why are we considering plasma physics? What exact problem are we solving? How can we implement a solution of this problem and judge of its efficiency?
- Then, Chapters 2, 3, 4, 5, and 8 are devoted to a specific method to solve the problem considered: the Particle-in-Cell method. Chapter 2 introduces this method, Chapter 3

gives an overview of our contributions, Chapters 4–5 explain the details of our work on this method, and Chapter 8 concludes this work.

- Chapter 6 is an independent chapter that describes another method to solve the same problem: the semi-Lagrangian method. It first introduces this method, explains our contributions, and concludes them by showing possible future work.
- Chapter 7 is another independent chapter which explains the numerical results obtained with our implementations. Although both Particle-in-Cell and semi-Lagrangian methods are discussed, this chapter can be understood without deep knowledge of those methods. The main purpose of this chapter is to verify our implementation, by showing that on particular test cases, it behaves as expected.

In summary, we have two chapters that introduce our work, followed by a brief third chapter explaining our contributions, and finally five chapters explaining the details of our contributions. The general dependencies of the different chapters in this manuscript can be depicted as follows:



Chapter 1

General Background

During this thesis, we worked on efficient implementations of computer simulations in the domain of plasma physics. This introductory chapter will give insights into the physical and mathematical backgrounds that surround the design of this implementation. If any of the notations used are hard to understand, please refer to Chapter A.

First, Section 1.1 motivates the need for simulations in the area of plasma physics, explains what is plasma, and gives some insights into the physics involved.

Section 1.2 then presents the mathematical equations that govern those simulations, and some methods to solve them numerically.

Section 1.3 finally introduces what is needed in terms of computer science to understand the optimizations performed throughout this thesis.

1.1 Some Physical Background

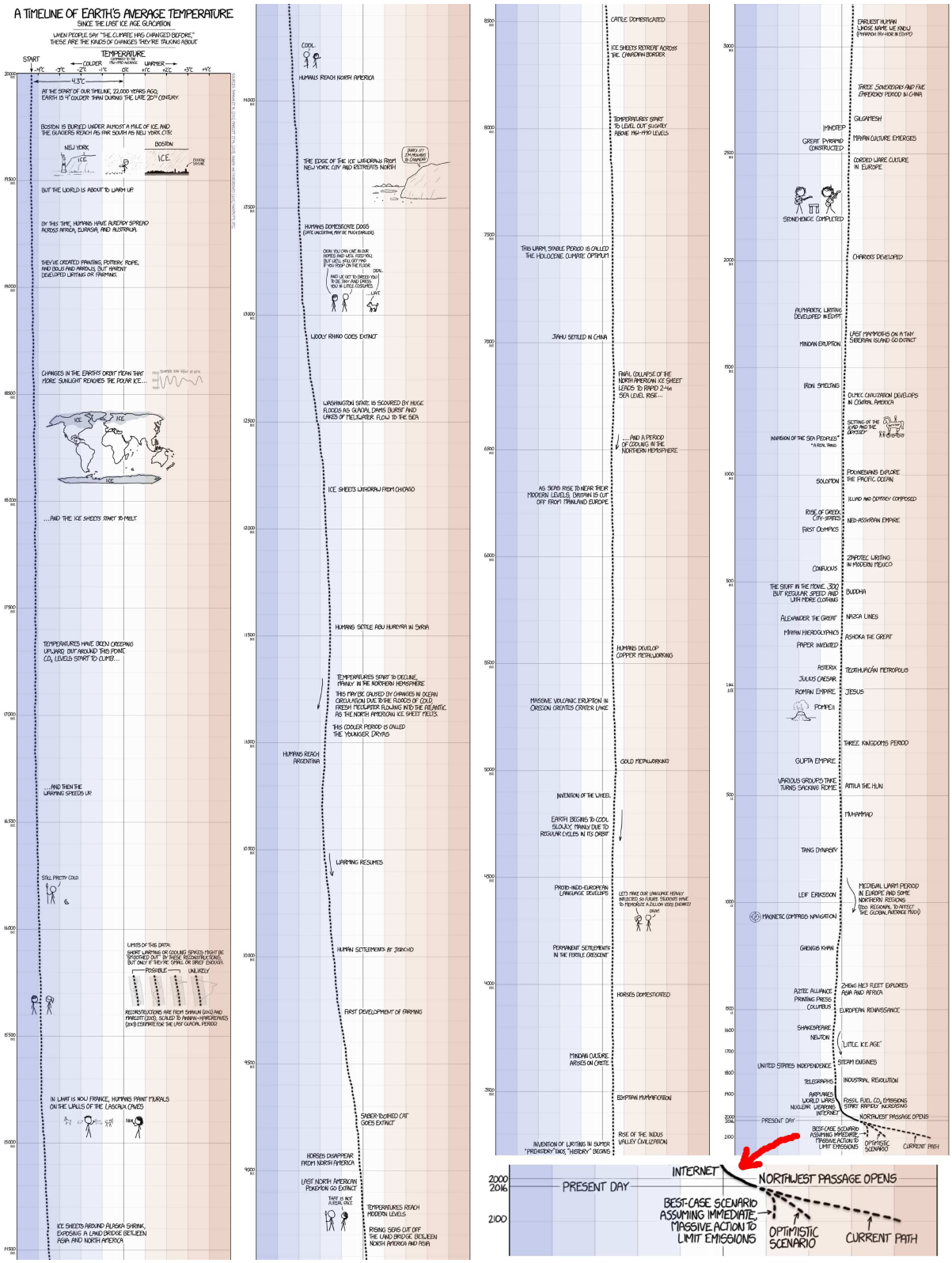
1.1.1 Energy

> “Today our planet is thoroughly wedded to fossil fuels in the form of oil, natural gas, and coal. Altogether, the world consumes about 14 trillion watts of power, of which 33 percent comes from oil, 25 percent from coal, 20 percent from gas, 7 percent from nuclear, 15 percent from biomass and hydroelectric, and a paltry .5 percent from solar and renewables.”
>
> M. Kaku [24, Chapter 5]

Mastering energy is not new in human history. After making other animals work for us, making some first use of “renewable” sources of energy (windmills, water wheels…), we then discovered new ways to use sources of energies: coal, then oil.

However, what is new with these kinds of energy we are now using in our everyday life is that they have massive effects on our planet. International summits (*e.g.*, the Kyoto Protocol in 1997), international research groups (*e.g.*, IPCC — Intergovernmental Panel on Climate Change, http://www.ipcc.ch) explain how everything might go wrong if we continue in this way, and are trying to devise paths to change our future and avoid massive global warming, see Figure 1.1.

Nowadays, a lot of people praise the “green” energy which they define as the one given by wind and sun: wind turbines, solar panels… It could be the future of our energy, because they release less carbon dioxide (CO_2). However, there are still problems in availability (we cannot control when wind blows and still need energy when it does not) and in storage (batteries are not powerful enough to store all the energy produced; pumped-storage hydroelectricity provide some storage facility, but in a very limited amount). Reducing CO_2 emissions with those renewables is only possible if we reduce our use of fossil fuels. Unfortunately, coal plants



"[After setting your car on fire] Listen, your car's temperature has changed before."

Credits — Randall Munroe, https://xkcd.com/1732/; Sources — Shakun *et al.* (2012) [168], Marcott *et al.* (2013) [161], Annan and Hargreaves (2013) [131], HadCRUT4 (https://www.metoffice.gov.uk/hadobs/hadcrut4/), IPCC (http://www.ipcc.ch/).

Figure 1.1 – A Timeline of Earth's Average Temperature.

(that release a lot of CO_2) are commonly used to produce energy when wind turbines and solar panels do not (*e.g.*, during the night).

A source of energy which exists nowadays and has the good property of releasing less CO_2 than the traditional fossil fuels is the energy produced in nuclear power plants, thanks to the fission of atoms. We must nevertheless acknowledge that nuclear power plants are a controversial topic today: the Three Mile Island (1979), Chernobyl (1986) and Fukushima (2011) nuclear accidents changed our acceptance of this energy. Political mistakes did not help either to make people accept them. We can think of mistakes involving bribing local politicians whose land have uranium resources, we can think of large amounts of money lost to buy uranium mines from which nothing can be extracted [196], etc.

Even though fission is today among the energy productions which release the least amount of CO_2, there remains one last problem: when producing energy with fission, some "nuclear wastes" are created whose half-lives can be greater than 200 000 years. One topic of interest is thus how to explain with clear symbols, at the places where we stock them, that those wastes are dangerous so that people will understand these symbols in thousands of years [129]. Another topic of interest is to actually build places that can accommodate nuclear wastes for such a long time [154].

> "Les scientifiques annoncent ainsi l'avènement des piles à combustible, de la fusion par laser ou par confinement magnétique, des véhicules à hydrogène ou à sustentation magnétique, et même des centrales solaires placées en orbite autour de la Terre.[1] [24, Chapter 5]
>
> G. Pitron [199]

Faced with these problems with energy production, researchers try to give life to new forms of energy production. One of them, which is a major application of our work, is the *controlled thermonuclear fusion*. Fusion is a reaction which can be seen as the opposite of fission. Fission creates energy by "breaking" big nuclei (*e.g.*, uranium) into smaller nuclei. Fusion creates energy by "merging" small nuclei (*e.g.*, hydrogen) into bigger nuclei. This is a first good point for fusion because hydrogen is easier to get than uranium: we can produce deuterium and tritium (the required hydrogen isotopes) and do not need to find mines containing them.

When producing energy with fusion, no direct radioactive wastes are created. But the massively energetic neutrons created will irradiate the surrounding structure, and there is still research ongoing to know how this can be tamed. It is thus still a major challenge to create energy thanks to fusion. The international ITER[2] project, located at Cadarache (France), aims in this direction.

Projects like ITER are very costly. Before using big devices such as a 29 m $\times$ 28 m *tokamak*[3] — see Figure 1.2 — we have to know what is inside and how it behaves.

> "La fusion est prometteuse, mais ne pourra guère être industrialisée avant la fin du XXI^{e} siècle.[4]
>
> B. Barré [192, Chapter 13]

Before explaining what is inside the big doughnut pictured on Figure 1.2, we must acknowledge that fusion will probably not make energy available for everyday use before the end of

[1] "Scientist announce the era of fuel cells, of laser fusion or controlled thermonuclear fusion, of vehicles propelled by hydrogen or by magnetic levitation, and even of solar plants in orbit around the earth." G. Pitron (translation by this manuscript's author)

[2] International Thermonuclear Experimental Reactor, "The way" (in Latin): http://www.iter.org

[3] Токамак: тороидальная камера с магнитными катушками (a toroidal chamber with magnetic coils)

[4] "Fusion is promising, but it will probably not be ready for widespread commercialization before the end of the XXIst century." B. Barré (translation by this manuscript's author)

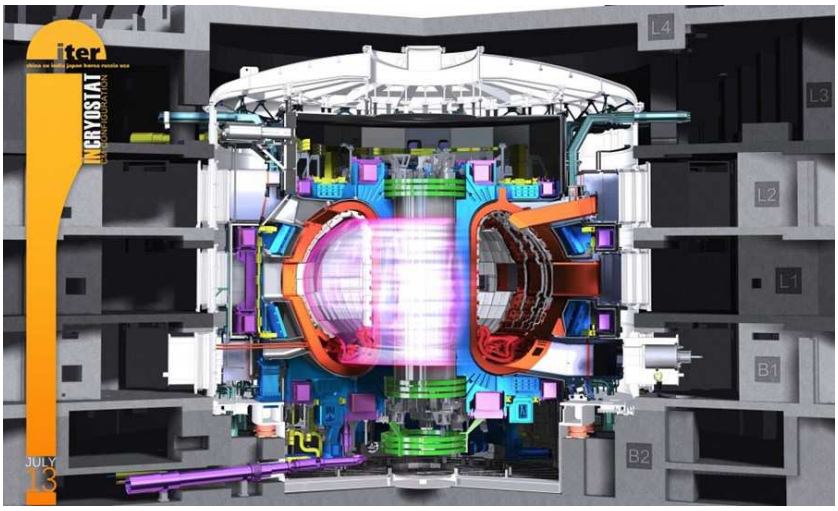


Figure 1.2 – The ITER tokamak — artist view. *Image courtesy of the ITER Organization*

our century. So, even if we think it is a good idea to invest our time and money into this promising source of energy, we also have to make changes in our everyday lives in order to use less energy until then. Let us finish this section with an optimistic view and bet that we will make clear political steps towards the direction of a more reasonable world, where economic growth will not always be our main goal.

1.1.2 Plasma

> “ This reminds [Langmuir] of [...] the way blood plasma carries around red and white corpuscles and germs. So he proposed to call our ‘uniform discharge’ a ‘plasma’.
>
> H. M. Mott-Smith [164] ”

To produce energy with fusion, the idea of the ITER project is to create a plasma inside a tokamak, then contain that plasma inside with a strong magnetic field. The plasma is the fourth state of matter, and is believed to form 99% of the mass of the visible universe. Matter reach this state at very high temperature ($> 10\,000$ K). At this temperature, *particles* (ions and electrons) behave differently. Plasma can be found, *e.g.*, in a lightning, in a neon tube, in the sun. Mastering controlled thermonuclear fusion is thus nothing else than putting the sun in a doughnut. More information about plasma can be found at `http://www.plasmas.org`.

To understand how a plasma behaves, we have to study the particles inside it, and thus we have to track their spatial positions $\vec{x} = (x, y, z) \in \mathbb{R}^3$ and their velocities $\vec{v} = (v_x, v_y, v_z) \in \mathbb{R}^3$ across time. To track the particles, there are essentially three models in plasma physics:

- the *N-body model*: in this model, we follow each of the N particles, taking into account interactions between every couple of particles, hence leading to $\Theta(N^2)$ interactions. This model is the most precise one. While there exist formulas for $N = 2$, this problem requires approximations even for $N = 3$. However, because of the complexity of this model, we may have to wait for quantum computers to use this method efficiently (in a plasma with $N > 10^{10}$ particles, there are more than $10^{20} = 100\,000\,000\,000\,000\,000\,000$ interactions);

- the *kinetic model*: instead of following every particle, we study the particle density f (a function of seven variables: three for positions, three for velocities, and one for time), which gives the probability of presence of particles around a given time, a given position and a given velocity. The different *species* of particles in the plasma have different impacts

on it, so we have to track one density function per species. Usually, we have electrons and one type of ions, so we have to track f_e (for electrons) and f_i (for ions);

- the *fluid model*: this model can be used when the density function f follows some properties. We can then have some less costly equations to solve, that give a macroscopic view of the plasma, rather than a microscopic one.

In this thesis, we focus on the kinetic model: we numerically solve the Vlasov–Poisson system of equations, see Section 1.2.1. Those equations take place in the so-called *phase space* (a six-dimensional space: three for positions, three for velocities). Sometimes we can use simpler models with less dimensions. For example in our thesis, we simulated test cases in a 1d1v phase space (1 dimension for positions, 1 dimension for velocities: 2 dimensions in total), but also in 2d2v, 2d3v, and 3d3v.

1.2 Mathematical Background

1.2.1 Set of equations

> “ [...] вследствие большой массы ионов в сравнении с электронами можно их перемещением пренебречь, т.е. считать ионы фактически неподвижными[5]
>
> А. А. Власов [178, Section 2] ”

The main equation we have to solve is due to Vlasov [178, Equation II]. It can be reformulated as “*the distribution function f is conserved along the trajectories of the particles which are determined by the mean electric field*” [35, Section 2.1]. In the non-relativistic case, it means that $\frac{\mathrm{d}f(\vec{x},\vec{v},t)}{\mathrm{dt}} = 0$, which leads to:

$$\underbrace{\frac{\mathrm{d}\vec{x}}{\mathrm{dt}}}_{\vec{v}} \cdot \nabla_{\vec{x}} f + \underbrace{\frac{\mathrm{d}\vec{v}}{\mathrm{dt}}}_{\vec{a}} \cdot \nabla_{\vec{v}} f + \underbrace{\frac{\mathrm{dt}}{\mathrm{dt}}}_{1} \frac{\partial f}{\partial t} = 0 \tag{1.1}$$

He also stated that the only force acting on particles is the electromagnetic force, also called the *Lorentz force*:

$$\vec{F}(\vec{x},t) = q(\vec{E}(\vec{x},t) + \vec{v}(t) \times \vec{B}(\vec{x},t)) \tag{1.2}$$

With the notations:

- $\vec{E}(\vec{x},t)$: electric field
- $\vec{B}(\vec{x},t)$: magnetic field

Newton’s second law states that, in the non-relativistic case, the sum of the forces is equal to the mass times the acceleration:

$$\sum \overrightarrow{\text{forces}} = m \cdot \vec{a}$$

This law together with (1.2) implies that $\vec{a} = \frac{q}{m}(\vec{E} + \vec{v} \times \vec{B})$. Replacing $\vec{a}$ by this value in (1.1), we get the Vlasov equation that we use throughout this thesis, see (1.3).

$$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f + \frac{q}{m}(\vec{E} + \vec{v} \times \vec{B}) \cdot \nabla_{\vec{v}} f = 0 \tag{1.3}$$

[5] “[...] because of the large mass of the ions as compared to that of the electrons we can neglect their displacements i.e., we can practically assume the ions to be immovable” A. A. Vlasov (translation by D. ter Haar)

This equation is coupled with the four Maxwell equations that allow to deduce the self-consistent fields $\vec{E_s}(x,t)$ and $\vec{B_s}(x,t)$, see (1.4).

$$\begin{cases} \operatorname{div} \vec{B_s} = \vec{0} & \text{Maxwell–Thomson} \\ \operatorname{rot} \vec{E_s} = -\dfrac{\partial \vec{B_s}}{\partial t} & \text{Maxwell–Faraday} \\ \operatorname{div} \vec{E_s} = \dfrac{\rho}{\epsilon_0} & \text{Maxwell–Gauss} \\ \operatorname{rot} \vec{B_s} = \mu_0 \left(\vec{J} + \epsilon_0 \dfrac{\partial \vec{E_s}}{\partial t} \right) & \text{Maxwell–Ampère} \end{cases} \tag{1.4}$$

With the notations:

- ϵ_0, μ_0: vacuum permittivity and vacuum permeability, related to the speed of light: $c = \dfrac{1}{\sqrt{\epsilon_0 \cdot \mu_0}}$
- $\rho(\vec{x},t) = q \int f(\vec{x},\vec{v},t)\,\mathrm{d}\vec{v}$: volume density of electric charge
- $\vec{J}(\vec{x},t) = q \int f(\vec{x},\vec{v},t)\vec{v}\,\mathrm{d}\vec{v}$: current density

In this thesis, there is no external electric field. In some cases, there are no external fields at all, which means that the fields are exactly the self-consistent ones: $\vec{E}(\vec{x},t) = \vec{E_s}(\vec{x},t)$ and $\vec{B}(\vec{x},t) = \vec{B_s}(\vec{x},t)$. In other cases (as in the tokamak of the ITER project), there is an external magnetic field $\vec{B_e}(\vec{x},t)$ which means that $\vec{B}(\vec{x},t) = \vec{B_s}(\vec{x},t) + \vec{B_e}(x,t)$.

In some cases, we can simplify the Maxwell equations: we use the simplifying hypothesis $||\vec{v}|| \ll c$, which implies that we can neglect the current density and the magnetic field. We are left with only one equation instead of four, a Poisson equation:

$$\boxed{-\Delta_{\vec{x}}\phi = \frac{\rho}{\epsilon_0}} \quad \text{where} \quad \vec{E}(\vec{x},t) = -\nabla_{\vec{x}}\phi(\vec{x},t)$$

Most of the time, this thesis will focus on the Vlasov–Poisson system, in the special cases where:

- there is no external field
- the self-consistent magnetic field is neglected
- the ions are supposed motionless (and neutralize the charge), so we only simulate the electrons of charge $q = -\mathrm{e}$ (where e is the elementary charge)

This gives the system of equations (1.5).

$$\begin{cases} \dfrac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f + \dfrac{q}{m} \vec{E} \cdot \nabla_{\vec{v}} f = 0 & \text{Vlasov} \\ -\Delta_{\vec{x}}\phi = \dfrac{\rho}{\epsilon_0} \left(= \dfrac{q}{\epsilon_0} \left(\displaystyle\int f(\vec{x},\vec{v},t)\,\mathrm{d}\vec{v} - 1 \right) \right) & \text{Poisson} \end{cases} \tag{1.5}$$

Moreover, in this thesis we will work in an dimensionless world where $\mathrm{e} = m = \epsilon_0 = 1$ (hence $q = -1$). Sometimes, we will add an external constant magnetic field, sometimes we will simulate both electrons and ions, and we will have to change the system accordingly.

1.2.2 Numerical Approximations

> “On pourroit aussi construire une table des différences premières & l'employer seule. [...] [L]e procédé est aussi exact que la méthode des secondes différences, mais il est moins commode. Ce qui vient en partie de ce que [...] pour les différences premières il faut les prendre dans la table subsidiaire avec $(u + \frac{1}{2}du)$.[6]
>
> J.-B. Delambre [142, Article X] ”

Recall that it is difficult to prove the existence of solutions for the general system in a rigorous way. Let us allow ourselves the shortcut $f(t) = \{f(\vec{x}, \vec{v}, t) \mid (\vec{x}, \vec{v}) \in \text{phase-space}\}$, which represents all the values of f at time t. We thus have no formula giving $f(t)$ for all t.

Numerical Schemes

We use a numerical method to have approximations of the solution:

- $f(0)$ is known (initial condition)
- choose a “small” Δt (time step) and n (number of time steps)
- deduce $f^*(\Delta t)$, an approximation of $f(\Delta t)$, from $f(0)$
- then $f^*(2\Delta t)$, an approximation of $f(2\Delta t)$, from $f^*(\Delta t)$
- then $f^*(3\Delta t)$, an approximation of $f(3\Delta t)$, from $f^*(2\Delta t)$
- ...
- and $f^*(n\Delta t)$, an approximation of $f(n\Delta t)$, from $f^*((n-1)\Delta t)$

Euler proposed a numerical scheme [13, Part I, Section II, Chapter VII, Problem 85] that we here sketch when working with functions from $\mathbb{R}$ to $\mathbb{R}$. The unknown is f; we know g, a differential equation $f'(t) = g(t)$, and $f(a)$; we wish to know $f(b)$. “Close enough” around any value x, f behaves like an affine function (its derivative), which is to say that, for $x \in [a; b]$ and for small values of h,

$$f(x+h) = f(x) + h \cdot f'(x) + O(h^2) \qquad \text{(Taylor expansion, first order)} \tag{1.6}$$

With n steps of size $\Delta t = \frac{b-a}{n}$, we may retrieve a good approximation of $f(b)$, by stating that:

$$\begin{cases} f^*\left(a + \frac{b-a}{n}\right) = f(a) + \frac{b-a}{n} \cdot g(a) \\ \forall 2 \leq k \leq n, f^*\left(a + k \cdot \frac{b-a}{n}\right) = f^*\left(a + (k-1) \cdot \frac{b-a}{n}\right) + \frac{b-a}{n} \cdot g\left(a + (k-1) \cdot \frac{b-a}{n}\right) \end{cases}$$

The Euler scheme is a *first order* method: the error is at most proportional to Δt[7]. A more accurate (*second order*) numerical scheme is the leap-frog method, which has been (re)discovered a lot of time, see *e.g.*, [142, Article X], [172, Chapter III], [176], hence its name, the Verlet–Störmer–Delambre algorithm. Its application to the Particle-in-Cell (PIC) method is explained in [5, Section 2.4].

Other well-known numerical schemes are the Runge–Kutta methods [18, Chapter 2], which can be of any order[8]. Last but not least, there are also some *splitting* methods which are specific

[6] “One might also construct a table of the first differences & use it alone. [...] [T]his process is as exact as the second differences method, but is less practical. This partly comes from the fact that [...] for the first differences, one has to take them in the auxiliary table with $(u + \frac{1}{2}du)$.” J.-B. Delambre (translation by this manuscript's author)

[7] We have n steps, and at each step the error is at most proportional to $(\frac{b-a}{n})^2$, see (1.6). The constant thus depends on the length of the interval $|b-a|$ and on $sup_{x\in[a;b]}|g'(x)|$.

[8] In fact, the Runge–Kutta method of first order is the Euler method.

to some kind of differential equations, *e.g.*, [94] for the Vlasov equation, which can be used in the semi-Lagrangian method.

Method of Characteristics

Using one of those numerical schemes is not straightforward to solve the Vlasov–Poisson system, because the Vlasov equation is a partial differential equation. It is easier to solve ordinary differential equations. This can be done with the help of the method of characteristics [15, Section 1.4][14, Section 3.2].

The idea is to use parametric functions instead of the variables t, $\vec{x}$ and $\vec{v}$. We thus introduce three functions $T : \mathbb{R} \to \mathbb{R}$, $\vec{X} : \mathbb{R} \to \mathbb{R}^3$ and $\vec{V} : \mathbb{R} \to \mathbb{R}^3$. We search a differential equation which involves $f(\vec{X}(s), \vec{V}(s), T(s))$. We now compute $\dfrac{\mathrm{d}f(\vec{X}(s), \vec{V}(s), T(s))}{\mathrm{d}s}$ which leads to:

$$\frac{\mathrm{d}\vec{X}(s)}{\mathrm{d}s} \cdot \nabla_{\vec{x}} f(\vec{X}(s), \vec{V}(s), T(s)) + \frac{\mathrm{d}\vec{V}(s)}{\mathrm{d}s} \cdot \nabla_{\vec{v}} f(\vec{X}(s), \vec{V}(s), T(s)) + \frac{\mathrm{d}T(s)}{\mathrm{d}s} \frac{\partial f}{\partial t}(\vec{X}(s), \vec{V}(s), T(s))$$

Now if we set $\dfrac{\mathrm{d}\vec{X}(s)}{\mathrm{d}s} = \vec{V}(s)$, $\dfrac{\mathrm{d}\vec{V}(s)}{\mathrm{d}s} = \dfrac{q}{m}\vec{E}(\vec{X}(s), s)$ and $\dfrac{\mathrm{d}T(s)}{\mathrm{d}s} = 1$, we recognize the left-hand side of the Vlasov equation (1.5). This is thus what we do. We choose $T(s) = s$ as a natural solution of $\dfrac{\mathrm{d}T(s)}{\mathrm{d}s} = 1$ and we thus find that the characteristics of the Vlasov equation are solution of the system of ordinary differential equations (1.7).

$$\begin{cases} \dfrac{\mathrm{d}\vec{X}(s)}{\mathrm{d}s} = \vec{V}(s) \\ \dfrac{\mathrm{d}\vec{V}(s)}{\mathrm{d}s} = \dfrac{q}{m}\vec{E}(\vec{X}(s), s) \qquad \text{(Newton's second law)} \end{cases} \tag{1.7}$$

We also have a very important property of f, which comes from the Vlasov equation (1.5). This equation tells us that $\dfrac{\mathrm{d}f(\vec{X}(t), \vec{V}(t), t)}{\mathrm{d}t} = 0$ which means that we have the following property:

Property of the Vlasov–Poisson system
If f is a solution of the Vlasov–Poisson system, then f is constant along the characteristics of the Vlasov equation.

This property allows us to follow f in time by simply following the characteristics.

Verification of the Implementation

In order to verify our implementation and understand how much error is made by the numerical simulation, we have some test cases on which we have almost-exact theoretical solutions, thanks to a dispersion analysis, *e.g.*, [157]. Some test cases and their theoretical solutions can be found in [35, Chapter 4]. We also designed new test cases with their theoretical solutions during this thesis [207].

Space Discretization

In addition to errors inherent to the discretization of time, we must also cope with errors inherent to the discretization in space. With computers that have only finite memory, we cannot have access to all the "values of $\vec{E}$ at time t": for a given t, a computer cannot store $\vec{E}(\vec{x}, t)$ for all $\vec{x}$. We store values on a grid, for example for the x-axis:

- the physical space in which the particles evolve can always be chosen as $[x_{\min}; x_{\max}]$:
  - when it is periodic (*e.g.*, a torus), we choose $x_{\max} - x_{\min} = \mathit{period}$ (periodic boundaries)
  - when it is closed (particles do not go further than a given limit), we know that f is 0 beyond this limit (free boundaries)
  - in some other cases, the physical space is evolving at each time step, i.e. $x_{\min}$ and $x_{\max}$ depend on time (moving window)
- we choose a "small" Δx and store only $\frac{x_{\max} - x_{\min}}{\Delta x}$ different values on the x-axis
- for each time step, we only store $\vec{E}$ values on the grid

1.3 Some Computer Science Background

Our goal in this thesis is to design efficient parallel algorithms. It is exactly the same task as when you want to cook as fast as possible with friends. The algorithm will be your recipe, and the parallelism will come from the different persons working at it.

- Butter and flour a soufflé dish.
- Put the dish into a fridge.
- Preheat the oven to 200 °C
- Wash and stalk 300 g of fresh spinach.
- Cook the spinach with 10 g of butter in a frying pan for 4-5 minutes.
- Drain the spinach and chop them with a knife.
- Separate 4 egg whites and yolks.
- Prepare 300 mL of béchamel sauce.
- Mix the béchamel, the yolks, the spinach and a pinch of curry powder.
- Whisk the egg whites with a pinch of salt.
- Incorporate them delicately to the previous mixture with a spatula.
- Fill the dish with that mixture.
- Cook it in the oven for 30 minutes, door closed.
- Serve it as soon as it is ready.

Figure 1.3 – Spinach soufflé recipe.

Figure 1.3 shows a really good cooking recipe: a spinach soufflé — adapted and translated from `https://cuisine.larousse.fr/recette/souffle-aux-epinards`. First, the recipe contains a lot of small orders, that everybody understands (butter, flour, put, preheat, wash, stalk, cook, drain, chop, separate, mix, whisk, incorporate, fill, serve). They correspond to what is called *instructions* of an algorithm. Then, there is a special line that we might not understand (prepare a béchamel). It corresponds to what is called a *function*. If we do not know what this function is doing, we can always go and check for the instructions given in its definition. There sure is a cooking recipe for a béchamel sauce elsewhere. Finally, there is an order in which those instructions have to be executed. Following the order given in the recipe will always work, but some reordering might be possible without changing the final result in some cases. For example here, you may separate the eggs at the very beginning of the recipe without changing the result (*e.g.*, if you fear to break the yolks, as they tend to break more easily when the eggs are warmer).

Knowing when we can reorder instructions is one of the keys to cook with friends. Trying to make multiple computers cooperate on a given program (the goal being to compute faster) is *parallelism*. You have multiple options to cook in parallel:

- *You can do different tasks in parallel.* A common solution to cook with friends is to have each of you participate on different parts (starter, meal, desert). Even inside one of those parts, sub-tasks can be performed at the same time (*e.g.*, the preparation of the spinach and of the béchamel sauce).
- *You can pick a task, and divide it among the participants.* Here, stalking the spinach takes some time, and you can do it efficiently in parallel: each person will take a portion of the spinach and stalk it, at the same time as others.
- *You can pipeline different tasks.* This option does not come naturally in our previous example, but comes naturally to our mind when we then think of the desert we will make after this delicious soufflé: an apple pie. It is quite natural and efficient to have someone peel the apples and someone else cut them. At first, the one in charge of cutting them has no apple to cut, then as soon as the first apple is peeled, he has work to do until the end. Conversely, the one in charge of peeling has work to do from the beginning, but has no apple to peel when the other one is cutting the last apple (things may vary a little if peeling and cutting an apple have two substantially different timings). The idea is to gain time because only one action is performed. Using this technique with humans should always be made with care [195], but it is great with specialized computer chips.

In our thesis, we will focus on the second item: divide tasks among participants. In a perfect world, asking 2 people to perform a task which would take 1 hour for 1 person should lead to a 30 minute task. The perfect *speedup* (timing when performing the task alone divided by timing when doing it with multiple people) is 2. However, some parts of a program do not behave that well. For example, even if we divide the soufflé in ramekins and no matter how many ovens we have, we still have to wait 30 minutes for the final cooking. The rest of the recipe takes approximatively around 30 minutes also for 1 person. We thus realize that, with 2 people working on the recipe, we cannot make it in less than 45 minutes. Which corresponds to a speedup of $\frac{4}{3} \approx 1.33$. We re-discovered the so-called Amdahl's law [130]. Amdahl roughly stated that the inherently sequential part of any program will become the main bottleneck for parallel programming. Fortunately, his prediction was not so good and in our target programs, those parts are so tiny that they are negligible even in parallel, and do not affect the speedup.

1.3.1 Parallelism

> “ Today parallelism is available in all computer systems, and at many different scales starting with parallelism in the nano-circuits that implement individual instructions, and working the way up to parallel systems that occupy large data centers.
>
> U. A. Acar & G. E. Blelloch [1, Section 1.1] ”

Realistic simulations involve trillions of bytes of memory[9], and are executed for millions of time steps. A typical personal computer has around 4 to 16 GB (billions of bytes) of memory. For example our everyday computer has 16 GB of memory. Asking for 8 additional GB from the usual 8 GB available on the model on which this manuscript was written costs 100€. This memory is far from being enough to run realistic simulations. Even computers specialized for scientific computation do not have enough memory. They typically have 100 GB of memory. We thus cannot avoid the use of multiple computers and therefore use clusters of multiple computers, or so-called *supercomputers*.

Executing programs on those machines requires to have at least some basic knowledge about parallel programming and about modern computer architecture. We will try to fill that gap, if needed.

[9] A byte is 8 bits, where a bit is 0 or 1, the basic unit of computation of a computer.

Because we are working on a cluster of computers, we cannot work without the distributed memory paradigm. Fortunately, this is the easiest model and in fact, everything can be programmed with this model. But if we want to unleash the full power of our machines, we have to also use the shared memory paradigm.

The difference between the two models is quite easy to understand. Suppose you run a flower company. You have one shop in Amiens (Somme, Hauts-de-France, France) and another one in Le Touquet (Pas-de-Calais, Hauts-de-France, France). A customer enters the first shop and asks for a bunch of yellow roses. The seller checks for availability, sees he can fulfill his client's desire, and sells him what he wants. In the other shop, quite at the same time, the same scenario occurs. No problem is involved. At the end of the week, the two shops synchronize together to know the total number of flowers of each kind. When working with distributed memory, you thus need to distribute your data (here, the flowers) among the different computing units (here, the shops), ensure that everybody can work with its own part of the data, and then put together all the computations done to get the final result and/or put back all the pieces of data together (here, get the flowers stock and the cash incomes). Whenever some unit wants to access parts of data it does not have, it can ask a colleague, but this will take some time (here, sending some flowers from one shop to the other will probably takes some hours), and you have to ensure that its colleague is expecting such demands — or else, it will never answer.

Suppose now that we are not in two different shops, but in the same shop with two different sellers. Suppose now this — not very likely in everyday life, but very likely on a computer — scenario occurs: two customers enter the shop. The first one asks for a bunch of five yellow roses, and the first seller goes into the storage room to check for availability and price. He sees seven yellow roses, and goes back to say that there is no problem. His client agrees to buy them at the indicated price. At the same time, another client wants a bunch of three yellow roses. The second seller goes into the storage room just after his colleague, still sees seven roses, and goes back to say that there is no problem. The first client asks for other flowers, while the second one wants to end his shopping. The second seller goes and fetches the three asked roses, and his client goes out with satisfaction. When the first client has finished choosing flowers, the first seller goes back into the storage room... only to find that there are now only four roses left. Of course, to avoid such scenarios, each seller can take the flowers with him when he checks for availability. In computer science, we would call that an `atomic` operation. Rather than just checking a number, then trying later to change its value, we do it in only one pass. If someone else wants to access the value in the meantime, he cannot, because the value is locked during atomic operations. This avoids this scenario, which is called a *race* (two different actors accessing the same datum while at least one of them is modifying it). Of course there is no problem when the two accesses are only reads (both sellers can check the price of a same item without problem). We here saw one benefit of shared memory, together with one of its common pitfalls. When sharing the storage room, the two sellers need not to synchronize to know how many flowers they still have. But they need to be careful when modifying the stock.

The previous example sketched a very common bug when programming with the shared memory model. However, we are not interested only in bug-free simulations, we also want fast simulations. Suppose now that the two sellers of the same shop avoided races by handling the flower stocks on different stages of the storage shelf. The first seller might have the responsibility to handle the roses, on the two lower stages of the shelf. The other one might have the responsibility to handle the sunflowers, on the two upper stages of the shelf. What can happen is that they both need flowers on the shelf at the same time. When this happen, they will probably bother each other, because they are both in front of the same shelf. This scenario occurs on computers if two different actors access two different data that are " too close" in memory: it is called a *false sharing*. First we have to understand that data is organized not number by number but block of numbers by block of numbers in memory. If two different actors access two numbers in the same block, the system cannot be sure that they access two different numbers, and

thus has to take care of the special case where they would access the same number inside this block. To be sure that this never happens, we can put the roses and sunflowers on two different shelves. Seeing that the two actors are operating on two different block of numbers, the system now knows for sure that no race can happen.

Following the work of many other colleagues in the scientific computing community, we will focus in this thesis on the `OpenMP` [182] language extension (for `C`, `C++` and `Fortran`) for the shared-memory parallelism. The different workers in `OpenMP` are called *threads*, they are the different sellers in our flowers example. There are a lot of possibilities from `OpenMP` that we will use in this thesis, but also many others that we will not use. For example `OpenMP` enables what is called *task-based parallelism* that is used in some PIC implementations, *e.g.*, OSIRIS [33, Section 8.6]. In this thesis, we did not use tasks.

1.3.2 Vectorization

> “ Skill and knowledge of vectorization is absolutely ESSENTIAL to gain performance on the Intel® Xeon Phi™ product family.
>
> https://software.intel.com/en-us/articles/vectorization-essential ”

We have seen distributed-memory parallelism and shared-memory parallelism. There is a third level of parallelism, the vector-level parallelism.

Vectorization is the transformation of the kind of code shown in Listing 1.1 into the one shown in Listing 1.2. In the example, `A`, `B` and `C` are arrays of `doubles`, and we assume that we can do 4 double floating-point operations at once, which requires vectors of size 256 bits (one `double` takes 64 bits of memory).

```
1 for (i = 0; i < 1024; i++)
2     C[i] = A[i] + B[i];
```

Listing 1.1 – Code without vectorization.

```
1 for (i = 0; i < 1024; i+=4)
2     C[i:i+3] = A[i:i+3] + B[i:i+3];
```

Listing 1.2 – Vectorized code.

Listing 1.1 uses scalar instructions. The computer loads `A[0]` and `B[0]` in two scalar registers, computes `A[0] + B[0]`, stores the result into `C[0]`, then loads `A[1]` and `B[1]`, computes `A[1] + B[1]`, stores the result into `C[1]`... On “modern” architectures, there is a possibility, if you apply the same instruction on multiple contiguous data (an array), to apply it block of elements by block of elements, instead of element by element. The computer uses vector instructions rather than scalar instruction. In Listing 1.2, the computer loads `A[0]A[1]A[2]A[3]` in a vector register, similarly loads `B[0]B[1]B[2]B[3]` in another vector register, computes the four additions in one vector instruction, and stores back the vector result into `C[0]C[1]C[2]C[3]`... If the data is not contiguous in memory — you may have indirect accesses (*e.g.*, `A[f(i)]`), non-unit strides (*e.g.*, when using array of structures) — then you may still vectorize the operations but you would need so-called *gather* and/or *scatter* operations to load and/or store the non-contiguous data from and/or to memory.

This parallelism is known under the name SIMD (Single Instruction Multiple Data), and to reach the peak computing performance given by architecture manufacturers, you have to use vector instructions.

In the work presented in this thesis, we carefully wrote our code in order to benefit from distributed memory, shared memory and vector parallelism. See in Section 4.3.4 for applications in a PIC implementation.

1.3.3 Efficiency

> Note that metrics such as flop/s or percentage-of-peak are less relevant for the predominantly memory-bound gyrokinetic PIC methods, as modern architectures require 10 flops per byte moved from DRAM in order to be compute-limited.
>
> W. Tang, B. Wang, S. Ethier, G. Kwasniewski, T. Hoefler, K. Z. Ibrahim, K. Madduri, S. Williams, L. Oliker, C. Rosales-Fernandez and T. Williams [83]

Most of the work achieved during this thesis concerns the optimization of a PIC implementation. Yet, many questions arise if we start to think of optimization:

- How to measure the performance of an implementation?
- How to read/understand performance results?
- How to compare the performance of two different implementations?

To study the performance of our implementation, throughout this thesis, performance measurements were performed with the addition of some lines in the code to get the execution time. It is not new that observing a phenomenon might change that very phenomenon [139]. We took great care to verify that this was not the case here. We can get more detailed information thanks to the use of more complex tools, but we chose the simplicity of this approach.

To study the parallel performance of an implementation, there are usually two tests.

The first test is the strong scaling: we start with a problem of size N solved by P processors. Then, we add some processors, while keeping the total problem size N constant. This shows how suited is the program to solve a given problem on bigger machines.

Ideal parallelism (strong scaling)
We say that we have an ideal parallelism if, when multiplying the number of processors by k, the problem is solved k times faster[a].

[a] Remark: sometimes, super-linear effects are observed: the problem is solved more than k times faster. This happens for example when the division of the problem into smaller ones leads to better cache re-use. In that case, it means that the sequential implementation can probably be improved to also benefit from better cache re-use.

The second test is the weak scaling: we start with a problem of size N solved by P processors, which gives us a problem size per processor N/P. Then, we add some processors, while keeping the problem size per processor N/P constant. This shows how suited is the program to solve bigger problems on bigger machines.

Ideal parallelism (weak scaling)
When considering a problem with linear complexity — $\Theta(N)$, like the PIC algorithm —, we say that we have an ideal parallelism if the execution time does not change while increasing the number of processors.

For example let us look at Figure 1.5, starting from 8 cores (the behavior from 1 to 8 cores is explained hereafter). We see that we have an almost ideal weak scaling up to 512 cores, then our execution time grows whereas it could be constant. This is due to the parallelization scheme chosen that adds a logarithmic factor in the execution time, as discussed in Section 2.6.

Once we know how much time each part of our code takes, it is easy to compare two different implementations. But this tells us nothing about the absolute efficiency of our implementation. In the abstract of our first article [204], we report that "*our code processes 65 million*

particles/second per core on Intel Haswell (without hyper-threading)". How good is that? Later on, we see that there is a "bump" in efficiency when going from 4 to 8 threads [204, Figure 7], see Figure 1.5. How bad is that? Let us find out.

To extract information from the performance results, and have an intuition of what should be a good or a bad result, we need to have at least some basic knowledge about computer architecture, and about properties of our implementation. We could escape this discussion when dealing with basics of parallelism, but now that this introduction is coming to an end, we cannot delay it anymore.

Whenever a computation is performed (*e.g.*, `C[i] = A[i] + B[i]`, line 2 of Listing 1.1), a computing unit will take care of the computation. Before the computation can take place, the data (here, `A[i]` and `B[i]`) has to be loaded from the main memory. After the computation, the result (here, `C[i]`) has to be put in the main memory. In some cases it is a little bit different, but if we are doing this operation on big arrays, this is a really good approximation of what is happening. There are thus two main architectural properties which will contribute to the global efficiency of this loop:

- At which rate can the computations be performed? This is the *frequency* of our processor.
- At which rate can the data be accessed? This is the *memory bandwidth* of our processor.

There is a simple but very useful model that tells us what we can expect from a given implementation, knowing those architectural parameters: the roofline model [180]. When we have a lot of memory to load and write but not a lot of operations to perform, we are limited by memory bandwidth. We say that we are *memory bound*. This is the case for a PIC algorithm (we have $\Theta(N)$ data to read and write, and $\Theta(N)$ operations to perform on those data). When we have not a lot of memory to load but a lot of operations to perform, we are limited by floating-point performance. We say that we are *compute bound*. This is the case for example for usual (dense) matrix multiplications (we have $\Theta(N^2)$ data to read and write, and $\Theta(N^3)$ operations to perform on those data[10]). The parameter we have to look for is thus the *operational intensity* of our implementation. How many floating-point operations do we have per byte we have to move from and to memory? In Listing 1.1, if the arrays contain `doubles` (one `double` takes 8 bytes for storage), we have one operation for 24 bytes moved. This operational intensity is extremely low on modern architectures, and thus our loop will be memory bound. This loop is in fact used in a benchmark to test the maximum memory bandwidth that can be reached: the Stream benchmark [162]. In general, the reachable peak is lower than the theoretical one.

Some paragraphs ago, we wanted to know whether "65 million particles/second" was a good performance result on 1 core. We can now draw the roofline model for our architecture and implementation, see Figure 1.4. Our architecture can attain 68 GB/s for 4 memory channels[11], thus 17 GB/s for 1 core. The frequency is 2.3 GHz and the maximum number of single precision floating-point operations per cycle is 32[12], thus the maximum number of operations per second is 73.6 GFlops/s. Our 2d implementation performs 62 operations per particle, and requires $\approx$ 103 bytes moved per particle. This gives us an operational intensity of $\approx$ 0.60. For this intensity, no implementation can exceed 8.3 GFlops/s, so our implementation that achieves 4.5 GFlops/s is not that bad. Please note that this graph is not sufficient to analyze the performance of a given implementation. Even if we attained the Stream line, nothing tells us that the performance of the implementation cannot be further improved. We later on designed algorithms that require less memory transfers. Thus, the operational intensity increases, and the maximum reachable performance increases.

[10] There exist algorithms for dense matrix multiplications with lower complexity, *e.g.*, the Strassen algorithm in $\Theta(N^{log_2(7)})$ [174] or more recently $\Theta(N^{2.3728639})$ [158], but for values of N that can be handled by a typical computer, using optimized variants of the naive $\Theta(N^3)$ algorithm leads to the best execution times.

[11] http://ark.intel.com/products/81705

[12] https://en.wikipedia.org/wiki/FLOPS

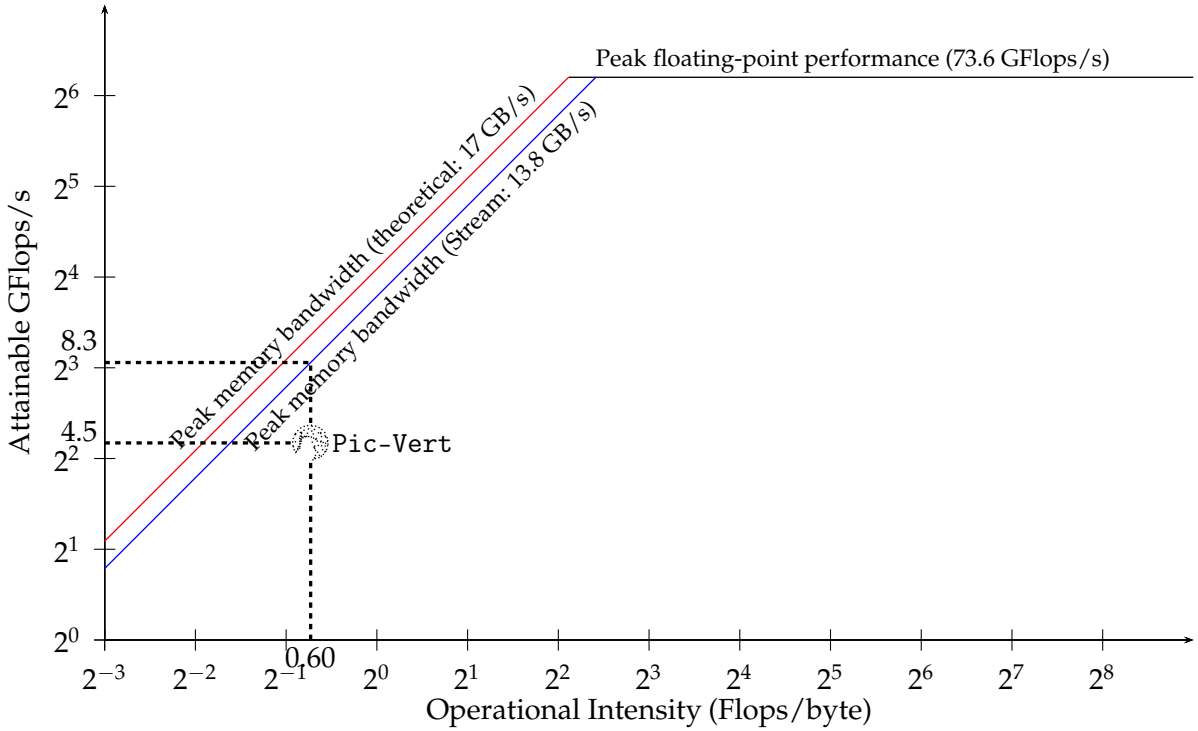


Figure 1.4 – Roofline model for our 2d implementation on 1 core of Intel Haswell.

This model tells us what can be achieved once we have fixed properties for our architecture and our implementation. There is one last step to understand when we are studying scaling properties of an implementation on a processor. On modern architectures, we have more cores than memory channels. Some paragraphs ago, we wanted to know whether it was expected or not that there was a "non-perfect" scaling on our implementation, when going from 4 to 8 cores. Our architecture has 8 cores but only 4 memory channels. With 1 core, this core can use a memory channel. With 2 or 4 cores, each can use a memory channel. But once we reach 8 cores, they will need to share the memory channels. Because our implementation is memory bound, it is thus not a surprise to have this kind of behavior, see the weak scaling (the problem size per processor is constant, thus the total problem size increases with the number of processors) on Figure 1.5.

Comparing efficiency of multiple implementations is a very difficult task. Direct comparison between the performance results from two papers is hard to make, for multiple reasons:

- The architecture parameters usually vary between two papers. We first need to normalize the results depending on the architecture parameters. In this thesis, we propose a normalization which is useful for PIC implementations as well as for any memory-bound implementation: we scale the results depending on the maximal memory bandwidth of the different architectures. This gives a first insight of how well a given implementation behaves with respect to another one, even if the maximal memory bandwidth is not the only useful parameter.

- The implementation details probably vary (for a PIC implementation it can be equations, initial conditions, precision, orders for interpolation and time-stepping...).

To truly compare two implementations, we would need to run them on the same architecture, with the same parameters. It is usually impossible because we rarely have access to the code from another paper, and when we have it, running it on a given architecture might not be fair, as this implementation might not have been optimized for this target architecture. The two

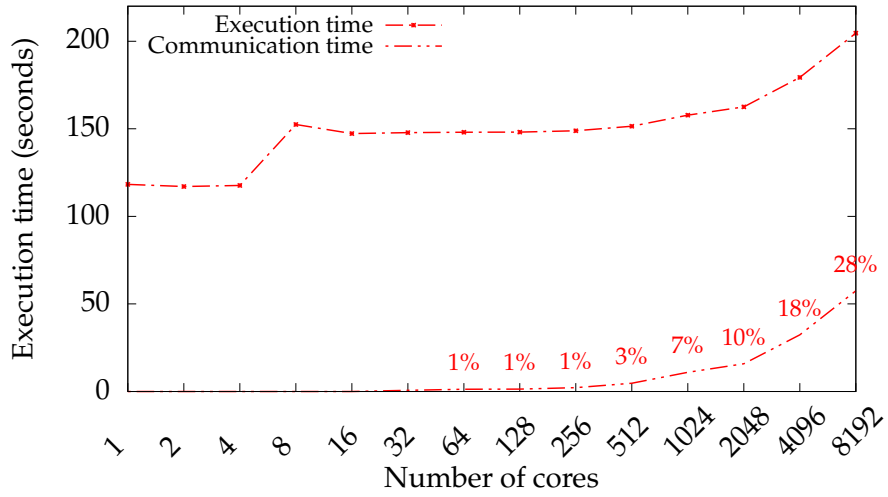


Figure 1.5 – Weak scaling on Curie (Intel Sandy Bridge) for our 2d implementation. Test case: 128×128 grid, 50 million particles per core, 100 iterations simulation (sorting every 50 iterations). Communication time is also shown as percentage of the execution time.

teams that wrote those codes would thus have to cooperate a lot, just in order to compare the performance of their own implementations. This huge amount of time needed is most often not available.

Chapter 2

Preliminaries to the Particle-in-Cell Methods

In this chapter, we introduce the Particle-in-Cell methods. The primary goal of this chapter is to give a general overview of the classical tools needed for this method, and will thus be useful mostly for someone encountering this method for the first time. Nevertheless, this chapter will also introduce a lot of material that will be useful in Chapters 4 and 5, devoted to the optimization of a Particle-in-Cell implementation.

Section 2.1 first gives the general idea of the method, together with a high-level description of the different steps needed. Then, we describe in detail the choices a programmer has to make for the different building blocks needed:

- the data structures, in Section 2.2;
- the particle initialization, in Section 2.3;
- the Poisson solver, in Section 2.4;
- the interpolation schemes, in Section 2.5;
- the process-level parallelism, in Section 2.6.

Finally, Section 2.7 gives links to some other PIC implementations.

2.1 Overview

Let us first recall that, in the case where there is no other external field and the self-consistent magnetic field is neglected, we solve the system (1.5), which we also give here:

$$\begin{cases} \partial_t f + \vec{v} \cdot \nabla_{\vec{x}} f + \dfrac{q}{m} \vec{E} \cdot \nabla_{\vec{v}} f = 0 & \text{Vlasov} \\ f(\vec{x}, \vec{v}, 0) = f_0 & \\ -\Delta \phi = \dfrac{\rho}{\epsilon_0} & \text{Poisson} \end{cases}$$

where

$$\rho(\vec{x}, t) = q \left(\int f(\vec{x}, \vec{v}, t) \, \mathrm{d}\vec{v} - 1 \right) \quad \text{and} \quad \vec{E}(\vec{x}, t) = -\nabla \phi(\vec{x}, t).$$

In this system, $f = f(\vec{x}, \vec{v}, t)$ stands for the distribution of one species of particles (with charge q and mass m) in a six-dimensional phase space (three dimensions for positions and three dimensions for velocities), ρ stands for the charge density, and E for the self-consistent electric field.

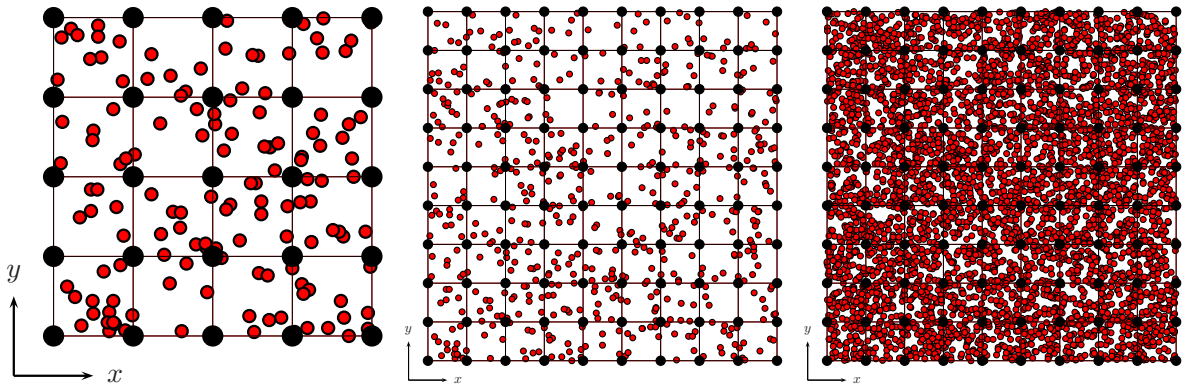


Figure 2.1 – Discretization of f (red) and of the physical space (black).

To solve this system with the PIC method, we approximate the distribution function by a collection of N macro-particles (or numerical particles), see formula (2.1), where δ is the distribution of Dirac [11, Section 15][1]. Theoretical properties of this approximation can be found in [77].

$$f(\vec{x},\vec{v},t) \approx \sum_{k=1}^{N} \text{weight}_k \cdot \delta(\vec{x} - \vec{x_k}(t)) \cdot \delta(\vec{v} - \vec{v_k}(t)) \tag{2.1}$$

One numerical particle represents many real-life particles. This is represented by weight_k which is the weight of the k^{th} particle. There are many ways to handle those weights.

A first question is: how to set the initial weights according to the initial density function f_0? One idea is to uniformly split the phase-space and to put, in each cell, a particle whose weight is equal to the integral of f_0 on this cell [52]. Another idea is to (pseudo- or quasi-) randomly pick particles according to f_0. In this case, all the particles have the same weight:

$$\forall k, \text{weight}_k = w = \frac{1}{N} \int f_0 \, \mathrm{d}\vec{x} \, \mathrm{d}\vec{v}.$$

A second question is: will the number of particles and their weights evolve in time? The δf PIC method does not try to resolve the full distribution function f, but, to reduce noise in the simulations (see next paragraph), expands the distribution function into an equilibrium solution F and a small perturbation δf. In this method, the weights depend on time [39, 76]. Even when coping with the full distribution function, there are methods that change the number of particles and their weights over time, by splitting and merging some particles when necessary [67]. It is nevertheless possible to avoid those considerations and to keep the number of particles and their weights constant over time.

Those particles interact via the self-consistent electric field, which is also discretized on a grid. Figure 2.1 represents particles in red and grid quantities in black. When using a low number of grid cells and a low number of particles per cell (Figure 2.1, left), two problems arise. First, the number of grid cells is not sufficient to take into account physical effects on small scale, thus we have to increase the grid sizes (Figure 2.1, middle), sometimes up to $\approx 1\,000$ per direction. Second, we face numerical noise when the number of particles is small, for two reasons: (a) when initializing particles randomly, we face the traditional stochastic convergence of Monte-Carlo methods in $\frac{1}{\sqrt{N}}$ [20, Section 2.4] and (b) for smooth particle simulations, one should increase the number of particles per cell [41, 77] (Figure 2.1, right), sometimes up to $\approx 10\,000 - 1\,000\,000$.

[1] Its cumulative distribution function is the Heaviside function: $H(x) = 0$ when $x < 0$ and $H(x) = 1$ when $x \geq 0$. As a small abuse of notation, we can see the δ distribution as having the following property: for each function f, $\int_{\mathbb{R}} f(x)\delta(x)\,\mathrm{dx} = f(0)$. This will be enough for our needs.

Parameters
N: number of particles.
$ncx \times ncy \times ncz$: number of grid cells.
Δt: time step.
f_0: initial distribution function.
q and m: particle charge and mass.

Variables
$particles[N]$: set of particles, with position $\mathbf{x_p}$ and velocity $\mathbf{v_p}$.
$\rho[ncx][ncy][ncz]$: charge density.
$\mathbf{E}[ncx][ncy][ncz]$: electric field.

Algorithm
1 Randomly initialize N particles following f_0
2 Compute initial ρ and $\mathbf{E}$
3 **Foreach** time step — Leap-frog
4 **If** $(condition)$, **then**
5 Sort the particles
6 Set all cells of ρ to 0
7 **Foreach** particle
8 Interpolate $\mathbf{E}$ at $\mathbf{x_p}$ — Stored in $\mathbf{E_p}$
9 Update $\mathbf{v_p}$ — $\mathbf{v_p}$ += $\frac{q}{m}\Delta t\, \mathbf{E_p}$
10 Update $\mathbf{x_p}$ — $\mathbf{x_p}$ += $\Delta t\, \mathbf{v_p}$
11 Accumulate charge from $\mathbf{x_p}$ on ρ
12 Compute $\mathbf{E}$ from ρ — Poisson solver

Figure 2.2 – High-level description of the Particle-in-Cell (PIC) method.

```
1 // Six doubles.
2 struct { double position_x, position_y, position_z,
3                 velocity_x, velocity_y, velocity_z; } particle1;
4
5 // Index plus offset.
6 struct { int    i_cell;
7          float  dx, dy, dz;
8          double vx, vy, vz; } particle2;
```

Listing 2.1 – Data structures for 3d particles.

Those numerical particles move in the phase space following the characteristics given in the system (1.7), which we also give here:

$$\begin{cases} \dfrac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \vec{v} \\ \dfrac{\mathrm{d}\vec{v}}{\mathrm{d}t} = \dfrac{q}{m}\vec{E} \end{cases}$$

In a PIC simulation, first the particles and the fields have to be initialized. Then, considering a leap-frog time-stepping — see Section 1.2.2 [5, Section 2.4] — (second-order in time), five operations have to be performed at each time step, as shown in Figure 2.2. First, the electric field is interpolated to the particles (line 8), in order to update the particle velocities (line 9). The particle positions are updated with those new velocities (line 10)[2]. Then, the particle charge is accumulated on the spatial grid (line 11). Finally, the Poisson equation is solved to obtain the grid electric field (line 12).

Before going into the details of those steps, we will first explain the main data structures and the initialization of the particles.

2.2 Data Structures

2.2.1 Particle Data Structure

One central aspect in the design of a PIC implementation is how the particles are stored in the shared memory. A first approach is to represent each 3d particle with 48 bytes (6 `doubles`) to describe its position and velocity, see Listing 2.1 (top).

[2] In the literature, the velocities and positions updates are often considered together in a unique "push" operation.

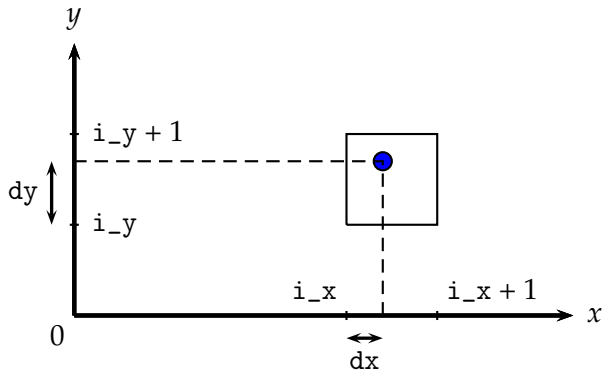


Figure 2.3 – A 2d particle: the physical space is normalized to $[0; ncx) \times [0; ncy)$.

A more efficient approach is the "index plus offset" representation [47, Section III.E.]. Positions are represented as a combination of a cell index and normalized offsets within this cell, see Figure 2.3.

We denote the physical space by $[x_{\min}; x_{\max}) \times [y_{\min}; y_{\max}) \times [z_{\min}; z_{\max})$. We use a grid on this physical space: ncx grid points on the x-axis, ncy grid points on the y-axis and ncz grid points on the z-axis, as introduced in Figure 2.2. We thus have grid spacings defined by $\Delta x = (x_{\max} - x_{\min})/ncx$, $\Delta y = (y_{\max} - y_{\min})/ncy$ and $\Delta z = (z_{\max} - z_{\min})/ncz$. The idea of the "index plus offset" representation is to store positions not on the physical space, but on the normalized space $[0; ncx) \times [0; ncy) \times [0; ncz)$. Thus, a particle positioned at $(x_{\text{physical}}, y_{\text{physical}}, z_{\text{physical}})$ in the physical space is represented on this normalized space at the position (x, y, z), where

$$x = \frac{x_{\text{physical}} - x_{\min}}{\Delta x}, \quad y = \frac{y_{\text{physical}} - y_{\min}}{\Delta y} \quad \text{and} \quad z = \frac{z_{\text{physical}} - z_{\min}}{\Delta z}.$$

Then, we consider the integers

$$i_x = \lfloor x \rfloor, \quad i_y = \lfloor y \rfloor \quad \text{and} \quad i_z = \lfloor z \rfloor,$$

and the normalized offsets (which are reals in $[0; 1)$)

$$\texttt{dx} = x - i_x, \quad \texttt{dy} = y - i_y \quad \text{and} \quad \texttt{dz} = z - i_z.$$

The cell index i_{cell} in $\{0, 1, \ldots, ncx \cdot ncy \cdot ncz - 1\}$ is the image of some one-to-one mapping depending on (i_x, i_y, i_z). Commonly in `C`, the row-major mapping, see formulas (2.2), is used.

$$\begin{aligned} (i_x, i_y, i_z) \mapsto i_{\text{cell}} &= (i_x \cdot ncy + i_y) \cdot ncz + i_z \\ i_{\text{cell}} \mapsto &\begin{cases} i_x = \left\lfloor \frac{i_{\text{cell}}}{ncz \cdot ncy} \right\rfloor \\ i_y = \text{mod}\left(\left\lfloor \frac{i_{\text{cell}}}{ncz} \right\rfloor, ncy \right) \\ i_z = \text{mod}(i_{\text{cell}}, ncz) \end{cases} \end{aligned} \tag{2.2}$$

With this representation, a particle is stored in memory with 40 bytes: one 32-bits `int` (i_{cell}), three `floats` (`dx`, `dy` and `dz`) and three `doubles` (`vx`, `vy` and `vz`), see Listing 2.1 (bottom).

There are two common data structures possible to store N particles with these 7 attributes. One can store the particles using one single array of size N; in this array, each cell contains the 7 attributes (*e.g.*, using the type `particle2`). This is the Array of Structures (AoS) data structure (Listing 2.2, top). One can also store the particles using seven arrays of size N: one array per attribute. This is the Structure of Arrays (SoA) data structure (Listing 2.2, middle).

One can also use more complex data structures. For example, it can be more efficient to keep, at each time step, all the particles that reside in the same cell together. This is known as the strict-binning approach. It can be implemented with the Packed Memory Array (PMA)

```
// Array of Structures (AoS), see Listing 2.1. Array of size N.
struct particle2* particles;

// Structure of Arrays (SoA). Arrays of size N.
int* i_cell;
float* dx; float* dy; float* dz;
double* vx; double* vy; double* vz;

// Strict-binning with SoA. Arrays of size (N * stretch), where stretch ≥ 1
// depends on implementation, see [144, 72].
float* dx; float* dy; float* dz;
double* vx; double* vy; double* vz;
int particle_sets_begin[ncx * ncy * ncz]; // Index of first particle in a cell.
int particle_sets_sizes[ncx * ncy * ncz]; // Number of particles in a cell.
```

Listing 2.2 – Data structures for the array of particles.

data structure (Listing 2.2, bottom [144, 72]): instead of having arrays of size N, arrays are "stretched" to be able to contain all the particles plus some empty cells, for efficiency.

The main drawback of the index plus offset representation is that we are limited in the grid discretization. With a 32-bit integer, the `i_cell` values cannot be greater than $2^{31} = 2\ 147\ 483\ 648$ (or $2^{32} - 1 = 4\ 294\ 967\ 295$ if we choose `unsigned int`[3]), which means that our grid size cannot exceed $\approx 1\ 000 \times 1\ 000 \times 2\ 000$. Of course a possible way to get rid of this restriction is to use 64-bits integers, but then this representation brings only small benefits (or even not at all). The cell index is implicit when using the strict-binning approach, and thus it avoids this problem for the particle data structure, while 64-bits integers can be computed on-the-fly to avoid this problem when accessing grid quantities.

As a remark, we note that we chose here to keep `doubles` for the representation of velocities. It makes sense because at each time step, we update the particle positions with $\vec{x}$ += $\vec{v}\Delta t$. With a high number of time steps, using only `floats` for velocities could accumulate too much error. There is a way to avoid the accumulation of errors known as Kahan's summation formula [155], but using it would require to store another `float` per particle (to account for the compensation in the running sum), and thus it is simpler to just use plain `doubles` for velocities.

2.2.2 E and ρ Data Structure

> "Redundant interpolation coefficients are stored. [...] Fields can be interpolated by accessing the cell's interpolator instead of jumping around 6 different arrays and across y- and z- strides within those arrays."
>
> K. J. Bowers [45]

The standard 3d representation of the electric field E and of the charge density ρ stores their values at the grid points, see Listing 2.3 (top).

In this case, the interpolation of the 3d arrays `Ex`, `Ey` and `Ez` asks for accessing memory locations that are not contiguous. A solution to partially overcome this problem consists in storing components of the field in only one array [54, Section IV, Case 3], see Listing 2.3 (middle).

Unfortunately, this data structure still leads to non-contiguous accesses. This problem is solved by using a redundant one-dimensional array of coefficients [45][4], see Listing 2.3 (bottom).

The redundant array `E_1d` stores, for each cell, the values of each of the three field arrays at the grid points on the 8 corners of the cell, contiguously in memory. `rho_1d` similarly stores,

[3] `unsigned int` is less efficient: https://software.intel.com/en-us/articles/common-vectorization-tips

[4] This data structure is also detailed in later presentations, *e.g.*, [44, Time 27'33"].

```
1  // 3d arrays.
2  double Ex[ncx][ncy][ncz], Ey[ncx][ncy][ncz], Ez[ncx][ncy][ncz];
3  double rho[ncx][ncy][ncz];
4
5  // One 3d array.
6  double Exyz[ncx][ncy][ncz][3];
7
8  // Redundant 1d arrays.
9  double E_1d[ncx*ncy*ncz][24];
10 double rho_1d[ncx*ncy*ncz][8];
```

Listing 2.3 – Data structures for E and ρ.

for each cell, the values of the charge density on the corners of that cell. In our implementation, we use the formulas (2.3).

$$\forall (i_x, i_y, i_z) \in \{0, \dots, ncx-1\} \times \{0, \dots, ncy-1\} \times \{0, \dots, ncz-1\},$$

$$\begin{aligned}
\rho_{1d}[i_{\text{cell}}(i_x,i_y,i_z)][0] &= \rho[\quad i_x \quad][\quad i_y \quad][\quad i_z \quad] \\
\rho_{1d}[i_{\text{cell}}(i_x,i_y,i_z)][1] &= \rho[\quad i_x \quad][\quad i_y \quad][\text{mod}(i_z+1,ncz)] \\
\rho_{1d}[i_{\text{cell}}(i_x,i_y,i_z)][2] &= \rho[\quad i_x \quad][\text{mod}(i_y+1,ncy)][\quad i_z \quad] \\
\rho_{1d}[i_{\text{cell}}(i_x,i_y,i_z)][3] &= \rho[\quad i_x \quad][\text{mod}(i_y+1,ncy)][\text{mod}(i_z+1,ncz)] \\
\rho_{1d}[i_{\text{cell}}(i_x,i_y,i_z)][4] &= \rho[\text{mod}(i_x+1,ncx)][\quad i_y \quad][\quad i_z \quad] \\
\rho_{1d}[i_{\text{cell}}(i_x,i_y,i_z)][5] &= \rho[\text{mod}(i_x+1,ncx)][\quad i_y \quad][\text{mod}(i_z+1,ncz)] \\
\rho_{1d}[i_{\text{cell}}(i_x,i_y,i_z)][6] &= \rho[\text{mod}(i_x+1,ncx)][\text{mod}(i_y+1,ncy)][\quad i_z \quad] \\
\rho_{1d}[i_{\text{cell}}(i_x,i_y,i_z)][7] &= \rho[\text{mod}(i_x+1,ncx)][\text{mod}(i_y+1,ncy)][\text{mod}(i_z+1,ncz)]
\end{aligned} \tag{2.3}$$

They take eight times more memory than the standard layout, but it has been shown that, for the charge density, it is more efficient because it opens the possibility to vectorize the accumulation step (line 11 in Figure 2.2)[87, Section 4.1.2]. Differences for this step between standard and redundant arrays are shown for a 2d implementation in Listing 2.4.

We can note that this redundant data structure for E is a good choice only if there are a lot of particles per cell. Of course, when there are roughly as many particles as grid points, multiplying by 8 the data for E almost doubles the memory transfers. With such a low number of particles, using a redundant data structure for E is detrimental.

In this thesis, we focus on simulations needing a lot of particles per cell. This data structure is hence useful in our case. We will show in Sections 4.3.3 and 4.5.2 how to gain performance through cache hit improvements, both for E and ρ, by using space-filling curves to order the cells of this data structure.

2.3 Particle Initialization

In PIC implementations, it is common to initialize the particle positions and velocities randomly following f_0, each with weight $w = \frac{1}{N} \int f_0 \, \mathrm{d}\vec{x} \, \mathrm{d}\vec{v}$[5]. There are some technical details hidden behind the word "randomly", that we will try to explain in this section.

- How to generate "good" uniform random numbers?
- How to deal with non-uniform numbers?
- Are we obliged to use randomness?

[5]The initial distribution functions considered have the same value $\int f_0 \, \mathrm{d}\vec{x} \, \mathrm{d}\vec{v} = (x_{\max} - x_{\min}) \cdot (y_{\max} - y_{\min}) \cdot (z_{\max} - z_{\min})$.

```
double rho[ncx][ncy]; // Standard 2d.
[...]
rho[i_x    ][i_y    ] += w * (1. - dx[i]) * (1. - dy[i]);
rho[i_x    ][i_y + 1] += w * (1. - dx[i]) * (     dy[i]);
rho[i_x + 1][i_y    ] += w * (     dx[i]) * (1. - dy[i]);
rho[i_x + 1][i_y + 1] += w * (     dx[i]) * (     dy[i]);

// VEC_ALIGN is architecture dependent, e.g., 32 with 256-bits vectors (AVX2).
// _Alignas(VEC_ALIGN) (c11) can be safely replaced with __attribute__((aligned(
    VEC_ALIGN))) (gcc 2.95.3).
#define NB_CORNERS_2D 4
_Alignas(VEC_ALIGN) double rho_1d[ncx * ncy][NB_CORNERS_2D]; // Redundant.
    Remark: padding is needed for vector sizes wider than 256-bits; use instead
    rho_1d[ncx * ncy][max(NB_CORNERS_2D, VEC_ALIGN / sizeof(double))].
_Alignas(VEC_ALIGN) float coeffs_x[NB_CORNERS_2D] = {  1.,  1.,  0., 0.};
_Alignas(VEC_ALIGN) float  signs_x[NB_CORNERS_2D] = { -1., -1.,  1., 1.};
_Alignas(VEC_ALIGN) float coeffs_y[NB_CORNERS_2D] = {  1.,  0.,  1., 0.};
_Alignas(VEC_ALIGN) float  signs_y[NB_CORNERS_2D] = { -1.,  1., -1., 1.};
[...]
#pragma omp simd aligned(coeffs_x, coeffs_y, signs_x, signs_y:VEC_ALIGN)
for (corner = 0; corner < NB_CORNERS_2D; corner++)
    rho_1d[i_cell[i]][corner] += w *
        (coeffs_x[corner] + signs_x[corner] * dx[i]) *
        (coeffs_y[corner] + signs_y[corner] * dy[i]);
```

Listing 2.4 – Accumulation in 2d: Standard VS Redundant (not thread-safe).

2.3.1 "Good" Random Uniform Numbers

> "An argument based on spatial statistics [32, p. 26] suggests that we need $N \gg 200\,n^2$ if there are n iid points in the problem."
>
> B. D. Ripley [117]

It is really important to pick a "good" pseudo-random number generator (PRNG) in Monte-Carlo simulations. Here are some examples of random number generators used in PIC implementations:

- The textbook ES1 implementation [5, Section 3.6] uses a *quiet start* (cf. Section 2.3.3), or a pseudo-random initialization with `Fortran`'s native `ranf` function together with scrambling; a more recent implementation [6] uses the "Minimal standard" generator [116].
- The VM_non_unif implementation [3] uses a "KISS" generator [113, Combination Generators][111, 118], and also a quiet start (cf. Section 2.3.3).
- The PIConGPU implementation [49] uses the CURAND library[6] from NVIDIA with the XORWOW generator [112], and also a quiet start (cf. Section 2.3.3).
- The SeLaLib [51] and PICSAR [87] implementations use `Fortran`'s native `random_number` (the GNU version[7] is the Xorshift1024* generator [112]).
- The GTC-P implementation [83] uses the "MRG32k3a" generator [109, 110].
- The VPIC [46] and SMILEI [56] implementations use the "Mersenne Twister" generator [114].

[6] https://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CURAND_Library.pdf

[7] https://gcc.gnu.org/onlinedocs/gfortran/RANDOM_005fNUMBER.html

```
1 double random_double(void) {
2     return ((double)random_integer()) / ((double)MAX_RAND_INT_VALUE + 1.);
3 }
```

Listing 2.5 – First attempt to convert an integer to a `double`: not all possible values are generated.

We should have the period p of our generator be much greater than $200\,n^2$ [117], where n is the number of random numbers generated. For a PIC simulation involving *e.g.*, 1 trillion particles, where each particle needs at least 6 but usually more random numbers, it means that we first need $p \gg 200 \cdot 10^{24}$. This is why we can already discard:

- the Intel version of `Fortran`'s native `random_number`[8] which uses two separate congruential generators together [108, 8] ($p \approx 10^{18}$)
- the Microsoft 15-bits version of `C`'s native `rand` generator ($p \approx 10^4$) and 32-bits versions of it ($p \approx 10^9$)
- `C`'s native `random` generator ($p \approx 10^{10}$)
- POSIX's native generator `drand48` ($p \approx 10^{13}$)

We can note that choosing a good pseudo-random number generator is easiest in `C++` than in `C` or `Fortran`, since good ones are included in the standard library `<random>` from the `C++11` version[9].

As a side note, we have to be careful if we want to produce `doubles` from a pseudo-random number generator that gives only integer values. One obvious-but-wrong way to do it is shown in Listing 2.5. This code does generate `doubles` in $[0;1)$, but if the generator outputs integers with less than 53 bits, it is not enough to generate all possible `double` values. In [106], a conversion function is shown with `MAX_RAND_INT_VALUE` $= 2^{32} - 1$. Listing 2.6 shows how to enable it for other `MAX_RAND_INT_VALUE` values. It assumes that `MAX_RAND_INT_VALUE` $\geq 2^{15} - 1$, but can be extended in a similar fashion if this is not the case (although this is unlikely to happen).

A detailed discussion of the desired properties required by a pseudo-random number generator can be found in [106]. Following the pieces of advice given in this article, we wrote the seed generator shown in Listing 2.7.

We now show how using different pseudo-random number generators affect a PIC simulation:

- `C`'s native generator `rand`, although it is well known that it should be avoided [107].
- `C`'s native generator `drand48`.
- The "KISS" [113, Combination Generators][111, 118] (Keep It Simple Stupid). We took the code from [106] ($p \approx 2^{127}$) which is a slight modification of the one given in the second citation.
- The "Mersenne Twister" [114] with $p = 2^{19\,937} - 1$. We linked our code with the GNU Scientific Library[10], but we could also have taken the code on its authors' webpage[11].
- The "WELL" [115] (Well Equidistributed Long-period Linear) with different periods available from $2^{512} - 1$ to $2^{44\,497} - 1$. We took the code from its authors' webpage[12].

[8] https://software.intel.com/en-us/node/693738
[9] http://en.cppreference.com/w/cpp/numeric/random
[10] https://www.gnu.org/software/gsl/manual/html_node/Random-Number-Generation.html
[11] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
[12] http://www.iro.umontreal.ca/~panneton/WELLRNG.html

```c
#define MASK_53_BITS 9007199254740991
#define TWO_POWER_53 9007199254740992.0
#define TWO_POWER_27        134217728.0
#define TWO_POWER_18           262144.0
/*
 * Combines different random_integer() calls to provide a double number with
 * all 53 random bits.
 *
 * N.B.: For performance issues, one could move the ifs at the exterior of the
 * function and use preprocessor #ifs instead.
 * N.B.: int64_t (stdint.h) can be safely replaced by unsigned long long (c99).
 */
double random_double(void) {
    int64_t max_plus_1 = (int64_t)MAX_RAND_INT_VALUE + 1;
    if (MAX_RAND_INT_VALUE < TWO_POWER_18)
        return (double)((
                 (int64_t)random_integer() +
                ((int64_t)random_integer() * max_plus_1) +
                ((int64_t)random_integer() * max_plus_1 * max_plus_1) +
                ((int64_t)random_integer() * max_plus_1 * max_plus_1 *
                   max_plus_1)
            ) & MASK_53_BITS) / TWO_POWER_53;
    else if (MAX_RAND_INT_VALUE < TWO_POWER_27)
        return (double)((
                 (int64_t)random_integer() +
                ((int64_t)random_integer() * max_plus_1) +
                ((int64_t)random_integer() * max_plus_1 * max_plus_1)
            ) & MASK_53_BITS) / TWO_POWER_53;
    else if (MAX_RAND_INT_VALUE < TWO_POWER_53)
        return (double)((
                 (int64_t)random_integer() +
                ((int64_t)random_integer() * max_plus_1)
            ) & MASK_53_BITS) / TWO_POWER_53;
    else
        return (double)((
                 (int64_t)random_integer()
            ) & MASK_53_BITS) / TWO_POWER_53;
}
```

Listing 2.6 – Second attempt to convert an integer to a `double`: all 53 random bits.

```c
/*
 * Hash function [10, Section 11] from http://burtleburtle.net/bob/hash/doobs.html
 * N.B.: int32_t (stdint.h) can be safely replaced by unsigned long (c99).
 */
inline int32_t mix(int32_t a, int32_t b, int32_t c) {
    a=a-b;  a=a-c;  a=a^(c >> 13);
    b=b-c;  b=b-a;  b=b^(a <<  8);
    c=c-a;  c=c-b;  c=c^(b >> 13);
    a=a-b;  a=a-c;  a=a^(c >> 12);
    b=b-c;  b=b-a;  b=b^(a << 16);
    c=c-a;  c=c-b;  c=c^(b >>  5);
    a=a-b;  a=a-c;  a=a^(c >>  3);
    b=b-c;  b=b-a;  b=b^(a << 10);
    c=c-a;  c=c-b;  c=c^(b >> 15);
    return c;
}

/*
 * Returns a 64-bit seed from time, process ID and host ID.
 * Also uses mpi_rank for the probable case where different MPI processes
 * are launched on the same machine.
 * One could alternatively use bits from /dev/(u)random
 * N.B.: int32_t (stdint.h) can be safely replaced by unsigned long (c99).
 * N.B.: int64_t (stdint.h) can be safely replaced by unsigned long long (c99).
 */
inline int64_t seed_64bits(int mpi_rank) {
    struct timeval tv;
    gettimeofday(&tv, (void*)0);
    int32_t time_value1 = (int32_t)tv.tv_sec;
    int32_t time_value2 = (int32_t)tv.tv_usec;
    int32_t process_id  = (int32_t)getpid();
    int32_t host_id     = (int32_t)gethostid() + (int32_t)mpi_rank;
    int64_t  low_32bits = (int64_t)mix(time_value1, process_id, host_id);
    int64_t high_32bits = (int64_t)mix(time_value2, process_id, host_id);
    return low_32bits | (high_32bits << 32);
}
```

Listing 2.7 – 64-bits seed generator.

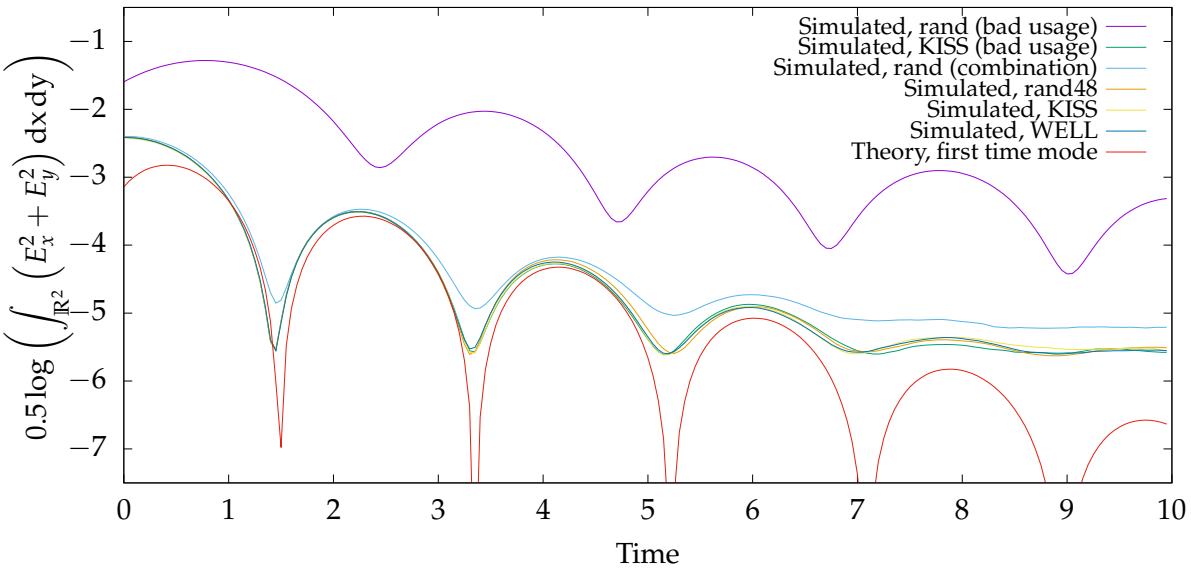


Figure 2.4 – Landau damping simulation with different generators. 1 billion particles, grid size: 256×256, Δt: 0.05, perturbation: 0.01.

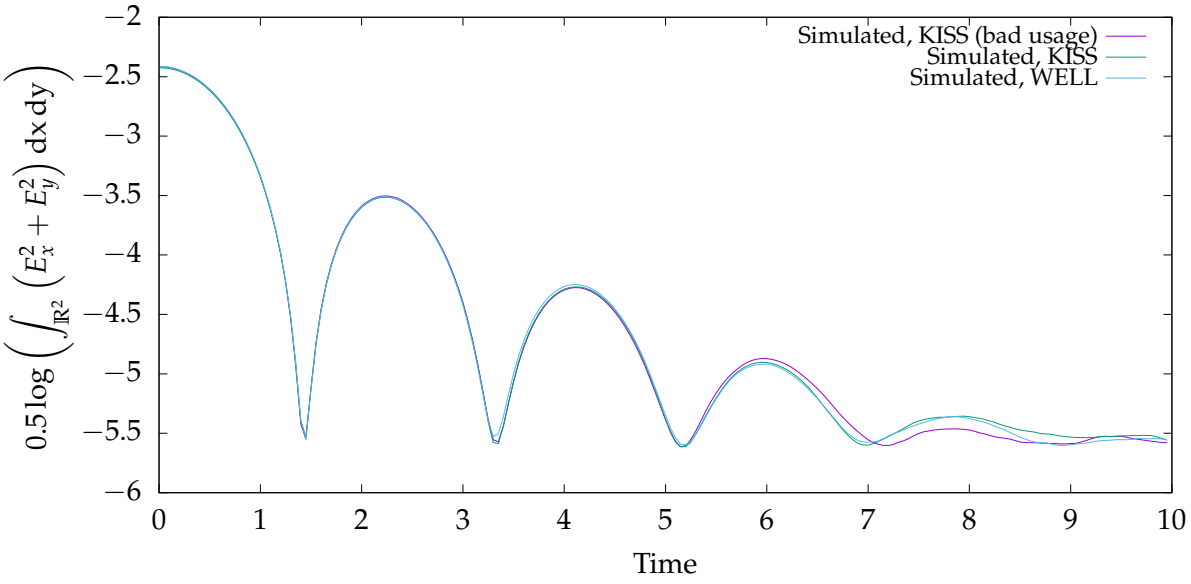


Figure 2.5 – Landau damping simulation with different generators (zoom from Figure 2.4). 1 billion particles, grid size: 256×256, Δt: 0.05, perturbation: 0.01.

Figure 2.4 shows a diagnostic for a simulation: we plot $0.5 \cdot \log(\int (E_x^2 + E_y^2)\,\mathrm{d}x\,\mathrm{d}y)$, closely related to the electric energy $\frac{1}{2}\int (E_x^2 + E_y^2)\,\mathrm{d}x\,\mathrm{d}y$). This computation is not the main goal of our simulations, but is here used to verify our implementation: on this particular test case, we have an almost-exact theoretical solution thanks to a dispersion analysis [35, Chapter 4], see details in Section 7.1. The fact that none of the simulated curves agree with the theoretical one at the beginning is normal: at the beginning, we need more than just the first time mode to explain the behavior. The fact that they also differ from this curve at the end is also expected: this is numerical noise caused by a "low" number of particles. This figure illustrates two things. First, one should not use `rand()`. On the target architecture, `rand()` outputs 32 bits integers with $p \approx 10^9$, and you can see that using Listing 2.5 yields incorrect results from the start. Even if combining two `rand()` calls (see Listing 2.6) yields "better" results, it is still far from what is attained by other generators. Second, when using a generator that outputs integers to produce `doubles`, we should pay attention to combine different calls to the generator if the integers have less than 53 bits. When using `rand()`, we had two problems: the low period and this one. With the KISS generator, there is no period problem, but this one remains. A zoom on this figure, see Figure 2.5, clearly demonstrates the superior effect of combining two KISS calls.

Suppose now that the only random number generator used in our implementation is the one given in Listing 2.5. We output diagnostics, see the top curve "rand (bad usage)", and we might think that there is a problem in the initialization. However, when looking at the distribution function with such a high number of grid points, it is hard to detect anything (see Figure 2.6, top left). To be able to see something, let us look at the distribution function on a coarser 32×32 grid (see Figure 2.6, top right). Now we clearly see that the initial distribution function has not been correctly sampled. Of course, the function looks correctly sampled when using *e.g.*, the WELL generator, see Figure 2.6 (bottom right)... but it also looks correctly sampled when using a combination of two `rand()` calls, see Figure 2.6 (bottom left), even though we have seen in Figure 2.4 that this method does not give satisfactory results. So, looking with our eyes at the initial repartition of the particles is not enough to judge the quality of a random number generator. It would be interesting to know if it is possible to use a tool to detect anomalies.

Figure 2.7 shows that the bad behavior of `rand()` is even worse when using much more particles, as expected. It also shows that, even though it is regarded as a bad generator in [106],

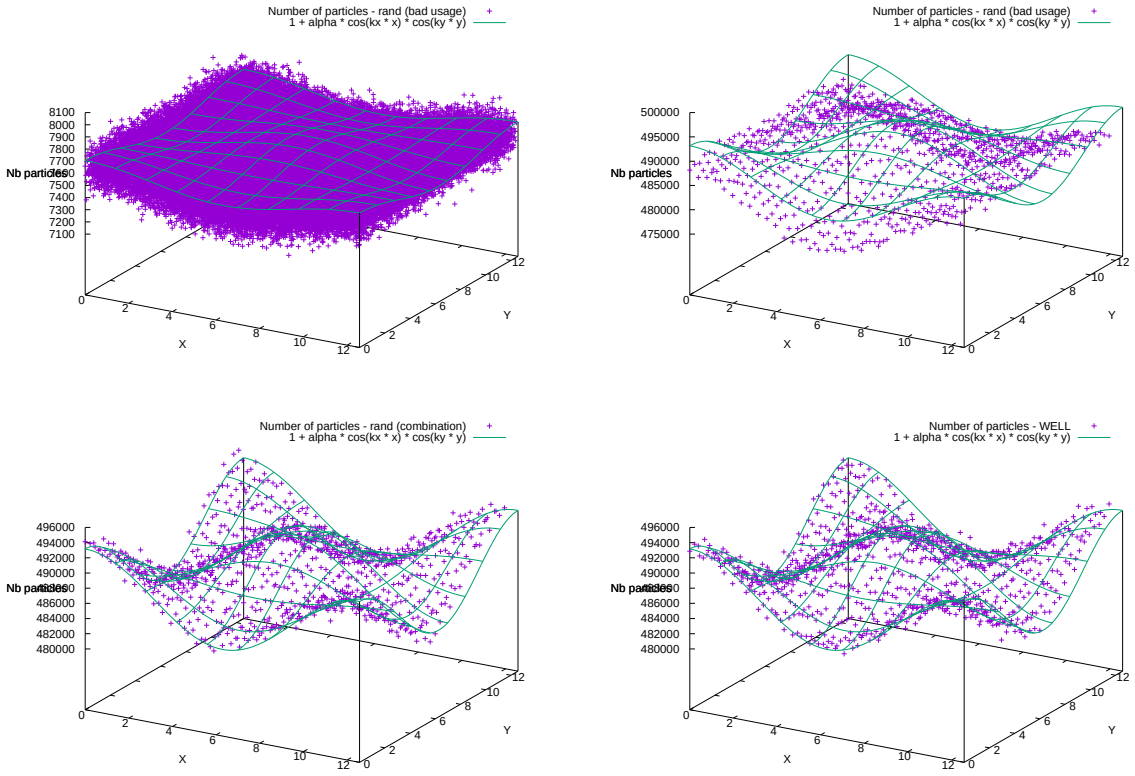


Figure 2.6 – Sampling 500 million particles following $(1 + 0.01 \cdot \cos(x/2) \cdot \cos(y/2))$.
Top left: 32-bits `rand()`, 256×256 grid. Top right: 32-bits `rand()`, 32×32 grid.
Bottom left: two 32-bits `rand()` combined, 32×32 grid. Bottom right: WELL, 32×32 grid.

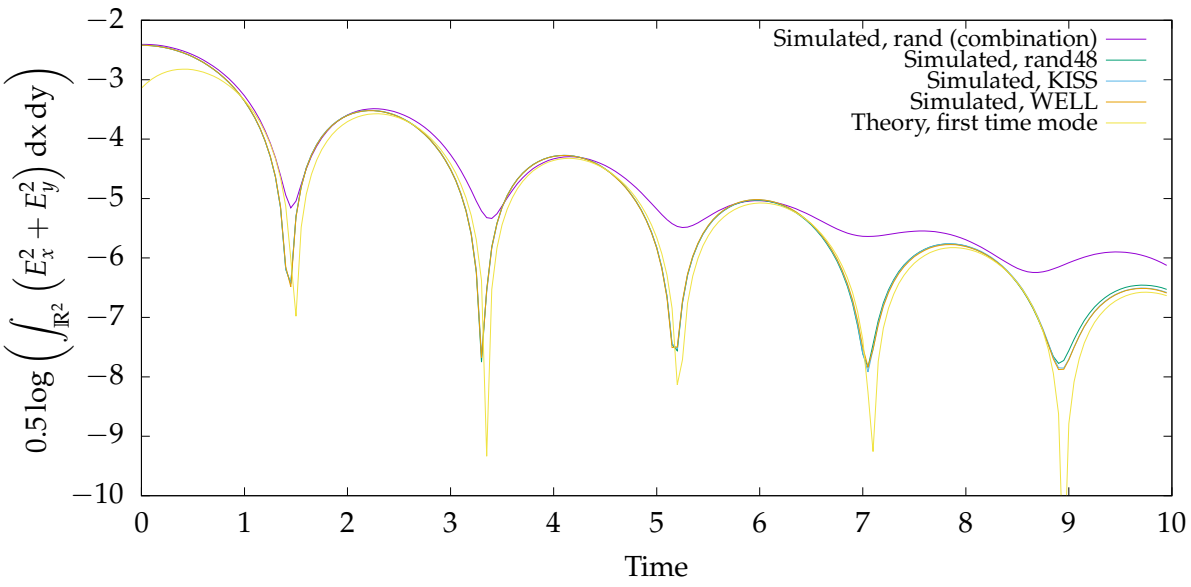


Figure 2.7 – Landau damping simulation with different generators. 100 billion particles, grid size: 256×256, Δt: 0.05, perturbation: 0.01.

Credits — left strip: Randall Munroe, https://xkcd.com/221/; right strip: Scott Adams and the Andrews McMeel Syndication, http://dilbert.com/strip/2001-10-25.

Figure 2.8 – xkcd's RNG (left) and Dilbert's one (right).

`drand48()` leads to results close to the other generators on this particular test case.

There are of course a lot of other random number generators, see *e.g.*, [16, Section 8.3]. Among them, Dilbert's and xkcd's random number generators seemed promising, see Figure 2.8, but they turned out to be unsatisfactory.

2.3.2 Non-Uniform Random Numbers

> “ Occasionally, a research paper will contend that the quality of the random numbers generated by some particular method, such as Box–Muller or the ratio-of-uniforms, is bad, but the quality ultimately depends only on the quality of the underlying uniform generator.
>
> J. E. Gentle [16, Section 4] ”

The uniform law has been studied in the previous subsection. We will here suppose that we have a generator for this law, and use it to generate other ones. Different methods for achieving this aim are described in [16, Section 4]. We will here only sketch the methods that were used in `Pic-Vert`.

If we are in a 1d setting, one option is to reverse the cumulative density function. See for example the Applied Statistics Algorithm 241 [120] if we need the normal distribution. It is

```
> f:=(x,y)->x**2 * exp(-(x**2 + y**2) / 2) / (2 * Pi);
> solve({diff(f(x,y),x)=0, diff(f(x,y),y)=0}, {x, y});
    {x = 0, y = y}, {x = RootOf(_Z**2 - 2), y = 0}
> evalf(f(0, y));
                          0.
> evalf(f(sqrt(2), 0));
                      0.1170996630
> evalf(f(-sqrt(2), 0));
                      0.1170996630
```

```
            |\^/|
          ._|\|   |/|_.
           \  MAPLE  /
           <____ ____>
                |
```

Figure 2.9 – Maximum search with Maple.

```
1 void polynomial_times_maxwellian_2d(double* vx, double* vy) {
2     double control_point, evaluated_point;
3     do {
4         *vx = 12. * pic_vert_next_random_double() - 6.0; // Uniform in [-6 ; 6)
5         *vy = 12. * pic_vert_next_random_double() - 6.0; // Uniform in [-6 ; 6)
6         control_point = 0.1171 * pic_vert_next_random_double();
7         evaluated_point = sqr(*vx) * exp(-(sqr(*vx) + sqr(*vy)) / 2.) / (2. * PI
              );
8     } while (control_point > evaluated_point);
9 }
```

Listing 2.8 – Acceptance / rejection algorithm for $f(x,y) = x^2 \cdot \frac{1}{2\pi} \exp\left(-\frac{x^2+y^2}{2}\right)$.

not always possible to give a formula for the reverse cumulative density function. However, it is always possible to numerically reverse it, *e.g.*, with a binary search [10, Exercises 2.3-5]. In any dimension, if we have to generate the normal distribution, we can use the Box–Muller algorithm instead [104].

In the general case we can always use the acceptance / reject method [16, Section 4.5]. This is the most used method in our implementation. This method requires that we can provide an upper bound of the function. Whenever we had a new function to generate with this method, we thus had to search for global maximums, which is quite easy with a formal calculus tool. First, we have to search for critical points, then look if the maximum of f lies on a critical point or on the boundaries of its definition. Figure 2.9 shows how to use Maple [188] to find the maximum of the 2d function $f(x,y) = x^2 \cdot \frac{1}{2\pi} \exp\left(-\frac{x^2+y^2}{2}\right)$ [207, Equation 5]. Here we find the value ≈ 0.1170996630, which is a maximum value: we can look at the curve of f to be sure, see Figure 2.10. The acceptance / rejection method for this function is then given in Listing 2.8. As a side note, let us remark that this algorithm is not the most efficient to generate this function. If we look at Figure 2.10, we feel that taking 0.1171 as a delimiter for f on the whole domain leads to a poor acceptance rate: around $1/16.8624 \approx 5.9\%$ because the integral of f over $[-6;6] \times [-6;6]$ is around 1 and the volume of the parallelepiped $[-6;6] \times [-6;6] \times [0;0.1171]$ is 16.8624. This solution has thus the advantage to be easy, but could be fastened by using a better majoring function than the constant function 0.1171, see details in [16, Section 4.5].

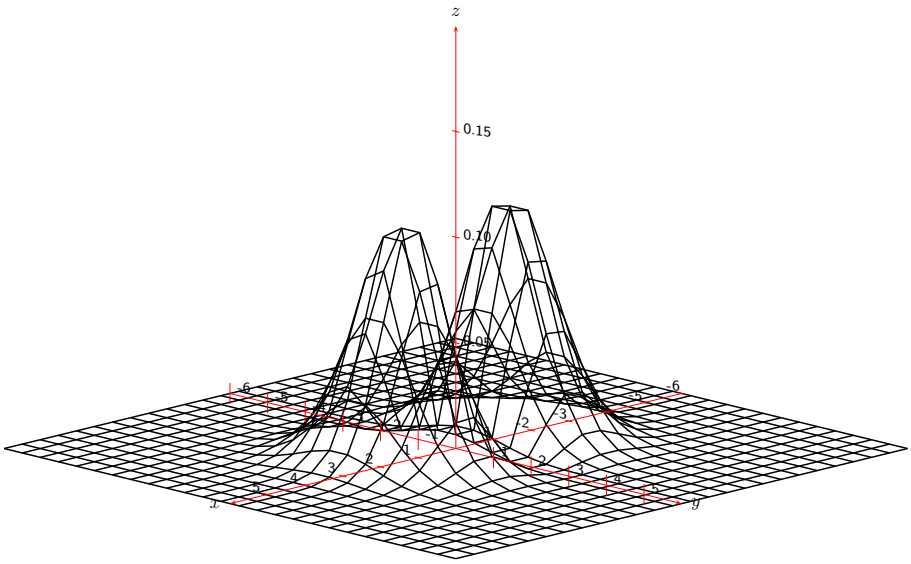


Figure 2.10 – $f(x,y) = x^2 \cdot \frac{1}{2\,\pi} \exp\left(-\frac{x^2+y^2}{2}\right)$ plotted on $[-6;6] \times [-6;6]$.

2.3.3 Non-Random Initialization: "Quiet Start"

> Nonrandom initializations [...] give quiet starts with low initial noise levels. [...] The noise level grows and, as in other quiet-start methods, the simulation remains quiet for a finite time only.
>
> J. Denavit & J. M. Walsh [55]

To generate particles, it is also possible to use quasi-random number generators. The main idea is the following: when we want to simulate what happens with a really precise initial condition (*e.g.*, "activate" only a given set of Fourier modes), we cannot do it precisely with a "truly" random initialization, unless we put a lot of particles [55]. The randomness will lead to the activation of unwanted Fourier modes. Hence the need to have a "noiseless" particle initialization [50]. Instead of using pseudo-random numbers, the idea is thus to use low discrepancy sequences: sequences of numbers that are deterministic, and as uniform as possible. They are called quasi-random numbers, even if they are not at all random. In some applications, it gets rid of the traditional stochastic convergence of Monte-Carlo methods in $\frac{1}{\sqrt{N}}$, and replaces it with a better convergence in $\frac{1}{N}$, although for a PIC method it applies only for the initial distribution function. Examples of such sequences are the Hammersley sequence [20, Section 3.3], the Sobol' sequence [119], the van der Corput sequence [105]...

If using those sequences with "simple" initializations, i.e. that require one random number per position and one per velocity (*e.g.*, uniform, the polar Box–Muller algorithm, the reverse cumulative density function algorithm... but not the acceptance / reject algorithm.), it is straightforward to parallelize the use of these sequences. With P processes, the i-th process just accesses the $\{i \cdot N/P, \ldots, (i+1) \cdot N/P - 1\}$ indices. However, when using those sequences in parallel together with an acceptance / rejection algorithm, one cannot know in advance the number of indices needed to generate the N particles. Thus, special care has to be taken, in order to avoid indices not accessed by any process or indices accessed by multiple processes.

2.4 Field Solve

The field solver is in charge of converting ρ to E with respect to the Poisson equation. The complexity of this part is then related to $\mathsf{nbCells} = ncx \cdot ncy \cdot ncz$.

In our implementation, the electric field is computed by solving the Poisson equation on a uniform Cartesian grid, by a Fourier method — using FFTW [149] together with `OpenMP` [182]. This algorithm performs $O(\mathsf{nbCells} \cdot \log \mathsf{nbCells})$ operations and has a huge asset: the first "F" in FFT means "Fast". For example, on a benchmark with a $64 \times 64 \times 64$ grid and 1 billion particles, the Poisson solve takes only 0.20% of the total execution time (or 3.5 ms per solve with 24 cores). In 2d with a 512×512 grid, we obtain roughly the same numbers. This gives a good insight of how powerful this method is.

> [The particle processing routines] times are negligible in comparison to the Poisson solve time.
>
> R. K. Narayanan & K. Madduri [74, Section 3.1]

Other applications need bigger grid sizes or less particles, and the percentage of the Poisson solve time will grow. In [74], the test case uses a few thousands particles on a $256 \times 256 \times 256$ grid. In our test cases, we have different boundary conditions, but with this grid size, the Poisson solve takes only 0.25 s per solve with 24 cores, and it is still low in comparison to the particle processing routines. In this article, a timing of 4.4 s per solve is reported for 8 cores, with a different algorithm: a multi-grid solver, which performs $O(\mathsf{nbCells})$ operations. For comparison, those timings correspond to 5.9 s for 1 core with FFT and 35 s for 1 core with their multi-grid solver. Of course, the number of iterations of the multi-grid solver explains the higher execution time, and this number of iterations depends on the boundary conditions. Still, in this paper they consider FFT as future research.

Compared to FFT, the multi-grid solver has a better asymptotic complexity, might be better suited for parallelism using domain decomposition, and might be easier to modify when changing the boundary conditions. It is thus interesting to keep in mind that, even though in our applications the FFT was a good choice, in other scenarios another solver might be more efficient.

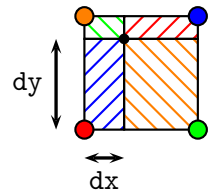
> We have primarily adopted the charge-conserving current deposition algorithms because they allow the field solve to be done locally, i.e., there is no need for a Poisson solve.
>
> R. A. Fonseca, L. O. Silva, F. S. Tsung, V. K. Decyk, W. Lu, C. Ren, W. B. Mori, S. Deng, S. Lee, T. Katsouleas, & J. C. Adam [58, Section 4]

As stated in the introduction, the Poisson equation is a simplification of the Maxwell equations (1.4). Solving the Maxwell equations (a) needs other operations on the particles to deposit the current in addition to deposit the charge, and (b) needs to account for the self-consistent magnetic field (which is neglected when using the Poisson equation instead), but (c) allows to solve the fields locally, hence allows to efficiently use domain decomposition. In this thesis, another approach is taken to parallelize the computations: replicate the grid on each `MPI` process, and distribute evenly the particles across them. This strategy is efficient in 2d or in 3d with small grid sizes, but becomes too costly with bigger grid sizes due to `MPI` communication of the full grid and due to the numerous cache misses if an `MPI` process may have particles anywhere on the domain.

2.5 Interaction Between Particles and Fields: Interpolation

There are two points in the algorithm where the particles interact with the grid quantities: when computing the electric field at the particles' positions, and when computing the charge density from the particles (lines 8 and 11 in Figure 2.2). Different schemes exist that are of arbitrary order. The most common ones are [23, Section 5-3]:

$$v+ = \frac{q}{m} E \Delta t$$




```
for (i = 0; i < N; i++) {
    vx[i] += q / m * delta_t * (
                (   dx[i])*(   dy[i])*E[i_cell[i]].x_ne
              + (1.-dx[i])*(   dy[i])*E[i_cell[i]].x_nw
              + (   dx[i])*(1.-dy[i])*E[i_cell[i]].x_se
              + (1.-dx[i])*(1.-dy[i])*E[i_cell[i]].x_sw);
    vy[i] += q / m * delta_t * (
                (   dx[i])*(   dy[i])*E[i_cell[i]].y_ne
              + (1.-dx[i])*(   dy[i])*E[i_cell[i]].y_nw
              + (   dx[i])*(1.-dy[i])*E[i_cell[i]].y_se
              + (1.-dx[i])*(1.-dy[i])*E[i_cell[i]].y_sw);
}
```

Figure 2.11 – First order interpolation in 2d with the redundant data structure.

- the nearest grid point (NGP), or first-order B-spline (degree 0 in space).
- the cloud-in-cell (CIC) model, or second-order B-spline, or linear splines (degree 1 in space).
- the triangular-shaped-cloud (TSC), or third-order B-spline, or quadratic splines (degree 2 in space).
- the quadratic-spline interpolation scheme (PQS), or fourth-order B-spline, or cubic splines (degree 3 in space).

Schemes of degree up to 5 in space are presented in [80, Table 4]. Increasing the degree of interpolation also increases the execution time. This increase is studied for example in 1d in [80, Figure 19] and in 3d in [57, Table 3].

If the degree of a scheme is k, then it uses $(k+1)^d$ points, where d is the number of dimensions of the physical space. For example, the cloud-in-cell model [42] that we will apply throughout this thesis uses 4 points in 2d, and 8 points in 3d.

The code for the accumulation step in 2d using the cloud-in-cell model was shown in Listing 2.4. Using the same model, Figure 2.11 shows the code for the interpolation step in 2d and Listing 2.9 shows the code for the accumulation step in 3d. In a number of physical applications, this model is not considered as sufficiently precise. Thus, a model of degree 2 or 3 is often used in practice.

2.6 Process-level Parallelism

Let us now focus on different ways to parallelize a PIC implementation across machines from a supercomputer (distributed memory, with `MPI` [184]).

When the grid over which the simulation takes place is very large, the cost of maintaining a copy of the entire grid on every machine is prohibitive. In such situations, one resorts to *domain decomposition*, thereby assigning the available machines to subdomains of the grid space. A first challenge in domain decomposition is to balance the load: find subdomains that bear approximatively the same number of particles. Possible solutions involve:

- space-filling curves [2], *e.g.*, PSC [59],
- Barnes–Hut trees (also called octrees) or their generalization spacetrees, *e.g.*, [134, 181, 89],
- rectilinear partitioning [167] (i.e., using parallelepipeds), *e.g.*, PICADOR [82],
- having processes with less particles help the ones with more particles, *e.g.*, EMSES [73].

A second challenge is associated with the significant amount of communication involved for redistributing the particles that move across the subdomain boundaries. A typical plasma

```
1  // VEC_ALIGN is architecture dependent, e.g., 32 with 256-bits vectors (AVX2).
2  // _Alignas(VEC_ALIGN) (c11) can be safely replaced with __attribute__((aligned(
       VEC_ALIGN))) (gcc 2.95.3).
3  #define NB_CORNERS_3D 8
4  _Alignas(VEC_ALIGN) double rho_1d[ncx * ncy * ncz][NB_CORNERS_3D]; // Redundant.
       Remark: padding is needed for vector sizes wider than 512-bits; use instead
       rho_1d[ncx * ncy * ncz][max(NB_CORNERS_3D, VEC_ALIGN / sizeof(double))].
5  _Alignas(VEC_ALIGN) float coeffs_x[NB_CORNERS_3D] =
6      {  1.,  1.,  1.,  1.,  0.,  0.,  0.,  0.};
7  _Alignas(VEC_ALIGN) float  signs_x[NB_CORNERS_3D] =
8      { -1., -1., -1., -1.,  1.,  1.,  1.,  1.};
9  _Alignas(VEC_ALIGN) float coeffs_y[NB_CORNERS_3D] =
10     {  1.,  1.,  0.,  0.,  1.,  1.,  0.,  0.};
11 _Alignas(VEC_ALIGN) float  signs_y[NB_CORNERS_3D] =
12     { -1., -1.,  1.,  1., -1., -1.,  1.,  1.};
13 _Alignas(VEC_ALIGN) float coeffs_z[NB_CORNERS_3D] =
14     {  1.,  0.,  1.,  0.,  1.,  0.,  1.,  0.};
15 _Alignas(VEC_ALIGN) float  signs_z[NB_CORNERS_3D] =
16     { -1.,  1., -1.,  1., -1.,  1., -1.,  1.};
17 [...]
18 #pragma omp simd aligned(coeffs_x, coeffs_y, coeffs_z, signs_x, signs_y, signs_z
       :VEC_ALIGN)
19 for (corner = 0; corner < NB_CORNERS_3D; corner++)
20     rho_1d[i_cell[i]][corner] += w *
21         (coeffs_x[corner] + signs_x[corner] * dx[i]) *
22         (coeffs_y[corner] + signs_y[corner] * dy[i]) *
23         (coeffs_z[corner] + signs_z[corner] * dz[i]);
```

Listing 2.9 – Accumulation in 3d: Redundant (not thread-safe).

simulation may involve a significant fraction of fast-moving particles that frequently cross subdomains boundaries, thus requiring heavy cross-machine communication. A possible solution is to overlap as much communication with computation as possible, *e.g.*, [48, p. 2835].

When the grid is not too large, *particle decomposition* may be used: particles are distributed evenly to the machines, each of which replicates the description of the electric field. The machines synchronize at every time step, by communicating the contribution of their particles to the charge density, in order to update the electric field. In that approach, the communication time is small if the shared domain itself is small, even though there is a logarithmic reduction step for synchronization.

A successful hybrid approach, known as *domain cloning* [64, 79], consists of using domain decomposition in order to create subdomains that are just small enough for particle decomposition to apply.

In this work, we handle the process-level parallelism with particle decomposition. The main advantages of this method are:

- its simplicity: everything is automatically work-balanced, because every `MPI` process keeps all its particles during the whole simulation;
- the only communication is via `MPI_ALLREDUCE` for the reduction of the charge array: no particle has to move from one process to another during the simulation;
- the scaling is automatically independent of the particle distribution and thus of the particle dynamics: the performance of the parallelism should be problem independent.

The bottleneck of this approach is that the scalability is highly limited by the global reduction step. Thus, two parameters should not be very large, otherwise they could severely slow down the simulation: the number of cells and the number of processes.

2.7 Some Particle-in-Cell Implementations

There are approximatively as many PIC implementations as there are research teams that use the PIC method. It would be practically unfeasible to list them all. Here are just a few implementations that we encountered during this thesis.

- EMSES (ElectroMagnetic Spacecraft Environment Simulator). Implementation introduced in 2009 ([71]; recent work: [72]
- ES1 (ElectroStatic 1 dimensional). Implementation introduced in 1985 ([5]); recent work: [6]; code repository: https://www.crcpress.com/downloads/IP586/DISKFILES.zip
- EUTERPE[13]. Implementation introduced in 1998 ([63]); recent work: [33]; website: http://montblanc-test.bsc.es/applications/euterpe
- GTC (Gyrokinetic Toroidal Code) / GTC-P (Princeton Gyrokinetic Toroidal Code). Implementation introduced in 1998 ([69]); recent work: [83]; website: http://www.nersc.gov/research-and-development/apex/apex-benchmarks/gtc-p/
- KEMPO (Kyoto university's ElectroMagnetic Particle cOde). Implementation introduced in 1985 ([27]); recent work: [30]; code repository: https://www.terrapub.co.jp/e-library/cspp/text/09.txt (Fortran), https://web.archive.org/web/20141022230251/http://www.rish.kyoto-u.ac.jp/isss7/KEMPO/[14] (matlab)
- ORB5[15]. Implementation introduced in 1998 ([84]); recent work: [62]; website: https://spc.epfl.ch/ORB5
- OSIRIS (Object-oriented SImulation Rapid Implementation System). Implementation introduced in 2000 ([22]); recent work: [57]; website: https://picksc.idre.ucla.edu/software/software-production-codes/osiris/
- PICSAR (Particle-In-Cell Scalable Application Resource). Implementation introduced in 2016 ([87]); website: https://picsar.net/
- PICADOR[16]. Implementation introduced in 2011 ([40]); recent work: [81]; website: http://hpc-education.unn.ru/en/research/overview/laser-plasma
- PIConGPU (Particle-in-Cell on Graphics Processing Units). Implementation introduced in 2010 ([48]); recent work: [90]; website: http://picongpu.hzdr.de/
- PSC (Plasma Simulation Code). Implementation introduced in 2006 ([7, Chapter 2]); recent work: [59]; website: http://www.plasma-simulation-code.net/
- QuickPIC (QUasi-statIC Particle-in-Cell). Implementation introduced in 2006 ([60]); recent work: [37]; website: https://picksc.idre.ucla.edu/software/software-production-codes/quickpic/
- SHARP (A Spatially Higher-order, Relativistic Particle-in-cell code). Implementation introduced in 2017 ([80])

[13] Not an acronym. In Greek mythology, Euterpe was one of the Muses: https://en.wikipedia.org/wiki/Euterpe.

[14] Archived from the original http://www.rish.kyoto-u.ac.jp/isss7/KEMPO/.

[15] Not an acronym.

[16] Not an acronym. A picador is a horseman in Spanish bullfights: https://en.wikipedia.org/wiki/Picador. It is also a ketchup brand: http://en.unitedfoodgroup.ru/brands/ketchup-picador/.

- SMILEI (Simulating Matter Irradiated by Light at Extreme Intensities). Implementation introduced in 2016 ([66]); recent work: [56]; website: http://www.maisondelasimulation.fr/projects/Smilei/html/index.html
- TRISTAN (TRIdimensional STANford code) / par-T (PARallel Tristan) / Apar-T[17]. Implementation introduced in 1993 ([9]); recent work: [29]; code repository: https://www.terrapub.co.jp/e-library/cspp/text/10.txt
- VORPAL[18]. Implementation introduced in 2001 ([75]); website: https://www.txcorp.com/vsim
- VPIC (Vector Particle-in-Cell). Implementation introduced in 2003 ([45]); recent work: [46]; code repository: https://github.com/lanl/vpic
- XOOPIC (X11-based Object-Oriented Particle-in-Cell). Implementation introduced in 1995 ([86]); recent work: [38]; website: https://ptsg.egr.msu.edu/

[17] Not an acronym. Aparté is a French word.

[18] Not an acronym. Vorpal is a nonsense word from the 1872 poem "Jabberwocky" by Lewis Carroll [194, Chapter 1].

Chapter 3

Contributions

The main contribution of our thesis is a software written in `C`, called `Pic-Vert`: it is an implementation of the Particle-in-Cell (PIC) method for plasma physics. Three important steps are required for such an implementation:

- (computer science) it should be efficient. PIC implementations are usually memory bound, as a recent paper [83] points out: "*metrics such as flop/s or percentage-of-peak are less relevant for the predominantly memory-bound gyrokinetic PIC methods, as modern architectures require 10 flops per byte moved from DRAM in order to be compute-limited.*". It is possible to verify this assertion in the roofline model [180]. Memory bandwidth usage should hence be a good metric to ensure efficiency of one's implementation.
- (mathematics) it should be verified. Some test cases have almost-exact theoretical solutions. One should verify that the implementation outputs the desired solution. For other test cases, some other diagnostics may be performed to assess correctness of the results, like the conservation of total energy.
- (physics) it should be validated. The physical model used in a given implementation always have some limitations. One should validate the model chosen with real-life simulations.

For performance, we provide an implementation that (a) achieves close-to-minimal number of memory transfers with the main memory, (b) exploits SIMD instructions for numerical computations and (c) exhibits a high degree of `OpenMP`-level parallelism. Throughout this manuscript, we show the memory bandwidth usage of our implementation, in the roofline model [180]. We also implemented distributed-memory parallelism through particle decomposition, even though for future work, domain decomposition is a better approach. This last layer of parallelism is thus not the core of our work. To put our work in perspective, we also designed a new metric to compare the efficiency of PIC implementations when using different multi-core architectures. `Pic-Vert` is compared to other recent implementations in Section 3.3.

For verification, we simulate classical 2d2v and 3d3v Landau-damping test cases [4, 23]. We also simulate a new test case that was designed by our co-authors [207]. It is possible to test the correctness of these simulations by comparing the simulated electric energy to its theoretical value obtained from the dispersion analysis.

For validation, we simulate a 2d3v electron hole test case [165]. We are able to reproduce the results of this paper, which are validated by satellite measurements of phase-space holes in various parts of the magnetosphere.

3.1 Organization of the Chapters Describing Our Contributions

Chapters 4–8 describe the main contributions of our thesis.

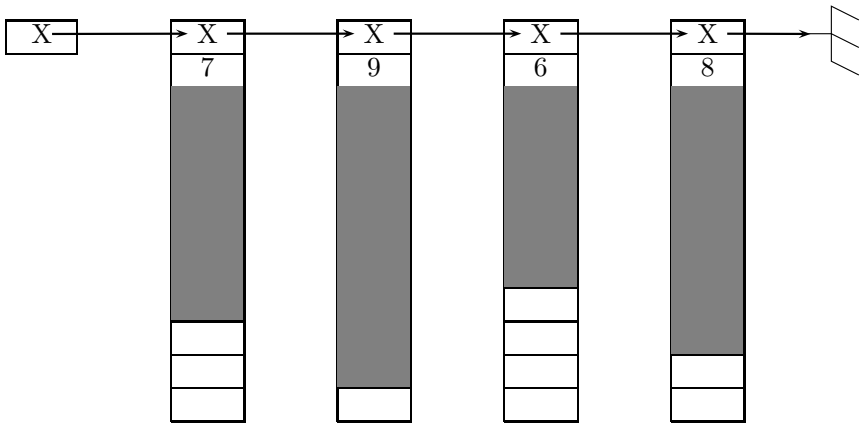


Figure 3.1 – Chunk bag data structure: chunks of size 10, particles stored in grey cells.

Chapters 4 and 5 first present the main features of `Pic-Vert`. Both chapters are written with the intent of providing the intuition behind the optimizations performed. Our goal would be accomplished if, when reading these chapters, the ideas seem easy and natural. The implementation itself is also described, and this sometimes leads to technical details which need more time to be understood. All in all, we tried to follow this piece of advice:

> “ Everything should be made as simple as possible, but not simpler.
>
> A. Einstein ”

Chapter 4 focuses on the optimization of a relatively standard algorithm, where we use AoS or SoA for the data layout of the particle array, with a periodic sorting of this array. Some optimizations presented in this chapter are quite classical — even though not always found in the PIC community — and some other ones are, to the best of our knowledge, entirely new. In our 2d tests, the optimizations presented in this chapter lead to the best execution times, see details in Chapter 8.

Chapter 5 then presents the happy wedding between chunked sequences — linked lists of fixed-capacity arrays, see Figure 3.1 — and the PIC method. In our 3d tests, the algorithms presented in this chapter lead to the best execution times, see details in Chapter 8.

Chapter 6 describes an implementation of the semi-Lagrangian (SL) method in 2d, using domain decomposition. This chapter is completely independent from the others, except from the initial Chapter 1. The data structures for this method are entirely different from the PIC ones, and the fact that we use domain decomposition leads to a truly different `MPI` behavior. An algorithm which gets rid of state-of-the-art limitations for this method is presented, and future directions for this implementation are also described. The source files for this implementation are not part of `Pic-Vert`.

Chapter 7 describes in detail the verification and validation of our implementation. It also details how to use both the PIC and the SL method together in a same implementation.

Chapter 8 concludes this thesis and shows some possible future directions.

3.2 Publications

This thesis is based on the following publications:

[i] Y. Barsamian, S. A. Hirstoaga, and É. Violard. “Efficient Data Structures for a Hybrid Parallel and Vectorized Particle-in-Cell Code”. In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, 2017, pp. 1168–1177.
DOI: 10.1109/IPDPSW.2017.74
Slides: http://www.barsamian.am/Slides/slides_2017-06-02.pdf.

[ii] Y. Barsamian, A. Charguéraud, and A. Ketterlin. "A Space and Bandwidth Efficient Multicore Algorithm for the Particle-in-Cell Method". In: *Parallel Processing and Applied Mathematics: 12th International Conference (PPAM)*. vol. 10777. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 133–144.
DOI: 10.1007/978-3-319-78024-5_13
Slides: http://www.barsamian.am/Slides/slides_2017-09-11.pdf.

[iii] Y. Barsamian, S. A. Hirstoaga, and É. Violard. "Efficient Data Layouts for a Three-Dimensional Electrostatic Particle-in-Cell Code". In: *Journal of Computational Science* 27 (2018), pp. 345–356.
DOI: 10.1016/j.jocs.2018.06.004.

[iv] Y. Barsamian, J. Bernier, S. A. Hirstoaga, and M. Mehrenberger. "Verification of $2D \times 2D$ and two-species Vlasov–Poisson solvers". In: *ESAIM: Proceedings and Surveys* 63 (2018), pp. 78–108.
DOI: 10.1051/proc/201863078
Slides: http://www.barsamian.am/Slides/slides_2016-08-25.pdf.

[v] Y. Barsamian, A. Charguéraud, S. A. Hirstoaga, and M. Mehrenberger. "Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors". In: *24th International Conference on Parallel and Distributed Computing (Euro-Par)*. Vol. 11014. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 749–763.
DOI: 10.1007/978-3-319-78024-5_33
Slides: http://www.barsamian.am/Slides/slides_2018-08-30.pdf.

This last publication comes with the following artifacts[1] ("Best Artifact Award"):

[vi] Y. Barsamian, A. Charguéraud, S. A. Hirstoaga, and M. Mehrenberger. *Software artifacts for Euro-Par 2018 paper: "Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors"*. Figshare. 2018.
URL: https://doi.org/10.6084/m9.figshare.6391796.

Chapter 6 is also based on a work presented but not published:

[vii] Y. Barsamian and M. Mehrenberger. "Semi-Lagrangian Simulations for Solving 2d2v Vlasov–Poisson Systems (one and two species)". In: *Platform for Advanced Scientific Computing (PASC), Minisymposium "Kinetic Simulations on HPC Platforms for Plasma Physics Applications (3/3): Parallelization and New Hardware"*. 2017.
Slides: http://www.barsamian.am/Slides/slides_2017-06-27.pdf.

We also wrote a technical report which is not contained in the present manuscript:

[viii] Y. Barsamian. "Maximum Subarray Problem in 1D and 2D via Weighted Paths in Directed Acyclic Graphs". Tech. rep. Université de Strasbourg, 2016.
URL: https://hal.archives-ouvertes.fr/hal-01585324.

3.3 Comparison Between `Pic-Vert` and Other Implementations

Tables 3.1–3.3 give performances that can be found (or deduced) in recent papers. We already emphasized that comparing efficiency of multiple implementations is a very difficult task, see the end of Section 1.3.3. In Section 2.5, we also saw some articles which explain in detail that the interpolation scheme changes the performance a lot. In those tables, almost all the implementations use linear interpolation (except PIConGPU which uses second order interpolation),

[1] Please forgive us for a mistake in the simulation files: one should read `!=` instead of `==` in the line `if (strcmp(parameters.sim_distrib_name, STRING_NOT_SET) == 0)`, or else the simulation will always load the default initial distribution, and not the one asked in the parameter file.

and all the implementations use leap-frog time-stepping. It is thus possible to compare the performance of those implementations. In papers, performance results are given in absolute figures, but architectures used vary a lot. Since PIC implementations are memory bound, we designed a new metric in order to compare the relative performances of PIC implementations.

Our metric to compare PIC implementations
In those tables, the normalized column gives the number of particles processed per second divided by the theoretical bandwidth of the architecture. We believe that this is a figure that makes sense to be compared (higher is better). This metric is not perfect: it is not hard to understand that the number of cores has also some impact on the performance. But we think that it is a relatively fair and really simple way of making comparisons.

One last thing that we must acknowledge is that our implementation solves the Vlasov–Poisson system (1.5), and most of those other implementations solve the Vlasov–Maxwell system (1.4). Exact comparison is thus not possible, but the difference between those two systems is not a lot. Roughly speaking, to solve the Vlasov–Maxwell system, one needs two additional arrays (magnetic field, current), and additional computations. In a 3d simulation, we perform 144 floating-point operations per particle (a new optimization makes us avoid 6 operations per particle, see Section 4.3.2) — 209 operations when normalized to single precision —, while it is reported that at least 246 operations are needed for a cold plasma with the Vlasov–Maxwell system [47]. The additional computations do not harm performance, since the implementation is memory bound — see the performance of our 2d3v implementation in Table 3.2 where those additional computations are performed. The additional accesses to the magnetic field should not be a problem since we make only one access per cell per iteration thanks to the strict-binning approach, see Chapter 5. The only thing that really matters is the deposit of the current, which has to be done on multiple cells when a particle changes cell. This could harm performance when we have fast particles, but in practice in most test cases and implementations found in the literature, particles do not move a lot per iteration.

In summary, those tables give what we view as a fair comparison between PIC implementations, when looking at the normalized column (we recall that higher is better). Most recent implementations do not show results in 2d nor 2d3v, so the most important table is Table 3.3. In this table we see that our implementation outperforms other ones on CPU. To prove that our metric is relevant and has not been forged to show a fake superiority of our implementation, we provide in this table results on an architecture close to the one used by other recent papers: on Intel Haswell (Intel Xeon E5-269X), we obtain a x3 speedup over other implementations which also use Intel Haswell with the same memory bandwidth — even though the other implementations had access to more cores.

2d Implementation	Architecture parameters	Nb. part./s (in millions)	Arithmetic (in GFlop-s/s)	Memory bandwidth	Normalized (in million part./GB)
UPIC 2014 [53]	Intel Xeon-X5650: 12 cores, 64 GB/s	383	23.0	15.3 GB/s	6.0
SeLaLib 2016 [51]	Intel Xeon E5-2670: 8 cores, 51.2 GB/s	132	-	9.5 GB/s	2.6
PSC 2016 [59]	AMD Opteron 6274: 16 cores, 51.2 GB/s	125	-	-	2.4
Pic-Vert 2017 [204]	Intel Xeon E5-2680: 8 cores, 51.2 GB/s	266	16.5	26.8 GB/s	5.2
Pic-Vert 2018 [202]	Intel Xeon E5-2697 v4: 18 cores, 76.8 GB/s	861	53.4	41.3 GB/s	11.2
UPIC 2014 [53]	NVIDIA Fermi M2090: 512 cores, 142 GB/s	1144	79	45.8 GB/s	8.1

Table 3.1 – Performances of some 2d PIC implementations (top: CPU; bottom: GPU).

2d3v Implementation	Architecture parameters	Nb. part./s (in millions)	Arithmetic (in GFlop-s/s)	Memory bandwidth	Normalized (in million part./GB)
OSIRIS 2011 [65]	Intel Core 2 Duo E7200: 1 core, 5.67 GB/s	4.9	-	-	0.9
Pic-Vert 2018 [205]	Intel Xeon Platinum 8160: 24 cores, 127.99 GB/s	910	104	59.1 GB/s	7.1
OSIRIS 2011 [65]	EVGA GeForce GTX 280: 240 cores, 141.7 GB/s	397	-	31%	2.8

Table 3.2 – Performances of some 2d3v PIC implementations (top: CPU; bottom: GPU).

3d Implementation	Architecture parameters	Nb. part./s (in millions)	Arithmetic (in GFlop-s/s)	Memory bandwidth	Normalized (in million part./GB)
VPIC 2008 [47]	IBM PowerXCell 8i 9 cores, 204.8 GB/s	173	43	49%	0.845
OSIRIS 2013 [57]	Intel Xeon E5-2680 8 cores, 51.2 GB/s	134	-	-	2.62
ORB5 2016 [61]	Intel Xeon E5-2670 8 cores, 51.2 GB/s	69.1	-	-	1.35
PICADOR 2016 [81]	Intel Xeon E5-2697 v3 14 cores, 68 GB/s	127	42.5	-	1.87
GTC-P 2016 [83]	Intel Xeon E5 2692 v2 12 cores, 59.7 GB/s	100	-	-	1.68
PIConGPU 2016 [90]	Intel Xeon E5-2698 v3 16 cores, 68 GB/s	111	58.9	-	1.63
Pic-Vert 2018 [205]	Intel Xeon Platinum 8160 24 cores, 127.99 GB/s	740	155	53.6 GB/s	5.78
Pic-Vert Chapter 8	Intel Xeon E5-2690 v3 12 cores, 68 GB/s	374	78	31.3 GB/s	5.49
PIConGPU 2013 [49]	NVIDIA Tesla K20X 2 688 cores, 250.0 GB/s	0.25[2]	498[2]	-	0.001
PIConGPU 2016 [90]	NVIDIA Tesla GK210 2 496 cores, 480 GB/s	336	196	-	0.7
ORB5 2016 [61]	NVIDIA Tesla K20X 2 688 cores, 250.0 GB/s	177	-	-	0.708
PICADOR 2016 [81]	Intel Xeon Phi 7250 (KNL) 68 cores, 115.2 GB/s	298	100	-	2.59
EMSES 2017 [72]	Intel Xeon Phi 7250 (KNL) 68 cores, 115.2 GB/s	1300	-	-	11.3

Table 3.3 – Performances of some 3d PIC implementations (top: CPU; bottom: GPU, MIC).

3.4 Implementation

`Pic-Vert` is available at `http://www.barsamian.am/Pic-Vert/`.

3.4.1 Dependencies

`Pic-Vert` has several additional softwares to install:

- a `C` compiler that supports `OpenMP` [182]. The compiler must at least support `OpenMP 3.0`, but it is better if it supports `OpenMP 4.0` (`#pragma omp simd`; without this support, those pragmas are desactivated in our implementation, and the vectorization would then be done with the automatic vectorization from the compiler which usually gives less satisfactory results). For example, `icc` supports `OpenMP 4.0` since version 15.0 and `gcc` since version 4.9.1.
- an `MPI` [184] wrapper. All the scripts are given for a standard installation of Open-MPI [151], but it is possible to modify them to use another wrapper if needed.

[2]The low number of particles processed per second is due to expensive far-field radiation computations ($\approx$ 95% of the execution time). This also explains the high number of operations per second.

- the FFTW3 library [149]. It is used to solve the Poisson equation.
- the HDF5 library [185]. This dependency is only needed to output HDF5 files, *e.g.*, for 2d visualizations of the particle density and of the charge density in Chapter 7. If needed, it is possible to output binary and/or ascii files instead of HDF5 files to get rid of this dependency, but this would decrease performance.

Here is how to install those dependencies under Ubuntu.
For the `C` compiler: it is possible to get the Intel compiler at https://software.intel.com/en-us/qualify-for-free-software/student; to get the GNU compiler, type:

```
sudo apt-get install gcc
```

For openmpi and hdf5 type:

```
sudo apt-get install libopenmpi-dev openmpi-bin libhdf5-openmpi-dev
```

For the FFTW library, type:

```
sudo apt-get install libfftw3-dev
```

3.4.2 Source Files

The `Pic-Vert` repository is composed of several folders:

- `include` and `src` contain all the modular source files for `Pic-Vert` (as `.h` and `.c` files):
  - `alignment_crash.h`: function to detect misalignment of data also on Intel i386 family of processors[3].
  - `compiler_test.h`: compiler test to know which features can be used.
  - `diagnostics.c/h`: electric energy diagnostics, see Chapter 7.
  - `fields.c/h`: redundant data structure for E, see Section 2.2.2.
  - `hdf5_io.c/h`: handling of HDF5 outputs.
  - `initial_distributions.c/h`: possible initial distributions of particles.
  - `math_functions.h`: useful mathematical functions not already in standard libraries.
  - `matrix_functions.c/h`: dynamic allocation of arrays.
  - `meshes.c/h`: data structure for the grid.
  - `output.c/h`: handling of energy and efficiency outputs.
  - `papi_handlers.c/h`: handling of PAPI [190] performance counters[4].
  - `parameter_reader.c/h`: reading parameter files for simulations.
  - `parameters.h`: useful parameters (architecture, mathematics, default parameters for simulations...)
  - `particle_type_XoX_XdXv.c/h`: AoS or SoA data structure for particles, see Chapter 4.
  - `particle_type_(concurrent_)chunkbags_of_XoX_XdXv.c/h`: chunk bag data structure for particles, see Chapter 5.
  - `poisson_solvers.c/h`: data structure for the FFT Poisson solver, see Section 2.4.
  - `random.c/h`: pseudo-random number generators, see Section 2.3.

[3] See http://orchistro.tistory.com/206.
[4] They can be activated by adding `-DPAPI_LIB_INSTALLED` in the compilation line, provided that the PAPI library is installed first.

  - `rho.c/h`: redundant data structure for ρ, see Section 2.2.2.
  - `space_filling_curves.c/h`: space-filling curves for E and ρ, see Chapter 4.
  - `variadic.h`: macros to write functions with default arguments in `C`[5].

- `simulations` contains the PIC simulations that we used throughout this thesis, and that uses the files in the previous item.
- `parameter_files` contains examples of parameter files for our simulations.
- `scripts_local` contains the compile and run scripts for our simulations, on typical workstations. It notably contains a file `architectural_configuration.sh` that needs to be updated with respect to the architectural configuration used.

Apart from the test cases we detail in this manuscript, other test cases are available. They are documented in the files `initial_distributions.c/h`. It is possible to add some initial distributions if needed. All the parameters can be modified in the parameter files.

[5]This somehow mimics the default argument feature, e.g. in `C++` or `Fortran`, see https://gustedt.wordpress.com/2010/06/03/default-arguments-for-c99/ and https://stackoverflow.com/questions/1472138/c-default-arguments#33786937.

Chapter 4

Pic-Vert in 2d or 3d with Periodic Sorting

During this thesis, one of our main contributions was the optimization of a Particle-in-Cell (PIC) implementation. In this chapter, we will explain our first optimization steps. Parts of what is explained in this chapter appeared in two articles [204, 203]. Some of the optimizations techniques described in this chapter are quite classical, but to the best of our knowledge the following ones are unique in the literature:

New optimizations

- We applied a transformation that we call "loop-not-so-invariant code motion": this transformation was inspired by the loop hoisting (or loop-invariant code motion) [34, Section 2.3.5.3]. This technique can be applied to other codes as long as some hypotheses are respected.
- We used space-filling curves to reduce cache misses in 2d. Although this technique is known in a number of regular applications, it has not been successfully used in a PIC implementation before our work, even though some authors acknowledged that it could be useful [44, Time 57'38"].
- We designed a new space-filling curve in 3d, to apply the previous technique to 3d.

The baseline of our implementation is a previous work in 2d from several colleagues [51]. This baseline, written in `Fortran` inside the library SeLaLib [183], is described in Section 4.1.

We first extracted the useful parts from this library and ported them to `C`. Once the code was ported, we started to optimize it. Section 4.2 describes the architectures on which the optimizations were tested.

Section 4.3 describes the optimizations that were useful in 2d on a single core. This section will describe in detail the methodology adopted and the optimization techniques used.

Section 4.4 explains some parallelization issues, both with `OpenMP` and `MPI`.

Finally, Section 4.5 gives new insights for the optimization methodology on multi-core, explains a new optimization and how to port the previous optimizations in the context of 3d simulations.

4.1 Baseline SeLaLib Implementation

First of all, we would like to express our gratitude to all the contributors of the SeLaLib library, that provided an initial 2d PIC implementation from which we could work. This implementation, detailed in [51], features:

- "Cell index plus offset" particle representation. See Section 2.2.1.

	icps-gc-6	Curie	Marconi A3
Processor	Intel Xeon E5-2650 v3 (Haswell)	Intel Xeon E5-2680 (SandyBridge)	Intel Xeon Platinum 8160 (Skylake)
RAM	16 GB	64 GB	96 GB
# memory channels	2	4	6
Memory bandwidth	34 GB/s	51.2 GB/s	127.99 GB/s
# cores	10	8	24
Clock frequency	2.3 GHz	2.7 GHz	2.1 GHz
Floating-point	736 GFlops/s	345.6 GFlops/s	1 612 GFlops/s

Table 4.1 – Architectural parameters of one socket of our test machines.

- Array of Structures for particle data structure. See Section 2.2.1.
- Redundant arrays for E and ρ. See Section 2.2.2.
- In-place particle sorting. See [43].
- Shared memory parallelism (`OpenMP`): parallel particle loop.
- Distributed memory parallelism (`MPI`): particle decomposition. See Section 2.6.
- Single Instruction Multiple Data (SIMD) parallelism (vectorization): not exploited.
- Throughput: 132 million particles per second on an 8-core Intel Sandy Bridge. See Table 3.1.

When porting a `Fortran` code to `C`, one can think of `f2c` [145]. This was not an option, since the `C` code had to be readable and since SeLaLib is written in modern `Fortran`. To be able to extract the needed parts from the library, an Integrated Development Environment (IDE) was more than helpful. IDEs for `Fortran` are less easy to find than for `C`. We used Code::Blocks IDE for Fortran [189].

4.2 Test Architectures

The results presented in this chapter come from simulations run on different computers.

Our Inria team machine "icps-gc-6". This machine features 2 sockets, and each of those sockets is an Intel Xeon E5-2650 v3 @2.3 GHz (Haswell) with 16 GB of RAM, 2 memory channels, and 10 cores. Its theoretical memory bandwidth peak is 34 GB/s (only 2 memory channels installed on a maximum of 4[1]), its theoretical single precision floating-point operation peak is 736 GFlops/s. On this machine, we had access to `gcc 6.2` and `icc 17.0.0`[2].

The GENCI supercomputer "Curie"[3] (5 040 nodes). Each node features 2 sockets, and each of those sockets is an Intel Xeon E5-2680 @2.7 GHz (SandyBridge) with 64 GB of RAM, 4 memory channels, and 8 cores). Its theoretical memory bandwidth peak is 51.2 GB/s, its theoretical single precision floating-point operation peak is 345.6 GFlops/s. On this machine, we had access to `gcc 6.1` and `icc 16.0.3.210`.

The A3 partition of the CINECA supercomputer "Marconi"[4] (2 304 nodes). Each node features 2 sockets, and each of those sockets is an Intel Xeon Platinum 8160 @ 2.1 GHz (Skylake) with 96 GB of RAM, 6 memory channels, and 24 cores. Its theoretical memory bandwidth peak is 127.99 GB/s, its theoretical single precision floating-point operation peak is 1 612 GFlops/s. On this machine, we had access to `icc 17.0.4`.

Table 4.1 summarizes the architectural parameters of those machines. Because our PIC implementation is memory-bound, the parameter that matters most is the memory bandwidth.

[1] http://ark.intel.com/products/81705

[2] Thanks to https://software.intel.com/en-us/qualify-for-free-software/student

[3] http://www-hpc.cea.fr/fr/complexe/tgcc-curie.htm

[4] https://www.cineca.it/en/content/marconi

Optimization	Number of codes
Loop Fission, see Section 4.3.2	2 (1 loop or 3 loops)
Loop-not-so-invariant code motion, see Section 4.3.2	2 (with or without)
Data structure for E and ρ, see Section 4.3.3	5 (2d or redundant with 4 different space-filling curves)
Vectorized update-positions code, see Section 4.3.4	2 (before or after)
Data structure for the particles, see Section 4.3.4	2 (AoS or SoA)

Table 4.2 – The different optimizations presented in this section.

4.3 Single Core Optimizations in 2d

4.3.1 Methodology

When a code transformation is performed, it can increase or decrease the efficiency. When another one is performed, the same applies. But what happens if we perform multiple optimizations? Even though two transformations each improve efficiency, it is not clear what happens to efficiency if we perform them together. Because different compilers perform different sets of automatic transformations, it is not obvious neither what happens if we change the compiler. Last but not least, it is far less obvious what happens if we change the target architecture.

In this section, we will fully investigate the first question: what happens when we perform multiple code transformations? When applying transformations to the code, we thus made sure to keep all versions of the code: we ended up with a lot of different versions. For the particle data structure, fully different codes where needed (this choice has implications everywhere in the code). We enabled the other optimizations with compilation flags or with command line options. In total, we had $2 \times 2 \times 2 \times 2 \times 5 = 80$ different codes automatically tested for this section, inside two standalone `C` files of around 3 100 lines each. Table 4.2 summarizes those optimizations, that will be presented in this section.

> “ [...] quis custodiet ipsos custodes?[5]
>
> Juvenal [197, Satire VI, Lines 347–348] ”

In high-performance programming, the programmer is the optimizer who improves the efficiency of his or her implementation. But who will optimize the optimizers? The answer might well be meta-programming. We indeed remark that some — but not all — of these transformations could have been generated automatically through meta-programming, *e.g.*, with the framework BOAST [177].

Another parameter has to be taken into account: the number of time steps between two successive sortings of the particle array (the step in lines 4–5 in Figure 2.2). This parameter can be tuned manually, or automatically according to a mathematical model [70]. For one of the space-filling curves, an additional parameter had to be tuned, and in our case it was manually tuned. The best value for this parameter depends on hardware, and it can also be automatically found via self-tuning, following what is done in the libraries FFTW [149], ATLAS [179], and many others.

Once we have the 80 different codes to test and the corresponding scripts to run them, the question is: how do we perform the tests? In this section, all the simulations were run on a single core of icps-gc-6 (see Table 4.1). We believe that, before starting to run simulations in parallel, the sequential version should be fully optimized.

It should be noted that when performing optimizations, we paid attention to achieve similar efficiency when using both Intel and GNU compilers. This means that each of those 80 different

[5] “[...] who will ward the warders?” Juvenal (translation by G. G. Ramsay)

Physical test case	Linear Landau damping [5, Section 5.15], initial distribution $f(x,y,v_x,v_y,t=0) = \left(1+0.01\cos\left(\frac{x}{2}\right)\cos\left(\frac{y}{2}\right)\right)\frac{1}{2\pi}\exp\left(-\frac{v_x^2+v_y^2}{2}\right)$
Spatial grid	$[0;4\pi)^2$ decomposed in 128^2 cells, periodic boundaries
Particle shape factor	Cloud-in-cell model [42]
Number of particles	50 million
Number of iterations	100 (sorting every 20 iterations)
Time step	0.1
Particle crossing: averaged, per iteration	58% of the particles move 1 cell away, 25% move 2 cells away, 3.4% move 3 cells away, 0.18% move further away

Table 4.3 – 2d test case for sequential optimizations.

codes were compiled both with `gcc` and `icc`. To avoid statistical noise, we ran each of our 160 compiled codes 10 times, and compared the average timings on those 10 runs[6].

As mentioned in Section 1.3.3, observing a phenomenon might change it. Everyone who has tried to instrument a code has probably some funny stories to tell. One of those stories is about a `C++` code whose performance degrades when the name of a class becomes too long... another one is about a code whose timings were reduced after adding a `noop` in the loop [153]. We did not encounter such extreme behaviors, but we did encounter one strange behavior: when adding some code in the particle loop to instrument it (as the code was entirely in one function, it was difficult to time it otherwise), there was at some point a dramatic increase in the timings, just because of the instrumentation code. We managed to reduce the number of operations to perform for this instrumentation, and at some point the dramatic increase of the timings stopped. We speculate that those extra computations were overflowing the L1 instruction cache, which would explain the dramatic increase in the timings (because we could observe this behavior no matter with which particular function call the timings were taken). What can be learned from this experience is that it is good to have an external way of checking the timings (in our case, `perf`) to check that the additional instrumentation does not change the performance.

All the runs were performed with a standard linear Landau damping test case in 2d, presented in Table 4.3. Theoretical results which allow to verify the implementation are available [5, 23]. We believe that this test case is general enough to provide an accurate view of how our implementation would behave on other physical test cases. It is anyway impossible to run all possible physical test cases.

4.3.2 Loop Fission and Loop-not-so-invariant Code Motion

This thesis was performed within two Inria teams. One of them, the CAMUS team, is mostly working on the optimization of codes that contain loops. Therefore, at the very beginning of this thesis, the first book encountered was [36]. This book describes in detail many loop optimization techniques. It was extremely useful to apply some of those techniques to our PIC implementation.

Loop Fission

The first optimization we implemented is the loop fission [36, Section 9.3]. More precisely, the loop "**Foreach** particle" in Figure 2.2 is broken into three parts: one loop to interpolate E and to update velocities, one to update positions, and one loop to accumulate the charge. There are two main reasons to use three loops instead of one: (a) we can efficiently vectorize the

[6] In our tests, the execution times do not vary much, and 10 runs are enough to have a precise average.

```
1  Foreach particle in particles
2      Interpolate E to particle
3      Update the velocity
4      Update the position
5      Accumulate particle charge to ρ
```

Figure 4.1 – PIC pseudo-code with one loop.

```
1  Foreach particle in particles   Update-velocities loop
2      Interpolate E to particle
3      Update the velocity
4  Foreach particle in particles   Update-positions loop
5      Update the position
6  Foreach particle in particles   Accumulate loop
7      Accumulate particle charge to ρ
```

Figure 4.2 – PIC pseudo-code with loop fission.

update-positions as a stand-alone loop and (b) a separate processing of the arrays of E and ρ in different loops leads to a better overall memory management.

This transformation transforms the pseudo-code in Figure 4.1 to the one in Figure 4.2, and speeds up our implementation 18% to 25% depending on the data structure (even without considering vectorization). Starting from now, we call "update-velocities loop" the loop which contains both the interpolation and the velocity update.

Loop-not-so-invariant Code Motion

We also improve runtime performance by removing as much computations as possible from the particle loops. When carefully inspecting the code, we applied an optimization that could be used in other codes, as long as some hypotheses are verified.

The idea is the following: when we have a computation that needs a constant inside a loop, there is a high probability that an optimization can be applied. We give here two classical examples: loop hoisting (or loop-invariant code motion) [34, Section 2.3.5.3] and strength reduction [36, Section 11.1.7]. When comparing Listing 4.1 and Listing 4.2, we understand that the two codes are equivalent. In the second one, we gain $N-1$ multiplications. When comparing Listing 4.3 and Listing 4.4, we also understand that the two codes are equivalent. It is however less clear why this is an optimization, until we learn that on some architectures, an addition is faster than an addition.

```
1
2  for (i = 0; i < N; i++)
3      A[i] += b * c;
```

Listing 4.1 – First sample code...

```
1  tmp = b * c;
2  for (i = 0; i < N; i++)
3      A[i] += tmp;
```

Listing 4.2 – ... with loop hoisting.

```
1
2  for (i = 0; i < N; i++)
3      B[i] = 3 * i;
```

Listing 4.3 – Second sample code...

```
1  tmp = 0;
2  for (i = 0; i < N; i++) {
3      B[i] = tmp;
4      tmp += 3;
5  }
```

Listing 4.4 – ... with strength reduction.

What those two examples underline is the fact that it is possible to write more efficient code if we apply some algebraic properties of computations, and if we allow ourselves to add some initialization code. Modern compilers are usually able to detect when it is possible to apply those optimizations. What we present in this subsection is a new optimization that cannot be done by the compilers, because it changes the data stored. As an introductory example, let us compare Listing 4.5 and Listing 4.6.

```
1 for (t = 0; t < nb_iter; t++) {
2     for (j = 0; j < m; j++)
3         E[j] = f(j, t);
4     for (i = 0; i < N; i++)
5         A[i] += E[g(i)] * c;
6 }
```

Listing 4.5 – Third sample code...

```
1 for (t = 0; t < nb_iter; t++) {
2     for (j = 0; j < m; j++)
3         E_[j] = f(j, t) * c;
4     for (i = 0; i < N; i++)
5         A[i] += E_[g(i)];
6 }
```

Listing 4.6 – ... with loop-not-so-invariant code motion.

A careful inspection of those two codes will make us understand why they are equivalent — provided that we do not use the array `E` elsewhere in the code. Of course, because `E_` is equal to `E * c`, then both updates of `A` are equivalent.

One might wonder why this transformation is an optimization. The answer lies in the numbers `m` and `N`. Listing 4.5 performs `nb_iter * N` multiplications, and Listing 4.6 performs `nb_iter * m` multiplications. When `m < N`, this transformation then reduces the number of operations needed.

In our PIC implementation, things are slightly more complicated. To understand the actual transformation performed, let us now compare Listing 4.7 and Listing 4.8.

```
1
2
3  for (t = 0; t < nb_iter; t++) {
4      for (j = 0; j < m; j++)
5          E[j] = f(j, t);
6      for (i = 0; i < N; i++) {
7          A[i] += E[g(i)] * c;
8          B[i] += A[i] * d;
9      }
10 }
```

Listing 4.7 – Fourth sample code...

```
1  for (i = 0; i < N; i++)
2      A_[i] = A[i] * d;
3  for (t = 0; t < nb_iter; t++) {
4      for (j = 0; j < m; j++)
5          E_[j] = f(j, t) * c * d;
6      for (i = 0; i < N; i++) {
7          A_[i] += E_[g(i)];
8          B[i] += A_[i];
9      }
10 }
```

Listing 4.8 – ... with loop-not-so-invariant code motion applied twice.

Those two codes are equivalent provided that we do not use the arrays `E` and `A` elsewhere in the code. If we remove line 8 from both listings, we recognize the transformation from Listing 4.5 to Listing 4.6, where we additionally multiplied `E` and `A` by `d`. We then understand that both updates of `B` on this line are equivalent.

Of course, we can still apply loop hoisting on line 5 of Listing 4.8. We let the compiler do this job. As for performance, Listing 4.7 performs `2 * nb_iter * N` multiplications and Listing 4.8 performs `N + 2 * nb_iter * m` multiplications (`N + 1 + nb_iter * m` if we apply loop hoisting). When `N + 1 + nb_iter * m < 2 * nb_iter * N`, this transformation then reduces the number of operations needed.

Finally, let us remark that there is no need to introduce new arrays `E_` and `A_`. We can update the arrays `E` and `A` in place.

In a PIC implementation, we can apply this transformation. This is due to the fact that: (a) to update the velocities of particles, we need to multiply the field by $\frac{\Delta t \cdot q}{m}$ (it is the `c` of our example) and (b) to update the positions of particles, we need to multiply the velocities by $\frac{\Delta t}{\Delta\{x,y\}}$ (it is the `d` of our example). These multiplications can hence be performed outside the particle loop if, instead of storing the field and the velocities, we store the field multiplied by $\frac{\Delta t^2 \cdot q}{m \cdot \Delta\{x,y\}}$ and the velocities multiplied by $\frac{\Delta t}{\Delta\{x,y\}}$. For the velocities, this can be done during the initialization of the particles and for the electric field, this can be done at each iteration during the Poisson solve step; compare Listing 4.9 and Listing 4.10. This optimization leads to a gain in time of 3.5% with Intel and of 2% with GNU.

Let us remember that PIC implementations are memory bound. This transformation only

reduces the number of computations, and is still able to increase efficiency. This transformation is then expected to be a lot more efficient if it can be applied to compute bound codes.

We must acknowledge that this transformation leads to a code somehow harder to read, since we now have to remember that the arrays E and v do not store the electric field and the velocities, but those values multiplied by a constant. Even if we change the name of those variables, it is still hard to understand how adding `dx` and `v` might yield to anything that has a physical sense on lines 28–29 of Listing 4.10.

4.3.3 Data Structure and Layout for E and ρ: Cache Misses, Visual Insight

> “ Use Hilbert space-filling curve voxel indexing
> Pro: Improved temporal locality during particle advance [...]
> Con: Tricky to implement [...]
> K. J. Bowers [44, Time 57’38”] ”

This sub-section shows how to use space-filling curves to enhance cache efficiency of a PIC implementation. Those results are, to the best of our knowledge, unique in the literature. Even though previous authors seem to have already implemented it, they did not explain this method in papers.

Related Work

Space-filling curves are known to enhance cache performances on applications with regular memory accesses such as linear algebra [121], or with irregular accesses such as the n-body problem [124]. Nevertheless, none of those results can directly apply to a PIC implementation, which has irregular accesses over the arrays E and ρ and not the particle array itself, as in the n-body problem.

The space-filling curves are also of interest in particle implementations at the inter-process level, to achieve load balancing when using domain decomposition [59] or to minimize communication between processes [122], which has no impact on cache performances.

A rather popular technique on modern PIC implementations is to organize the particles by so-called super-cells, *e.g.*, PIConGPU [49], UPIC [53], ORB5 [61], PICADOR [81], PICSAR [87]. The space-filling curve layout we propose leads to a similar organization of the particles: particles are kept together in memory also in a block-like fashion, cf. Figure 4.5; nevertheless we do not apply a reordering at each time step, and when we reorder, we perform this operation by cell and not by super-cell.

Data Layouts for the Redundant Data Structure

As mentioned in Section 2.2.2, we have at our disposal two data structures for E and ρ: the standard 2d and the redundant one. The latter has been shown in [87, Section 4.1.2.] to be effective for SIMD architectures since it enables vectorization of the accumulate loop, see Listing 2.4.

We next show that using other memory layouts for the redundant data structure decreases the number of cache misses. We emphasize the fact that in PIC implementations, memory accesses are a major bottleneck. Every time there is an access to a cell of E or ρ, a contiguous portion of that array is loaded into the cache: we want to do all the computations that use these data cells while they are still there, avoiding to reload them later from the main memory.

Since particles are moving at each iteration, a periodic sorting of the particles needs to be applied in order to improve data locality. In this manner, two particles contiguous in memory are in the same grid cell and thus, they access the same E (or ρ) cell during the update-velocities (or accumulate) loop. Nevertheless, sorting at every iteration would be computationally expensive and therefore we have to find a memory layout of the cells such that the cache benefits

```
// Update-velocities
for (size_t i = 0; i < num_particle; i++) {
    [...]
    vx[i] += dt_q_over_m * E_x_particle; // Reads array Ex
    vy[i] += dt_q_over_m * E_y_particle; // Reads array Ey
}
// Update-positions
for (size_t i = 0; i < num_particle; i++) {
    x = (i_cell[i] / ncy) + dx[i] + dt_over_dx * vx[i];
    y = (i_cell[i] % ncy) + dy[i] + dt_over_dy * vy[i];
    [...]
}
```

Listing 4.9 – Update-velocities and update-positions loops, multiplications inside.

```
void poisson_solver([...] double** Ex, double** Ey) {
    [...]
    for (size_t i = 0; i < ncx; i++) {
        for (size_t j = 0; j < ncy; j++) {
            Ex[i][j] *= dt_q_over_m * dt_over_dx;
            Ey[i][j] *= dt_q_over_m * dt_over_dy;
        }
    }
}

void particle_initialization([...]) {
    [...]
    // WARNING : after, v doesn't represent the speed, but speed * dt / dx.
    for (size_t i = 0; i < num_particle; i++) {
        vx[i] *= dt_over_dx;
        vy[i] *= dt_over_dy;
    }
}

// Update-velocities
for (size_t i = 0; i < num_particle; i++) {
    [...]
    vx[i] += E_x_particle; // Reads array Ex
    vy[i] += E_y_particle; // Reads array Ey
}
// Update-positions
for (size_t i = 0; i < num_particle; i++) {
    x = (i_cell[i] / ncy) + dx[i] + vx[i];
    y = (i_cell[i] % ncy) + dy[i] + vy[i];
    [...]
}
```

Listing 4.10 – Update-velocities and update-positions loops, with loop-not-so-invariant code motion.

from the sorting last as long as possible. More precisely, using notations from Section 2.2.1, our aim is to find a mapping $(i_x, i_y) \mapsto i_{\text{cell}}$ so that we obtain, when a particle changes a cell, a high probability that its new cell-index i_{cell} is close to the old one.

We remark that the mapping of the row-major ordering in equation (4.1) has advantageous data locality when a particle moves along the y-axis: if i_y increases by one, the new cell-index also increases by one (except for particles on the right edge of the grid), becoming the index accessed by the following particles in the particle array. However, when a particle moves along the x-axis, this good behavior is lost: if i_x increases by one, the cell-index changes by ncy which implies cache misses for E and ρ.

$$\begin{aligned} (i_x, i_y) \mapsto i_{\text{cell}} &= i_x \cdot ncy + i_y \\ i_{\text{cell}} \mapsto \begin{cases} i_x = \left\lfloor \frac{i_{\text{cell}}}{ncy} \right\rfloor \\ i_y = \bmod (i_{\text{cell}}, ncy) \end{cases} \end{aligned} \tag{4.1}$$

Four different strategies for ordering the cells have been tested. They are listed below from the least to the most computational-intensive, in terms of the computation of the mapping $(i_x, i_y) \mapsto i_{\text{cell}}$:

(i) Scan-order (or row-major order), cf. Figure 4.3: the canonical `C` memory layout.
(ii) "Column-major of row-major"-order (or L4D-order), cf. [121, Section 2.1.] and Figure 4.5. We re-designed algorithms to convert from and to this ordering.
(iii) Morton-order ($\mathcal{N}$-order) or Lebesgue-order (Z-order), cf. [125] and Figure 4.4. Algorithms to convert from and to this ordering can be found in [126].
(iv) Hilbert-order, cf. [123] and Figure 4.6. Algorithms to convert from and to this ordering can be found in [127].[7]

Cache Misses Improvements

We present in Figure 4.7 and Figure 4.8 the evolution in time of the number of cache misses for each of these orderings, for the L2 and L3 cache levels. As for the L1 level, we obtain very close values for all the orderings, see also Table 4.4. At the first iteration, particles are sorted according to the given ordering. Then, we remark the steep descent of the cache misses number every 20 iterations due to sorting.

Clearly, the general idea those figures underline is that the three non-canonical layouts entail less cache misses than the row-major one. Going further, we see that all the curves give similar results when the particles become randomized (at least for the L2 cache). However, the good data locality due to the sorting keeps longer in time for the three non-canonical curves than for the row-major.

As a drawback, we notice that the time to compute i_{cell} to/from i_x and i_y is greater for these layouts than for the row-major one. Thus, we need to compare in the following the overall performance of these space-filling curves. The L4D and Morton curves give the best results overall as reported in Table 4.5. The update-positions loop takes much larger times when using the Hilbert ordering, the cause being that, to the best of our knowledge, there is no "efficient-enough" algorithm for computing this bijection. Therefore being the slowest in overall simulation time, the Hilbert ordering has to be discarded. We note that the total column is obtained by adding to the sum of the three first columns the time of sorting. Those orderings are faster than the row-major ordering for the accumulate loop (a 15% gain using Intel and an 11% gain using GNU), that was already better than the standard 2d structure thanks to vectorization. The redundant data structure with row-major ordering is not better than the

[7] With a small typo: in `for(i=n-1;i>=0;i--) X[i] ^= X[i-1];`, replace `i>=0` with `i>0` or it causes to read the nonexistent cell `X[-1]`.

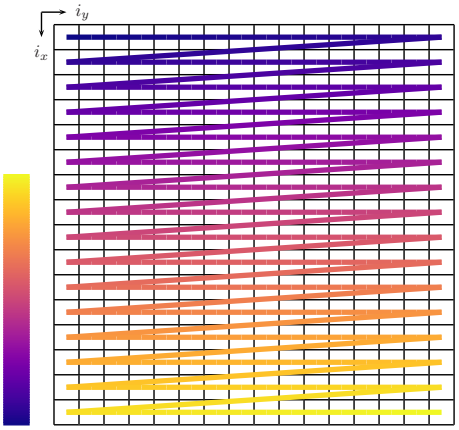


Figure 4.3 – Row major layout of a 16×16 matrix.

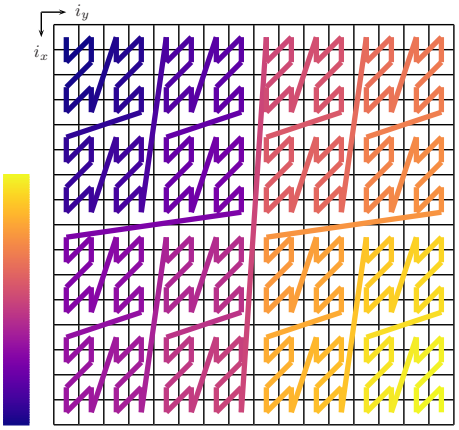


Figure 4.4 – Morton layout of a 16×16 matrix.

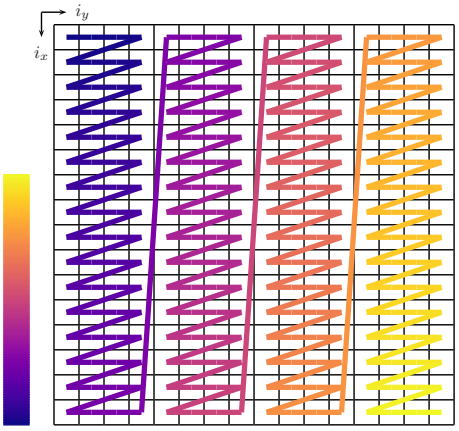


Figure 4.5 – L4D layout of a 16×16 matrix, SIZE=4.

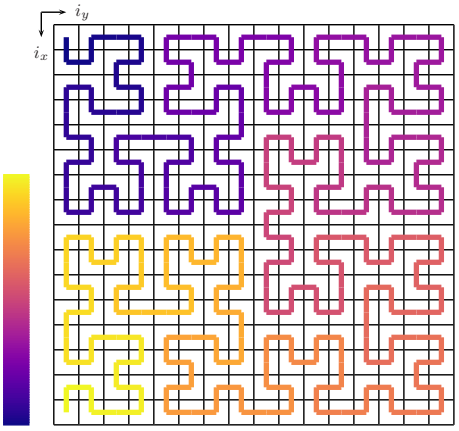


Figure 4.6 – Hilbert layout of a 16×16 matrix.

standard 2d structure for the update-velocities (5% lost with Intel, no time change with GNU), but with those orderings, we are able to gain time (3% gained with Intel, 9% gained with GNU). This might not seem significant but we emphasize that the new data structure needs four times more memory than the standard 2d one.

The results in Table 4.5 show 3 extra seconds in the update-positions loop for the L4D and Morton layouts. The reason is that for these two layouts we store, in addition to the cell-index i_{cell}, the indices i_x and i_y for each particle. We also tested the computation of i_x and i_y from i_{cell}, instead of storing them and we remarked that this is a slower approach. In contrast, for the row-major layout, that computation can be done in only one operation and therefore we do not need to store i_x and i_y.

The computation of i_{cell} can be achieved via different algorithms. In [126], two algorithms for the Morton layout are proposed: one that takes 12 operations and one that takes 5 operations plus two loads from a lookup table. We implemented the same idea for the L4D layout. In both cases, the lookup table creates an indirection which is not vectorizable and therefore this approach has to be discarded. Thus, we chose the Algorithm 5 from [126] for the Morton layout and we propose the one in (4.2) for the L4D-order.

$$
\begin{aligned}
(i_x, i_y) \mapsto i_{\text{cell}} &= ncx \cdot \text{SIZE} \cdot \lfloor i_y / \text{SIZE} \rfloor + \text{SIZE} \cdot i_x + \text{mod}(i_y, \text{SIZE}) \\
i_{\text{cell}} \mapsto &\begin{cases} i_x = \lfloor \text{mod}(i_{\text{cell}}, ncx \cdot \text{SIZE}) / \text{SIZE} \rfloor \\ i_y = \text{mod}(i_{\text{cell}}, \text{SIZE}) + \text{SIZE} \cdot \lfloor i_{\text{cell}} / (ncx \cdot \text{SIZE}) \rfloor . \end{cases}
\end{aligned}
\tag{4.2}
$$

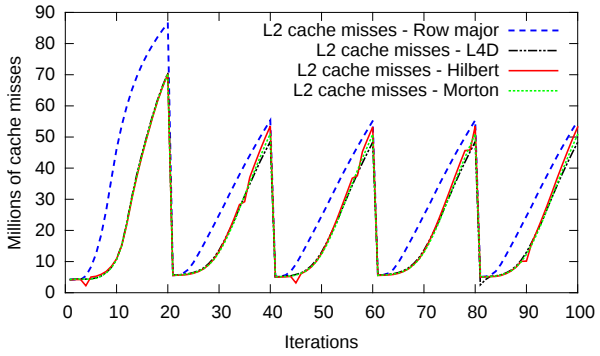


Figure 4.7 – Millions of cache misses per iteration for the cache level 2 during the update-velocities and accumulate loops. Test case in Table 4.3.

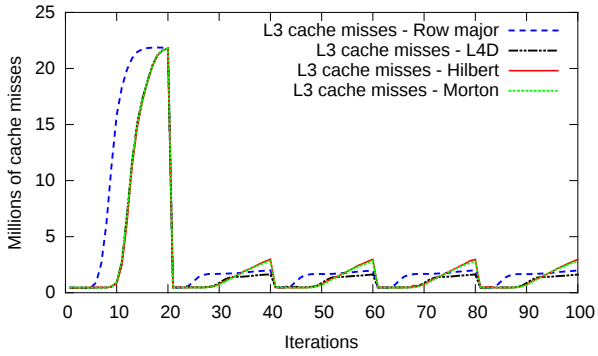


Figure 4.8 – Millions of cache misses per iteration for the cache level 3 during the update-velocities and accumulate loops. Test case in Table 4.3.

	L1	L2	L3
Row-major	95.4	43.3	4.94
L4D	92.0	27.8	3.14
Morton	91.1	27.0	3.20
Hilbert	90.9	27.1	3.29
Improvement	−3.5%	−36%	−36%

Table 4.4 – Average cache misses per iteration (in millions) for the update-velocities and accumulate loops. Test case in Table 4.3.

	Up. v	Up. x	Acc.	Total
2d standard[8]	30.6	12.5	20.7	74.3
Row-major	32.3	12.8	14.9	70.5
L4D	29.7	15.9	12.7	68.8
Morton	29.6	15.3	12.7	69.0
Hilbert	30.0	133.1	12.8	185.8

Table 4.5 – Time spent in the different loops (in seconds). Test case in Table 4.3.

Additional Remarks

For the L4D-order, we have to choose carefully the SIZE number depending of the cache sizes. In our tests, SIZE=8 led to the best times. The Morton-order gives the same speedup as the L4D-order, and does not depend on cache sizes: the update-velocities and accumulate loops become cache-oblivious [150].

As a remark, this optimal choice of SIZE=8 is a divisor of our grid size 128. But it is to note that choosing a value of SIZE that does not divide *ncy* is possible: then, there will be a few allocated cells that correspond to physical positions outside the boundaries and that will never be accessed.

Can You See it at a Glance?

> “In order to convince ourselves of the presence or of the quality of an object, we like to see and to touch it. [...] We prefer, of course, a short and intuitive argument to a long and heavy one: *Can you see it at a glance?*
>
> G. Pólya [31, Part I, Section 13] ”

[8] The 2d standard data structure for E and ρ is storing them as `E[ncx][ncy]` and `rho[ncx][ncy]`, see Section 2.2.2.

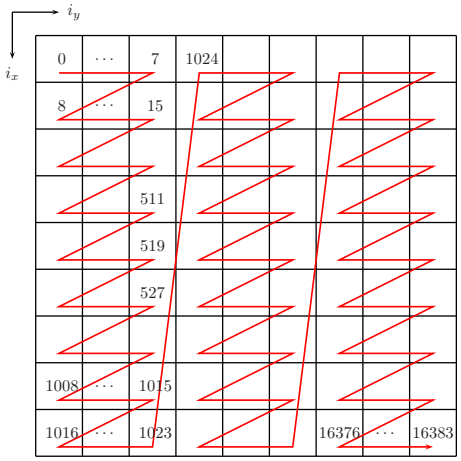


Figure 4.9 – L4D layout of a 128 × 128 matrix, SIZE=8.

Figure 4.9 allows us to explain why the L4D-order makes less cache misses than the row-major order. In the following explanation, we chose SIZE=8 as in the picture. It can be replaced with other values, as long as they are not too large for the cache (notice that SIZE=*ncy* corresponds to the row-major ordering). In this context, when a particle moves horizontally, $\frac{7}{8}$ of the time, it will lead to a new index close to the old one (i_{cell} changed to $i_{\text{cell}} + 1$); only $\frac{1}{8}$ of them will thus generate cache misses twice. All the vertical moves lead to a new index close to the old one (i_{cell} changed to $i_{\text{cell}} + 8$) - except on the boundary. Contrast this with the row-major ordering, in which all the horizontal moves are good (i_{cell} changed to $i_{\text{cell}} + 1$) and all the vertical ones are bad (i_{cell} changed to $i_{\text{cell}} + 128$); assuming that the particles movement is isotropic (particles move along the x and y-axes with no favorite direction), it means that we can expect up to 43% $\left(= \frac{7}{8} \times 50\%\right)$ less cache misses on E and ρ. The overall improvement shown in Table 4.4 is nevertheless lower since we have to take into account cache misses from the particle array.

We now give snapshots of our Landau damping simulation, to give a visual intuition of why this works. We recall that our test case is roughly isotropic, and that different conclusions may have to be drawn with test cases that behave very differently. We show in Figures 4.10–4.17 snapshots of the same 2d simulation, one using the row-major curve and the other using the L4D curve. For each ordering, two snapshots are taken: a first snapshot just after a sorting, and another one 3 iterations later. For each ordering, a coloring is applied on the particles to indicate which particles are close in memory. We will here focus on the black particles in the center of the row-major ordering just after the sorting, and on the deep purple particles in the center of the L4D ordering just after the sorting. Just after the sorting, the black particles occupy only the center cells. Three iterations later, they are spread on 7 as many cells (some move up to 3 cells up, some move up to 3 cells down - the ones that move horizontally do not incur the use of new cells). Just after the sorting, the deep purple particles occupy only the center cells. Three iterations later, they are spread on 3 as many cells (they are spread on a 14 × 14 square instead of a 8 × 8 square). This explains why, when fetching the E values and when updating the ρ values, the L4D curve will lead to less cache misses than the row-major one.

4.3.4 Vectorization of the Update-Positions Step

> “ While I cannot strongly recommend this paper, I would highly recommend it to someone who was learning vectorization for the first time, especially in a similar problem domain.
>
> Anonymous referee, reviewing our first article [204, Section IV.C] ”

In this section, we will explain how to efficiently vectorize the update-positions loop. As pointed by an anonymous referee, this section mostly serves the purpose of explaining the technical details behind the vectorization in general. Mastering those details is mandatory to unleash the power of modern architectures.

SIMD architectures can handle several operations at once: they compute on vectors rather than on scalars. For example, with vectors of size 256 bits, it takes “as much” time to compute

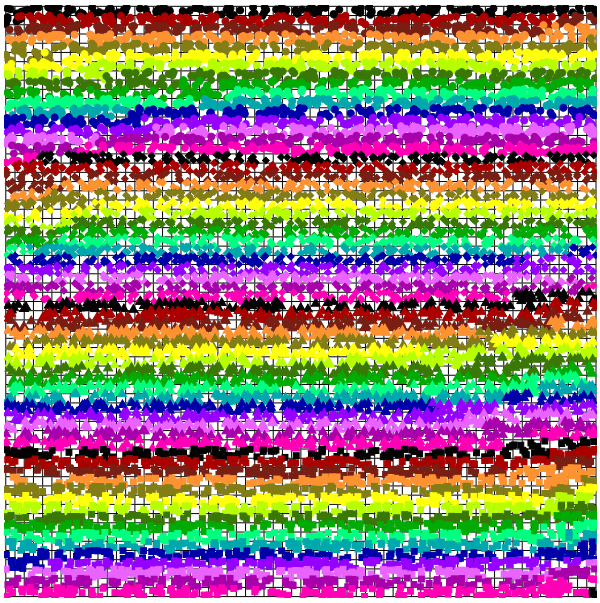


Figure 4.10 – Row-major, just after a sorting.

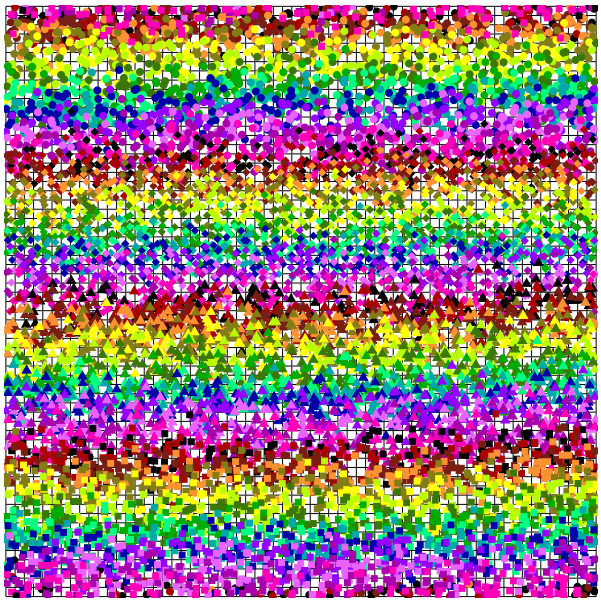


Figure 4.11 – Row-major, 3 iterations after a sorting.

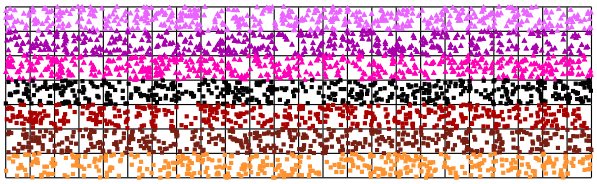


Figure 4.12 – Row-major, just after a sorting (zoom).

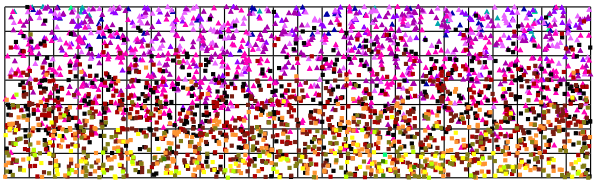


Figure 4.13 – Row-major, 3 iterations after a sorting (zoom).

4 multiplications on double-precision real numbers (each of size 64 bits) than to compute only one multiplication. The `-ftree-vectorize` compilation flag (activated from `-O2` with the Intel compiler, from `-O3` with the GNU compiler) is one possibility to automatically use vector operations. Another possibility is to use `#pragma omp simd` from `OpenMP 3.0`. However, in order to enable real vector performances, we need to rewrite the code in addition to the use of an appropriate data structure.

Array of Structures (AoS) or Structure of Arrays (SoA)?

To achieve the full power of vectorization requires that the Single Instruction operates on Multiple Data that are contiguous in memory. Using AoS for the particles leads to a stride of 4 or 8 between two data to be vectorized[9]. The GNU compiler does not vectorize such a code, as shows a compilation report with `-fopt-info-vec-all 2> vect_info.txt`.

```
note: Detected interleaving of size 8
note: Data access with gaps requires scalar epilogue loop
[...]
note: not vectorized: complicated access pattern.
note: bad data access.
```

[9] If we use `struct { int i_cell; float dx, dy; double vx, vy; }` for particles, then `p[i].dx` and `p[i+1].dx` are separated by memory equivalent to 8 `floats`; `p[i].vx` and `p[i+1].vx` are separated by memory equivalent to 4 `doubles`.

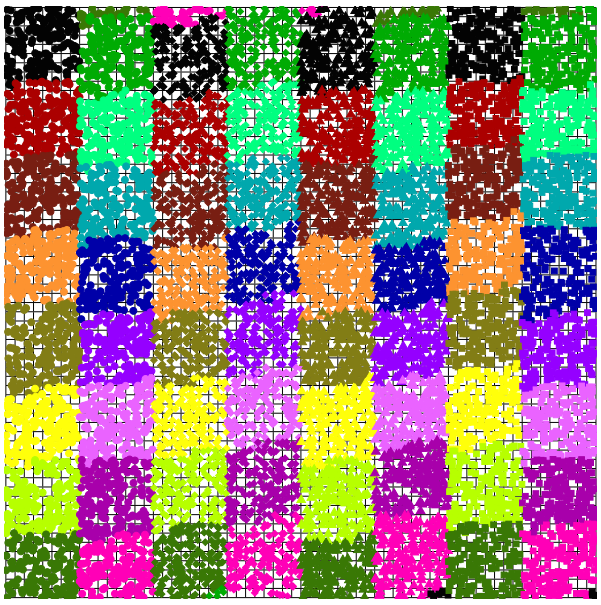


Figure 4.14 – L4D, just after a sorting.

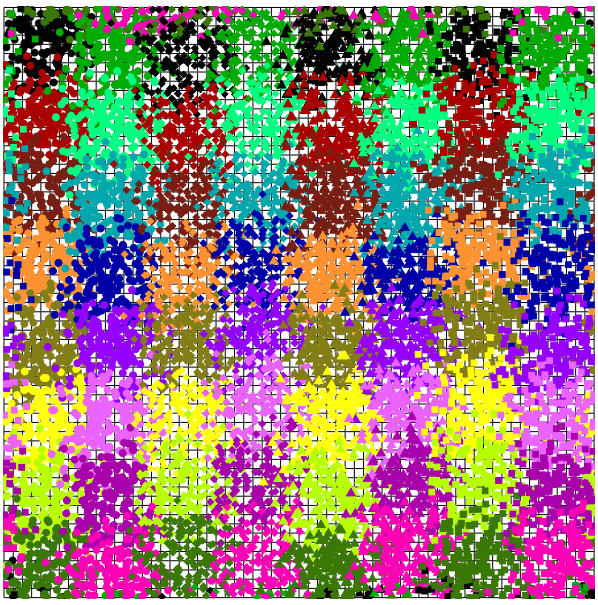


Figure 4.15 – L4D, 3 iterations after a sorting.

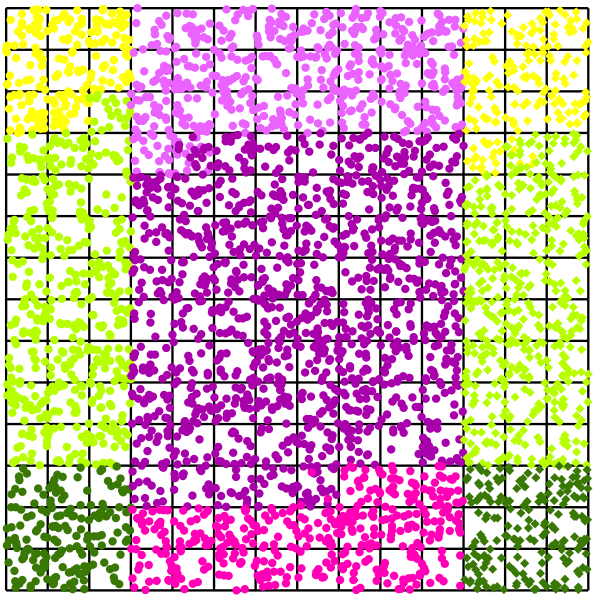


Figure 4.16 – L4D, just after a sorting (zoom).

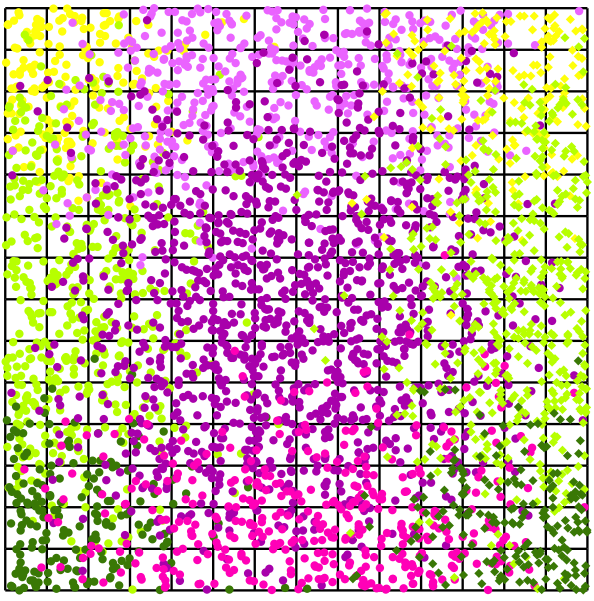


Figure 4.17 – L4D, 3 iterations after a sorting (zoom).

The Intel compiler has to use non-unit stride loads and stores, as shows a compilation report with `-qopt-report=5 -qopt-report-phase=vec`. Thus, using SoA guarantees the best timings.

```
non-unit strided load was generated for the variable <x->dx[i]>, stride is 8
[...]
unmasked strided loads: 7
unmasked strided stores: 3
```

Remove the `ifs`

When updating the positions of the particles in a periodic setting, we need to get in the physical domain the particles going outside. Usually, this is done by testing if the new position is still inside the grid. Without any concern for vectorization, `ifs` in a program are expensive (when incorrectly predicted, they cause rollbacks and inhibit the filling of arithmetic pipelines). Moreover, they prevent automatic vectorization (for the GNU compiler) or at best give unsatisfactory efficiency results. Therefore, our goal is to remove the `ifs` thanks to an efficient rewriting of the code.

As shown in Section 2.2.1, the positions of the particles are stored each with an integer (for the nearest lower grid position) and a real number (for the distance to that grid position). Since periodic boundary conditions are used, if a particle leaves the grid from one side, it goes to the beginning of the grid from the opposite side. This can be implemented by using an extension of the modulo[10] over the reals, see Listing 4.11. One obvious way to remove the `if` in this listing would be to... remove the line 3 that contains an `if`. It would work because the modulo produces the correct value even when $x \in [0; ncx)$. But then, computing the modulo for each particle would add a lot of computations. Can we rewrite this step in a more efficient way?

```
1 double modulo(double a, double b) { return a - floor(a / b) * b; }
2
3 if (x < 0. || x >= ncx)
4     x = modulo(x, ncx);
5 i_x = (int)floor(x);
6 dx[i] = x - i_x;
```

Listing 4.11 – Update-positions step, initial code.

Two ideas for removing the `ifs` are proposed in [54]. However, they are only useful in the accumulate step. For the update-positions step, however, given the assumption that no particle will cross the full grid[11], a code is shown that needs slightly less operations, see Listing 4.12.

To enable efficient vectorization, we can modify this listing by using the fact that in `C`, a test returns 1 for `true` or 0 for `false`, see Listing 4.13.

```
1 if (x <  0.)  x += ncx;
2 if (x >= ncx) x -= ncx;
3 i_x = (int)floor(x);
4 dx[i] = x - i_x;
```

Listing 4.12 – Update-positions step, adapted from [54].

```
1 x += ((x < 0.) - (x >= ncx)) * ncx;
2 i_x = (int)floor(x);
3 dx[i] = x - i_x;
```

Listing 4.13 – Update-positions step, optimized from [54].

Remove Function Calls

In the previous listings, one can see a call to the `floor` function. The Intel compiler vectorizes such a function call, but this is not the case for the latest GNU compiler. In order to solve this

[10] $modulo(a, b)$ is the unique real number in $[0; b)$ such that $a - modulo(a, b)$ is an integer multiple of b.

[11] Which seems reasonable: how can the simulation be precise if a particle can travel more than a full grid away during one single iteration?

problem, we can rewrite the code as in Listing 4.14. Because the modulo operator % returns a value whose sign is the sign of the input value, we add b before computing the modulo. This also uses the fact that no particle crosses the full grid.

```
1 #define modulo(a, b) ((a + b) % b)
2
3 floor_x = (int)x - (x < 0.);
4 i_x = modulo(floor_x, ncx);
5 dx[i] = x - floor_x;
```

Listing 4.14 – Update-positions step, integer computations.

Line 3 of Listing 4.14 computes `(int)floor(x)`[12] because the cast to an integer `(int)x` removes what is after the comma[13]. Thus if x is negative, we have to remove 1. This is once more done with a test which returns 1 for `true` or 0 for `false`.

Rewriting the code in this way enables automatic vectorization by the GNU compiler too. In addition, the Intel compiler produces faster vectorized code (31% time improvement on the update-positions loop).

As a remark, we can note that an additional optimization may be profitable when ncx is a power of two. When b is a power of two, computing `modulo(a, b)` is equivalent to computing a bitwise AND between a and `b - 1`, because the numbers are encoded in binary. For example, if we have 128 grid cells, we have to compute modulo 128: computing `modulo(a, 128)` in binary is the same as taking the seven least significant bits; in other words, it is exactly a bitwise AND between a and $127_{10} = 1111111_2$[14]. This optimization slightly improved efficiency on one core. In the assembly code, there are only 4 more lines when using the classical `% ncx` compared to using the less readable `& ncx_minus_one` when ncx is known at compile time, compare Listing 4.15 and Listing 4.16. If ncx is not known at compile time, there is only one more line, compare Listing 4.17 and Listing 4.18.

```
1 sarl $31, %ecx
2 shrl $25, %ecx
3 addl %ecx, %edx
4 andl $127, %edx
5 subl %ecx, %edx
```

Listing 4.15 – Using `% 128`.

```
1 andl $127, %edx
```

Listing 4.16 – Using `& 127`.

```
1 cltd
2 idivl %ebp
3 movl %edx, -32(%rcx)
```

Listing 4.17 – Using `% ncx`.

```
1 andl %ebp, %eax
2 movl %eax, -32(%rcx)
```

Listing 4.18 – Using `& ncx_minus_one`.

4.3.5 Overall Gains and Comparisons

In this section, we presented sequential optimizations. When using all the optimizations together, we remark that the simulation runs slightly faster with the Intel compiler than with the GNU one (1.8%). Our optimizations are thus summarized in Table 4.6 with the former. In this table, the baseline is an version of the code with the standard 2d data structure for E and ρ and the Array of Structures for the particles. The gains (in %) are computed with respect to the previous line and the accumulated gains are computed with respect to the baseline.

[12] In C, the `floor` function returns a `double`, hence the cast of `floor(x)` to an `int`.
[13] If x is 2.3, then `(int)x` is 2. However, if x is -3.14, then `(int)x` is -3 which is not the floor of x.
[14] It works also for negative numbers, thanks to the two's complement.

	Time (s)	Gains	Accumulated gains
Baseline	120.4	0.0%	0.0%
+ Loop-not-so-invariant code motion	113.4	5.8%	5.8%
+ Loop Fission	97.9	13.7%	18.7%
+ Redundant arrays (E and ρ)	94.0	4.0%	21.9%
+ Structure of Arrays (*particles*)	76.0	19.1%	36.9%
+ Space-filling curves (E and ρ)	72.6	4.5%	39.7%
+ Optimized update-positions loop	68.8	5.2%	42.8%

Table 4.6 – Total execution time, gains and accumulated gains. Test case in Table 4.3.

	Decyk & Singh work [53]	Present work	Present work
Push	19.9	15.6	9.1
Accumulate	9.0	4.3	2.6
Reorder	0.3	-	-
Sorting	-	1.9	2.0
Total	29.2	21.8	13.7
Particle crossing	12%	87%	87%
Processor	Nehalem	Sandy Bridge	Haswell
Theoretical bandwidth	10.7 GB/s	12.8 GB/s	17 GB/s
Normalized total	312	279	233

Table 4.7 – Top: time spent per particle per iteration, in nanoseconds (lower is better, not directly comparable). Middle: simulation and architecture parameters. Bottom: the normalized total is Total $\times$ Theoretical bandwidth (lower is better, can be compared).

Review of different optimizations

- Loop-not-so-invariant code motion: $O(ncx \times ncy)$ operations instead of $O(N)$ operations.
- Loop fission: better memory management, allows to vectorize the update-positions loop.
- Redundant arrays: vectorized accumulation loop.
- Structure of Arrays: stride-1 vectorization in the update-positions loop.
- Space-filling curves on the redundant data structure: 36% less cache misses on the accumulate and update-velocities loops.
- Optimized update-positions loop (no control flow, no function call): 31% less time on that loop, able to vectorize with the GNU compiler too.

Overall, these optimizations result in 75 million particles processed per second, on one hyper-threaded core on Intel Haswell architecture (65 million particles processed per second without hyper-threading). Those performances are compared in Table 4.7 to the electrostatic 2d Vlasov-Poisson implementation presented in [53], which uses domain-decomposition. The loop called "push" in that paper corresponds to the two loops we call update-velocities and update-positions. After that push, some of the particles move from one sub-domain to another. This is treated in a step called "reorder" which is not a full sorting, like in our algorithm. The reorder from that paper and the sorting in our work are separated in Table 4.7 because they are not directly comparable. For the sorting, we ran several simulations and we found that the optimal number of iterations between two sorting steps is 50 on Sandy Bridge architecture, and 20 on Haswell architecture. The results in Table 4.7 are thus presented for these sorting frequencies. As stated in Section 4.3.1, this underlines that it is interesting to implement an automatic finding of this optimal number.

Before explaining the parallelism that we used, let us finish this section by showing the baseline code, in Listing 4.19 and the optimized one, in Listing 4.20. When looking at them, one can wonder how come there is a difference of 42.8% in performance between two codes that look so similar. The two codes are available at `http://www.barsamian.am/Pic-Vert/` to reproduce those results on other architectures[15].

4.4 Parallel Optimizations in 2d

In `Pic-Vert`, we used the same kind of parallelism as in [51]: particle decomposition for distributed memory parallelism (`MPI`), see Section 2.6, and parallel loops among threads for shared memory parallelism (`OpenMP`).

The new features compared to that paper are the parallelization of the sorting among threads and `OpenMP 4.5` reduction on array sections. We will describe those two features in the next subsections, and show some performance results.

4.4.1 Particle Sorting

> “ [The sorting] problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools.
>
> T. H. Cormen, C. E. Leiserson, R. L. Rivest & C. Stein [10, Section 1] ”

Before explaining how to parallelize a sorting algorithm, let us first explain the sorting algorithm and its goal in our application. Sorting is taking a set of elements and arrange them following a given order. For example, in the Acknowledgements section of a thesis, there is a set of persons and institutions that the author wishes to thank, and he or she has to abide by a formal order in which those persons and institutions should appear.

In our PIC application, in the usual case when the particles are stored[16] in an array, it has been found that sorting this array with respect to the physical dimensions leads to substantial gains in the execution time [54, Section VI]. However, to sort an array of N cells is generally time-consuming: it usually needs $\Omega(N \log N)$ comparison operations [10, Theorem 8.1]. Fortunately, we are not interested here in a full sorting of the particles with respect to their positions. We just need them to be sorted grid cell by grid cell. Let us take for example the coinche[17] hand of cards of Figure 4.18.

You might be interested in sorting this hand exactly, with the order ♣ < ♦ < ♠ < ♥ for the colors, and inside each color, the order $7 < 8 < 9 < 10 <$ *Jack* $<$ *Queen* $<$ *King* $<$ *Ace*. This will take you some time, and you will end up with the left hand of Figure 4.19.

Now if you just want the cards ordered color by color, the sorting is faster. For example for each color you can just let the cards be in their initial order, and you will end up with the right hand of Figure 4.19. In the following, we will be interested only in this last sorting. The algorithm we will use to achieve such a goal is called the counting sort [10, Section 8.2]. Together with the radix sort [10, Section 8.3], it is one of the most popular algorithm to sort particle arrays in the literature (*e.g.*, [47, 59]). There are only two requirements for the counting sort: that each array element has a key associated, and that there is a finite number of keys.

When the number of keys is $O(N)$ (which is true here, because we always have more particles than grid points), the main advantage of this sorting algorithm is that its complexity is $O(N)$ instead of $O(N \log(N))$.

The main drawback of this sorting algorithm is that it requires an auxiliary array to be really fast, thus doubling the memory requirements. The initial SeLaLib implementation used

[15] Respectively in `sim2d_aos_1valueERho_baseline.c` and in `sim2d_soa_4corners_ipdpsw2017.c`.
[16] Do not mix stored and sorted :)
[17] A French card game, see `https://en.wikipedia.org/wiki/Coinche`

```
for (size_t i = 0; i < num_particle; i++) { // Particle loop
    i_x = particles[i].i_cell / ncy;
    i_y = particles[i].i_cell % ncy;
    Ex = (     particles[i].dx) * (     particles[i].dy) * E1[i_x + 1][i_y + 1]
       + (1. - particles[i].dx) * (     particles[i].dy) * E1[i_x    ][i_y + 1]
       + (     particles[i].dx) * (1. - particles[i].dy) * E1[i_x + 1][i_y    ]
       + (1. - particles[i].dx) * (1. - particles[i].dy) * E1[i_x    ][i_y    ];
    Ey = (     particles[i].dx) * (     particles[i].dy) * E2[i_x + 1][i_y + 1]
       + (1. - particles[i].dx) * (     particles[i].dy) * E2[i_x    ][i_y + 1]
       + (     particles[i].dx) * (1. - particles[i].dy) * E2[i_x + 1][i_y    ]
       + (1. - particles[i].dx) * (1. - particles[i].dy) * E2[i_x    ][i_y    ];
    particles[i].vx += dt_q_over_m * Ex;
    particles[i].vy += dt_q_over_m * Ey;
    x = i_x + particles[i].dx + dt_over_dx * particles[i].vx;
    y = i_y + particles[i].dy + dt_over_dy * particles[i].vy;
    if (x < 0. || x >= ncx)
        x = modulo(x, ncx);
    if (y < 0. || y >= ncy)
        y = modulo(y, ncy);
    i_x = (int)floor(x);
    i_y = (int)floor(y);
    particles[i].dx = x - i_x;
    particles[i].dy = y - i_y;
    particles[i].i_cell = i_x * ncy + i_y; // Row-major
    rho_2d[i_x    ][i_y    ] += (1. - particles[i].dx) * (1. - particles[i].dy);
    rho_2d[i_x    ][i_y + 1] += (1. - particles[i].dx) * (     particles[i].dy);
    rho_2d[i_x + 1][i_y    ] += (     particles[i].dx) * (1. - particles[i].dy);
    rho_2d[i_x + 1][i_y + 1] += (     particles[i].dx) * (     particles[i].dy);
}
```

Listing 4.19 – Baseline 2d code.

```
for (size_t i = 0; i < num_particle; i++) { // Update velocities
    vx[i] += (     dx[i]) * (     dy[i]) * field_accu[i_cell[i]].Ex_ne
           + (1. - dx[i]) * (     dy[i]) * field_accu[i_cell[i]].Ex_nw
           + (     dx[i]) * (1. - dy[i]) * field_accu[i_cell[i]].Ex_se
           + (1. - dx[i]) * (1. - dy[i]) * field_accu[i_cell[i]].Ex_sw;
    vy[i] += (     dx[i]) * (     dy[i]) * field_accu[i_cell[i]].Ey_ne
           + (1. - dx[i]) * (     dy[i]) * field_accu[i_cell[i]].Ey_nw
           + (     dx[i]) * (1. - dy[i]) * field_accu[i_cell[i]].Ey_se
           + (1. - dx[i]) * (1. - dy[i]) * field_accu[i_cell[i]].Ey_sw;
}
#pragma omp simd
for (size_t i = 0; i < num_particle; i++) { // Update positions
    x = icx[i] + dx[i] + vx[i];
    y = icy[i] + dy[i] + vy[i];
    ic_x = (int)x - (x < 0.);
    ic_y = (int)y - (y < 0.);
    icx[i] = (ic_x + ncx) % ncx;
    icy[i] = (ic_y + ncy) % ncy;
    dx[i] = x - ic_x;
    dy[i] = y - ic_y;
    i_cell[i] = COMPUTE_I_CELL(icx[i], icy[i]); // L4D (macro)
}
for (size_t i = 0; i < num_particle; i++) { // Accumulate
    #pragma omp simd
    for (corner = 0; corner < NB_CORNERS_2D; corner++)
        charge_accu[NB_CORNERS_2D * i_cell[i] + corner] +=
            (coeffs_x[corner] + signs_x[corner] * dx[i]) *
            (coeffs_y[corner] + signs_y[corner] * dy[i]);
}
```

Listing 4.20 – Optimized 2d code.

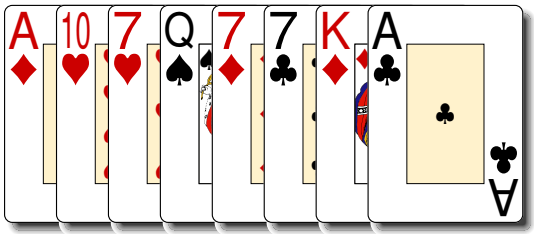


Figure 4.18 – Sample hand

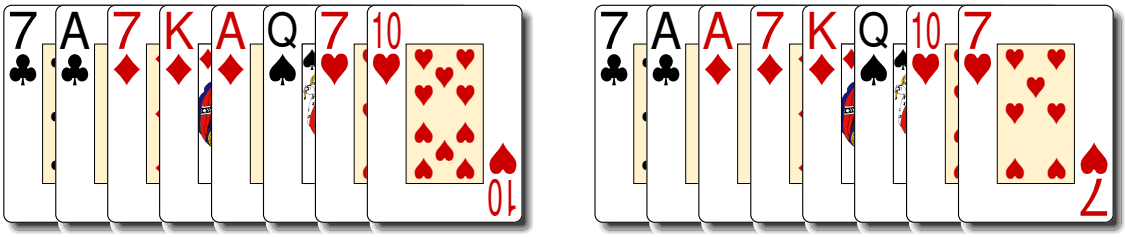


Figure 4.19 – Sorted hands

an *in-place* (without auxiliary array) sorting, and we thus replaced it with an *out-of-place* (with an auxiliary array) one since the beginning, to improve performances.

Let us follow this algorithm in its out-of-place version to sort our coinche hand. We have our initial hand, stored in an array `cards` of 8 cells. Our first goal is to count the number of cards of each color. We will store this number in the `nb_card` array.

cards								
color	♦	♥	♥	♠	♦	♣	♦	♣

color	♣	♦	♠	♥
nb_card	2	3	1	2

Now, we will put the cards in another `tmp` array, sorted by color. It is easy to see, thanks to the `nb_card` array, that the club cards will be in the first two cells, then the diamond cards in the next three cells, then the spade card in the next cell, and the heart cards in the last two cells. We would like to have an easy way, whenever we look at a card in the `cards` array, to know directly in which cell of the `tmp` array to put it. To do this, first we compute a *prefix sum*[18] of the `nb_card` array. This is just to know directly that the first club card will go in index 0, the first diamond card will go in index 2 (after the two club cards), the first spade card will go in index $2+3=5$ (after the two club cards and the three diamond cards), and the first heart card will go in index $2+3+1=6$ (after the two club cards, the three diamond cards, and the spade card). We will call this array `next_index`, because it tells us which is the index of the next card we have to put, for each color. For now, this array is:

color	♣	♦	♠	♥
next_index	0	2	5	6

The rest of the algorithm is straightforward. We go through the cells of `cards`, we put each card at the `next_index` of its color, and increment this value for the next card of this color. The different steps of this algorithm are depicted in Figure 4.20.

We can now focus on the counting sort inside a PIC implementation. To have a shorter code to look at, we will suppose that:

- our N particles are stored as an array of structures, i.e. we have a `particles` array, where `particles[i]` holds the offset and velocity of the i-th particle.
- we have an additional array `i_cell`, where `i_cell[i]` holds the cell index of the grid cell containing the i-th particle.

[18] Also called *scan* of an array A, defined as $S[i] = \sum_{k=0}^{i-1} A[k]$

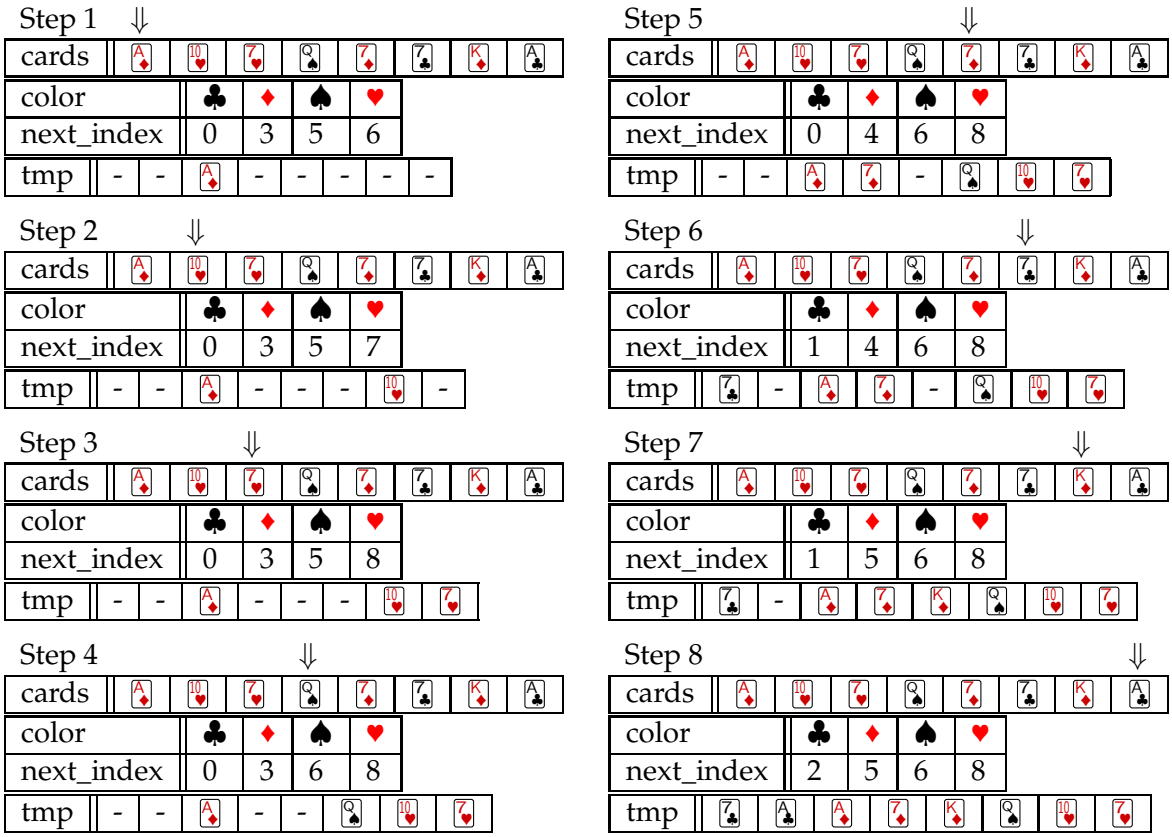


Figure 4.20 – Counting sort on a coinche hand.

```
1 struct { float  dx, dy;
2          double vx, vy; } particle3;
3 particle3* particles;
4 int* i_cell;
5 particle3* parts_tmp;
6 int* icell_tmp;
```

Listing 4.21 – Data structures for the counting sort.

- `parts_tmp` and `icell_tmp` are the required auxiliary arrays that contains the particle data sorted at the end of the algorithm. A simple pointer swap enables to have the arrays `particles` and `i_cell` sorted. In the case where a pointer swap is impossible, one could recopy the auxiliary arrays into the useful arrays.

Those data structures are summarized in Listing 4.21. In the actual code, we have four arrays for the offsets and velocities, following a structure of arrays data structure. This adds a lot of noise in the code, which explains why we present the code with this modification. The sequential code of our counting sort is given in Listing 4.22.

Now, let us try to parallelize such a code, with `OpenMP`. The number of *threads* that compute in parallel in `OpenMP` can be chosen before the program starts. In several parts of our program, we can ask them to work together (we open a parallel *region* with `#pragma omp parallel`), or we can let one of them do the job (outside a parallel region or if we add a *single* region inside a parallel one with `#pragma omp single`).

First, because $nb\_cell = o(N)$, we know that the "scan" part (lines 4–7) can be treated in sequential without changing much the scalability, because its cost is negligible. There exist a dedicated algorithm to do this in parallel in $O(N/p + \log p)$ with $p \leq N$ threads [138, 21], but we can avoid it here, and just treat it sequentially.

```
1  // Count the particles in each cell
2  for (i = 0; i < N; i++)
3      nb_particle[i_cell[i]]++;
4  // Scan : S[i] = sum_{k=0}^{i-1} A[k]
5  next_index[0] = 0;
6  for (j = 1; j < nb_cell; j++)
7      next_index[j] = next_index[j-1] + nb_particle[j-1];
8  // Everything in its right place
9  for (i = 0; i < N; i++) {
10     old_index = next_index[i_cell[i]]++; // Store value before incrementation
11     parts_tmp[old_index] = particles[i];
12     icell_tmp[old_index] = i_cell[i];
13 }
```

Listing 4.22 – Sequential code for the counting sort.

Let us now look at the two other parts. All the threads will be able to work on our arrays. Remember that, because they will work on the same array, there is a risk of data races. If we run those loops in parallel, where are the possible data races? This happens if two threads access the same data and at least one of them is modifying it. Where are the modifications?

- line 3: if two different threads are treating two particles that lie in the same grid cell, there is a possible race.
- line 10: the same goes here.
- lines 11–12: if we took care of avoiding races on the previous line, two different threads will always have different values for `old_index`, thus no race can happen.

To parallelize this code, we can avoid the first race with `#pragma omp atomic`, and the second race with `#pragma omp atomic capture`. We have to use two different *pragmas* because the second operation is different from the first one. The first one just increments a value, while the second one increments and "captures" its initial value. This first attempt leads to the code given in Listing 4.23. The `#pragma omp for` divides the for loop iterations between the threads.

We saw that an atomic operation induces a lock on the data. You could imagine that it thus takes more time to do the operation, and you would be right. This first, naive, attempt to parallelize our sort is bug-free, but it is far from being optimal. How can we avoid atomic operations while still avoiding data races? A second idea would be to divide the cells among the different threads, and to have each thread only handle the particles that lie on the cells it has in charge. If we assume that it is easy to make this division, and that each thread will handle more or less the same number of particles in the cells it is in charge of, then it can be legitimate to think of the code given in Listing 4.24. This time, we do not need anymore to divide the `for` loop among the threads. Instead, each thread will iterate on the full loop, but will only "do something" when it stumbles upon a particle it has to handle. Note that we need a synchronization point (`#pragma omp barrier`) after the first for loop, to be sure that all the threads finished updating the `nb_particle` array, before the prefix sum on this array is performed. There was no need to add a barrier in the previous code, because a loop divided among the threads with a `#pragma omp for` has an implicit barrier at its end.

The resulting code shows some acceptable timings, but of course we feel that something better can be done, because even if the cost of a test is really small in front of the cost of moving data from `particles` to `tmp`, we should succeed in writing a code without those additional tests.

Why did we put this test? We wanted to be sure that each thread would have its own set of particles that it handles, without interfering with other threads. This test was easy to design, because if we divide particles cell by cell, no conflict may appear. The downside is that each thread has to scan the whole particle array. Could we just divide the particle array among the

```
#pragma omp parallel private(i)
{
    // Count the particles in each cell
    #pragma omp for
    for (i = 0; i < N; i++) {
        #pragma omp atomic
        nb_particle[i_cell[i]]++;
    }
    // Scan : S[i] = ∑_{k=0}^{i-1} A[k]
    #pragma omp single
    {
        next_index[0] = 0;
        for (j = 1; j < nb_cell; j++)
            next_index[j] = next_index[j-1] + nb_particle[j-1];
    } // End single region
    // Everything in its right place
    #pragma omp for
    for (i = 0; i < N; i++) {
        #pragma omp atomic capture
        old_index = next_index[i_cell[i]]++;
        parts_tmp[old_index] = particles[i];
        icell_tmp[old_index] = i_cell[i];
    }
} // End parallel region
```

Listing 4.23 – First (naive) attempt to parallelize our counting sort.

```
// For example, we can set a constant number of cells per thread
// int num_cells_per_thread = nb_cell / omp_get_num_threads();
// Then in_charge(thread_id, cell_id) can be defined as
// cell_id / num_cells_per_thread == thread_id;
#pragma omp parallel private(i, thread_id)
{
    thread_id = omp_get_thread_num(); // id of the current thread
    // Count the particles in each cell
    for (i = 0; i < N; i++)
        if (in_charge(thread_id, i_cell[i]))
            nb_particle[i_cell[i]]++;
    #pragma omp barrier
    // Scan : S[i] = ∑_{k=0}^{i-1} A[k]
    #pragma omp single
    {
        next_index[0] = 0;
        for (j = 1; j < nb_cell; j++)
            next_index[j] = next_index[j-1] + nb_particle[j-1];
    } // End single region
    // Everything in its right place
    for (i = 0; i < N; i++) {
        if (in_charge(thread_id, i_cell[i])) {
            old_index = next_index[i_cell[i]]++;
            parts_tmp[old_index] = particles[i];
            icell_tmp[old_index] = i_cell[i];
        }
    }
} // End parallel region
```

Listing 4.24 – Second attempt to parallelize our counting sort.

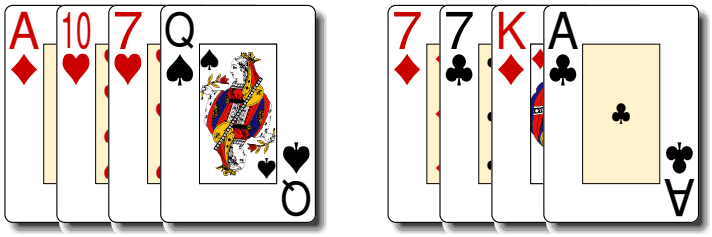


Figure 4.21 – Sample hand divided among two threads

threads, while still ensuring that no conflict will appear? Yes we can. Think of our prefix sum array. It reserves, for each cell index, a sufficient amount of space in the `tmp` array, to allow putting particles in this array at their correct place whichever is their cell index. Now that we think of its use, the solution is (almost) straightforward: each thread just has to reserve some space for the particles it will handle, for each cell index.

Let us take back our card example, and let us try to parallelize the sorting of the hand of Figure 4.18 among two threads. The first thread will put the 4 first cards at their correct locations, and the second thread will put the 4 last cards at their correct locations, as depicted in Figure 4.21.

Thus, thread 0 has no club cells to reserve, 1 diamond cell to reserve (the first among the 3 needed), 1 spade cell to reserve, and 2 heart cells to reserve. Thread 1 has 2 club cells to reserve and 2 diamond cells (the last two among the 3 needed).

tmp	7♣	A♣	A♦	7♦	K♦	Q♠	10♥	7♥
thread_id	1	1	0	1	1	0	0	0

To achieve such a division of the reservation, we need to change our `nb_particle` and `next_index` arrays. Instead of counting the total number of particles, they will count them only thread by thread. Thus, now, `nb_particle[thread_id][i]` is the number of particles that are in the i-th grid cell and handled by the `thread_id`-th thread, and `next_index[thread_id][i]` is the index where the `thread_id`-th thread should put the next particle that is in the i-th grid cell it encounters. The prefix sum also changes a little bit. We still have to count, for each cell index, the number of particles in the previous cell indices, but we also have to count the number of particles in the same cell index that are handled by previous threads. There remains only one last technical question. Remember that we have two loops on the particles. How can we be sure that each particle will be treated by the same thread in both loops? The `schedule static` ensures that the two loops are divided among threads in the same way. The static schedule will divide the number of particles evenly among threads. This should lead to a perfect load-balancing because the cost of moving each particle from `particles` to `tmp` should be roughly the same. This leads to the code shown in Listing 4.25.

In this third listing, the prefix sum computation requires more time, because we have now `num_threads * nb_cell` cells in the array instead of just `nb_cell`. But this is still really small in front of N.

A last optimization that is quite common when dealing with counting sort is to avoid having an auxiliary array for `i_cell`. Because the goal is to sort the particles according to it and we counted, for each value i, the number of particles `nb_particle[i]` that have i_{cell} value i, we can just put i inside the `i_cell` array `nb_particle[i]` times. This allows to scan the `i_cell` array with better locality and to avoid using a temporary array, compared to updating an auxiliary array `icell_tmp` together with `parts_tmp`. This last optimization leads to the code shown in Listing 4.26.

```
#pragma omp parallel private(thread_id, i, k)
{
    num_threads = omp_get_num_threads(); // number of threads
    thread_id = omp_get_thread_num();    // id of the current thread
    // Count the particles in each cell handled by each thread
    for (i = 0; i < nb_cell; i++)
        nb_particle[thread_id][i] = 0;
    #pragma omp for schedule(static)
    for (i = 0; i < N; i++)
        nb_particle[thread_id][i_cell[i]]++;
    // Scan : S[i_thr][j_cel] = sum_{k=0}^{i_thr-1} sum_{l=0}^{nb_cell-1} A[k][l] + sum_{l=0}^{j_cel-1} A[i_thr][l]
    next_index[thread_id][0] = 0;
    for (i = 0; i < thread_id; i++)
        next_index[thread_id][0] += nb_particle[i][0];
    for (k = 1; k < nb_cell; k++) {
        next_index[thread_id][k] = next_index[thread_id][k - 1];
        for (i = thread_id; i < num_threads; i++)
            next_index[thread_id][k] += nb_particle[i][k - 1];
        for (i = 0; i < thread_id; i++)
            next_index[thread_id][k] += nb_particle[i][k];
    }
    // Everything in its right place
    #pragma omp for schedule(static)
    for (i = 0; i < N; i++) {
        old_index = next_index[thread_id][i_cell[i]]++;
        parts_tmp[old_index] = particles[i];
        icell_tmp[old_index] = i_cell[i];
    }
} // End parallel region
```

Listing 4.25 – Third attempt to parallelize our counting sort.

4.4.2 Array section from OpenMP 4.5

The update-velocities and update-positions loops can be made parallel with `#pragma omp for`. The only problem arises for the accumulate loop. Only using the `#pragma omp for`, we have race conditions: particles from different threads will update the same ρ values. `OpenMP 4.5` can handle this by adding `reduction(+:rho[0:ncx*ncy][0:4])` to the pragma. Nevertheless, this `OpenMP 4.5` feature was not available with the latest Intel compiler. To exploit the previous advantages from this compiler, we rewrote this feature by hand (our hand-coded version showed no overhead with `gcc 6.2` when compared to the `OpenMP 4.5` feature).

4.4.3 Parallel Results on Curie

Our parallel results come from simulations executed on the supercomputer Curie. Each node has 2 sockets of 8 cores each, hence for the hybrid `MPI` + `OpenMP` results, we used one `MPI` process per socket and 8 threads per process. For the pure `MPI` results, we used one `MPI` process per core.

Figure 4.22 shows a weak scaling from 1 core to 8 192 cores (512 nodes, 10% of the total number of nodes of the Curie supercomputer). These simulations run with 50 million particles per core in order to use the full memory of each socket. We can see that up to 8 192 cores, the overhead due to `MPI_ALLREDUCE` stays acceptable with the hybrid parallelism, whereas it becomes a major bottleneck when using only `MPI`. The hybrid parallelization achieves 543 million particles processed per second on one node (2×8 cores).

Table 4.8 shows a strong scaling up to 8 cores, when using 50 million particles over one socket. This table and Figure 4.22 illustrate that our implementation reaches near-ideal scalability up to 4 cores, but not for 8 cores. The reason is that PIC implementations are memory-bound and therefore, the 4 memory channels per socket limit the scalability when using more

```
#pragma omp parallel private(thread_id, i, k)
{
    num_threads = omp_get_num_threads(); // number of threads
    thread_id = omp_get_thread_num();    // id of the current thread
    // Count the particles in each cell handled by each thread
    for (i = 0; i < nb_cell; i++)
        nb_particle[thread_id][i] = 0;
    #pragma omp for schedule(static)
    for (i = 0; i < N; i++)
        nb_particle[thread_id][i_cell[i]]++;
    // Scan : S[i_thr][j_cell] = sum_{k=0}^{i_thr-1} sum_{l=0}^{nb_cell-1} A[k][l] + sum_{l=0}^{j_cel-1} A[i_thr][l]
    next_index[thread_id][0] = 0;
    for (i = 0; i < thread_id; i++)
        next_index[thread_id][0] += nb_particle[i][0];
    for (k = 1; k < nb_cell; k++) {
        next_index[thread_id][k] = next_index[thread_id][k - 1];
        for (i = thread_id; i < num_threads; i++)
            next_index[thread_id][k] += nb_particle[i][k - 1];
        for (i = 0; i < thread_id; i++)
            next_index[thread_id][k] += nb_particle[i][k];
    }
    // parts_tmp in its right place
    #pragma omp for schedule(static)
    for (i = 0; i < N; i++) {
        old_index = next_index[thread_id][i_cell[i]]++;
        parts_tmp[old_index] = particles[i];
    }
    // i_cell in its right place
    for (i = 0; i < nb_cell; i++) {
        stop_index = next_index[thread_id][i];
        start_index = stop_index - nb_particle[thread_id][i];
        for (k = start_index; k < stop_index; k++)
            i_cell[k] = i;
    }
} // End parallel region
```

Listing 4.26 – Fourth (and last) attempt to parallelize our counting sort: no auxiliary array for `i_cell`.

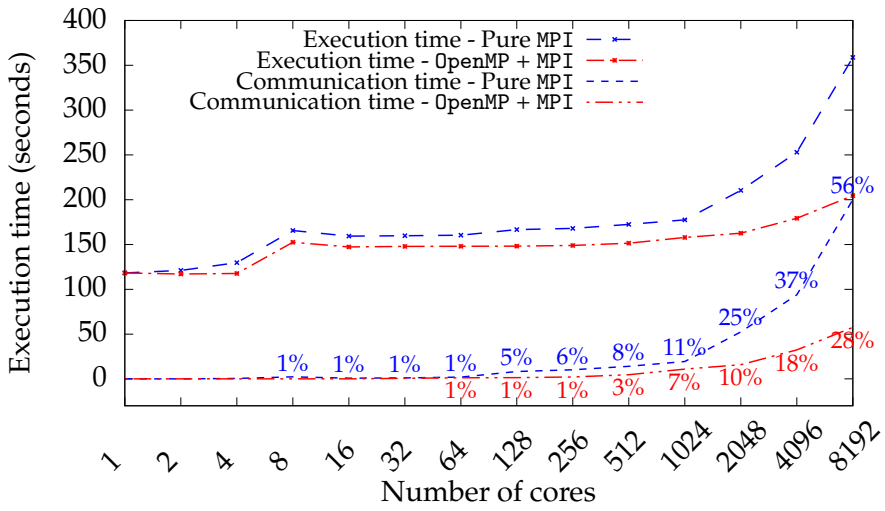


Figure 4.22 – Weak scaling on Curie : Hybrid `OpenMP + MPI` VS Pure `MPI`. Test case: 128 × 128 grid, 50 million particles per core, 100 iterations simulation (sorting every 50 iterations). Architecture: Sandy Bridge. Communication time is also shown as percentage of the execution time.

```
#pragma omp for private(i)
for (i = 0; i < num_particle; i++) {
    vx[i] += (     dx[i]) * (     dy[i]) * E_field[i_cell[i]].x_val.north_east
           + (1. - dx[i]) * (     dy[i]) * E_field[i_cell[i]].x_val.north_west
           + (     dx[i]) * (1. - dy[i]) * E_field[i_cell[i]].x_val.south_east
           + (1. - dx[i]) * (1. - dy[i]) * E_field[i_cell[i]].x_val.south_west;
    vy[i] += (     dx[i]) * (     dy[i]) * E_field[i_cell[i]].y_val.north_east
           + (1. - dx[i]) * (     dy[i]) * E_field[i_cell[i]].y_val.north_west
           + (     dx[i]) * (1. - dy[i]) * E_field[i_cell[i]].y_val.south_east
           + (1. - dx[i]) * (1. - dy[i]) * E_field[i_cell[i]].y_val.south_west;
}
// 2 * 8 * num_particle * 2 + // double vx, vy[num_particle] (read + write)
// 2 * 4 * num_particle +     // float dx, dy[num_particle] (read)
//     4 * num_particle +     // int i_cell[num_particle] (read)
// 8 * 8 * ncx * ncy          // double E_field[ncx*ncy][8] (read)
// => 44 num_particle + 64 ncx*ncy
```

Listing 4.27 – Memory bandwidth (in bytes) of the update-velocities step.

```
#pragma omp for private(x, y, ic_x, ic_y) firstprivate(ncxminusone, ncyminusone,
    icell_param)
for (i = 0; i < num_particle; i++) {
    x = icx[i] + dx[i] + vx[i];
    y = icy[i] + dy[i] + vy[i];
    ic_x = (int)x - (x < 0.);
    ic_y = (int)y - (y < 0.);
    icx[i] = ic_x & ncxminusone;
    icy[i] = ic_y & ncyminusone;
    dx[i] = (float)(x - ic_x);
    dy[i] = (float)(y - ic_y);
    i_cell[i] = COMPUTE_I_CELL_2D(icell_param, ic_x & ncxminusone, ic_y &
        ncyminusone);
}
// 2 * 8 * num_particle +     // double vx, vy[num_particle] (read)
// 2 * 2 * num_particle * 2 + // short int icx, icy[num_particle] (read + write)
// 2 * 4 * num_particle * 2 + // float dx, dy[num_particle] (read + write)
//     4 * num_particle       // int i_cell[num_particle] (write)
// => 44 num_particle
```

Listing 4.28 – Memory bandwidth (in bytes) of the update-positions step.

than 4 cores. To go further, we show in Figure 4.23 the memory bandwidth of our implementation, compared to that of the Stream benchmark [162]. On one hand, this histogram underlines that the update-velocities and accumulation steps are far to reach the peak memory bandwidth and thus, they have a good scaling up to 8 cores. Their low memory bandwidth is explained by the high number of cache misses on the E and ρ arrays, despite the use of space-filling curves. On the other hand, the update-positions step reaches the same memory bandwidth as the Stream benchmark (the theoretical peak on 8 cores is 51.2 GB/s). Accordingly, this step cannot be further fastened when using 8 cores.

The memory bandwidth of each step was computed as the number of bytes moved from/to main memory for this step divided by the execution time of this step. The number of bytes moved was counted by hand by inspection of the code, see Listings 4.27–4.30. The first three steps are done once per iteration, so we multiply by the number of iterations (here 100). The sorting step is done every 50 iterations, so we only have $\lfloor (num\_iterations - 1)/50 \rfloor$ (here 1) to count.

In consideration of these comments, it is not straightforward to extrapolate to 8 cores the overall gain results presented in Section 4.3.5. On the sequential implementation, we demonstrated that applying loop fission coupled with the SoA layout was the best choice even if fre-

```
1  thread_id = omp_get_thread_num();
2  offset = thread_id * NB_CORNERS_2D * num_cell_2d;
3
4  #pragma omp for private(i, corner)
5  for (i = 0; i < num_particle; i++) {
6      #pragma omp simd aligned(i_cell, dx, dy, coeffs_x, coeffs_y, signs_x,
           signs_y:VEC_ALIGN)
7      for (corner = 0; corner < NB_CORNERS_2D; corner++) {
8          charge_accu[offset + NB_CORNERS_2D * i_cell[i] + corner] +=
9              (coeffs_x[corner] + signs_x[corner] * dx[i]) *
10             (coeffs_y[corner] + signs_y[corner] * dy[i]);
11     }
12 }
13 // 4 * 8 * ncx * ncy * num_threads * 2 + // double charge_accu[ncx*ncy*
       num_threads*4] (read + write)
14 // 2 * 4 * num_particle +                 // float dx, dy[num_particle] (read)
15 //     4 * num_particle                   // int i_cell[num_particle] (read)
16 // => 12 num_particle + 64 ncx*ncy*num_threads
```

Listing 4.29 – Memory bandwidth (in bytes) of the accumulation step.

Number of cores	1 core	2 cores	4 cores	8 cores
Million particles/s	45.8	89.9	170	266
Million particles/s - ideal	45.8	91.6	183	366

Table 4.8 – Strong scaling on one socket of Curie (Pure `OpenMP`). Test case: 128×128 grid, 50 million particles, 100 iterations simulation (sorting every 50 iterations). Architecture: Sandy Bridge.

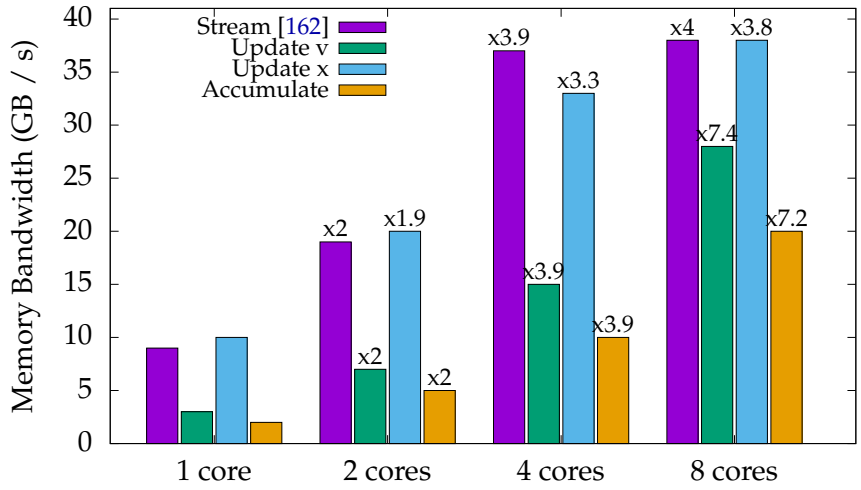


Figure 4.23 – Memory bandwidth on one socket of Curie (Pure `OpenMP`). Test case: 128×128 grid, 50 million particles, 100 iterations simulation (sorting every 50 iterations). Architecture: Sandy Bridge.

```
num_threads = omp_get_num_threads();
thread_id = omp_get_thread_num();

// Count how many particles are in each cell, per thread
for (i = 0; i < num_cell_2d; i++)
    num_particle_per_cell[thread_id][i] = 0;
#pragma omp for schedule(static)
for (i = 0; i < num_particle; i++)
    num_particle_per_cell[thread_id][i_cell[i]]++;

// Prefix sum, by thread
index_next_particle[thread_id][0] = 0;
for (i = 0; i < thread_id; i++)
    index_next_particle[thread_id][0] += num_particle_per_cell[i][0];
for (k = 1; k < num_cell_2d; k++) {
    index_next_particle[thread_id][k] = index_next_particle[thread_id][k - 1];
    for (i = thread_id; i < num_threads; i++)
        index_next_particle[thread_id][k] += num_particle_per_cell[i][k - 1];
    for (i = 0; i < thread_id; i++)
        index_next_particle[thread_id][k] += num_particle_per_cell[i][k];
}

// Update the _tmp arrays
#pragma omp for schedule(static)
for (i = 0; i < num_particle; i++) {
    icx_tmp[index_next_particle[thread_id][i_cell[i]]] = icx[i];
    icy_tmp[index_next_particle[thread_id][i_cell[i]]] = icy[i];
    dx_tmp [index_next_particle[thread_id][i_cell[i]]] = dx [i];
    vx_tmp [index_next_particle[thread_id][i_cell[i]]] = vx [i];
    dy_tmp [index_next_particle[thread_id][i_cell[i]]] = dy [i];
    vy_tmp [index_next_particle[thread_id][i_cell[i]]] = vy [i];
    index_next_particle[thread_id][i_cell[i]]++;
}

// Update i_cell
for (i = 0; i < num_cell_2d; i++) {
    stop_index = index_next_particle[thread_id][i];
    start_index = stop_index - num_particle_per_cell[thread_id][i];
    for (k = start_index; k < stop_index; k++)
    i_cell[k] = i;
}
//      4 * ncx * ncy * num_threads    // unsigned num_particle_per_cell[ncx*ncy*
    num_threads] (write)
//      4 * ncx * ncy * num_threads + // unsigned index_next_particle[ncx*ncy*
    num_threads] (write)
// 2 * 8 * num_particle * 2 +         // double vx, vy[num_particle] (read + write)
// 2 * 4 * num_particle * 2 +         // float dx, dy[num_particle] (read + write)
// 2 * 2 * num_particle * 2 +         // short icx, icy[num_particle] (read + write)
//      4 * num_particle * 3          // int i_cell[num_particle] (2 reads + 1
    write)
// => 68 num_particle + 8 ncx*ncy*num_threads
```

Listing 4.30 – Memory bandwidth (in bytes) of the sorting step.

AoS, 1 loop	AoS, 3 loops	SoA, 1 loop	SoA, 3 loops
30.9 s	22.7 s	23.1 s	18.3 s

Table 4.9 – Time spent in the simulation on 8 threads (Pure `OpenMP`). Test case: 128 × 128 grid, 50 million particles, 100 iterations simulation (sorting every 50 iterations). Architecture: Sandy Bridge.

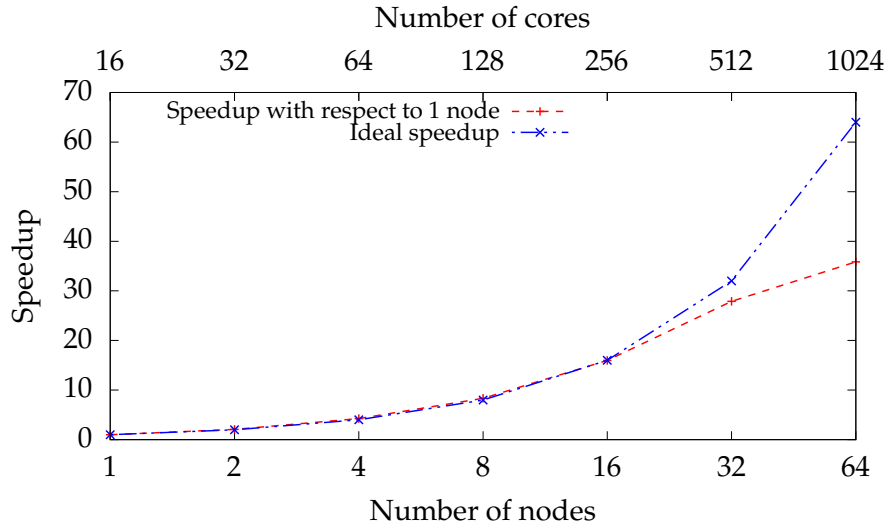


Figure 4.24 – Strong scaling on Curie (Hybrid: `OpenMP` + `MPI`). Test case: 256 × 256 grid, 800 million particles, 100 iterations simulation (sorting every 20 iterations). Architecture: Sandy Bridge.

quent memory movement occurs. In Table 4.9 we show that using the SoA layout with 3 loops is still the best option on 8 cores.

Figure 4.24 shows the strong scaling of the hybrid parallelism when using 800 million particles (maximum memory on one node). The last timing on 1 024 cores is less than 5 seconds. We thus remark that the speedup is far from ideal, when running over 64 nodes (128 processes). This is an expected result: the communication time as percentage of the total time grows with the increasing processes number, while the computation time per process decreases (since the number of particles per process decreases). Thus, in the case of the 64 nodes, only 6.25 million particles are distributed per process and the `MPI` communications take 32% of the total time. Going back to Figure 4.22, when using 400 million particles per process, the same number of processes leads to far better results (constant weak scaling) since communications take in this case only 7% of the total time.

4.5 Further Parallel Optimizations in 2d and 3d

4.5.1 Methodology

In this section we describe different strategies to gain performance in 3d simulations. All the simulations in this section were conducted on the Marconi supercomputer (see Table 4.1) on which we were granted the use of 64 nodes with two 24-cores sockets each. This can be viewed as an extension of Section 4.3 where we performed a similar analysis on a single core. Our new approach is motivated by the differences between the optimization strategies on single core and on multiple cores that share memory, due to the different ratios between computational performance and memory bandwidth of the two configurations. As we wrote in Section 4.4.3, it is not straightforward to extrapolate on the full architecture the results obtained on single

Physical test case	Linear Landau damping [5, Section 5.15], initial distribution $f(x,y,z,v_x,v_y,v_z,t=0) =$ $\left(1+0.01\cos\left(\frac{x}{2}\right)\cos\left(\frac{y}{2}\right)\cos\left(\frac{z}{2}\right)\right)\frac{1}{(2\pi)^{3/2}}\exp\left(-\frac{v_x^2+v_y^2+v_z^2}{2}\right)$
Spatial grid	$[0;4\pi)^3$ decomposed in 64^3 cells, periodic boundaries
Particle shape factor	Cloud-in-cell model [42]
Number of particles	1 billion
Number of iterations	100 (sorting every 10 iterations)
Time step	0.05
Particle crossing: averaged, per iteration	49% of the particles move 1 cell away, 0.0015% of the particles move 2 cells away

Table 4.10 – 3d test case for `OpenMP` optimizations.

core.

In many cases, 3d simulations benefit from the same optimizations as 2d simulations. For example, as in the 2d case, the loop fission improves our 3d implementation. In 3d, it increases efficiency by 8.9%. However, we discuss in Section 4.5.3 an additional optimization that behaves differently in 2d and in 3d.

The results shown in this section were all obtained with the test case presented in Table 4.10. In addition, we also simulated nonlinear Landau damping and two-stream instability test cases. Theoretical results which allow to verify the implementation are available in [5, 23]. Thus, we checked the numerical conservation of the total energy and the numerical evolution in time of the electric field, see details in Section 7.1.

4.5.2 Data Structure and Layout for E and ρ: L6D Curve, Implementation

Section 4.3.3 showed that space-filling curves can optimize the cache performance in 2d. To extend this idea in 3d, we designed a new space-filling curve, the L6D, and we demonstrate that we achieved this aim in 3d also. We thus study in 3d similar strategies for ordering the cells: Row-major order, cf. Figure 4.25; L6D-order, cf. Figure 4.27; Morton-order, cf. Figure 4.26; Hilbert-order, cf. Figure 4.28.

Bijection Algorithms

The aim of this section is to detail the formulas and their implementation needed for the application of space-filling curves in a PIC implementation. When using the cell index plus offset representation for the particles, the update-positions loop can be written as in Listing 4.31.

Section 4.3 explained the correctness of this loop, and a way of removing the computations `* delta_t / delta_{x,y,z}` (lines 2–4). Below, we only focus on the `i_{x,y,z}_from` and `i_cell_from` functions.

In 2d, the mentioned functions are given in Listing 4.32 and were defined in (4.2). They can be set as macros or inline functions. In our code, the constant parameters are part of the macro arguments; they are here omitted to present a shorter code, as it will not affect performance. First, the tile size is given, as global parameter (line 1). Then, depending on the size of the grid, we have a constant which should be set at the beginning of the simulation (line 3): the number of cells per column gives the number of cells that are in a full column of size `ncx * TILE_SIZE`, 64 in Figure 4.5.

The algorithm can be explained as follows: first, we have to know in which column the index is. This is computed as `i_y / TILE_SIZE`. The starting index of this column is thus given by the computation `num_cells_per_column * (i_y / TILE_SIZE)`. Then, in this column of size `ncx * TILE_SIZE`, we recognize the row-major curve, which means we have to add

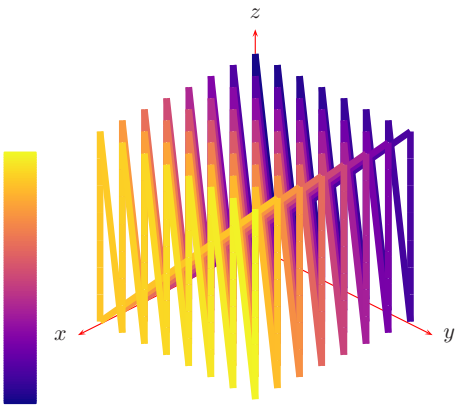


Figure 4.25 – Row major layout of a $8 \times 8 \times 8$ matrix.

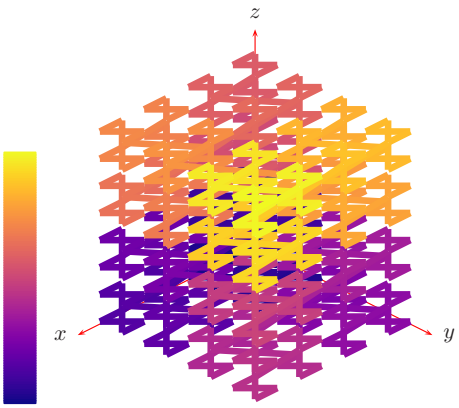


Figure 4.26 – Morton layout of a $8 \times 8 \times 8$ matrix.

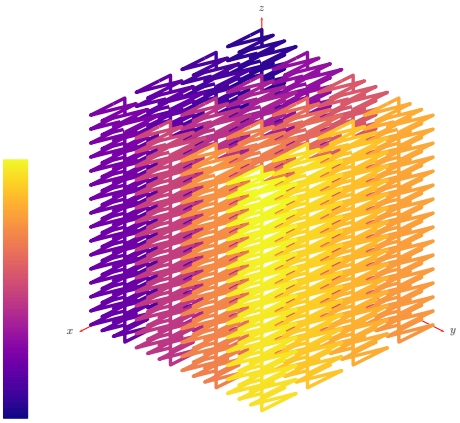


Figure 4.27 – L6D layout of a $16 \times 16 \times 16$ matrix, SIZE=4.

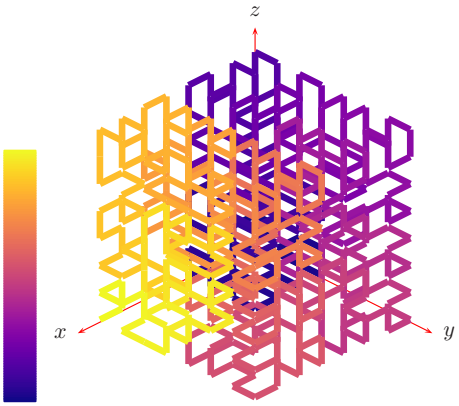


Figure 4.28 – Hilbert layout of a $8 \times 8 \times 8$ matrix.

`i_col * TILE_SIZE + j_col`. On the x-axis, the index is just `i_x`, and on the y-axis, the index is `mod(i_y, TILE_SIZE)`.

In 3d, the mentioned functions are given in equation (4.3) and Listing 4.33. Previous remarks on the 2d functions similarly apply. First, the tile size is given, as global parameter (line 1). Then, depending on the size of the grid, we have constants which should be set at the beginning of the simulation (line 3). The number of cells per tower gives the number of cells that are in a full tower of size `TILE_SIZE * TILE_SIZE * ncz`, 256 in Figure 4.27. The number of cells per wall gives the number of cells that are in a full row of towers. In Figure 4.27, there are 4 towers per wall (there are 4 towers of width `TILE_SIZE` $= 4$ in a row of $ncx = 16$ cells).

The explanation of the algorithm is similar to the 2d case. We have to know in which wall parallel to the (Oxz) plane the index is, then inside this wall in which tower it is, then this is a column-major ordering inside this tower.

$$
\begin{aligned}
(i_x; i_y; i_z) \mapsto i_{\text{cell}} &= ncz \cdot \text{SIZE}^2 \cdot \left(\lfloor i_x/\text{SIZE} \rfloor + \lfloor i_y/\text{SIZE} \rfloor \cdot \lceil ncx/\text{SIZE} \rceil \right) \\
&\quad + i_z \cdot \text{SIZE}^2 + \text{mod}(i_y, \text{SIZE}) \cdot \text{SIZE} + \text{mod}(i_x, \text{SIZE}) \\
i_{\text{cell}} \mapsto \begin{cases}
i_x = \text{mod}(i_{\text{cell}}, \text{SIZE}) + \text{SIZE} \cdot \left\lfloor \text{mod}(i_{\text{cell}}, ncz \cdot \text{SIZE}^2 \cdot \lceil ncx/\text{SIZE} \rceil) / (ncz \cdot \text{SIZE}^2) \right\rfloor \\
i_y = \text{SIZE} \cdot \left\lfloor i_{\text{cell}} / \left(ncz \cdot \text{SIZE}^2 \cdot \lceil ncx/\text{SIZE} \rceil \right) \right\rfloor + \left\lfloor \text{mod}(i_{\text{cell}}, \text{SIZE}^2)/\text{SIZE} \right\rfloor \\
i_z = \left\lfloor \text{mod}(i_{\text{cell}}, ncz \cdot \text{SIZE}^2) / (\text{SIZE}^2) \right\rfloor
\end{cases}
\end{aligned}
\tag{4.3}
$$

```
for (i = 0; i < num_particle; i++) {
    x = i_x_from(i_cell[i]) + dx[i] + vx[i] * delta_t / delta_x;
    y = i_y_from(i_cell[i]) + dy[i] + vy[i] * delta_t / delta_y;
    z = i_y_from(i_cell[i]) + dz[i] + vz[i] * delta_t / delta_z;
    i_x = (int)x - (x < 0.); // floor(x)
    i_y = (int)y - (y < 0.); // floor(y)
    i_z = (int)z - (z < 0.); // floor(z)
    dx[i] = (float)(x - i_x);
    dy[i] = (float)(y - i_y);
    dz[i] = (float)(z - i_z);
    i_cell[i] = i_cell_from((i_x + ncx) % ncx,
                            (i_y + ncy) % ncy,
                            (i_z + ncz) % ncz);
}
```

Listing 4.31 – Sample C code for the update-positions loop. We have to add the grid sizes before applying the % operator because indices might become negative. It assumes that no particle will cross the full grid, see discussion on page 73.

```
#define TILE_SIZE 8 // Depends on architecture.

int num_cells_per_column = ncx * TILE_SIZE;

#define i_cell_from(i_x, i_y) (TILE_SIZE * (i_x) + \
    ((i_y) % TILE_SIZE) + num_cells_per_column * ((i_y) / TILE_SIZE))
#define i_x_from(i_cell) (((i_cell) % num_cells_per_column) / TILE_SIZE)
#define i_y_from(i_cell) (((i_cell) % TILE_SIZE) + \
    TILE_SIZE * ((i_cell) / num_cells_per_column))
```

Listing 4.32 – Efficient C code for L4D bijection functions.

```
#define TILE_SIZE 8 // Depends on architecture.
#define SQR_TILE_SIZE (TILE_SIZE * TILE_SIZE)

int num_cells_per_tower = ncz * SQR_TILE_SIZE;
int num_cells_per_wall  = num_cells_per_tower * \
    ((ncx + TILE_SIZE - 1) / TILE_SIZE); // ceiling(ncx/TILE_SIZE)

#define i_cell_from(i_x, i_y, i_z)                                    \
    (((i_x) / TILE_SIZE) * num_cells_per_tower +                      \
    ((i_y) / TILE_SIZE) * num_cells_per_wall + (i_z) * SQR_TILE_SIZE + \
    ((i_y) % TILE_SIZE) * TILE_SIZE + ((i_x) % TILE_SIZE))
#define i_x_from(i_cell) (((i_cell) % TILE_SIZE) + \
    (((i_cell) % num_cells_per_wall) / num_cells_per_tower) * TILE_SIZE)
#define i_y_from(i_cell) (((i_cell) / num_cells_per_wall) * TILE_SIZE + \
    ((i_cell) % SQR_TILE_SIZE) / TILE_SIZE)
#define i_z_from(i_cell) (((i_cell) % num_cells_per_tower) / SQR_TILE_SIZE)
```

Listing 4.33 – Efficient C code for L6D bijection functions.

	Up. v	Up. x	Acc.	Sort	Tot.
2d standard	59.0	39.8	41.9	28.6	171
Row-major	63.6	39.7	42.8	28.6	177
L4D arr.	57.6	48.2	33.5	41.1	183
Morton arr.	60.2	48.0	29.4	40.7	180
Hilbert arr.	64.9	49.6	30.7	40.5	193
L4D	57.5	40.0	32.0	28.6	161
Morton	59.3	39.8	29.8	28.4	160
Hilbert	59.0	323.7	33.6	28.6	452

Table 4.11 – 2d space-filling curves timings. Time spent in the different loops (in seconds). Test case: Landau damping 2d on a $[0; 4\pi)^2$ grid decomposed in 512^2 cells, 1 billion particles, 100 iterations (sorting every 20 iterations), $\Delta t = 0.1$. "arr." indicates that we use auxiliary arrays instead of recomputing the indices.

	Up. v	Up. x	Acc.	Sort	Tot.
3d standard	126.7	55.3	31.5	21.5	236
Row-major	92.6	55.3	31.5	21.4	202
L6D arr.	92.8	79.0	30.4	29.5	233
Morton arr.	96.5	79.0	30.3	27.5	234
Hilbert arr.	95.3	80.4	31.1	26.9	235
L6D	85.5	55.5	29.9	20.9	193
Morton	89.4	56.7	33.5	19.8	200
Hilbert	87.3	244.4	29.2	20.3	382

Table 4.12 – 3d space-filling curves timings. Time spent in the different loops (in seconds). Test case in Table 4.10. "arr." indicates that we use auxiliary arrays instead of recomputing the indices

Results

We present in Tables 4.11 and 4.12 the performance gains when using the space-filling curves we previously described. Section 4.5.2 only showed results on one core. On modern architectures, there are usually more cores than memory channels: it is thus not straightforward to extrapolate the one core results on the full multi-core architecture; we therefore show here results on the full processor. Moreover, additional arrays to store the indices i_x and i_y were used in Section 4.5.2. We show now that additional gains can be obtained with efficient computations of the bijection functions. In the tables, "arr" means that we use additional arrays to store the indices, otherwise we recompute them.

We focus on three meaningful comparisons in Tables 4.11 and 4.12. The first one is on the data structure: is it beneficial to use the redundant one for the E arrays? The only point where the code changes is in the update-velocities loop. We see in the tables that in 2d, it is detrimental to use it if we stick to the row-major curve, but it is already beneficial with the canonical curve in 3d. We recall that we use here many particles per grid cell, and that when using only a few particles per cell, the redundant data structure is not a good choice, see Section 4.5.4 for a detailed comparison.

The second comparison concerns the data layout. Is it possible to obtain notable gains by changing the ordering of the grid cells? There are two places in the algorithm where the changes might become significant: in the interpolation and in the accumulation. This time, we can answer positively. Yes, by taking another order than the canonical one, we can save time (thanks to a reduction in the cache misses, see Table 4.4). In 2d, the L4D and Morton curves seem to give similar and optimal timings, while in 3d, the L6D curve allows additional gains and seems to be the best one.

The last comparison is on the particle data structure needed for the non-canonical orderings. We can either store the indices in additional arrays (here, arrays of `short int`), or re-compute them at each time step. When storing them, it requires more memory for the particles, therefore we need more time in the update-positions loop and in the sorting step.

Last but not least, we see a surprising timing of the update-positions loop when using the Hilbert ordering without additional arrays. It is due to the fact that the computation (i_x, i_y, i_z) from i_{cell} is expensive and not vectorized. This is thus the only curve for which using additional arrays is profitable. Nevertheless, even with additional arrays, this curve does not improve performances compared to the standard layout, and consequently has to be discarded.

```
1  Foreach subset of k particles in particles
2      Foreach particle in this subset
3          Interpolate E to particle
4          Update the velocity
5      Foreach particle in this subset
6          Update the position
7      Foreach particle in this subset
8          Accumulate particle charge to ρ
```

Figure 4.29 – PIC pseudo-code with strip-mining.

```
1  Foreach particle in particles
2      Interpolate E to particle
3      Update the velocity
4  Foreach subset of k particles in particles
5      Foreach particle in this subset
6          Update the position
7      Foreach particle in this subset
8          Accumulate particle charge to ρ
```

Figure 4.30 – PIC pseudo-code with strip-mining on the two last loops only.

Additional Remarks

As in 2d, we have to choose carefully `TILE_SIZE` depending of the cache sizes. In our tests, a value of `TILE_SIZE` $= 8$ led to the best timings. It can be replaced with other values, as long as they are not too large for the cache.

It should be noted that choosing a value of `TILE_SIZE` that does not divide the grid sizes is possible: then, there will be a few allocated cells that correspond to physical positions outside the boundaries and that will never be accessed.

As a side note, we can remark that if grid sizes are powers of two and if the architecture represents integers with two's complement, we can save some computations on each modulo operation (lines 11–13 in Listing 4.31). For example for the modulo in the x-axis, we can use the variable `int ncx_minus_one = ncx - 1` and then compute `mod(i_x, ncx)` as `i_x & ncx_minus_one`. We showed that this is more efficient than `(i_x + ncx) % ncx` on one core. However, when using the full 24-cores architecture, this small optimization brings no significant gains.

We also note that it is possible to use space-filling curves without using the redundant data structure for E. We do not show here the corresponding results, but the conclusion is that when using a high number of particles per cell (as is our case) the redundant data structure turns out to be the best approach. This is clearly not true when using a low (*e.g.*, less than a hundred) number of particles per cell.

4.5.3 Strip-mining

Even though the loop fission gives satisfactory results, it needs to scan some particle arrays three times, thus putting a lot of pressure on the memory bus. A loop transformation that naturally comes to mind is thus the strip-mining [36, Section 9.8]. Instead of having three loops each scanning all the particles, we split the particle arrays in sub-arrays of size k (where k has to be chosen, depending on the architecture), and have the three loops operate only on k particles. Thus, for each particle, instead of having to fetch its properties in the main memory for each loop, it is now possible to fetch its properties in the cache for the two last loops. This transformation leads to the pseudo-code shown in Figure 4.29, and improves efficiency by 22% in 2d.

We can note that if we choose $k = 1$, we are back to the base pseudo-code, which was not optimal. If we choose $k = N$, we are back to the previous pseudo-code with loop fission. In our 2d experiments, choosing a strip-size k between 64 and 256 gives similar optimal results.

Unfortunately, in 3d this strip-mining does not improve performances. This is explained by the fact that the cache is filled with too many E values, thus the expected gain in performance coming from the cache reuse of the particle arrays is out of reach. Thus, in 3d, to be able to efficiently reuse the particle data, the strip-mining has to be done only on the two last loops. This transformation leads to the pseudo-code shown in Figure 4.30, and improves efficiency by 12%.

	Time (s)	Gains	Accumulated gains
Baseline	258.7	0.0%	0.0%
+ Loop Fission	235.8	8.9%	8.9%
+ Space-filling curves (L6D)	192.9	18.2%	25.4%
+ Strip-mining	169.5	12.1%	34.5%

Table 4.13 – Gains of different optimizations in our implementation. Total execution time, gains and accumulated gains. Test case in Table 4.10.

In our 3d experiments, choosing a strip-size k between 32 and 128 gives similar optimal results.

This strip-mining technique was more or less already used in VPIC [47], which advanced 4 particles at a time for vectorization ($k = 4$ with our notations). However, in VPIC, an additional assumption was made — none of those 4 particles should cross cell boundaries — and scalar code was generated for the particles that crossed cell boundaries. In our implementation, the update-positions loop is vectorized without exceptions.

4.5.4 Overall Gains and Comparisons

The optimizations presented in this section are summarized in Table 4.13. In this table, the baseline is a version of the code with the standard 3d data structure for E, the redundant one for ρ, and an optimization applied that, in particular, removes the `* delta_t / delta_{x,y,z}` computations (lines 2–4 in Listing 4.31), see Section 4.3.2. The gains (in %) are computed with respect to the previous line of the table and the accumulated gains are computed with respect to the baseline.

Review of different optimizations

- Loop fission: better memory management for E and ρ, allows to vectorize the update-positions loop.
- Redundant arrays for E together with appropriate use of space-filling curves: less cache misses in the interpolation and in the accumulation. We note that the redundant arrays are only useful when using at least a hundred particles per grid cell, see Table 4.14.
- Strip-mining: allows to reuse particle data between loops.

Overall, these optimizations result in 590 million particles processed per second, on 24 cores on Intel Skylake architecture, without hyper-threading (25 million particles per second per core), or 1.48 ns per particle per time step (35.47 ns per particle per time step per core).

Those performances are compared in Table 4.14 to another recent PIC implementation [61], solving the same equations with the same precision (both implementations use double precision and first order interpolations). We ran simulations with the parameters chosen in this paper, which differ from our previous ones. Simulations presented in this paper were conducted on the Piz Daint supercomputer consisting of 8 Sandy-Bridge cores @2.6 GHz with a theoretical memory bandwidth of 51.2 GB/s (or 6.4 GB/s/core), and we recall that we used the Marconi supercomputer consisting of 24 Skylake cores @2.1 GHz with a theoretical memory bandwidth of 127.99 GB/s (or 5.3 GB/s/core). PIC implementations being memory bound, comparing performances per core on those two architectures makes sense.

In the simulations of that paper, there are only 7.6 particles per cell (when using 1 million particles), and 122 particles per cell (when using 16 million particles). The redundant data structure for E is a good choice only if there are a lot of particles per cell (in the present work, we use 3 815 particles per cell for our test case in Table 4.10). Of course, when there are roughly as many particles as grid points, multiplying by 8 the data for E almost doubles the memory transfers. With such a low number of particles, using a redundant data structure for E is

Implementation \ Nb. particles	10^6	$16 \cdot 10^6$	10^9
Jocksch *et al.* work [61] (8 cores, 6.4 GB/s/core)	153.9	115.8	-
Present work (24 cores, 5.3 GB/s/core)	854.3	93.07	36.87

Table 4.14 – Comparison with another implementation. Time spent per particle per iteration per core, in nanoseconds. Test case [61]: Grid of $512 \times 256 \times 1$ cells. Initial particle distribution uniform in space ($[0;512) \times [0;256) \times [0;1)$) and velocity ($[-1;1)^3$).

detrimental, and we would have to make another study for such a configuration. But with 122 particles per cell, the data layouts and code transformations presented in this section become useful. At the scale of 1 billion particles, our implementation needs only 36.87 ns per particle per iteration per core (no data for this particle number in [61] to compare).

4.5.5 Parallel Results on Marconi A3

In 2d, we used particle decomposition to parallelize our implementation on distributed memory, see Section 2.6, and showed that this kind of parallelism is "good enough". Even though the scalability is harmed by a logarithmic `MPI_ALLREDUCE` step, we showed that this overhead remained fairly limited in practice, see Figure 4.22. In 3d, we also used particle decomposition and the same applies: when using a large number of processes, the communication becomes too costly. But in practice, if we use the full memory on each `MPI` process to put particles, this is not the main drawback. The main bottleneck of this approach in 3d is that in realistic simulations (using a grid bigger than our $64 \times 64 \times 64$ grid), there are so many cells that the computations on one process will be inefficient, due to the high number of cache misses involved. This behavior is quite common for example in matrix computations, where you can see super-scaling behaviors on large matrices, due to the large reduction of cache misses when computing only on sub-blocks. This is the reason why the technique we propose in this work should be seen as only a small brick inside a more complex scheme: one should probably use domain decomposition on top of the efficient `OpenMP` algorithm we provide.

We next show a strong scaling on up to 24 cores of two versions of our implementation. Figure 4.31 shows a strong scaling of the implementation with loop fission (pseudo-code given in Figure 4.2) and Figure 4.32 show a strong scaling of our implementation with strip-mining (pseudo-code given in Figure 4.30).

Figure 4.33 shows both implementations on a single graph, for comparison. On 1 thread, the loop fission implementation is 6.7% faster, but on 24 threads the strip-mining implementation is 12% faster.

These figures illustrate the importance of having an efficient implementation on one core, but also the importance of precise performance analysis to enhance multi-core efficiency. On the comparison graph, we see that up to 8 cores, the implementation with 3 loops performs better, then the implementation with strip-mining performs better. To understand why, we can look at the two other figures, where we show the memory bandwidth of our implementation, compared to that of the triad test in the Stream benchmark [162]. On one hand, these histograms underline that the update-velocities, accumulation and sorting steps are far to reach the peak memory bandwidth and thus, they have a good scaling up to 24 threads. On the other hand, the update-positions step reaches the same memory bandwidth as the Stream benchmark (the theoretical peak on 24 threads is 127.99 GB/s). Accordingly, this step cannot be further accelerated when using 24 threads.

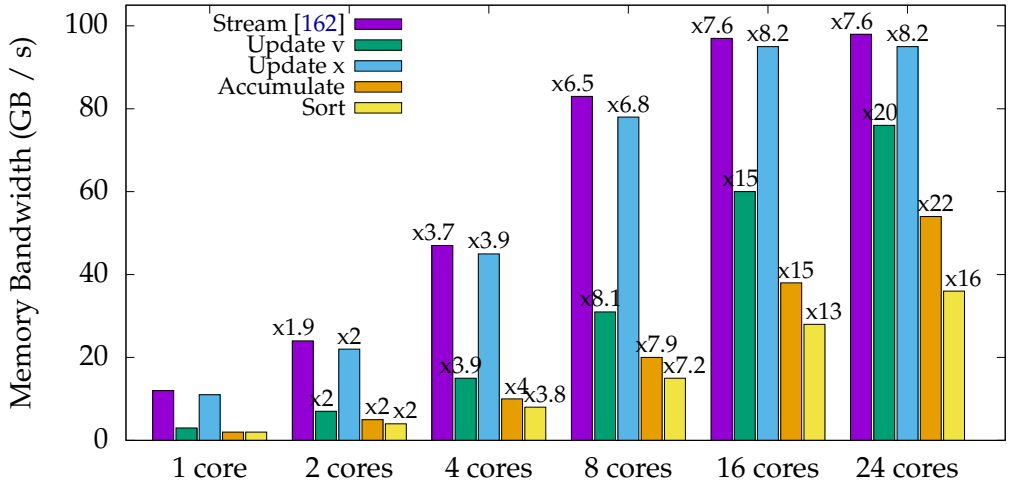


Figure 4.31 – Memory bandwidth with loop-fission (3 loops). Test case in Table 4.10.

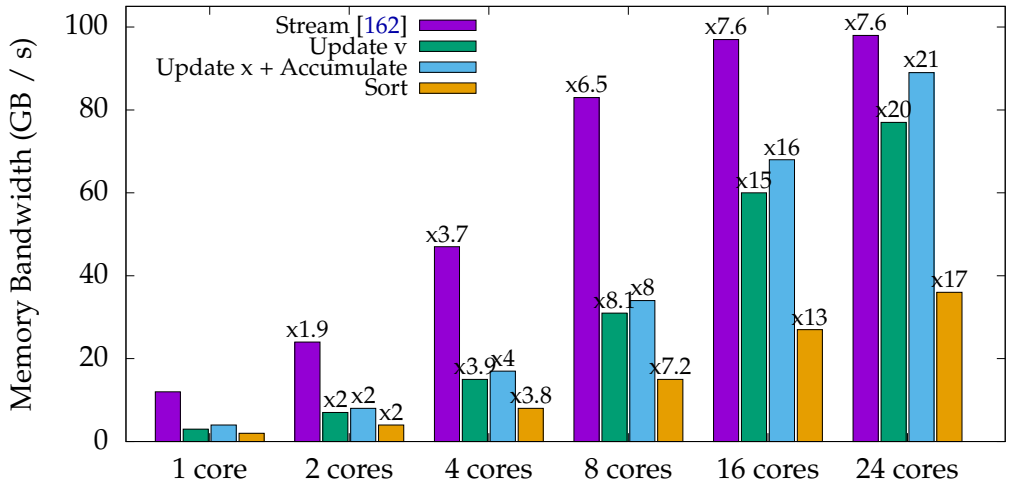


Figure 4.32 – Memory bandwidth with strip-mining (2 loops). Test case in Table 4.10.

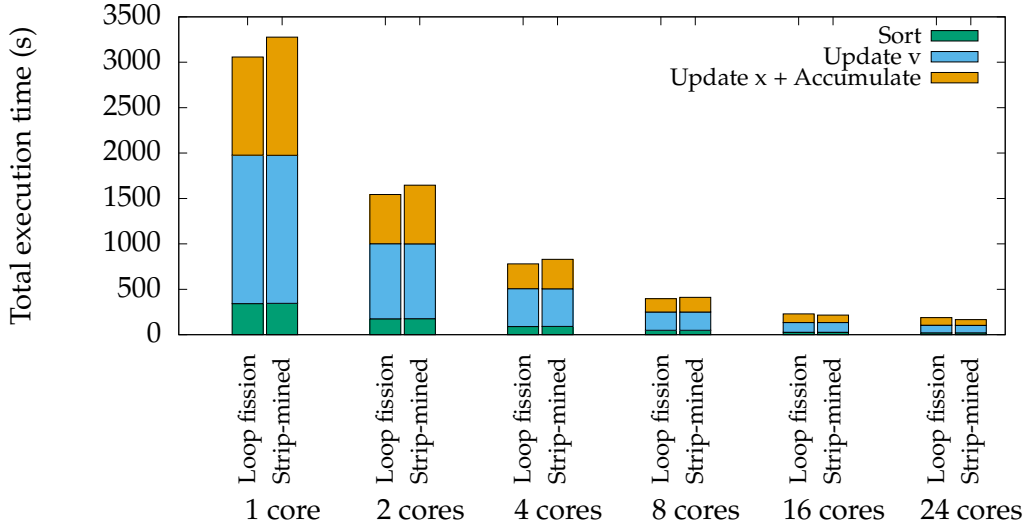


Figure 4.33 – Loop fission (3 loops) VS Strip-mining (2 loops). Test case in Table 4.10.

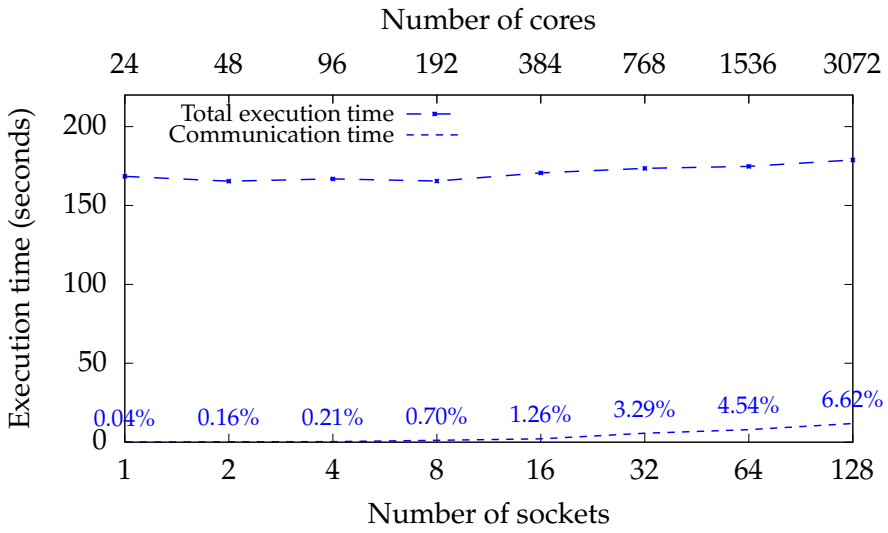


Figure 4.34 – Weak scaling on Marconi. Test case: $64 \times 64 \times 64$ grid, 1 billion particles per socket, 100 iterations simulation (sorting every 10 iterations). Architecture: Skylake. Communication time is also shown as percentage of the execution time.

Can You See it at a Glance?

> In order to convince ourselves of the presence or of the quality of an object, we like to see and to touch it. [...] We prefer, of course, a short and intuitive argument to a long and heavy one: *Can you see it at a glance?*
>
> G. Pólya [31, Part I, Section 13]

The strip-mining idea should be natural when looking at Figure 4.31: when the update-positions loop cannot be further sped up by more cores, due to memory bandwidth limits, merging it with another loop becomes a good idea. To merge two loops, we can either "undo" part of our loop fission, by applying loop fusion [36, Section 9.2]; or we can use strip-mining. To preserve the high efficiency of this vectorized loop, it is better to use strip-mining rather than loop fusion, which would break the vectorization opportunity.

Last but not least, this transformation is easy to implement, compare Listing 4.34 and Listing 4.35.

4.5.6 Weak scaling on 3 072 cores with `MPI`

Our parallel results come from simulations executed on the supercomputer Marconi. Each node has 2 sockets of 24 cores each, hence we used one `MPI` process per socket and 24 threads per process.

Figure 4.34 shows a weak scaling from 1 core to 3 072 cores (64 nodes). These simulations run with 1 billion particles per socket in order to use the full memory. We can see that up to 3 072 cores, the overhead due to `MPI_ALLREDUCE` stays acceptable, thanks to the hybrid parallelism `OpenMP` + `MPI`.

```
#pragma omp for [...]
for (size_t i = 0; i < num_particle; i++) {
    [...] // Update v
}
#pragma omp for simd [...]
for (size_t i = 0; i < num_particle; i++) {
    [...] // Update x
}
#pragma omp for [...]
for (size_t i = 0; i < num_particle; i++) {
    [...] // Accumulate
}
```

Listing 4.34 – 3d code with 3 loops.

```
// STRIP_SIZE is an architecture-dependent parameter (depends on cache size).
#define STRIP_SIZE 32

#pragma omp for [...]
for (size_t i = 0; i < num_particle; i++) {
    [...] // Update v
}
#pragma omp for [...]
for (size_t big_i = 0; big_i < num_particle; big_i += STRIP_SIZE) {
    #pragma omp simd [...]
    for (size_t i = big_i; i < min(num_particle, big_i + STRIP_SIZE); i++) {
        [...] // Update x
    }
    for (size_t i = big_i; i < min(num_particle, big_i + STRIP_SIZE); i++) {
        [...] // Accumulate
    }
}
```

Listing 4.35 – 3d code with a strip-mining on the two last steps.

Chapter 5

Pic-Vert in 2d or 3d with Chunks

The main difference between `Pic-Vert` and other PIC implementations is the use of a specialized data structure: the chunk bags. We introduced this idea in two articles [202, 205]. In this chapter, we discuss in detail the advantages of this data structure for a PIC algorithm. In a nutshell:

Chunk Bags

- are a dynamic data structure with concurrent accesses (defined in Listings 5.1 and 5.4);
- allow to keep particles sorted at all times (the so-called *strict-binning* approach), which minimizes the cache misses, allows vectorization of core loops and reduces the memory footprint of a particle;
- let the strict binning approach efficiently handle fast-moving particles.

Section 5.1 first describes the architectures on which this data structure was tested. Section 5.2 then describes the state-of-the-art in details, and Section 5.3 explains what are the limitations from the state-of-the-art that we are able to remove with chunks, and how. Section 5.4 discusses our first results with this data structure in 2d, Section 5.5 our next results with this data structure in 3d.

5.1 Test Architectures

The results presented in this chapter come from simulations run on different computers.

Our Inria team machine "icps-gc-6". This machine features 2 sockets, and each of those sockets is an Intel Xeon E5-2650 v3 @2.3 GHz (Haswell) with 16 GB of RAM, 2 memory channels, and 10 cores. Its theoretical memory bandwidth peak is 34 GB/s (only 2 memory channels installed on a maximum of 4[1]), its theoretical single precision floating-point operation peak is 736 GFlops/s. On this machine, we had access to `gcc 6.2` and `icc 17.0.0`[2].

The A1 partition of the CINECA supercomputer "Marconi"[3] (1 512 nodes). Each node features 2 sockets, and each of those sockets is an Intel Xeon E5-2697 v4 @ 2.3 GHz (Broadwell) with 64 GB of RAM, 4 memory channels, and 18 cores. Its theoretical memory bandwidth peak is 76.8 GB/s, its theoretical single precision floating-point operation peak is 1 325 GFlops/s. On this machine, we had access to `icc 17.0.1`.

The A3 partition of the CINECA supercomputer "Marconi"[3] (2 304 nodes). Each node features 2 sockets, and each of those sockets is an Intel Xeon Platinum 8160 @ 2.1 GHz (Skylake) with 96 GB of RAM, 6 memory channels, and 24 cores. Its theoretical memory bandwidth peak

[1] http://ark.intel.com/products/81705
[2] Thanks to https://software.intel.com/en-us/qualify-for-free-software/student
[3] https://www.cineca.it/en/content/marconi

	icps-gc-6	Marconi A1	Marconi A3
Processor	Intel Xeon E5-2650 v3 (Haswell)	Intel Xeon E5-2697 v4 (Broadwell)	Intel Xeon Platinum 8160 (Skylake)
RAM	16 GB	64 GB	96 GB
#mem. channels	2	4	6
Memory bandwidth	34 GB/s	76.8 GB/s	127.99 GB/s
#cores	10	18	24
Clock frequency	2.3 GHz	2.3 GHz	2.1 GHz
Floating-point	736 GFlops/s	1 325 GFlops/s	1 612 GFlops/s

Table 5.1 – Architectural parameters of one socket of our test machines.

is 127.99 GB/s, its theoretical single precision floating-point operation peak is 1 612 GFlops/s. On this machine, we had access to `icc 17.0.4`.

Table 5.1 summarizes the architectural parameters of those machines. Because PIC implementations are memory-bound, the parameter that matters most is the memory bandwidth.

5.2 Related Work and Motivation

A recent paper [83] studied GTC-P performance in details, and points out that: "*metrics such as flop/s or percentage-of-peak are less relevant for the predominantly memory-bound gyrokinetic PIC methods, as modern architectures require 10 flops per byte moved from DRAM in order to be compute-limited.*" The authors then present a model able to predict execution time based on the amount of data transfer performed. Data transfers include both inter-node network communication (15% to 30% of the execution time) and intra-node loads and stores on shared memory (60% to 80% of the execution time). Intra-node transfers are decomposed between in-cache accesses, contiguous accesses, and random accesses — the latter being the most costly. In other words, memory bandwidth is a limiting factor. This study shows that, to improve the performance of multi-core (intra-node) processing in PIC simulations, we must decrease the amount of costly memory accesses. Of course, we must do so by preserving the `OpenMP`-level parallelism as well as the crucial use of SIMD instructions.

One central aspect in the design of a PIC implementation is how the particles are stored in the shared memory, and how the particles are assigned to the various threads acting over this shared memory. In the remainder of this chapter, we will assume that every algorithm uses the "index plus offset" representation [47, III.E.], see Section 2.2.1.

We organize the following discussion by focusing on three main criteria: strict or non-strict binning, the representation of particles, and the treatment of data races arising when two threads push data onto a same target cell.

A common approach, which was the focus of Chapter 4, consists of storing the particles in a static array, either in an Array of Structures (AoS) fashion or in a Structure of Arrays (SoA) fashion. Prior work has investigated the benefits of sorting this array by cells, to improve locality when accessing the electric field and charge arrays [54, 43]. Sorting may be performed either in between every iteration to maximize locality, or only every so many iterations, so as to tame the overheads of the sorting process by leveraging the fact that not all particles move to arbitrary other cells (*e.g.*, [47, 59]). Depending on the option, performance suffers either from numerous costly random accesses or from suboptimal locality even when efficient, specialized sorting algorithms are used. As already remarked in the previous chapter, the best frequency for sorting is not so easy to select: it is both architecture-dependent (due to the relative benefits of locality) and input-dependent (particles move faster in a "hot" plasma): it is thus also a good option to choose the frequency of the sorting dynamically [70].

Rather than sorting cell by cell, other algorithms rely on *coarse-grained binning* [49, 83, 61,

	Sort	Update v	Update x	Accumulate	Total
Do not sort	0.0	98.0	64.6	35.9	199.0
Sort every 100	3.6	78.3	64.4	25.6	177.0
Always sort	209.0	66.3	64.2	13.4	353.0

Table 5.2 – Time spent in the different loops (in seconds). Test case: 200 million particles, 128×128 grid, $\Delta t = 0.1$, 500 iterations. Architecture: Marconi A1.

81, 87]. Particles are organized in super-cells (of size, *e.g.*, $10 \times 10 \times 10$). This coarse-grained binning greatly reduces the number of cache misses for the electric field and charge arrays on the L3 cache level, but still incurs a lot of misses on the L1 and L2 levels since particles are not sorted by cell inside a super-cell. Various data structures may be used to store particles. For example, PIConGPU [49] use *attribute tiles* to store particles on GPU, which are doubly linked lists of fixed-capacity arrays. This data structure is analogous to our chunks but unlike in our work, particles in an attribute tile are processed in place. If a particle moves to a different super-cell, it is migrated to a transfer buffer, and its slot is marked as a *hole* in an auxiliary bitmap. Subsequently, holes are filled with particles incoming from neighboring super-cells. Remaining holes, if any, are filled using particles taken from the end of the attribute tile. In GTC-P [83], particles that change super-cell are also first updated in place, then migrated to a different super-cell. In ORB5 [61], particles are sorted by super-cell at each iteration. These implementations thus also double the memory traffic associated with particles that cross super-cell boundaries. In contrast, in our algorithm, particles are directly moved to their target bin — they get moved exactly once per time step.

Going further in terms of sorting, one may try to keep the particles sorted by cell at all times. In other words, instead of storing particles directly in an array, one stores the particles in nbCells distinct sets of particles. This approach known as *strict binning* has three main benefits. Two of them are shown in Table 5.2: locality is exploited at its best, and it enables the vectorization of the update-velocities step. A last benefit is that we save 4 bytes per particle as there is no need to store the cell index. While this approach overcomes several of the aforementioned limitations, it is nontrivial to implement efficiently: as the table shows, the naive idea to apply a sorting at each time step is too costly.

> “To be, or not to be, that is the question:
> Whether ’tis nobler in the mind to suffer
> The slings and arrows of outrageous fortune,
> Or to take Arms against a Sea of troubles,
> And by opposing end them: to die, to sleep
> No more;”
>
> W. Shakespeare [201, Act III]

To sort, or not to sort, that is thus the question; whether ’tis better in efficiency to suffer from outrageous cache misses, or to take arms against the challenge of the representation of sets of particles: to write code, to sleep no more. The challenge here lies in the fact that the size of these sets may vary dynamically as the particles move across the grid.

A first approach is to “hope” that the distribution of particles does not become very unbalanced, at least no more than by some constant factor (*e.g.*, 2). Under this assumption, one may represent each set as a fixed-size array. The resulting representation is an Array of Arrays of Structures [85, 53]. The arrays have their size fixed at the beginning of the simulation. If, at some point in the simulation, the number of particles in a given cell exceeds this size, an error is triggered and the simulation must be interrupted. This approach is thus not very robust.

Efficient, general-purpose implementations of strict binning must cope with dynamically-sized bins. In the Particle-Particle/Particle-Mesh algorithm [23, Section 8.4.], each set is repre-

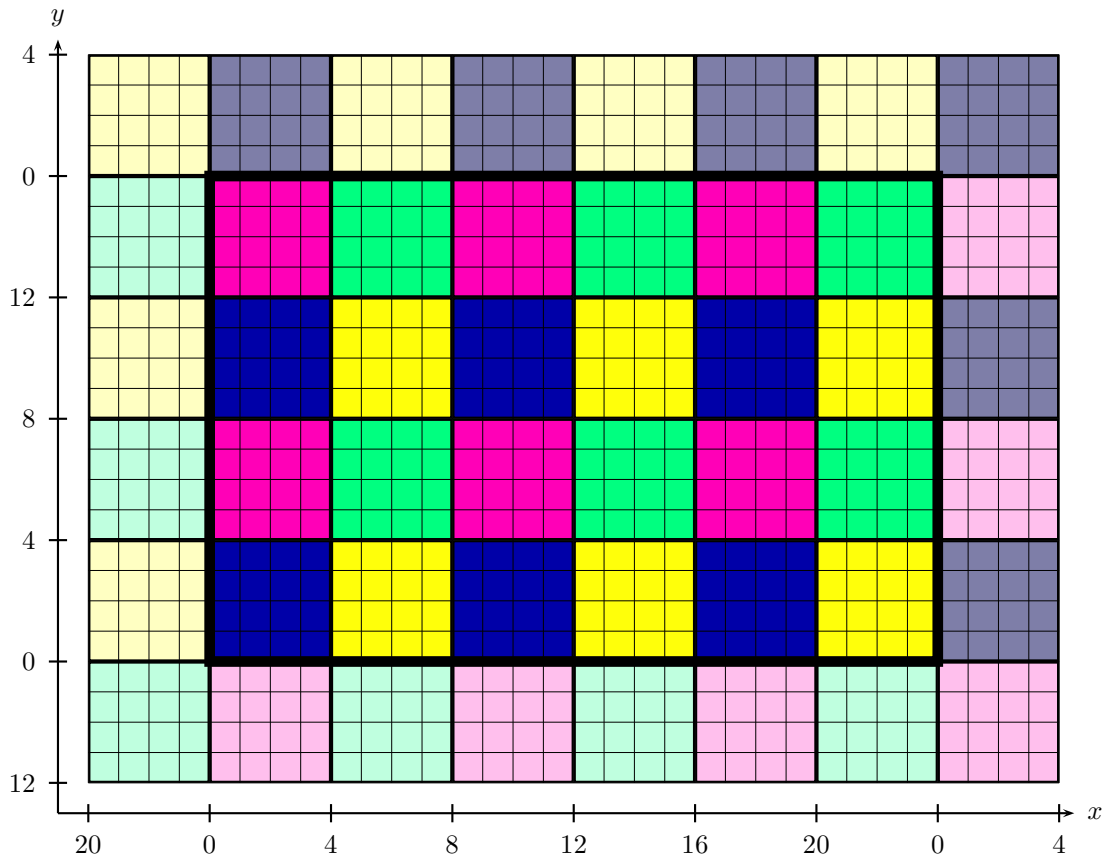


Figure 5.1 – A coloring with 4 colors with 4×4 tiles of a 24×16 grid with periodic boundaries.

sented as a linked list. Yet, this data structure lead to tremendous overheads both in terms of space (to represent list cells) and time (to follow indirections). A textbook data structure for resizable arrays is the vector. However, vectors suffer from a prohibitive 2x space overhead and involve costly resize operations. Despite their $O(1)$ amortized cost, those resize operations induce a significant slowdown in practice. We rewrote our `C` code to make it compatible with `C++` and compared our chunks to the standard `std::vector` from `C++`. Using `std::vector` in simulations with an average of 2 288 particles per vector incurred a 50% slowdown compared to our chunks (without concurrent accesses). Furthermore, we are not aware of a concurrent vector implementation that would be more efficient that the one we provide for our chunks.

Other researchers have investigated more elaborate dynamic set data structures, combining arrays with trees, such as the Packed Memory Arrays (PMA) [135, 144]. This structure consists of a big array containing a fraction of unused cells, called *gaps* or *holes*, and that supports dynamic rebalancing of these gaps. Yet, dealing with the gaps and rebalancing them increases the number of memory operations, resulting in poorer performance. Furthermore, the parallelization scheme proposed for PMA [12, Chapter 5] incurs additional overheads, as the structure then needs to be scanned twice.

Nakashima *et al.* [72] propose a data structure that we view as a parallelism-friendly version of PMAs. To tame the frequency of rebalancing operations, the authors introduce thread-local *overflow buffers*. Yet, this approach suffers from two important limitations. First, as particles move, maintaining the variable-size gaps requires costly operations for shifting particles. Second, the algorithm, which uses a coloring scheme [65] — see Figures 5.1–5.3 — to avoid data races, does not handle *fast-moving particles* well (particles moving more than a couple cells away at a given time step): it resorts to sequential processing for these particles[4].

These two limitations are exacerbated when the percentage of *crossing particles* (particles

[4]The percentage of fast-moving particles heavily depends on the simulation.

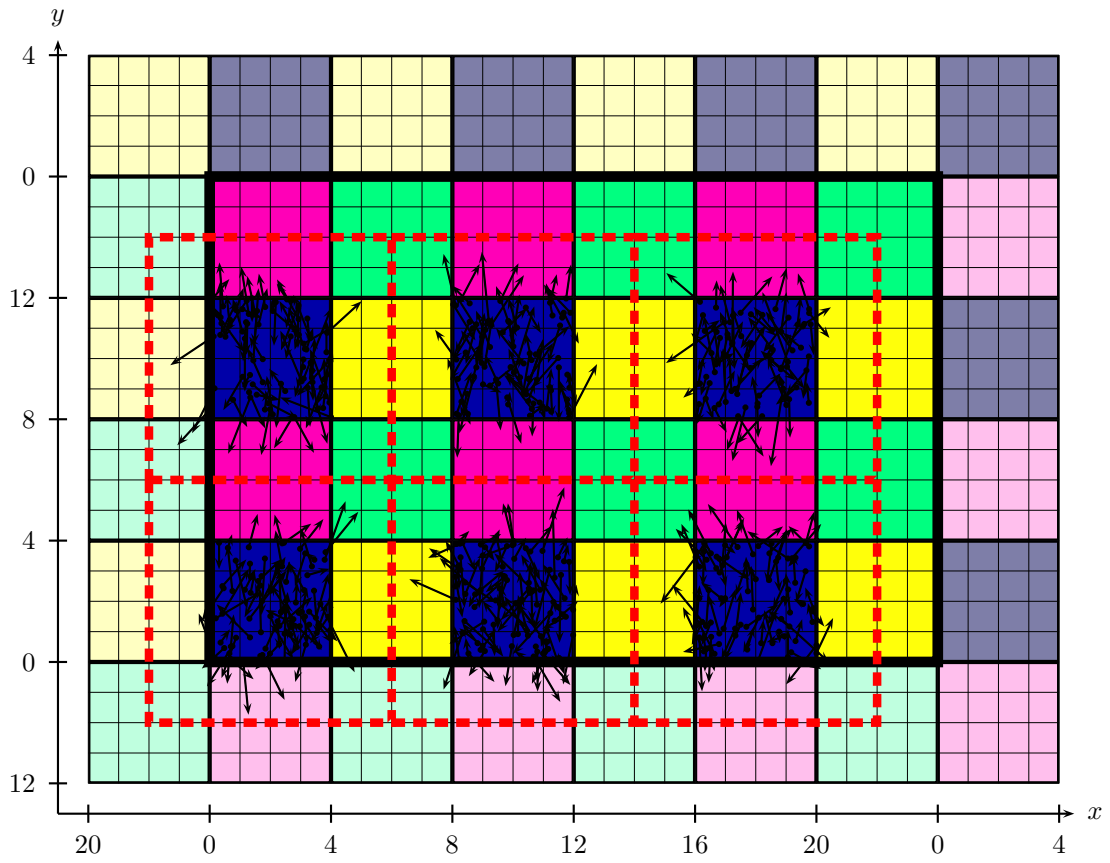


Figure 5.2 – Tiles of a same color are processed in parallel. If particles stay in their initial tile or move at most 2 cells from it, no data race can occur when particles move to their new cell.

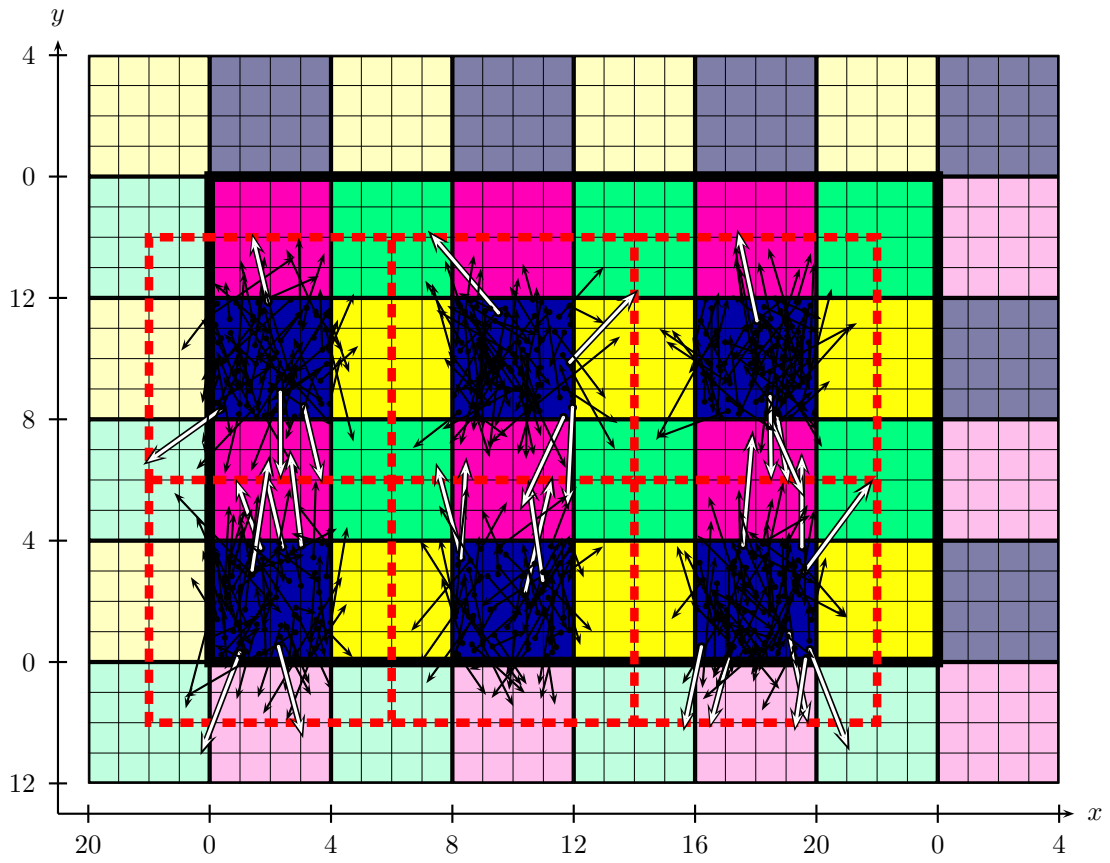


Figure 5.3 – Special care is required for particles that move further away (drawn in white): two distinct threads might process particles that move in the same cell, and data races can occur.

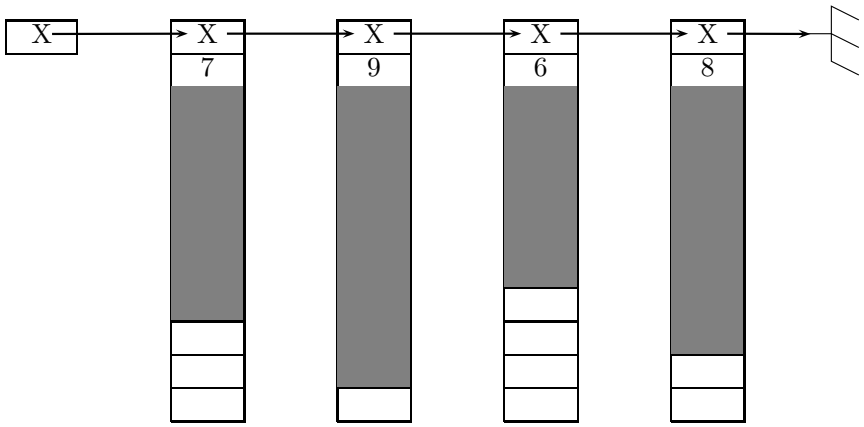


Figure 5.4 – Chunk bag data structure: chunks of size 10, particles stored in grey cells.

changing cells at each time step) increases, to the point of possibly becoming a major bottleneck. For example, in a parallel execution using 64 threads, having as little as 0.5% of fast-moving particles can result in a 32% slowdown on the total execution time due to the sequential processing of these particles[5]. In contrast, we are able to integrate the processing of these particles within the main parallel loop.

5.3 Overview

We propose an algorithm implementing strict-binning for the PIC method that addresses the limitations mentioned in Section 5.2, while still supporting efficient `OpenMP`/SIMD parallelization of all critical loops. Our algorithm leverages the use of *chunk bags*, i.e. linked lists of fixed-capacity arrays, to achieve SIMD-friendly storage of particles with limited memory overheads. The idea of this data structure is illustrated in Figure 5.4. Theoretical analysis of this data structure and comparison with similar ones can be found in [128].

Section 5.4 and Section 5.5 will show two possible implementations of those chunks. In this chapter, we will denote chunkSize the number of particles that can be stored in one chunk. We will also denote memoryOf(object) the amount of memory, in bytes, required to store one particular object[6]. In particular, we will see that the memory footprint for a chunk is, in both implementations:

$$\mathsf{memoryOf(chunk)} = 64 + \mathsf{chunkSize} \cdot \mathsf{memoryOf(particle)}$$

For efficiency, chunkSize should enable efficient vectorization (chunkSize is a multiple of 16 for 512-bit registers), and at the same time be large enough to tame the cost of following a pointer from a chunk to the next (*e.g.*, 128 or 256).

This chapter presents different variants of a novel parallel algorithm for PIC simulations on multi-core architectures, featuring at the same time: asymptotically-optimal memory consumption, minimal bandwidth usage, competitive constant factors on the execution time, and excellent scalability.

This algorithm minimizes the amount of memory transfers in the following sense: at each time step, each particle gets read from and written to memory exactly once. In particular, no further move or reordering is ever required, regardless of the percentage of fast-moving particles.

[5] Let t denote the single-thread execution time. Assume 0.5% of sequential execution, and 99.5% using 64 threads. The parallel execution time is: $0.005t + 0.995t/64 = 1.32t/64$.

[6] It is the equivalent of the `sizeof` operator in `C`, but we wish to avoid confusion between the memory footprint of a chunk (memoryOf(chunk)) and the number of particles that can be stored in a chunk (chunkSize).

The memory consumption is asymptotically-optimal in the sense that the particle representation is minimal, and that in addition to the minimal amount of space required for storing the particles, the space overhead is constant for a fixed grid (and a fixed hardware for the first variant). In particular, the space usage does not depend on the number of crossing particles.

> “ Le Poète est semblable au prince des nuées
> Qui hante la tempête et se rit de l’archer ;
> Exilé sur le sol au milieu des huées,
> Ses ailes de géant l’empêchent de marcher.[7]
>
> C. Baudelaire [193, L’albatros] ”

Our algorithm enables fast rides in the sky of simulations that need a lot of particles. But, stranded on the ground of simulations that use only a few particles, its large chunks prevent it to be efficient. We must therefore acknowledge that our algorithm is efficient when the average number of particles per cell exceeds a couple hundreds. Although laser-driven particle acceleration simulations can use as few as 30 particles per cell [49], large-scale, high-precision simulations may involve hundreds to thousands of particles per cell [85, 47, 59].

Chunk bags can be used privately by one thread, but also support atomic push operations. This atomic operation reserves a particle slot in a chunk in a thread-safe manner and is used in practice to handle fast-moving particles within the main parallel loop.

The four variants (hereafter denoted Variant 1, Variant 2, Variant 3 and Variant 4) are summarized in Table 5.3:

- Variant 1, explained in Section 5.4, removes the data races by allocating a chunk bag for each thread, on each cell.
- Variant 2, explained in Section 5.5, removes most of the data races by using a coloring scheme. It handles the grid cells with a cubic tiling: as long as particles do not move further than half a tile away at each time step, no data race is involved. The remaining data races, associated with fast-moving particles, are handled by allocating one chunk bag per cell on which threads write with an atomic operation.
- Variant 3, explained in Section 5.5.4, also uses cubic tiles but does not use a coloring scheme. Instead, it allocates 2^d private bags per cell “close” to the borders of each tile — where d is the spatial dimension —, to cope with the fact that multiple threads might handle neighboring tiles concurrently. The remaining data races, associated with particles moving “far” away from a tile, are handled by allocating one chunk bag per cell on which threads write with an atomic operation. The greater number of chunk bags to handle involves more indirections and is, in our tests, not better than the previous variant even if it exhibits 2^d more parallelism.
- Variant 4 uses neither tiling nor a coloring scheme, but instead an atomic operation to update all the particles. It is the variant that allocates the least additional memory, but it is too naive and incurs too much time overhead because of atomic operations, therefore it will not be further discussed.

These different variants illustrate possible trade-offs between the number of atomic operations involved, the memory involved, and the amount of parallelism available. In fact, this trade-off can be customized at will: more private bags requires more memory but fewer atomic operations.

[7] The Poet, like this monarch of the clouds,
Despising archers, rides the storm elate.
But, stranded on the earth to jeering crowds,
The great wings of the giant baulk his gait. C. Baudelaire (translation by Roy Campbell)

Variant	1	2	3	4
Maximum number of parallel tasks	nbCells	$\frac{1}{2^d} \cdot \frac{\text{nbCells}}{\text{tileSize}^d}$	$\frac{\text{nbCells}}{\text{tileSize}^d}$	nbCells
Additional memory needed	C_1	C_2	C_3	C_4
Number of synchronization points / time step	3	$2 + 2^d$	3	3
Use of atomics	Never	For particles that move further than $\frac{1}{2} \cdot$ tileSize away from their tile	For particles that move further than borderSize[8] away from their tile	Always

Table 5.3 – The four variants of our strict-binning algorithm. In the table, d is the spatial dimension, $C_1 \approx 2 \cdot \text{nbThreads} \cdot \text{nbCells} \cdot \text{memoryOf(chunk)}$, $C_2 \approx 4 \cdot \text{nbCells} \cdot \text{memoryOf(chunk)}$, $C_3 \approx 2 \cdot \left(\frac{(\text{tileSize} + 2 \cdot \text{borderSize})^d}{\text{tileSize}^d} + 1 \right) \cdot \text{nbCells} \cdot \text{memoryOf(chunk)}$, and the last constant $C_4 \approx 2 \cdot \text{nbCells} \cdot \text{memoryOf(chunk)}$ — note that $C_4 < C_2 < C_3 < C_1$.

```
1 struct { float dx, dy; double vx, vy; } particle_2d;
2 struct chunk { struct chunk* next; int size;
3                particle_2d array[CHUNK_SIZE]; } chunk;
4 struct { chunk* front, back; particle* back_end, back_head; } bag;
```

Listing 5.1 – Chunk bag of AoS data structure, in 2d.

5.4 Variant 1 of our Strict-Binning Algorithm

This section will explain the details of the first variant of our strict-binning algorithm. Pseudo-code and results are given in 2d, but can be modified in other dimensions.

5.4.1 The Chunk Bag Data Structure (AoS inside)

Our approach is based on a realization of the sets of particles using a data structure, which we here refer to as *chunk bag*. This data structure is an optimized variant of a relatively standard structure for representing extensible sequences. A chunk bag essentially consists of a linked list of fixed-capacity arrays, called *chunks*. Each chunk stores a pointer to the next chunk (possibly a null pointer), a fixed-capacity array of particles, and a size field. Each bag stores pointers to its first chunk and to its last chunk from that linked list.

As an optimization, a bag also keeps pointers to the next available location in the array of the back chunk, and to the location one past the last location inside the back chunk. These auxiliary pointers save an indirection each time we add a particle to the data structure — such optimizations are typical for container data structures [152]. As an exception, we do not maintain the size field of the back chunk, since it can be deduced from the two auxiliary pointers. This data structure is illustrated in Figure 5.5, with corresponding code in Listing 5.1.

The bag data structure supports the following operations, whose code is given in Listing 5.2:

- **Add**: inserts a particle into a bag, with complexity $O(1)$. An insertion may require allocating a new chunk, but the associated overhead is amortized over the size of a chunk.
- **Iter**: iterates over all the particles in the bag. This operation is almost as efficient as iterating over a static array. Most importantly, chunks may be deallocated while the iteration

[8] borderSize is a parameter of our algorithm that can be set between 0 and $\frac{1}{2} \cdot$ tileSize.

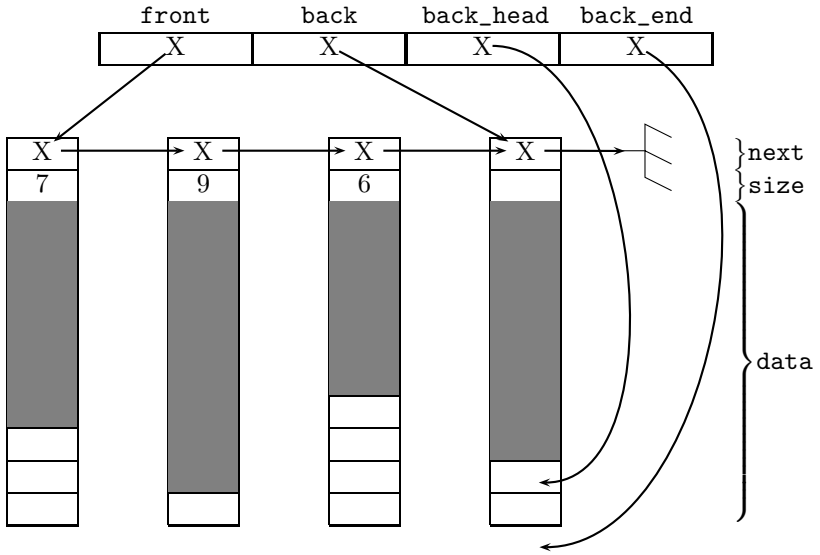


Figure 5.5 – Chunk bag data structure: chunks of size 10, particles stored in grey cells.

over the bag proceeds (line 19 in Figure 5.7). This possibility enables us to perform our operations as in an out-of-place algorithm, yet without having to pay for the twofold space overhead associated with out-of-place algorithms.

- **Merge**: two bags may be merged in-place, with complexity $O(1)$, by concatenating the two linked lists involved. Importantly, no compaction is involved. In particular, after a merge, a non-full chunk may appear in the middle of a linked list of chunks, as appears in Figure 5.6.

- **Memory management**: since all chunks have the same size, allocation and deallocation are optimized using free lists, thus avoiding costly and frequent `malloc` and `free` calls. This kind of dynamic storage allocation is a variant of *obstack* [152]: we are here in the special case where we do not deallocate the obstack until the end of the program. This memory management is also called a *memory pool*. In the more general case where the objects to manage do not have all the same size, see the algorithms in [25, Section 2.5].

At a given iteration of the simulation, we use one bag per cell from the grid, for storing the particles in this cell. To prepare for the next iteration, we need to distribute particles to different bags, which are associated with the next iteration. In order to avoid data races between the several threads that move the numerous particles, we allocate one bag for each cell and for each thread. With all these bags at hand, each thread may place the particles that it processes directly in a bag associated with the destination cell, without having to worry about races with other threads. Once all particles are distributed in these bags, the algorithm merges, for each cell, the bags associated with that cell (there are as many such bags as threads). Since each merge operation takes constant time, as it amounts to an in-place concatenation of two linked lists, the overall cost of merging all these bags is $O(\mathsf{nbThreads} \times \mathsf{nbCells})$. In practice, this cost is small compared to the processing of all the particles. Once the bags are merged, the particles are readily sorted for the next iteration.

One might worry about the memory overhead associated with the numerous bags involved. Yet, the total memory footprint of our algorithm is equal to the minimal amount of space required for representing all the particles, plus a fixed memory overhead of size $2 \cdot \mathsf{nbThreads} \cdot \mathsf{nbCells} \cdot \mathsf{memoryOf(chunk)}$. For example, in a simulation on a 128x128 grid, with chunks of size 512, executing on 18 cores, the memory overhead is 7.3 GB. This may be significant in absolute terms, nevertheless it is much less than what is required by competing algorithms whose memory overheads are proportional to the number of particles, *e.g.*, accounting for 50% of the

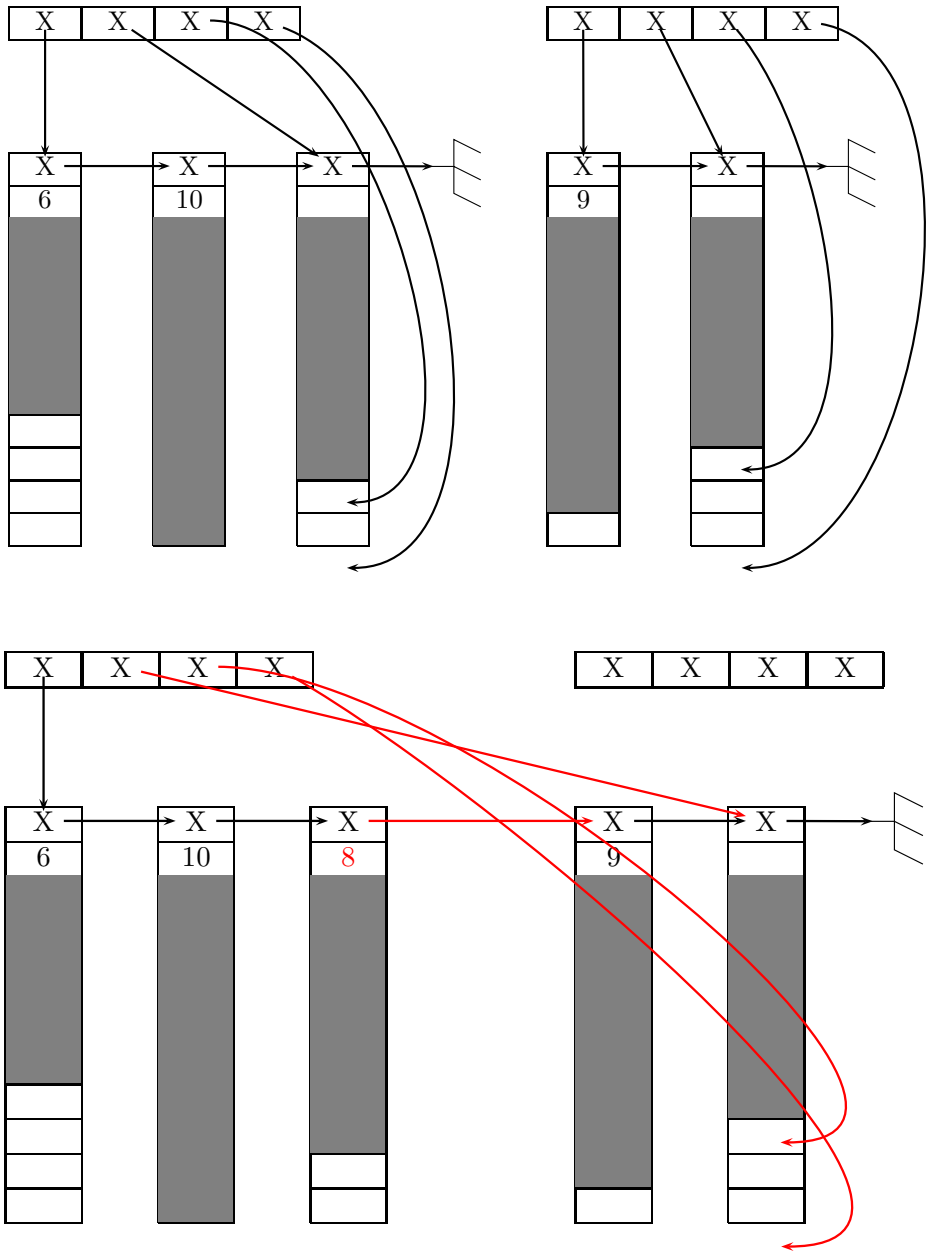


Figure 5.6 – Chunk bag data structure: the merge operation. Top: two chunks. Bottom: those chunks merged.

```
#define BYTES_PER_CACHE_LINE 64
#define INTS_PER_CACHE_LINE (64 / sizeof(int))
int[num_threads][INTS_PER_CACHE_LINE] free_index; // Avoids false sharing
#define FREE_INDEX(i) free_index[i][0]
unsigned int max_freelist_size = num_particle / CHUNK_SIZE / num_threads;
chunk* free_chunks[num_threads][max_freelist_size];

// Take a chunk from the free list if possible, else allocate a new one.
chunk* chunk_alloc(int thread_id) {
  return (FREE_INDEX(thread_id) > 0)
    ? free_chunks[thread_id][--FREE_INDEX(thread_id)];
    : (chunk*) malloc(sizeof(chunk));
}

// Put the chunk at the end of the freelist if possible, else free it.
void chunk_free(chunk* c, int thread_id) {
  if (FREE_INDEX(thread_id) < max_freelist_size)
    free_chunks[thread_id][FREE_INDEX(thread_id)++] = c;
  else
    free(c);
}

// Allocate a new chunk and put it at the back of a chunk bag.
chunk* new_back_chunk(bag* b, int thread_id) {
  chunk* c = chunk_alloc(thread_id); // c->size needs not be initialized
  c->next = (void*)0;
  b->back = c;
  b->back_head = &(c->array[0]);
  b->back_end  = &(c->array[CHUNK_SIZE]);
  return c;
}

// Initialize a bag (already allocated) with only one empty chunk.
void bag_init(bag* b, int thread_id) {
  chunk* c = new_back_chunk(b, thread_id);
  b->front = c;
}

// Merge other into b; other becomes empty.
void bag_append(bag* b, bag* other, int thread_id) {
  b->back->size = CHUNK_SIZE - (b->back_end - b->back_head); // Pointer
      arithmetic.
  b->back->next = other->front;
  b->back       = other->back;
  b->back_end   = other->back_end;
  b->back_head  = other->back_head;
  bag_init(other, thread_id);
}

// Add p into the last chunk of bag b (allocate a new chunk before if needed).
void bag_push(bag* b, particle p, int thread_id) {
  if (b->back_head == b->back_end) {
    chunk* old_back = b->back;
    chunk* c = new_back_chunk(b, thread_id);
    old_back->size = CHUNK_SIZE;
    old_back->next = c;
  }
  *(b->back_head) = p;
  b->back_head++;
}
```

Listing 5.2 – 2d chunk operations; chunks data structure in Listing 5.1.

2d Particle-in-Cell multi-core algorithm	Memory usage, in bytes[10]	Largest N for 64 GB, in billions
Out-of-place periodic sort, AoS [47]	$32 \cdot 2\,N$	0.9
Out-of-place periodic sort, SoA [204]	$28 \cdot 2\,N$	1.0
Always sorted, static arrays [85]	$\geq 24 \cdot 1.5\,N$	≤ 1.6
Always sorted, packed arrays [144, 12]	$24 \cdot (1.4\,N + M)$	$1.0 \leq N \leq 1.7$
In-place periodic sort, AoS [43]	$32 \cdot N$	1.8
Sort by super-cell each time step, SoA [61]	$28 \cdot (N + 0.1\,M)$	$1.8 \leq N \leq 2.0$
Always sorted by super-cell, frame lists [49]	$(28 + \frac{64}{\mathsf{frameSize}})\,(N + 0.1\,M)$	$1.8 \leq N \leq 2.0$
Always sorted, SoA [72]	$24 \cdot 1.17\,N$	2.0
Always sorted, chunk bags (Variant 1)	$(24 + \frac{64}{\mathsf{chunkSize}}) \cdot N + C_1$	2.1

Table 5.4 – Memory usage of 2d PIC implementations. N is the number of particles, M is the maximum number of particles crossing cell boundaries on one iteration (M can be up to N in our simulations), and $C_1 \approx 2 \cdot \mathsf{nbThreads} \cdot \mathsf{nbCells} \cdot \mathsf{memoryOf(chunk)}$ is a constant for a given grid and hardware — we here use a grid of size 128×128 and a hardware with 18 threads.

total memory usage. Table 5.4 summarizes the memory usage of the algorithms mentioned in Section 5.2, to compare against our proposal, which, asymptotically, requires a smaller amount of memory. The last column shows that, for 64 GB of total memory or more, our algorithm is able to fit a much larger number of particles in memory. We recall that, in 2d, the "index plus offset" representation requires 28 bytes per particle if stored in an SoA fashion, but 32 bytes per particle if stored in an AoS fashion, due to necessary padding[9]. When using the strict-binning approach, the cell index does not need to be stored, bringing the size requirement down to 24 bytes per particle.

5.4.2 Our Strict-Binning Algorithm

The pseudo-code of our algorithm appears in Figure 5.7. The key ideas have been described in Section 5.4.1. An important addition is the loop fission that we have applied in order to exploit the Single Instruction on Multiple Data (SIMD) feature: lines 11 and 13 we have two loops instead of only one loop on the particles of a chunk. Particles update their velocity by interpolating the value of the electric field at their position. Since the interpolation formula is the same for all particles from a same cell, it may be implemented using vectorized instructions. To that end, we isolated the velocity update operations (line 12). As long as the data from one chunk fits into the L1 cache, this does not increase the number of accesses to higher cache levels (nor to the main memory). Otherwise, an additional level of tiling can be applied. In our tests with L1 cache size equal to 32 KB, chunk sizes between 128 and 512 leads to the best performance, see Figure 5.8 and Figure 5.9. We see on those figures that there is a limit on the chunk size: when it is too big, there is not enough memory to store the chunks (a chunk size of 4096 is too big on icps-gc-6 on 10 threads and a chunk size of 1024 is too big on Marconi A1 on 18 threads). When we have enough memory, bigger chunk sizes lead to L2 cache accesses — slower than L1 cache accesses — and smaller chunk sizes lead to frequent irregular memory accesses, which explains the shape of those histograms.

We note that the use of chunks can be seen as strip-mining the main loop: in each cell, lines

[9] Even though the particle data fits on 28 bytes, padding for the remaining 4 bytes is required for double values to be aligned. In some implementations, those remaining 4 bytes may be used for the particle weight.

[10] In [85], the factor 1.5 allows each cell to contain up to 50% more particles than the average; above that threshold, the simulation must be interrupted. In [144], the factor 1.4 comes from the fact that 40% of the array is reserved for unused cells (holes). In [72], the factor 1.17 similarly corresponds to 6% unused cells and overflow buffers. In [49] and [61], the factor 0.1 is the expected fraction of particles leaving supercells. In our work, the term $\frac{64}{\mathsf{chunkSize}}$ accounts for the size of the fields `next` and `size` of each chunk (we here use $\mathsf{frameSize} = \mathsf{chunkSize} = 512$).

```
1  bag particles[0..nbCells−1]; // Particles by cell, at current time step
2  bag particlesNext[0..nbThreads−1][0..nbCells−1];
3  double ρ[0..ncx][0..ncy], E[0..ncx][0..ncy];
4  double ρNext[0..nbThreads−1][0..nbCells−1][0..3]; // 4 corners per cell
5  Foreach time step
6      Set in parallel // OpenMP parallel
7          particlesNext[0..nbThreads−1][0..nbCells−1] to empty, using an empty chunk
8          ρNext[0..nbThreads−1][0..nbCells−1][0..3] and ρ[0..ncx][0..ncy] to zero
9      Parallel Foreach idCell in [0..nbCells−1] // OpenMP parallel
10         Read E[x][y], foreach (x, y) among the 4 corners of cell idCell
11         Foreach chunk in particles[idCell]
12             Foreach particle in that chunk // SIMD vectorized
13                 Update particle velocity
14             Foreach particle in that chunk
15                 Update particle position
16                 Compute idCellNext, the index of the cell containing the particle
17                 Add the particle into particlesNext[currentThreadId][idCellNext]
18                 Accumulate its charge into ρNext[currentThreadId][idCellNext][0..3]
19             Deallocate that chunk
20     Parallel Foreach idCell in [0..nbCells−1] // OpenMP parallel
21         Set particles[idCell] to particlesNext[0][idCell]
22         For idThread in [1..nbThreads−1]
23             Merge particlesNext[idThread][idCell] into particles[idCell]
24         For idThread in [0..nbThreads−1], For i in [0..3]
25             ρ[x][y] += ρNext[idThread][idCell][i], where (x, y) is i−th corner of cell idCell
26     Compute E from ρ using a Poisson solver // FFTW
```

Figure 5.7 – Variant 1 of our algorithm for the 2d PIC method on multi-core architectures.

9 and 11 feature a loop over the chunks then, nested, a loop over the particles in that chunk, instead of just a loop over all the particles in the cell. This technique is thus comparable to the strip-mining technique shown in Section 4.5.3. We showed in this previous section that a strip size between 64 and 256 gave similar optimal results, and here we show that a chunk size between 128 and 512 give similar optimal results. The reason for that difference is that with chunks, we have only one electric field value to read for all the particles in a given cell (line 10), whereas in the previous approach, we had as many electric field values to read as the number of different cells in which particles of a common strip would be scattered, thus filling the cache with those different values.

To summarize, our algorithm has three key features:

- First, at each step, each particle is read from and written into the main memory exactly once (read on line 13, still in cache for lines 15–18, and write on line 17). Thus, our algorithm does not perform unnecessary accesses to the main memory.
- Second, each time step involves only three synchronization points: one at the end of each parallel loop (lines 6, 9, and 20).
- Third, thanks to the use of thread-indexed data structures for ρ and for particlesNext, we avoid data races and do not need atomic operations. Note that for ρ, this can be done automatically with the pragma `reduction(+:rho[0:nbCells][0:4])` from `OpenMP 4.5`.

With this pseudo-code in mind, we can now prove our upper bound C_1 on the number of additional chunks needed.

Theorem 1. *For a test case with N particles, a grid size of nbCells, an architecture with nbThreads threads and a chunk size of chunkSize, the Variant 1 of our algorithm uses, for the particles, a memory*

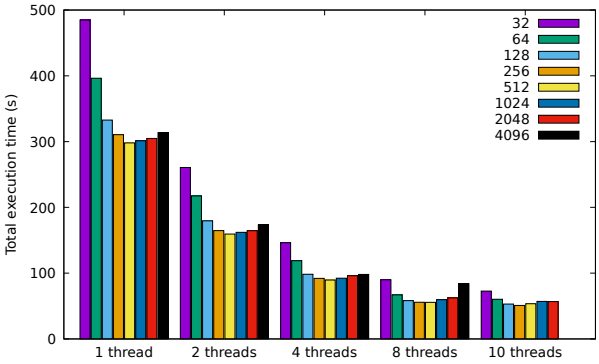


Figure 5.8 – Influence of chunk size on performance (icps-gc-6, 200 million particles, 128 × 128 grid). Variant 1 in Table 5.3. An absence of data means that the associated chunk size is too large for the memory.

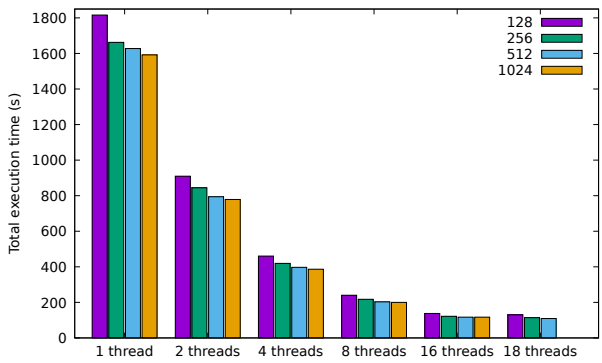


Figure 5.9 – Influence of chunk size on performance (Marconi A1, 900 million particles, 128 × 128 grid). Variant 1 in Table 5.3. An absence of data means that the associated chunk size is too large for the memory.

(in bytes) which is given by

$$\mathit{memoryOf(chunk)} \cdot \left\lceil \frac{N}{\mathit{chunkSize}} \right\rceil + C_1$$

where $C_1 = \mathit{memoryOf(chunk)} \cdot \mathit{nbThreads} \cdot (2 \cdot \mathit{nbCells} + 1) + 32 \cdot \mathit{nbCells} \cdot (\mathit{nbThreads} + 1)$.

Proof. Each bag structure weighs 32 bytes (four pointers, see Listing 5.1). We have one bag per grid cell in the particles variable, and one bag per grid cell per thread in the particlesNext variable. The total memory for the bag structure is $32 \cdot \mathsf{nbCells} \cdot (\mathsf{nbThreads} + 1)$.

Now let us look at the chunks inside those bags. We recall that N is the total number of particles, nbCells the number of cells, nbThreads the number of threads and chunkSize the number of particles that can be stored in a chunk. We additionally denote by:

- $N_{current}$ the number of particles still in the variable particles
- N_{next} the number of particles already in the variable particlesNext
- $C_{current}$ the number of chunks still in the variable particles
- C_{next} the number of chunks already in the variable particlesNext

We want to show that $C_{current} + C_{next} \leq \left\lceil \frac{N}{\mathsf{chunkSize}} \right\rceil + \mathsf{nbThreads} \cdot (2 \cdot \mathsf{nbCells} + 1)$. To that end, we use the fact that at any point in the simulation, $N_{current} + N_{next} = N$. A particle is either still in particles, either it has already been processed and is now in particlesNext. We now look at the number of chunks:

- first, notice that in the C_{next} chunks already in the variable particlesNext, there are at most $\mathsf{nbCells} \cdot \mathsf{nbThreads}$ non-full chunks (when a thread adds a particle in a bag, it uses the last chunk of this bag until it is full, thus only the last chunk can be non-full, see `bag_push` in Listing 5.2). The other (full) chunks contain chunkSize particles. We obtain:

$$\mathsf{chunkSize} \cdot (C_{next} - \mathsf{nbThreads} \cdot \mathsf{nbCells}) \leq N_{next}$$

- then, notice that in the $C_{current}$ chunks still in the variable particles, there are at most $\mathsf{nbThreads} \cdot (\mathsf{nbCells} + 1)$ non-full chunks (the chunk which is being processed by each thread, and the non-full chunks coming from the previous iteration when merging bags, see the previous point). The other (full) chunks contain chunkSize particles. This gives us:

$$\mathsf{chunkSize} \cdot (C_{current} - \mathsf{nbThreads} \cdot (\mathsf{nbCells} + 1)) \leq N_{current}$$

Summing up those two inequalities, we get:

$$\mathsf{chunkSize} \cdot (C_{current} + C_{next} - \mathsf{nbThreads} \cdot (2 \cdot \mathsf{nbCells} + 1)) \leq N_{current} + N_{next} = N$$

which gives

$$\begin{aligned} C_{current} + C_{next} &\leq \frac{N}{\mathsf{chunkSize}} + \mathsf{nbThreads} \cdot (2 \cdot \mathsf{nbCells} + 1) \\ &\leq \left\lceil \frac{N}{\mathsf{chunkSize}} \right\rceil + \mathsf{nbThreads} \cdot (2 \cdot \mathsf{nbCells} + 1) \end{aligned}$$

Whatever happens during a simulation, a total number of chunks of $\lceil \frac{N}{\mathsf{chunkSize}} \rceil + \mathsf{nbThreads} \cdot (2 \cdot \mathsf{nbCells} + 1)$ is thus enough. □

Application: On 18 cores with a 2d simulation where $\mathsf{memoryOf(particle)} = 24$ and a grid size of 128×128, it leads to 7.3 GB of auxiliary memory. See Table 5.4 to put this in perspective.

5.4.3 Performance Results

Our experiments were conducted on the A1 partition of the Marconi supercomputer (see Table 5.1), on which we were granted the use of 64 nodes with 2 sockets each. Each socket is an Intel Xeon E5-2697 v4 @2.3 GHz (Broadwell), with 64 GB of RAM, 4 memory channels, and 18 cores. Our `C` code was compiled using Intel C Compiler 17.0.1, using the FFTW3 library [149] for the Poisson solver, and storing 512 particles per chunk.

We ran simulations on two classical test cases [5, 23] and checked that they matched the expected mathematical results, see details in Section 7.1. We used periodic boundary conditions, and the following initial distributions:

$$\begin{aligned} &\left(1 + 0.01 \cos\left(\tfrac{x}{2}\right) \cos\left(\tfrac{y}{2}\right)\right) \tfrac{1}{2\pi v_{th}^2} \exp\left(-\tfrac{v_x^2+v_y^2}{2v_{th}^2}\right) && \text{Landau damping} \\ &\left(1 + 0.1 \left(\cos\left(\tfrac{y}{2}\right) + \cos\left(\tfrac{x+y}{2}\right)\right)\right) \tfrac{v_x^2}{2\pi v_{th}^2} \exp\left(-\tfrac{v_x^2+v_y^2}{2v_{th}^2}\right) && \text{Two-stream instability} \end{aligned}$$

One important challenge faced by prior work is that performance significantly depends on the percentage of particles crossing cell boundaries at each time step. In contrast, the performance of our algorithm should, by design, not depend so much on the percentage of crossing particles. To empirically verify this claim, we increased particle velocities by a factor 100 (raising v_{th} from 0.01 to 1.0). For Landau damping, this increased the percentage of crossing particles from 1.8% to 87%, but increased execution time by only 4.64%. For two-stream instability, this increased the percentage of crossing particles from 12% to 98%, but increased execution time by only 4.59%.

The next experiments all use the Landau damping test case with $v_{th} = 1.0$, summarized in Table 5.5[11].

Figure 5.10 reports a strong scaling for our algorithm, and compares it our prior work using SoA shown in Section 4.4, carefully optimized for the same architecture. Although the SoA algorithm is slightly faster when using 4 cores or less, our chunk algorithm, which puts less pressure on the memory bus, outperforms it for more cores. With 18 cores, the chunk algorithm is 36% faster and is able to update 861 million particles per second. Equivalently, one thread is able to process one particle at one time step in no more than 48 cycles, all inclusive. Note that this experiment simulates 900 million particles, which is the maximum that out-of-place sorting can accommodate with 64 GB, whereas our algorithm could handle more than twice as many particles, see Table 5.4.

[11] It is the same test case as in Table 4.3 in Chapter 4.

Physical test case	Linear Landau damping [5, Section 5.15], initial distribution $f(x,y,v_x,v_y,t=0) = \left(1+0.01\cos\left(\frac{x}{2}\right)\cos\left(\frac{y}{2}\right)\right)\frac{1}{2\pi}\exp\left(-\frac{v_x^2+v_y^2}{2}\right)$
Spatial grid	$[0;4\pi)^2$ decomposed in 128^2 cells, periodic boundaries
Particle shape factor	Cloud-in-cell model [42]
Number of iterations	100
Time step	0.1
Particle crossing: averaged, per iteration	58% of the particles move 1 cell away, 25% move 2 cells away, 3.4% move 3 cells away, 0.18% move further away

Table 5.5 – 2d test case.

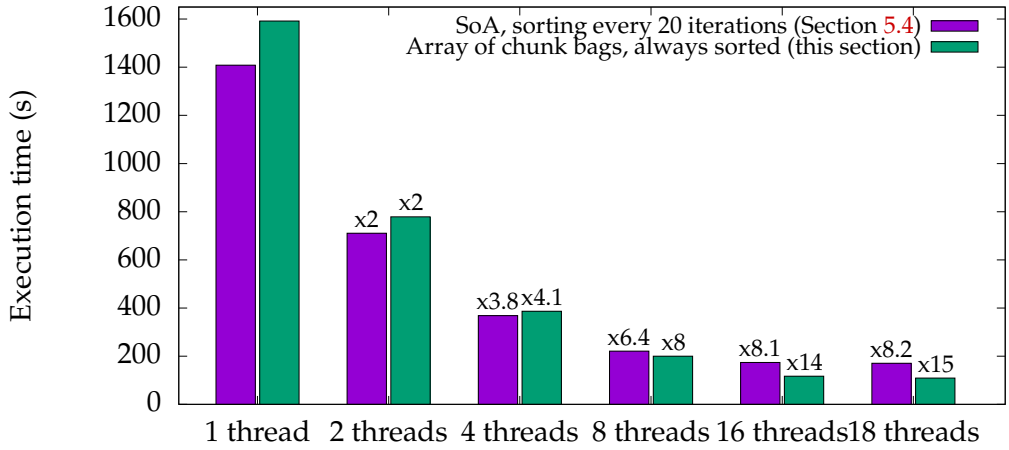


Figure 5.10 – Strong scaling with 900 million particles. Test case in Table 5.5. Variant 1 in Table 5.3.

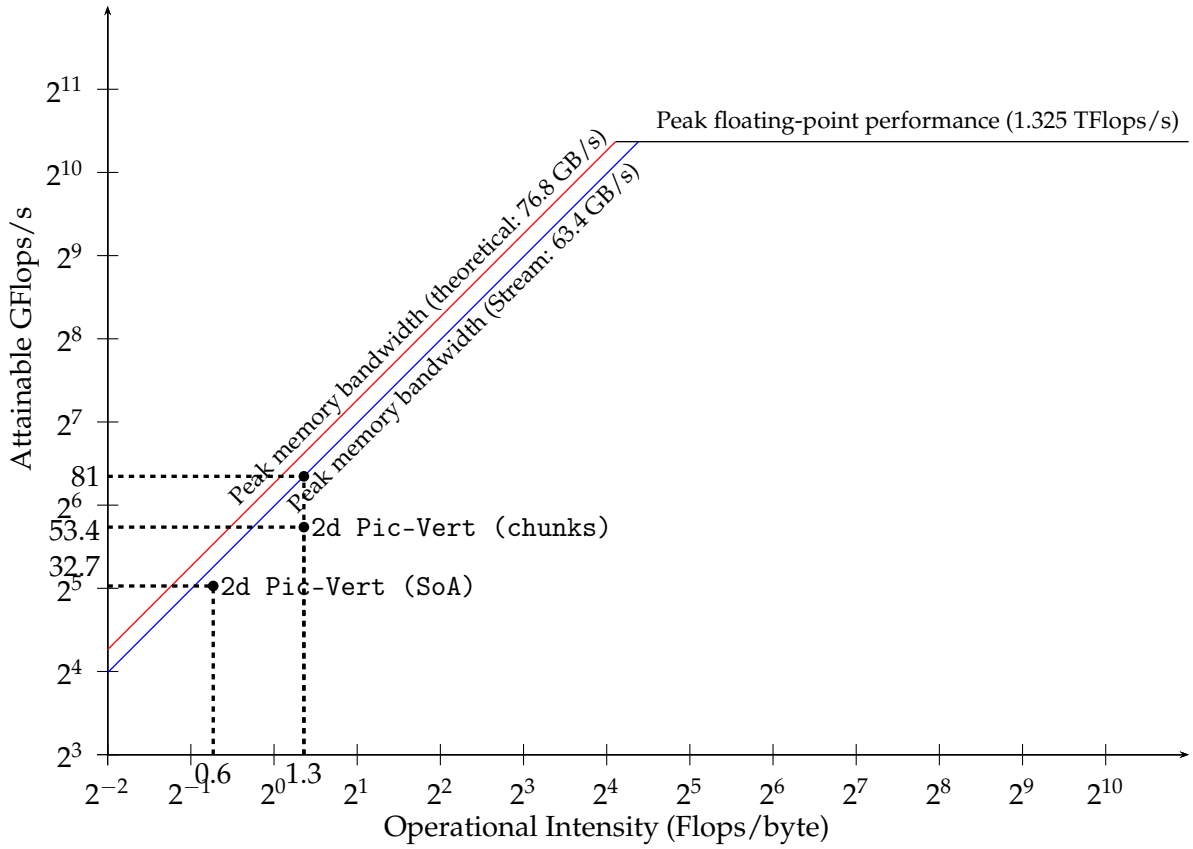


Figure 5.11 – Roofline model with 900 million particles. Test case in Table 5.5. Variant 1 in Table 5.3.

An analysis of those results in the roofline model, shown in Figure 5.11, allows us to understand why our previous SoA implementation cannot scale better on 18 threads: it almost reached the maximum memory bandwidth. We thus understand how precious is an algorithm which incurs less memory accesses. As a side note, let us nevertheless acknowledge that the SoA implementation was run without the strip-mining technique discussed in Section 4.5.3 (which we did not implement at the time of those experiments). The Marconi A1 partition is now no more available to allow comparison with this implementation on this architecture, but we will show a detailed comparison on other architectures in Chapter 8.

Figure 5.12 shows the memory bandwidth of our implementation when performing a weak scaling. We take as reference the Stream benchmark [162], which aims at evaluating the maximal bandwidth that can be reached in practice. The Stream benchmark reaches 63.4 GB/s, which corresponds to 83% of the theoretical peak of our hardware (76.8 GB/s). On 18 cores, our algorithm reaches more than 65% of the reference memory bandwidth. Since our algorithm does not perform unnecessary accesses to the main memory, we conclude that our implementation is not far from exploiting the machine at its best.

Figure 5.13 reports on the performance of hybrid parallelism, with a weak scaling of our implementation on 128 sockets (2 304 cores), using one `MPI` process per socket, and 18 `OpenMP` threads per socket, i.e. one thread per core. The results show an almost perfect scaling, with only 8% overhead when scaling from 1 to 128 sockets. This overhead is expected, due to the (logarithmic) communication costs involved in the `MPI_ALLREDUCE` communication, as we used particle decomposition to parallelize our implementation on distributed memory, see Section 2.6. This experiment demonstrates the efficiency of our parallel algorithm at the scale of 230 billion particles.

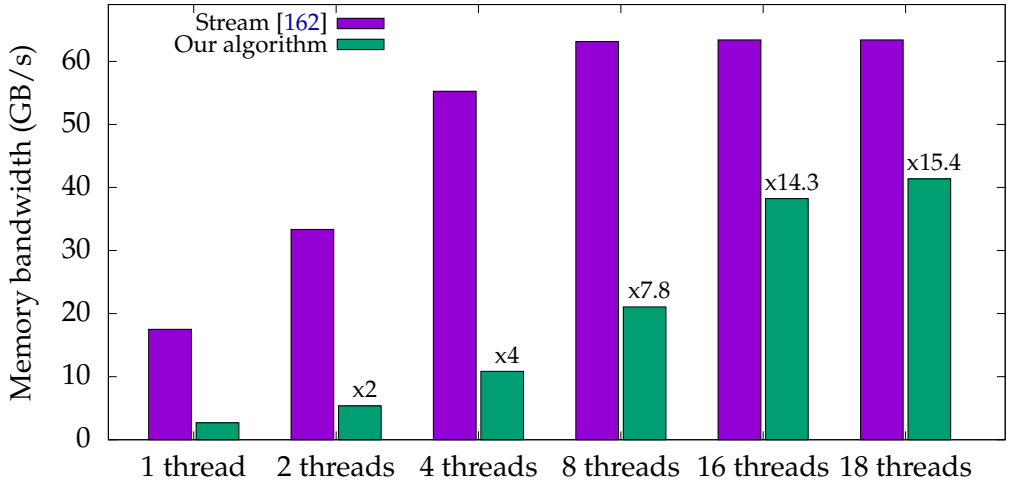


Figure 5.12 – Memory bandwidth with 100 million particles per core (up to 1.8 billion particles in total), measured as: nbIterations · nbParticles · memoryOf(particle) · 2 / executionTime. Test case in Table 5.5. Variant 1 in Table 5.3.

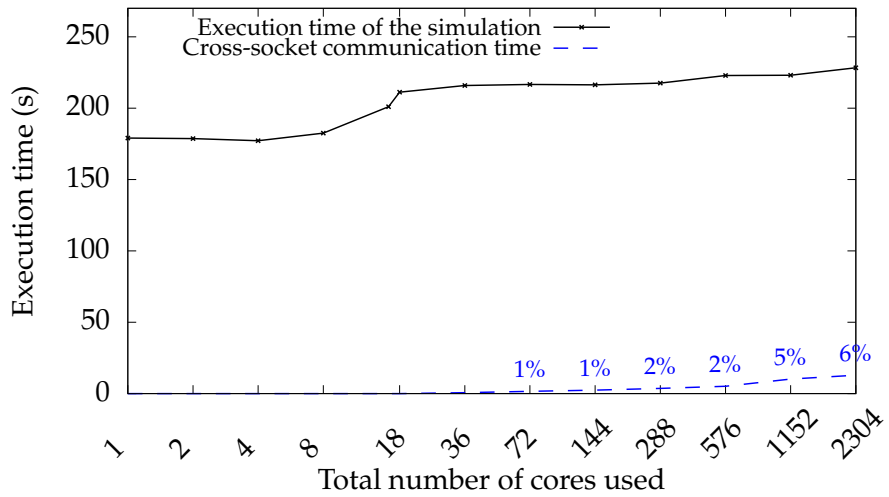


Figure 5.13 – Weak scaling with 100 million particles per core (up to 230 billion particles in total), on up to 128 18-core sockets. Test case in Table 5.5. Variant 1 in Table 5.3.

```
struct { float dx, dy, dz; double vx, vy, vz; } particle_3d;
struct chunk { struct chunk* next; int size;
               particle_3d array[CHUNK_SIZE]; } chunk;
struct { chunk* front, back; particle* back_end, back_head; } bag;
```

Listing 5.3 – Chunk bag of AoS data structure, in 3d.

```
// Strict-binning with chunk bags of SoA (linked-lists of fixed-size arrays).
// Total number of chunks: depends on implementation.
// CHUNK_SIZE is an architecture-dependent parameter (depends on cache size).
#define CHUNK_SIZE 128
// VEC_ALIGN is architecture dependent, e.g. 32 with 256-bits vectors (AVX2).
// _Alignas(VEC_ALIGN) (c11) can be safely replaced with __attribute__((aligned(
    VEC_ALIGN))) (gcc 2.95.3).
struct chunk { struct chunk* next; int size; // 0 <= size <= CHUNK_SIZE
               _Alignas(VEC_ALIGN) float  dx[CHUNK_SIZE];
               _Alignas(VEC_ALIGN) float  dy[CHUNK_SIZE];
               _Alignas(VEC_ALIGN) float  dz[CHUNK_SIZE];
               _Alignas(VEC_ALIGN) double vx[CHUNK_SIZE];
               _Alignas(VEC_ALIGN) double vy[CHUNK_SIZE];
               _Alignas(VEC_ALIGN) double vz[CHUNK_SIZE]; } chunk;
struct { chunk* front, back; } bag; // linked list of chunks
bag* particle_sets[ncx * ncy * ncz];
```

Listing 5.4 – Chunk bag of SoA data structure, in 3d.

5.5 Variant 2 of our Strict-Binning Algorithm

This section will detail the second variant of our strict-binning algorithm. The code used for this section is available in the figshare repository [206] ("Best Artifact Award" at Euro-Par 2018).

5.5.1 The Chunk Bag Data Structure (SoA inside)

In 2d, using AoS or SoA for the chunks uses the same memory because there is no padding in the `particle_2d` structure of Listing 5.1. In 3d, using AoS as described in Listing 5.3 would incur a 4-byte padding in the `particle_3d` structure. It is thus more efficient to use SoA. Benchmarking of our algorithm reveals that this SoA layout, which enables better vectorization, improves performance compared to the AoS layout — an observation consistent with the findings of Nakashima *et al.* [72]. It is thus the best layout, both for memory usage and for execution time.

With a SoA layout, it is not efficient to use the auxiliary pointers `back_end` and `back_head` as used in the AoS layout of Listings 5.1 and 5.3. Indeed, with SoA we would need one `back_head` pointer for each of the six arrays, plus one `back_head` pointer (for any of those six arrays). It is thus more efficient to update the `size` field each time we add a particle in the chunk. Moreover, by updating this field at all times, it is possible to efficiently implement a thread-safe atomic insertion operation, which is one of the new features of this section.

The memory layout we use for the particles is summarized on Listing 5.4, and an example is given in Figure 5.14. As in Section 5.4, chunk bags support $O(1)$ insertion of a particle (adding a fresh chunk if needed), $O(1)$ merge of two bags thanks to the *back* field (note that chunk compaction is not needed), see Figure 5.15, and $O(n)$ iteration over the contents, all with excellent constant factors.

Furthermore, unlike chunks introduced in the previous section, these ones are devised to support a thread-safe atomic insertion operation. Atomic insertions are central to the handling of fast-moving particles, as detailed further on. We implement atomic insertion using a fetch-and-add instruction to atomically reserve a slot in the chunk where to push the particle. In an

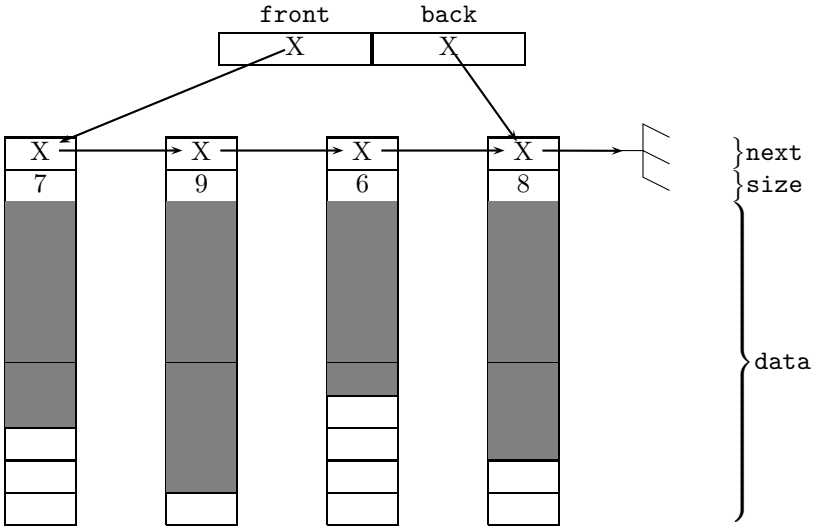


Figure 5.14 – Chunk bag data structure: chunks of size 10, particles stored in grey cells.

atomic insertion, if a thread attempts to reserve the one-past-the-end slot, it acquires responsibility to extend the bag with a fresh chunk, in which case it sets the *next* pointer of the fresh chunk to the current *front* pointer of the bag, and sets the *front* pointer of the bag to the address of the fresh chunk[12]. The `C` code of this atomic insertion is given in Listing 5.5. We can note that in Listing 5.2 we inserted new particles in the last chunk, and here we insert in the first chunk. Both options can be used.

When processing a chunk of particles, the algorithm first updates velocities and positions, then migrates the particles to different chunks, depending on the cell associated with their new position. Once all particles from the chunk are processed, the chunk is stored into a (per-thread) free list, so as to be subsequently reused to extend a bag whose last chunk becomes full. Our algorithm preserves the following invariant: at the beginning of a time step, all the particles are stored in at most $\lceil \frac{N}{\mathsf{chunkSize}} \rceil + 2 \cdot \mathsf{nbCells}$ chunks, where N denotes the total number of particles, and $\mathsf{chunkSize}$ denotes the number of particles per chunk.

To dispatch particles according to their target cells, we associate two bags with each cell: a *private bag*, accessed at most by one thread at a time; and a *shared bag*, accessed concurrently, to handle fast-moving particles. To initialize these two bags, we need an additional $2 \cdot \mathsf{nbCells}$ empty chunks. In total, we need $\lceil \frac{N}{\mathsf{chunkSize}} \rceil + 4 \cdot \mathsf{nbCells}$ chunks. We proved that this number of chunk suffices at any point of a simulation, regardless of how particles move. Thus, the space used by our algorithm, in addition to the minimal amount of memory needed to represent the particles, grows in proportion with $4 \cdot \mathsf{nbCells} \cdot \mathsf{memoryOf(chunk)}$[13].

Table 5.6 summarizes the memory usage of the algorithms mentioned in Section 5.2, to compare against our proposal, which, asymptotically, requires a smaller amount of memory. The last column shows that, for 96 GB of total memory or more, our algorithm is able to fit a much larger number of particles in memory. We recall that, in 3d, the "index plus offset" representation requires 40 bytes per particle. When using the strict-binning approach, the cell index does not need to be stored, leading to 36 bytes per particle. We note that this table uses $\mathsf{chunkSize} = 128$; it would not even be possible to use $\mathsf{chunkSize} = 256$ with the approach from the previous section: all the memory would be used for the additional chunks.

[12] Assigning the front pointer can be implemented with a non-atomic write, as long as it is preceded by a memory fence, to ensure preservation of the order of write operations.

[13] Although we proved a tight bound on the number of chunks used, in practice, we allocate some extra chunks per thread, to give some slack and avoid the need for dynamic load balancing of free chunks. Note that it is quite unlikely for one of these additional chunks to ever be required: due to the presence of partially-filled chunks at the end of each source bag, threads free source chunks at a slightly faster rate than they fill target chunks.

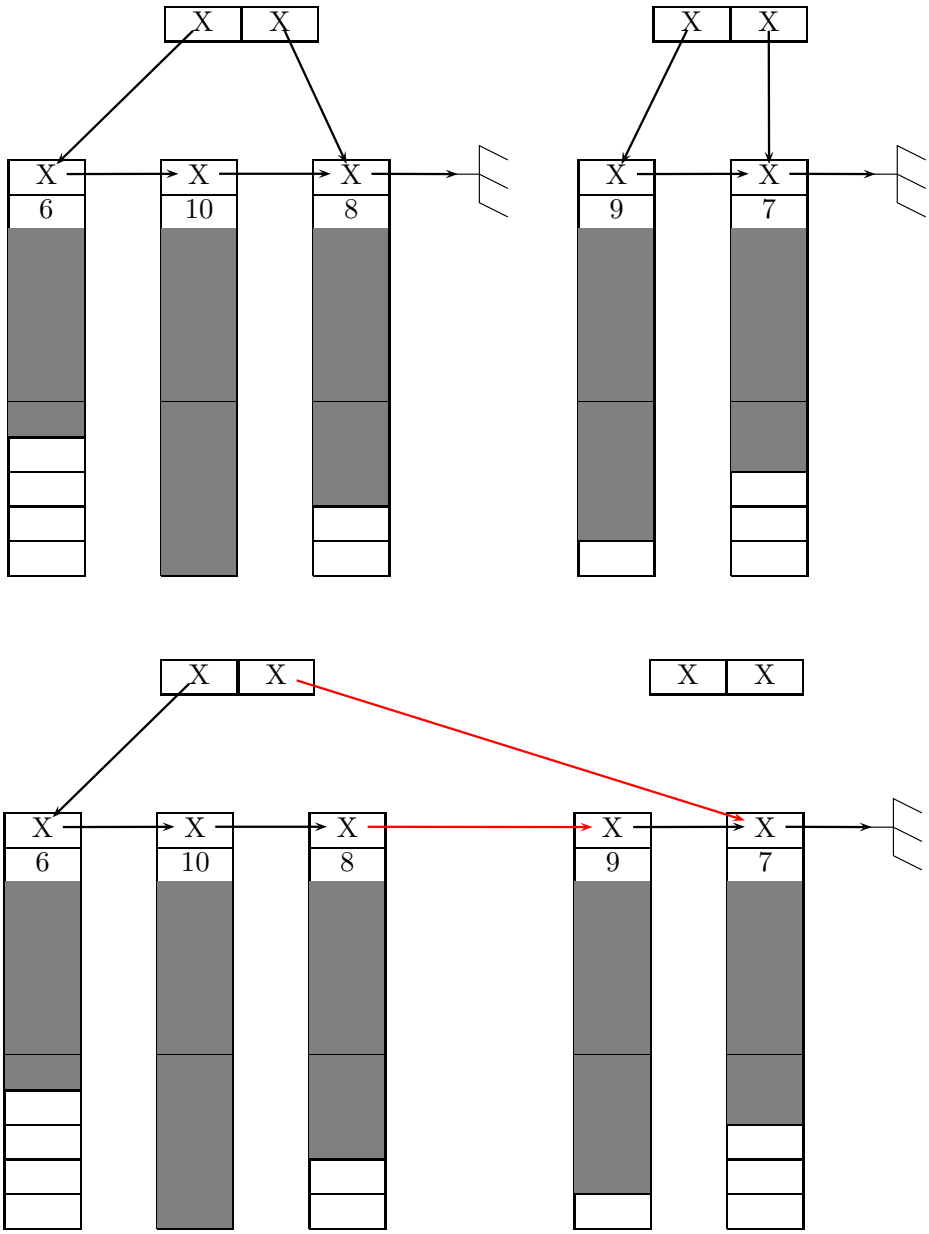


Figure 5.15 – Chunk bag data structure: the merge operation. Top: two chunks. Bottom: those chunks merged.

```
1  // Allocate a new chunk and put it at the front of a chunk bag.
2  // Two atomic writes to ensure the order of operations, so that other threads
3  // read the correct c->size on line 30 after reading b->front on line 45.
4  void add_front_chunk(bag* b, int thread_id) {
5    chunk* c = chunk_alloc(thread_id); // See Listing 5.2.
6    #pragma omp atomic write
7    c->size = 0;
8    c->next = b->front;
9    #pragma omp atomic write
10   b->front = c;
11 }
12
13 // Guarantee that *p is read atomically. Adapted from Example atomic.2.c in
14 // https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf
15 chunk* atomic_read(chunk** p) {
16   chunk* value;
17   #pragma omp atomic read
18   value = *p;
19   return value;
20 }
21
22 // Add p into the first chunk of bag b (allocate a new chunk after if needed).
23 // Adapted from Example atomic.3.c also in openmp-examples-4.5.0.pdf.
24 void bag_push_concurrent(bag* b, float dx, float dy, float dz, double vx, double
       vy, double vz, int thread_id) {
25   chunk* c;
26   int index;
27   while (true) { // Until success.
28     c = b->front;
29     #pragma omp atomic capture
30     index = c->size++;
31     if (index < CHUNK_SIZE) { // The chunk is not full, we write the particle.
32       c->dx[index] = dx; c->dy[index] = dy; c->dz[index] = dz;
33       c->vx[index] = vx; c->vy[index] = vy; c->vz[index] = vz;
34       if (index == CHUNK_SIZE - 1) // The chunk is now full, we extend the bag.
35         add_front_chunk(b, thread_id);
36       return;
37     } else {
38       // The chunk is full: another thread has just pushed a particle in it
39       // and is now extending the bag. First, we cancel our additional
40       // "c->size++" (c->size = CHUNK_SIZE is more efficient than c->size--).
41       #pragma omp atomic write
42       c->size = CHUNK_SIZE;
43       // Then, we wait until the bag is extended. The atomic_read forces the
44       // thread to read in the main memory, and not in its temporary view.
45       while (atomic_read(&b->front) == c) {}
46     }
47   }
48 }
49
50 // Add p into the first chunk of bag b (allocate a new chunk after if needed).
51 void bag_push_serial(bag* b, float dx, float dy, float dz, double vx, double vy,
       double vz, int thread_id) {
52   chunk* c = b->front;
53   int index = c->size++;
54   c->dx[index] = dx; c->dy[index] = dy; c->dz[index] = dz;
55   c->vx[index] = vx; c->vy[index] = vy; c->vz[index] = vz;
56   if (index == CHUNK_SIZE - 1) // The chunk is now full, we extend the bag.
57     add_front_chunk(b, thread_id);
58 }
```

Listing 5.5 – 3d chunk atomic insertion operation; chunks data structure in Listing 5.4.

3d Particle-in-Cell multi-core algorithm	Memory usage, in bytes[15]	Largest N for 96 GB, in billions
Always sorted, chunk bags (Variant 1)	$(36 + \frac{64}{\mathsf{chunkSize}}) \cdot N + C_1$	0.9
Out-of-place periodic sort [47, 203]	$40 \cdot 2\,N$	1.1
Always sorted, static arrays [85]	$\geq 36 \cdot 1.5\,N$	≤ 1.6
Always sorted, packed arrays [144, 12]	$36 \cdot (1.4\,N + M)$	$1.0 \leq N \leq 1.7$
Sort by super-cell each time step, SoA [61]	$40 \cdot (N + 0.2\,M)$	$1.8 \leq N \leq 2.1$
Always sorted by super-cell, frame lists [49]	$(40 + \frac{64}{\mathsf{frameSize}})\,(N + 0.2\,M)$	$1.8 \leq N \leq 2.1$
Always sorted, SoA [72]	$36 \cdot 1.17\,N$	2.1
In-place periodic sort, AoS [43]	$40 \cdot N$	2.2
Always sorted, chunk bags (Variant 2)	$(36 + \frac{64}{\mathsf{chunkSize}}) \cdot N + C_2$	2.3

Table 5.6 – Memory usage of 3d PIC implementations. N is the number of particles, M is the maximum number of particles crossing cell boundaries on one iteration (M can be up to N in our simulations), and $C_2 \approx 4 \cdot \mathsf{nbCells} \cdot \mathsf{memoryOf(chunk)}$ is a constant for a given grid, enhanced from $C_1 \approx 2 \cdot \mathsf{nbThreads} \cdot \mathsf{nbCells} \cdot \mathsf{memoryOf(chunk)}$ — we here use a grid of size $64 \times 64 \times 64$ and a hardware with 24 threads.

5.5.2 Our Strict-Binning Algorithm

In order to maximize the number of insertions into private bags while preserving a high degree of `OpenMP` parallelism, we follow the coloring scheme proposed by Kong *et al.* [65], and generalized from 2d to 3d by Nakashima *et al.* [72]. The idea is to fill the space with *tiles*, of size $2 \times 2 \times 2$ (or more), in a regular manner. Tiles are colored using 8 different colors in such a way that two adjacent tiles have distinct colors. At each of the 8 color phases, $\frac{1}{8}$ of the tiles are processed, in parallel by $\mathsf{nbThreads}$ threads[14]. Because cells processed in parallel by distinct threads are at least 2 cells away from each other, all the particles that move, at a given time step, no more than one cell away (no more than half a tile away, in general) can be pushed into private bags, in a thread-safe manner.

The pseudo-code of our algorithm appears in Figure 5.16. Particles from a same cell are processed sequentially by a same thread. To benefit from SIMD performance, we apply loop fission on the particle loop over each chunk, making the assumption that one chunk fits into the L1 cache. Otherwise, an additional level of tiling can be applied. First, the algorithm updates velocities (line 12). Second, it computes the new positions (line 14), introducing an auxiliary array for storing the new cell indices. Third, it sequentially pushes each particle into the chunk associated with its target cell. If the target cell lies in the current tile, or lies in the closer half of an immediate neighboring tile, a non-atomic insertion is performed on a private bag (line 19). Otherwise, an atomic insertion is performed on a shared bag (line 21). Note that the boolean condition involved can be evaluated using a simple arithmetic test.

Once all the particles are processed, the algorithm merges, for each cell, its private bag with its shared bag (line 27). No chunk compaction is performed at this point; as a result, the bag associated with one cell may contain up to 2 non-full chunks (corresponding to the head chunk of the private bag and that of the shared bag). Thus, there are at most $\lceil \frac{N}{\mathsf{chunkSize}} \rceil + 2 \cdot \mathsf{nbCells}$ nonempty chunks at the beginning of the next time step. It follows that at least $2 \cdot \mathsf{nbCells}$ empty chunks must have been freed during the current time step. This number corresponds exactly

[14] For a $2 \times 2 \times 2$ tiling, at the i-th coloring phase, the algorithm processes cells whose coordinates satisfy: $((x/2) \mod 2) + 2 \cdot ((y/2) \mod 2) + 4 \cdot ((z/2) \mod 2) = i$. Using larger tiles is possible. It may slightly reduce the number of accesses in shared bags, however it greatly reduces the number of tiles that can be processed independently in parallel at each phase. For example, a $4 \times 4 \times 4$ tiling divides the number of tiles by 8, and might thus result in increased idle time. Using tiles of size $1 \times 1 \times 1$ is also possible, but requires 27 colors.

[15] Remarks in footnote 10 on page 110 apply, except that the expected fraction of particles leaving supercells is 0.2 in 3d, and that we here use $\mathsf{frameSize} = \mathsf{chunkSize} = 128$.

```
1  bag particles[0..nbCells−1]; // Particles by cell, at current time step
2  bag particlesNextPrivate[0..nbCells−1], particlesNextShared[0..nbCells−1];
3  double ρ[0..ncx][0..ncy][0..ncz], E[0..ncx][0..ncy][0..ncz];
4  double ρNext[0..nbThreads−1][0..nbCells−1][0..7]; // 8 corners per cell
5  Foreach time step
6    Foreach color in [0..7] // 8 coloring phases
7      Parallel Foreach tile of that color // OpenMP parallel
8        Foreach cell idCell in that tile
9          Read E[x][y][z], foreach (x, y, z) among the 8 corners of cell idCell
10         Foreach chunk in particles[idCell]
11           Foreach particle in that chunk // SIMD vectorized
12             Update particle velocity
13           Foreach particle in that chunk // SIMD vectorized
14             Update particle position
15             Compute idCellNext, the index of the cell containing the particle
16           Foreach particle in that chunk
17             If the particle moves inside its tile
18             Or it moves to the closer half of a neighbor tile
19               Add the particle into particlesNextPrivate[idCellNext]
20             Else
21               Atomically add the particle into particlesNextShared[idCellNext]
22             Add its charge into ρNext[currentThreadId][idCellNext][..] // SIMD
23         Put a pointer to that chunk into the freelist of the current thread
24   Compute the cumulative sum of the free lists sizes
25   Parallel Foreach idCell in [0..nbCells−1] // OpenMP parallel
26     Set particles[idCell] to particlesNextPrivate[idCell]
27     Merge particlesNextShared[idCell] into particles[idCell]
28     Set particlesNextPrivate[idCell] to empty, using an empty chunk
29     Set particlesNextShared[idCell] to empty, using an empty chunk
30   Parallel Foreach (x, y, z) in [0..ncx]x[0..ncy]x[0..ncz] // OpenMP parallel, collapsed
31     Foreach of the 8 pairs (idCell,i) such that (x,y,z) is i−th corner of idCell
32       Foreach idThread in [0..nbThreads−1]
33         ρ[x][y][z] += ρNext[idThread][idCell][i]
34         ρNext[idThread][idCell][i] = 0
35   Compute E from ρ using a Poisson solver and set ρ to 0 // FFTW + OpenMP
```

Figure 5.16 – Variant 2 of our algorithm for the 3d PIC method on multi-core architectures.

to the number of chunks needed to initialize the private and the shared bags for the next time step. Our algorithm performs this initialization efficiently in parallel (using a prefix sum array, based on the sizes of the per-thread free lists).

With this pseudo-code in mind, we can now prove our upper bound C_2 on the number of additional chunks needed.

Theorem 2. *For a test case with N particles, a grid size of nbCells, an architecture with nbThreads threads and a chunk size of chunkSize, the Variant 2 of our algorithm uses, for the particles, a memory (in bytes) which is given by*

$$\mathit{memoryOf(chunk)} \cdot \left\lceil \frac{N}{\mathit{chunkSize}} \right\rceil + C_2$$

where $C_2 = \mathit{memoryOf(chunk)} \cdot (4 \cdot \mathit{nbCells} + \mathit{nbThreads}) + 48 \cdot \mathit{nbCells}$.

Proof. Each bag structure weighs 16 bytes (two pointers, see Listing 5.4). We have one bag per grid cell in the particles variable, one bag per grid cell in the particlesNextPrivate variable, and one bag per grid cell in the particlesNextShared variable. The total memory for the bag structure is $48 \cdot$ nbCells.

Now let us look at the chunks inside those bags. We recall that N is the total number of particles, nbCells the number of cells, nbThreads the number of threads and chunkSize the number of particles that can be stored in a chunk. We additionally denote by:

- $N_{current}$ the number of particles still in the variable particles
- N_{next} the number of particles already in the variables particlesNext{Private, Shared}
- $C_{current}$ the number of chunks still in the variable particles
- C_{next} the number of chunks already in the variables particlesNext{Private, Shared}

We want to show that $C_{current} + C_{next} \leq \lceil \frac{N}{\mathsf{chunkSize}} \rceil + 4 \cdot \mathsf{nbCells} + \mathsf{nbThreads}$. To that end, we use the fact that at any point in the simulation, $N_{current} + N_{next} = N$. A particle is either still in particles, either it has already been processed and is now in particlesNextPrivate or particlesNextShared. We now look at the number of chunks:

- first, notice that in the C_{next} chunks already in the variables particlesNext{Private, Shared}, there are at most $2 \cdot \mathsf{nbCells}$ non-full chunks (when a thread adds a particle in a bag, it uses the last chunk of this bag until it is full, thus only the last chunk can be non-full, see `bag_push_concurrent` and `bag_push_serial` in Listing 5.5). The other (full) chunks contain chunkSize particles. We obtain:

$$\mathsf{chunkSize} \cdot (C_{next} - 2 \cdot \mathsf{nbCells}) \leq N_{next}$$

- then, notice that in the $C_{current}$ chunks still in the variable particles, there are at most $\mathsf{nbThreads} + 2 \cdot \mathsf{nbCells}$ non-full chunks (the chunk which is being processed by each thread, and the non-full chunks coming from the previous iteration when merging bags, see the previous point). The other (full) chunks contain chunkSize particles. This gives us:

$$\mathsf{chunkSize} \cdot (C_{current} - \mathsf{nbThreads} - 2 \cdot \mathsf{nbCells}) \leq N_{current}$$

Summing up those two inequalities, we get:

$$\mathsf{chunkSize} \cdot (C_{current} + C_{next} - \mathsf{nbThreads} - 4 \cdot \mathsf{nbCells}) \leq N_{current} + N_{next} = N$$

which gives

$$\begin{aligned} C_{current} + C_{next} &\leq \frac{N}{\mathsf{chunkSize}} + \mathsf{nbThreads} + 4 \cdot \mathsf{nbCells} \\ &\leq \left\lceil \frac{N}{\mathsf{chunkSize}} \right\rceil + \mathsf{nbThreads} + 4 \cdot \mathsf{nbCells} \end{aligned}$$

Whatever happens during a simulation, a total number of chunks of $\lceil \frac{N}{\mathsf{chunkSize}} \rceil + 4 \cdot \mathsf{nbCells} + \mathsf{nbThreads}$ is thus enough. □

We next describe the treatment of the charge density and the electric field (ρ and E).When processing particles from one cell, the algorithm first reads from memory the values of the electric field on the 8 corners of that cell (line 9). Importantly, thanks to the strict-binning approach, this data needs only to be loaded once from memory. As particles are processed and moved to their target cells, the charge of each particle is accumulated (line 22) into the array ρNext, which, at the end of the time step, is used to update E for the next iteration. We exploit a recently-proposed, ingenious technique allowing to accumulate the charge on the 8 corners using SIMD instructions [87]. Concretely, the array ρNext involves some amount of redundancy: for each cell, 8 values are stored adjacently in memory, describing the charge on the 8 corners of that cell. At the end of a time step, the charge at a grid point is computed by summing the values associated with the 8 cells that have this grid point as one of their corners (line 33).

We considered two different possibilities for updating ρNext. The first possibility is to decompose ρNext into a *private* array and a *shared* array, just like we do for bags of particles. In this approach, only the deposit of the charge of fast-moving particles triggers atomic operations; for all others particles, we can use SIMD operations. The second possibility is to decompose ρNext

into nbThreads arrays. In this approach, each thread has exclusive access to its charge array, so all accesses use SIMD operations. The downside is a slight increase in the memory footprint, and in the time needed to sum up the values. However, under our assumption of a reasonably large number of particles per cell, these additional costs in memory and in time are tiny in front of the gains. Thus, we opted for the latter approach.

Under the assumption of (at least) hundreds of particles per cell in average, the operations for manipulating chunks (following pointers, pushing/popping in free lists) and for manipulating per-cell information are all well amortized. Overall, the kernel of our algorithm is not far from optimal in terms of memory transfers.

Optimization when particles move at most one cell per time step. For simulations whose physical parameters ensure that movement is restricted to immediate neighboring cells (*e.g.*, [47, 59]), we can optimize our algorithm by removing the shared bags altogether. In this case, our algorithm requires only $\lceil \frac{N}{\mathsf{chunkSize}} \rceil + 2 \cdot \mathsf{nbCells} + \mathsf{nbThreads}$ chunks, and does not need any atomic insertion operation. Likewise, ρNext can be stored in a single array (indexed by cells and by corners).

5.5.3 Performance Results

To assess correctness and performance of our implementation, we considered two classical test cases: a 3d Landau-damping simulation and a 2d3v electron hole simulation. Section 7.1 presents details on these experiments, and argues that the numerical results produced by our simulation match the expected results. In the remaining of this section, we discuss performance results.

Our experiments were conducted on the A3 partition of the Marconi supercomputer (see Table 5.1), on which we were granted the use of 64 nodes with 2 sockets each. Each socket is an Intel Xeon Platinum 8160 @ 2.1 GHz (Skylake), with 96 GB of RAM, 6 memory channels, and 24 cores. Our `C` code was compiled using Intel C Compiler 17.0.4, and the FFTW3 library [149] for the Poisson solver.

The algorithm depends on two parameters. First, we use tiles of size $2 \times 2 \times 2$ for the coloring. Tiles of size $4 \times 4 \times 4$ lead to similarly good performance. Using larger cubic tiles of side TILE_SIZE degrades performance, especially when NUMBER_CELLS $\equiv 1 \bmod$ TILE_SIZE (where NUMBER_CELLS $\in \{ncx, ncy, ncz\}$), which leads to tiny tiles of size 1 that cause too much imbalance. Second, we use chunkSize $= 256$ for the chunk capacity. Larger values of chunkSize increase the space usage and do not reduce the execution time. Smaller values of chunkSize increase the execution time overheads: +12% for chunkSize $= 128$, and +52% for chunkSize $= 64$. Note that, for chunkSize $= 256$, the memory "slack", which is equal to $4 \cdot \mathsf{nbCells} \cdot \mathsf{memoryOf(chunk)}$, represents in the Landau-damping simulation only 13% of the amount of memory strictly required for representing the particles.

In summary, for simulations with sufficient particle density, there exists values of chunkSize, such as 256, that suffice to properly amortize the cost of following pointers indirections between chunks, and at the same time allow fitting close to the maximal number of particles that the hardware can possibly accommodate — in other words, allowing to implement a strict-binning algorithm that achieves both time and space efficiency.

Achieved throughput. For the end-user of a simulation, the metric that matters is the number of particles processed per second. Our implementation achieves:

- 740 million particles per second (30.8 million per second per core) in the 3d Landau-damping simulation, where 31% of the particles change cell at each iteration;
- 910 million particles per second (37.9 million per second per core) in the 2d3v electron hole simulation, where 32% of the particles change cell at each iteration.

Analysis in the roofline performance model. As argued in Section 5.5.2, our algorithm performs not far from the minimal number of memory operations — a key feature for PIC simulations hit by the memory bandwidth bottleneck. With this property in mind, it is interesting to compare the memory bandwidth achieved by our algorithm against the capacity of the hardware. Consider the Landau-damping simulation. The memory bandwidth achieved is 53.6 GB/s[16]. The *theoretical peak* advertised by the manufacturer is 127.99 GB/s. The Stream benchmark [162], which aims at evaluating the *practical peak* using a few microbenchmark programs, and which is commonly used as a baseline, provides the measure 98.2 GB/s. Our algorithm thus achieves 42% of the theoretical peak and 55% of the practical peak bandwidth. Reaching higher percentage in a PIC simulation appears to be very challenging.

> “I could only run the experiments on a lower-performance node than what is presented in the paper, however, memory bandwidth results are consistent with what reported in the paper, as my experiment reached [...] 52% of the available bandwidth.
>
> Anonymous referee, reviewing the artifacts from our last article [205, Section 3]”

The code and scripts for reproducing our bandwidth measurements on other architectures are available in [206]. Let us note that the scripts provided do not allow hyper-threading, although we noticed that it is beneficial, see Chapter 8.

Our algorithm is memory bound. In general, an algorithm may be *compute bound* (i.e. limited by the number of floating-point operations per second) or *memory bound* (i.e. limited by the number of bytes per second transfered from main memory) depending on its *operational intensity*, defined as the number of operations performed divided by the number of bytes moved from or to the main memory. We computed the operational intensity of the 3d implementation by counting the number of floating point operations per particle (79 operations in single-precision and 65 in double-precision, which leads to 209 operations when normalized to single-precision), and counting the number of bytes used to represent a particle (36 bytes, plus 0.25 byte to account for chunk headers)[17]. We thus derive that our 3d implementation has an operational intensity equal to $209/(2 \cdot 36.25) \approx 2.9$. Similarly, we computed the operational intensity for the 2d3v implementation to be $114/(2 \cdot 32.25) \approx 1.8$.

Figure 5.17 represents the bounds on computation and memory bandwidth, in a chart showing the operational intensity on the x-axis, and the computation performance on the y-axis [180]. Note that both axes are log-scale. The computation bound is an horizontal line, at 1 612 GFlops/s (billion floating-point operations per second), a figure provided by the hardware manufacturer. The theoretical and practical memory bounds (bytes/s) are diagonal lines, because the bound in performance (flop/s) is equal to the operational intensity (flop/byte) multiplied by the memory bandwidth (bytes/s). Each diagonal line meets the horizontal line at the point of break-even between memory bound and compute bound.

Efficient processing of fast-moving particles. In addition to being memory efficient, our algorithm also benefits from another key feature not found in prior strict-binning algorithms: fast-moving particles are handled efficiently within the main parallel loop. For a particle moving more than half a tile away, we require only one extra atomic operation. Moreover, the contention associated with this atomic operation is relatively limited. Indeed, for two atomic operations to be issued on the same memory cell at the “same time” (i.e., close enough in time

[16] The bandwidth is obtained by multiplying the size of a particle (36 bytes, plus $\frac{64}{\text{chunkSize}}$ bytes to account for chunk headers) by the number of particles processed per second (740 million), and by a factor 2 (one read plus one write). Comparisons with other implementations are shown in Section 3.3.

[17] Counting the number of bytes moved per particle simply as the size of a particle is correct because all particles are read from and written to the main memory at each iteration; there is essentially no cache reuse for particle data between iterations.

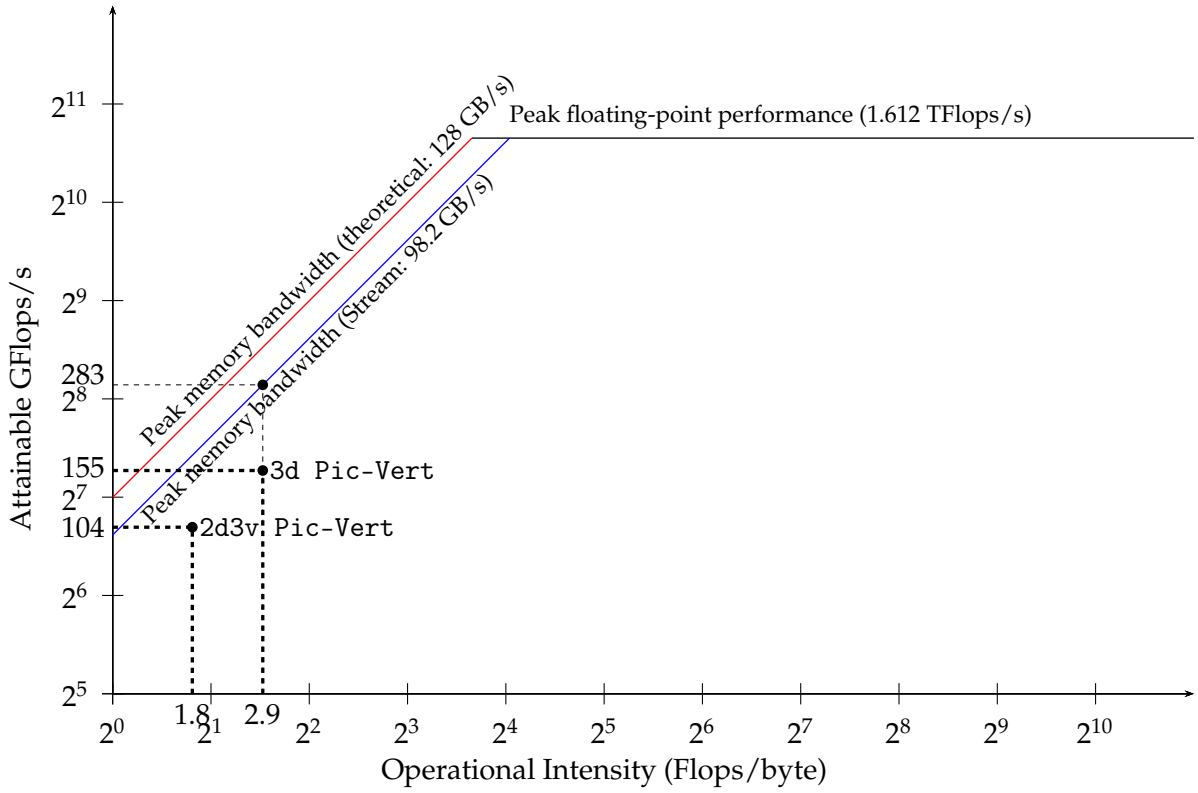


Figure 5.17 – Analysis of performance in the roofline model. Landau damping 3d3v test case in (7.4) and (7.5). 2d3v electron hole test case in (7.6) and (7.7). Variant 2 in Table 5.3.

for a race on the cache line to occur), it must be the case that two particles taken from two distinct tiles spaced away by at least one full tile are moving towards the same cell of a third tile, at the "same time". Thus, the performance of our algorithm should be relatively independent form the particle velocities.

To empirically evaluate the impact of fast-moving particles, we consider a simulation in which we artificially varied the initial distribution of particle velocities. To that end, we manually tuned these distributions in such a way as to obtain several test cases with increasing number of fast-moving particles. Each test case is reflected by a column from Table 5.7. More specifically, the three first rows show the percentage of particles that move away from 1, 2 or 3 cells from their current grid cell at each time step (no particle move further away). Initial particle velocities in these experiments follow the sum of two Gaussian distributions, like in the bump-on-tail instability. Initial particle positions are taken uniformly in $[0; 4\,\pi)^3$. More precisely, the initial distribution function for each row is given by the following formula and values of parameters (see [206] to easily reproduce the results):

$$\begin{aligned} &f_0(\mathbf{x}, vx, vy, vz) = g(vx) \cdot g(vy) \cdot g(vz), \\ &\text{with } g(w) = \frac{1}{\sqrt{2\,\pi}\,v_{th}} \left(p_{\text{drift}} \exp\left(-\frac{(w - v_{\text{drift}})^2}{2 v_{th}^2} \right) + (1 - p_{\text{drift}}) \exp\left(-\frac{w^2}{2 v_{th}^2} \right) \right). \end{aligned} \tag{5.1}$$

- Row 1: $p_{\text{drift}} = 0.\quad ; v_{\text{drift}} = \quad 0.; v_{th} = 0.339$
- Row 2: $p_{\text{drift}} = 0.02; v_{\text{drift}} = 11.; v_{th} = 0.126$
- Row 3: $p_{\text{drift}} = 0.02; v_{\text{drift}} = 13.; v_{th} = 0.178$
- Row 4: $p_{\text{drift}} = 0.02; v_{\text{drift}} = 15.; v_{th} = 0.234$
- Row 5: $p_{\text{drift}} = 0.02; v_{\text{drift}} = 17.; v_{th} = 0.287$
- Row 6: $p_{\text{drift}} = 0.02; v_{\text{drift}} = 20.; v_{th} = 0.355$
- Row 7: $p_{\text{drift}} = 0.02; v_{\text{drift}} = 22.; v_{th} = 0.3585$

Particles that move 1 cell away	8.0%	8.0%	8.0%	8.0%	8.0%	8.0%	8.0%
Particles that move 2 cells away	0	0.7%	1.9%	3.1%	4.3%	5.6%	4.4%
Particles that move 3 cells away	0	0	0	0	0	0.2%	1.4%
Particles pushed atomically (line 21)	0.0%	0.4%	1.0%	1.6%	2.2%	3.1%	3.7%
Slowdown w.r.t. first column	0	0.0%	0.9%	3.8%	4.4%	4.2%	7.0%

Table 5.7 – Impact on performance of increasing the percentage of fast particles. 3d3v test case in (5.1). Variant 2 in Table 5.3.

By instrumenting the code, we measured the number of push operations that trigger an atomic write (line 21 from Figure 5.16). These numbers, relative to the total number of particles, appear in the fourth line of the table: they vary from 0% to 3.7%. The last row of Table 5.7 gives the corresponding slowdown on the total execution time. Figures show that even when the percentage of particles whose move require an atomic operation is as high as 3.7%, the cost of processing these fast moving particles remains fairly limited: +7.0%. In comparison, any alternative algorithm that sequentially processes 3.7% of the particles in a 24-core execution would suffer at least from a +85% slowdown compared with a fully-parallel implementation[18].

Scaling. Although inter-node parallelism is mostly orthogonal to the focus of the present work, we used particle decomposition (see Section 2.6) to scale our algorithm on 128 Skylake sockets (each with 24 cores, 12.3 TB of RAM in total), using one `MPI` process per socket. We simulated Landau-damping with 256 billion particles, achieving a throughput of 89.6 billion particles per second: a 123x speedup with respect to one socket.

Technical note on the coloring scheme. Most of the time, this note can be forgotten, if you have full control over the grid size. However, we are never too careful, and having a code which is valid in every scenario is somehow a good practice.

When NUMBER_CELLS is not an even multiple of TILE_SIZE (where NUMBER_CELLS $\in \{ncx, ncy, ncz\}$), then the tiles at the borders of the concerned direction (or the ones just before the borders, depending on the case) cannot use as much private bags as written in Figure 5.16. In most cases, it will be possible to modify slightly the grid sizes so that this does not happen. For example, if you have tiles of size 5 and in one dimension the number of cells is 128, setting the number of cells to 130 is probably the best solution. To the best of our knowledge, previous works with the coloring scheme all implicitly use grid sizes that are even multiples of the tile size and do not discuss this corner case. Let us nevertheless explain what is the problem and how we can solve it.

Figures 5.18–5.20 show a non-perfect tiling. For the green tiles, there is no problem: as before, borders of width $\lfloor 5/2 \rfloor = 2$ can be used on the full grid. When processing the blue tiles, however, special care is needed. The x-axis starts with the blue color and ends with the blue color. For this axis, it is mandatory to use atomics whenever we leave the tile for the first and last tile. The y-axis ends with a pink tile, but it is too small to enable a border of size 2 for the top and bottom tile. We here chose to use atomics whenever we leave the tile on this direction, too (note that we could have used borders of size 1 for the top of the top tile or for the bottom of the bottom tile, a choice we did not make to keep the symmetry).

As a final remark, we notice on the drawing that we could also choose to have a border of size 3 for half of the tiles on which to use private bags. We did not choose that for the sake of symmetry.

[18] Let t denote the single-core execution time. Assume 3.7% of sequential execution, and 96.3% using 24 cores. The parallel execution time is: $0.037t + 0.963t/24 = 1.85t/24$.

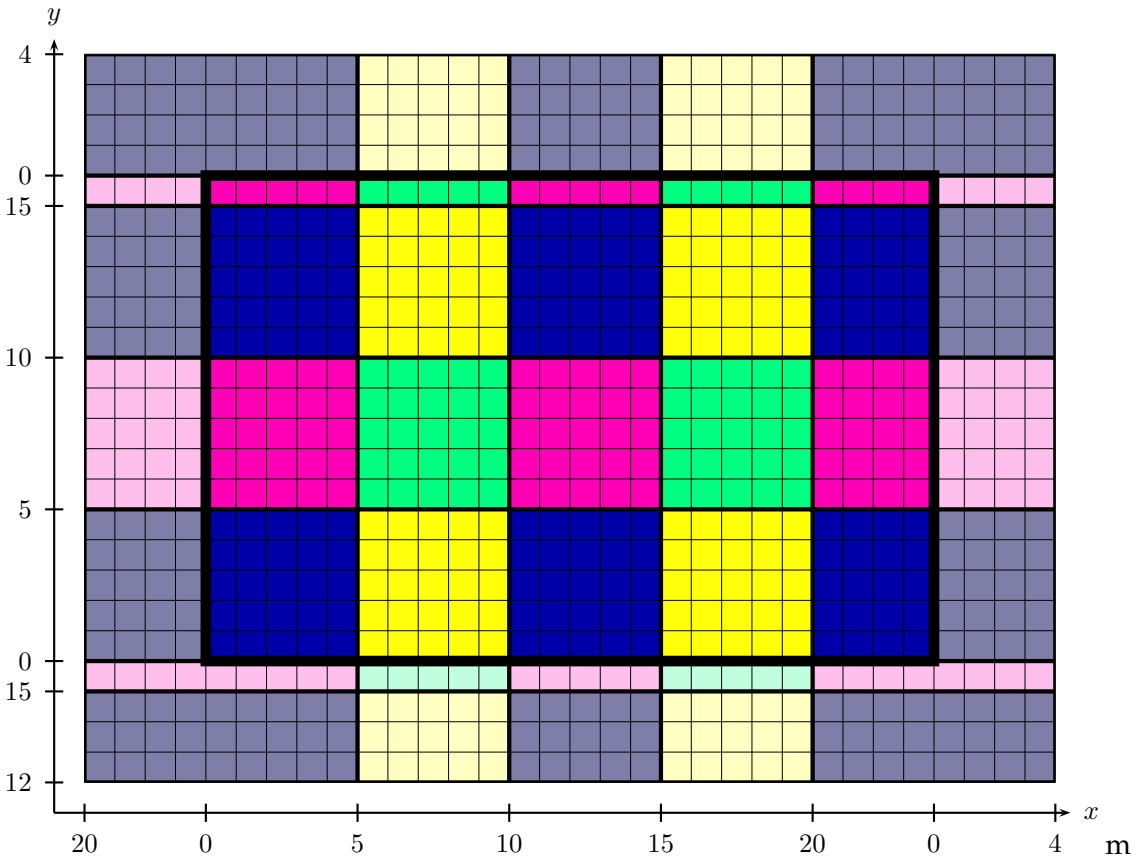


Figure 5.18 – A coloring with 4 colors with 5×5 tiles of a 24×16 grid with periodic boundaries.

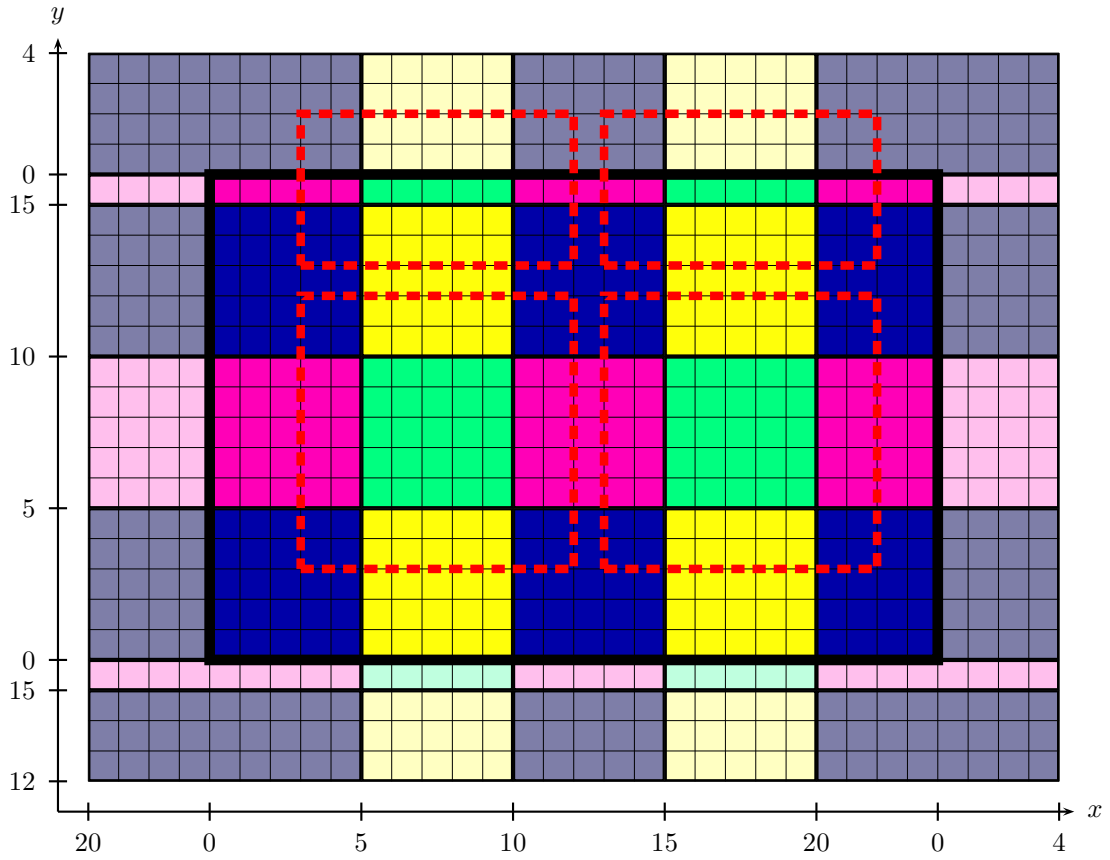


Figure 5.19 – Green tiles can be processed in parallel as in Figures 5.1–5.3: atomics are only needed for particles that move further than half a tile size away (2 cells) from their tile.

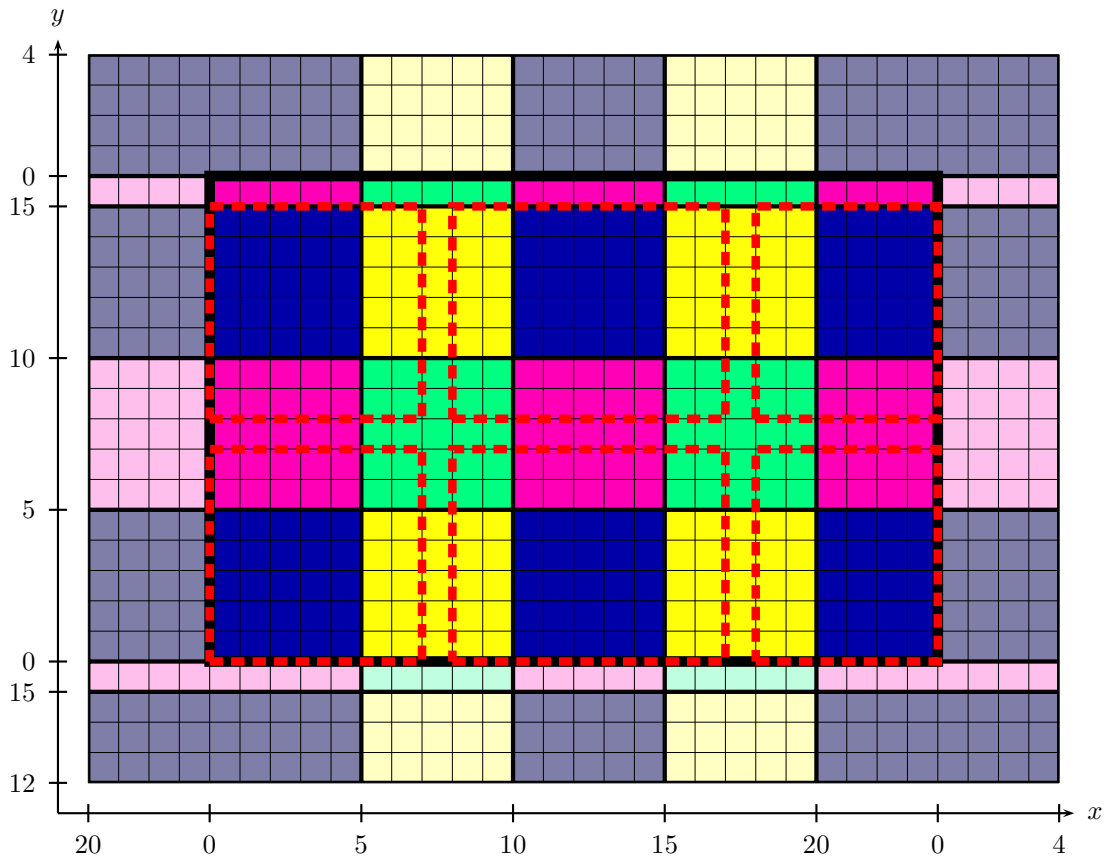


Figure 5.20 – Blue tiles need special care. Here we chose to use atomics whenever we leave the tile on directions where we cannot use half a tile size (2 cells) for borders.

5.5.4 Variant 3 of our Algorithm, Without Coloring

We previously showed results using a coloring scheme. In addition to that, we also developed another algorithm which uses a little more memory but avoids the need to color the tiles, thus saving 7 synchronization points. For simulations involving a large number of particles per cell (thousands or more), an alternative algorithm might deliver improved performance. The idea is to allocate, for each cell, 1 shared bag and a number of private bags equal to the number of tiles that can overlap on this cell (up to 2 in 1d, up to 4 in 2d, up to 8 in 3d). Two parameters can be set: the size of the tiles, and the size of the borders on which tiles can overlap (which has to be at most equal to half the tile size). Figure 5.21 shows an example with 4×4 tiles and a border of 1. The average number of private bags per cell is thus $\frac{6^2}{4^2} = 2.25$. On the figure, we have shown, for one particular tile, the number of private bags that are used in each cell. The cells on corners are at the intersection of 4 "tiles plus borders": they need 4 private bags. The cells on edges, but not on corners, are at the intersection of 2 "tiles plus borders". The cells on the interior are only inside one tile[19]. If we use 4×4 tiles and a border of 2, this time all the cells would need 4 private bags.

In this implementation, all the tiles can be processed in parallel, which can be much more efficient on architectures with a lot of threads (we exhibit 8 times more parallelism than with the coloring scheme). For example, with a grid size of $64 \times 64 \times 64$ and a tile size of $4 \times 4 \times 4$, we have 4 096 tiles in total. With a coloring scheme, it means that for each color, only 512 tiles can be processed in parallel. This is enough work as long as the number of threads do not exceed 50 (a good rule of thumb is that when the number of tasks is less than 10 times the number of threads, we cannot exploit all the parallelism possible due to, *e.g.*, load imbalance).

[19] Which leads to an equivalent computation for the average number of bags per cell: $\frac{4 \cdot 4 + 8 \cdot 2 + 4 \cdot 1}{4^2} = 2.25$.

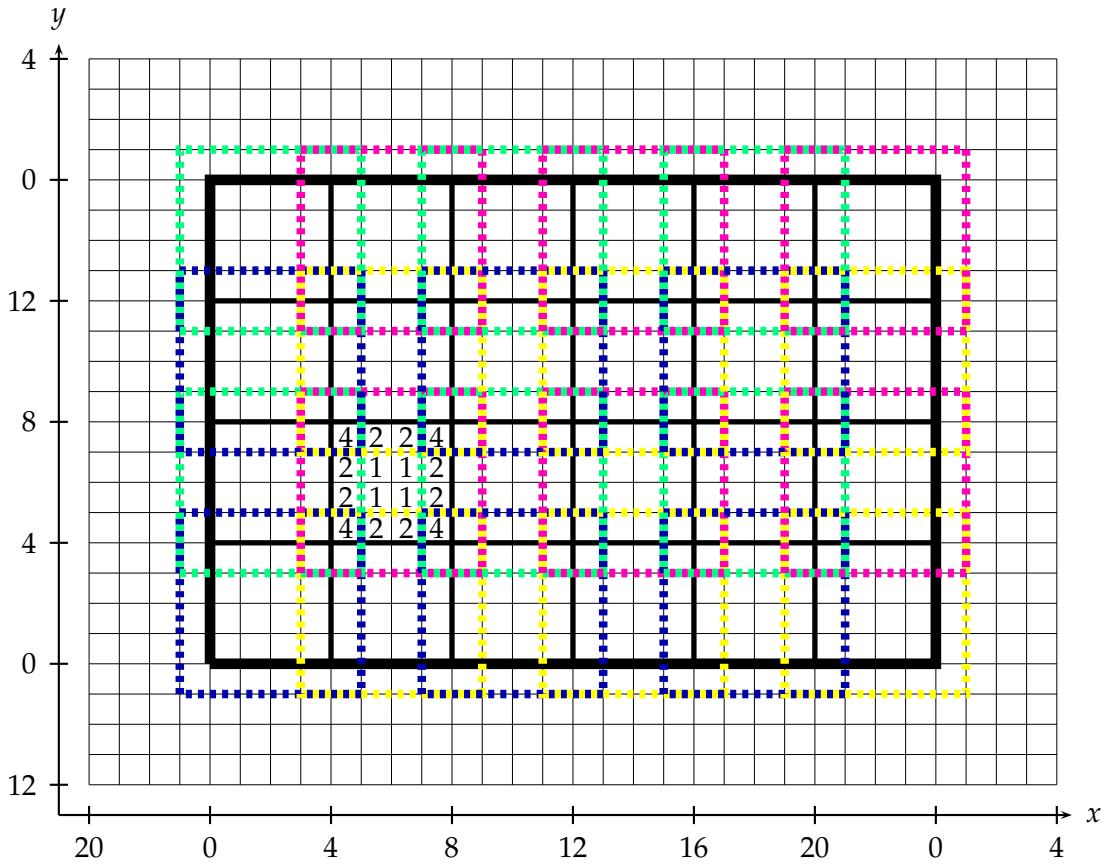


Figure 5.21 – A tiling with 4×4 tiles and borders of 1 of a 24×16 grid with periodic boundaries. All the tiles can be processed in parallel. Atomics are needed when particles move more than 1 cell away from their tile. In other cases, green tiles use the bag with identifier 0, red tiles use the bag with identifier 1, blue tiles use the bag with identifier 2 and yellow tiles use the bag with identifier 3. The numbers inside cells on the figure indicate the number of different private bags that are needed.

```
1 // i{x,y,z}_min is the index of the first cell of the tile on the {x,y,z}-axis
2 ((ix_min % (2 * TILE_SIZE)) != 0) + 2 * ((iy_min % (2 * TILE_SIZE)) != 0) + 4 *
      ((iz_min % (2 * TILE_SIZE)) != 0)
```

Listing 5.6 – Computation of the private bag identifier of a tile.

```
1  int idSharedBag = 8;
2  bag particles[0..nbCells−1]; // Particles by cell, at current time step
3  bag particlesNext[0..8][0..nbCells-1]; // 0..7 for private bags, and 8 for the shared bag.
4  double ρ[0..ncx][0..ncy][0..ncz], E[0..ncx][0..ncy][0..ncz];
5  double ρNext[0..nbThreads−1][0..nbCells−1][0..7]; // 8 corners per cell
6  Foreach time step
7    Parallel Foreach tile // OpenMP parallel
8      Compute idPrivateBag for that tile // See Listing 5.6
9      Foreach cell idCell in that tile
10       Read E[x][y][z], foreach (x, y, z) among the 8 corners of cell idCell
11       Foreach chunk in particles[idCell]
12         Foreach particle in that chunk // SIMD vectorized
13           Update particle velocity
14         Foreach particle in that chunk // SIMD vectorized
15           Update particle position
16           Compute idCellNext, the index of the cell containing the particle
17         Foreach particle in that chunk
18           If the particle moves inside its tile
19           Or it moves no further than borderSize cells from its tile
20             Add the particle into particlesNext[idPrivateBag][idCellNext]
21           Else
22             Atomically add the particle into particlesNext[idSharedBag][idCellNext]
23           Add its charge into ρNext[currentThreadId][idCellNext][..] // SIMD
24       Put a pointer to that chunk into the freelist of the current thread
25   Compute the cumulative sum of the free lists sizes
26   Parallel Foreach idCell in [0..nbCells−1] // OpenMP parallel
27     Set particles[idCell] to particlesNext[idSharedBag][idCell]
28     Set particlesNext[idSharedBag][idCell] to empty, using an empty chunk
29     For idBag in [0..7]
30       Merge particlesNext[idBag][idCell] into particles[idCell]
31       Set particlesNext[idBag][idCell] to empty, using an empty chunk
32   Parallel Foreach (x, y, z) in [0..ncx]x[0..ncy]x[0..ncz] // OpenMP parallel, collapsed
33     Foreach of the 8 pairs (idCell,i) such that (x,y,z) is i−th corner of idCell
34       Foreach idThread in [0..nbThreads−1]
35         ρ[x][y][z] += ρNext[idThread][idCell][i]
36         ρNext[idThread][idCell][i] = 0
37   Compute E from ρ using a Poisson solver and set ρ to 0 // FFTW + OpenMP
```

Figure 5.22 – Variant 3 of our algorithm for the 3d PIC method on multi-core architectures. Changes from Variant 2 in Figure 5.16 are in red.

When using, *e.g.*, Intel KNL with 68 cores that support up to 4 threads per core (272 thread in total), we believe that this alternative algorithm could help.

The pseudo-code of this alternative algorithm is given in Figure 5.22. When processing a tile, each thread begins by computing the identifier of the private bag it can use, at line 8, by the formula in Listing 5.6. In fact this computation is just a bijection between the 8 colors of the coloring scheme and $\{0, \dots 7\}$ — any alternative bijection would also work. The same restrictions on the borders that applied to the coloring scheme also apply here when the tiling is not perfect — see the previous technical note on the coloring scheme.

This alternative algorithm uses more memory: instead of having just two bags per cell for the variable `particlesNext`, it can have up to 9 bags per cell. On Marconi A3, the extra memory was not a problem (we could fit 2.1 billions particles with this alternative algorithm), but the

fact that we have more bags per cell implies that the chunks are less filled, thus we lose more time in indirections. As said in the introduction of this subsection, this algorithm is expected to be superior to the coloring scheme only with a lot of particles per cell and a lot of threads.

Chapter 6

The Semi-Lagrangian Method in 2d with Domain Decomposition

During this thesis, another contribution was the implementation and optimization of the semi-Lagrangian (SL) method in 2d, using domain decomposition. In this chapter, we will first explain what is the semi-Lagrangian method in Section 6.1, then we will explain our design choices and optimization steps. Parts of what is explained in this chapter was presented in a minisymposium of PASC'17[1].

The baseline of our implementation is a previous work in 2d from several colleagues [93]. This baseline, written in `Fortran` inside the library SeLaLib [183], is described in Section 6.2. We first extracted the useful parts from this library and ported them to `C`.

Section 6.3 describes the architectures on which the optimizations were tested and explains the new parallelization strategies that we designed and implemented, in `MPI`. Throughout this chapter, we will denote by P the number of `MPI` processes used for simulations. This section also describes optimizations that could be tested, for future work.

Finally, Section 6.4 concludes the work shown in this chapter.

6.1 Preliminaries to the Semi-Lagrangian Methods

6.1.1 Additional Grid

Semi-Lagrangian methods need to store f values on a grid (as opposed to have particles, *e.g.*, in the Particle-in-Cell method). Because f depends on both $\vec{x}$ and $\vec{v}$, we need a grid of the phase-space for it (for E and ρ we need only a spatial grid). We saw different ways of dealing with the spatial grid in Section 1.2.2. For the velocities, it is different. There is no physical reason to consider periodicity; there is no reason neither to think that the interval of velocities to consider will be easy to retrieve at each time step. But this does not mean that we need a grid for all the possible values in $\mathbb{R}$: we have at least one hard constraint which is the speed of light limitation. A natural idea is thus to mimic the "free boundaries":

- we choose "appropriate" $v_{\min}$ and $v_{\max}$ (we "forget" values outside $[v_{\min}; v_{\max}]$)
- we choose a "small" Δv and store $ncvx = \dfrac{v_{\max} - v_{\min}}{\Delta v}$ different values on the v_x-axis
- for each time step, we only store f values on the grid

[1] Y. Barsamian, and M. Mehrenberger. "Semi-Lagrangian Simulations for Solving 2d2v Vlasov–Poisson Systems (one and two species)". In: *Platform for Advanced Scientific Computing (PASC), Minisymposium "Kinetic Simulations on HPC Platforms for Plasma Physics Applications (3/3): Parallelization and New Hardware"*. 2017.
Slides: http://www.barsamian.am/Slides/slides_2017-06-27.pdf.

Remark: in total, there are $ncx \times ncvx$ different values of f stored for a 1d1v implementation, $ncx \times ncy \times ncvx \times ncvy$ in 2d2v, or $ncx \times ncy \times ncz \times ncvx \times ncvy \times ncvz$ in 3d3v[2].

6.1.2 Splitting

Semi-Lagrangian methods use a splitting method, and the characteristic curves to solve each split equation. In [94], Cheng and Knorr present a way to split the Vlasov–Poisson system (1.5) that has to be solved at each time step, that we recall here:

$$\begin{cases} \dfrac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f + \dfrac{q}{m} \vec{E} \cdot \nabla_{\vec{v}} f = 0 & \text{Vlasov} \\ \nabla_{\vec{x}} \vec{E} = \dfrac{\rho}{\epsilon_0} & \text{Poisson} \end{cases}$$

into two easier systems that have each to be solved for different sub-steps:

$$\begin{cases} \dfrac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f = 0 \\ \nabla_{\vec{x}} \vec{E} = \dfrac{\rho}{\epsilon_0} \end{cases} \tag{6.1}$$

$$\begin{cases} \dfrac{\partial f}{\partial t} + \dfrac{q}{m} \vec{E} \cdot \nabla_{\vec{v}} f = 0 \\ \dfrac{\partial \vec{E}}{\partial t} = \vec{0} \end{cases} \tag{6.2}$$

These systems are easier to solve, because $\dfrac{\partial g}{\partial t} + a \dfrac{\partial g}{\partial x} = 0$ is a movement called *advection*. If a is a constant, it is really simple: when t moves, g retains the same values, translated, as shown in Figure 6.1.

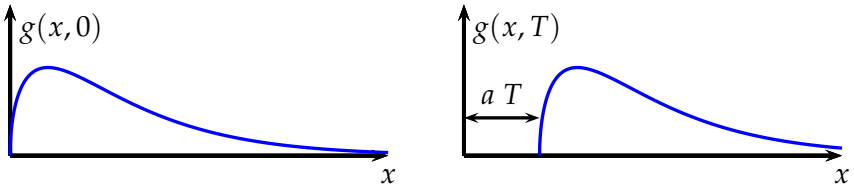


Figure 6.1 – Advection $\dfrac{\partial g}{\partial t} + a \dfrac{\partial g}{\partial x} = 0$.

In other words, g is constant along a well-chosen curve — the characteristic curve whose equation is $g(x(0) + a\ t, t) = g(x(0), 0)$. During each sub-step where we solve system (6.1), we indeed have a constant advection coefficient ($\vec{v}$ does not depend on t nor on $\vec{x}$). During each sub-step where we solve system (6.2), we also have a constant advection coefficient because with this splitting, $\vec{E}$ is constant in time during this sub-step[3]. We advect f along the characteristics given in the system (1.7), which we also give here:

$$\begin{cases} \dfrac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \vec{v} \\ \dfrac{\mathrm{d}\vec{v}}{\mathrm{d}t} = \dfrac{q}{m} \vec{E} \end{cases}$$

There are many ways to arrange the advections on $\vec{x}$ and on $\vec{v}$: we are looking for one that gives as little error as possible. Cheng and Knorr present a scheme that is second order in Δt (i.e. the error is no more than a constant times Δt^2): the Strang splitting [173][4]. Higher-order

[2] ncx, ncy and ncz being the number of points for the spatial grid, as in Section 2.2.1.

[3] If we had the full electromagnetic force and not just the electric part as here, we would have to take additional care for the splitting — *e.g.*, by further splitting the system on each of the velocities directions to have constant advection coefficients.

[4] Solving (6.1) for half a time step, then solving (6.2) for a full time step, and (6.1) for half a time step.

Parameters
$ncx \times ncy$: size of the spatial grid.
$ncvx \times ncvy$: size of the velocities grid.
Δt: time step.
S: number of iterations.
f_0: initial distribution function.
q and m: particle charge and mass.

Variables
$f[ncx][ncy][ncvx][ncvy]$: distribution function.
$\rho[ncx][ncy]$: charge density.
$\mathbf{E}[ncx][ncy]$: electric field.

Algorithm

1	Initialize f following f_0	
2	**For** i from 1 to S	Strang splitting
3	Advection of f on $\vec{x}$ for $\frac{1}{2}\Delta t$	$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f = 0$
4	Compute ρ from f	Integration in $\vec{v}$
5	Compute $\mathbf{E}$ from ρ	Poisson solver
6	Advection of f on $\vec{v}$ for Δt	$\frac{\partial f}{\partial t} + \frac{q}{m}\vec{E} \cdot \nabla_{\vec{v}} f = 0$
7	Advection of f on $\vec{x}$ for $\frac{1}{2}\Delta t$	$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f = 0$

Figure 6.2 – High-level description of the Semi-Lagrangian method.

1	Initialize f following f_0	
1.5	Advection of f on $\vec{x}$ for $\frac{1}{2}\Delta t$	$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f = 0$
2	**For** i from 1 to $S-1$	
4	Compute ρ from f	Integration in $\vec{v}$
5	Compute $\mathbf{E}$ from ρ	Poisson solver
6	Advection of f on $\vec{v}$ for Δt	$\frac{\partial f}{\partial t} + \frac{q}{m}\vec{E} \cdot \nabla_{\vec{v}} f = 0$
7	Advection of f on $\vec{x}$ for Δt	$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f = 0$
8	Compute ρ from f	Integration in $\vec{v}$
9	Compute $\mathbf{E}$ from ρ	Poisson solver
10	Advection of f on $\vec{v}$ for Δt	$\frac{\partial f}{\partial t} + \frac{q}{m}\vec{E} \cdot \nabla_{\vec{v}} f = 0$
11	Advection of f on $\vec{x}$ for $\frac{1}{2}\Delta t$	$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f = 0$

Figure 6.3 – SL pseudo-code, half-advections merged (modifications from Figure 6.2 in red).

splittings exist, *e.g.*, the 6^{th} order splitting of Blanes and Moan [137], or splittings obtained by the triple jump technique [19, Section II.4], or high-order splittings specific to the Vlasov–Poisson system [93]. During each time step of the semi-Lagrangian scheme, there is thus either an advection on $\vec{x}$ or an advection on $\vec{v}$, whose precise coefficients depend on the splitting.

> “ Finally we notice that the horizontal shifting (6.1) by half a time step may be connected with the subsequent horizontal shifting of the next time step. In effect a horizontal and a vertical shifting, by one step each, alternate.
>
> C. Z. Cheng & G. Knorr [94] ”

The pseudo-code for a Strang splitting is shown in Figure 6.2. As remarked in the quote from [94], let us just note that when we do not need the simulation values at each iteration, we can merge the two advections lines 3 and 7: two advections, one after the other, on $\vec{x}$ are equivalent to only one advection. This optimization is shown in Figure 6.3.

6.1.3 Details of an Advection

Let us now focus on advections. Without loss of generality, we can focus on an advection on $\vec{x}$, and suppose that we know, for a particular sub-step, how to compute $\overrightarrow{\Delta x}$ such that we have to evaluate:

$$f(\vec{x}, \vec{v}, \text{after sub-step}) = f\left(\vec{x} - \overrightarrow{\Delta x}, \vec{v}, \text{before sub-step}\right)$$

The general scheme to apply to retrieve this value is given in Figure 6.4. At each sub-step, we have to compute the values of f on the grid. But we then have to solve a problem that

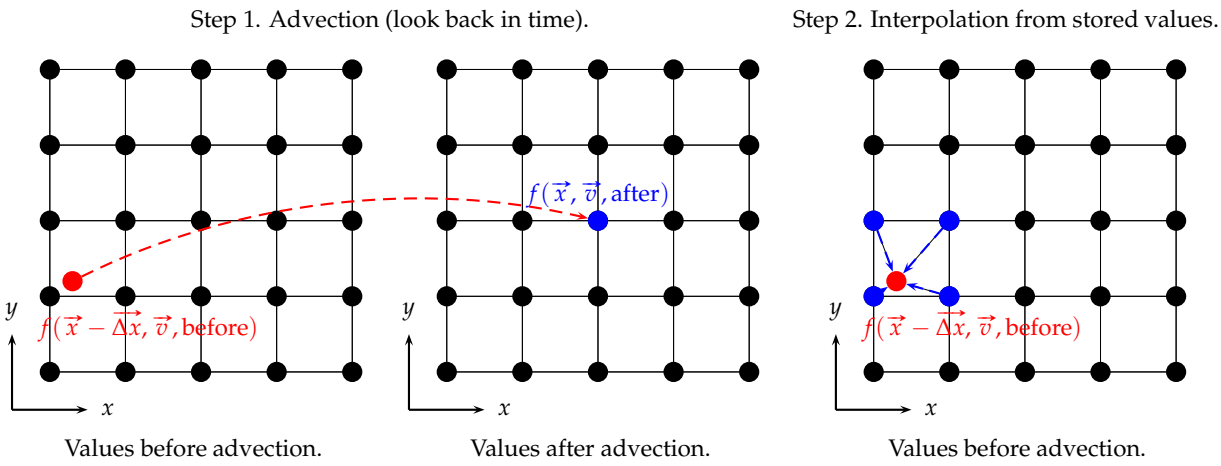


Figure 6.4 – Semi-Lagrangian framework. A value on the grid after the sub-step is equal to a value not necessarily on the grid before the sub-step. This value is interpolated from the neighboring grid values.

comes from the finite numbers of values of f stored: when $\vec{x}$ is on the grid, $\vec{x} - \overrightarrow{\Delta x}$ might not be on it. This is done by interpolating, as shown in step 2 of Figure 6.4. On this figure, we made a simple interpolation that only takes two points in each direction. This would lead to too much error. In practice, we use for example a Lagrangian interpolation of a higher degree. The pseudo-code for the advection and interpolation on $\vec{x}$ is shown in Figure 6.5.

More details on the semi-Lagrangian method can be found *e.g.*, in [28, Deuxième partie] or [103].

6.2 Baseline SeLaLib Implementation

6.2.1 Parallelization Strategy: Transposition

To explain how to parallelize an advection, let us still consider an advection on $\vec{x}$, and a simulation where we have $2^6 = 64$ points per direction: because we work on a 2d2v phase space ($\vec{x} = (x, y)$ and $\vec{v} = (v_x, v_y)$), it means we have $2^{24} = 16\,777\,216$ points in total.

The parallel strategy stems from [91, Section 3] and [95]. During the advection on $\vec{x}$, all the computations can be made in parallel, provided that the needed values are available on the process. The idea for parallelization is simple: because we advect on $\vec{x}$, we only need other values on the position axes, not on the velocity axes. So, we will split the array on the velocity directions, and each process will have in memory a portion of the full array that will be enough for his computation. Thus, the two outermost loops (lines 1–2 on Figure 6.5) are split among the P processes.

The advection on $\vec{v}$ is treated in a similar fashion: this time, the memory has to be split on the position directions. Hence, another efficient parallel algorithm is needed to go from one memory representation (split in $\vec{v}$) to the other (split in $\vec{x}$). This change in memory is a *transposition* of the f array. Figure 6.6 gives an example of this transposition for $P = 4$ processes. The picture is in 2d but we recall that the f array is 4d.

This change in memory is made by `MPI` communications. The parallel pseudo-code for advection is changed as shown in Figure 6.7. In the example given on Figure 6.6, the process 0 would have $k\_min = 0, k\_max = 31, l\_min = 0, l\_max = 31$, the process 1 would have $k\_min = 0, k\_max = 31, l\_min = 32, l\_max = 63$, etc.

Parameters
real $f[0..ncx-1][0..ncy-1][0..ncvx-1][0..ncvy-1]$: the repartition function (before advection).
integer $degree$, the degree of the interpolation.

Output
real $new\_f[0..ncx-1][0..ncy-1][0..ncvx-1][0..ncvy-1]$: the repartition function (after advection).

Algorithm
1 **For** k from 0 to $ncvx-1$
2 **For** l from 0 to $ncvy-1$
3 Compute the displacement (d_x, d_y) and the interpolation coefficients $coeff$.
4 **For** i from 0 to $ncx-1$
5 **For** j from 0 to $ncy-1$
6 $new\_f[i][j][k][l] \leftarrow$ interpolate$(coeff, i-\lfloor d_x \rfloor, j-\lfloor d_y \rfloor, k, l)$

real interpolate(real $coeff[-degree..degree][-degree..degree]$, integer i, integer j, integer k, integer l)
i $value \leftarrow 0$
ii **For** a from $-degree$ to $+degree$
iii **For** b from $-degree$ to $+degree$
iv $value \leftarrow value + coeff[a][b] \times f[(i+a) \mod ncx][(j+b) \mod ncy][k][l]$
v return $value$

Figure 6.5 – Advection on $\vec{x}$ pseudo-code.

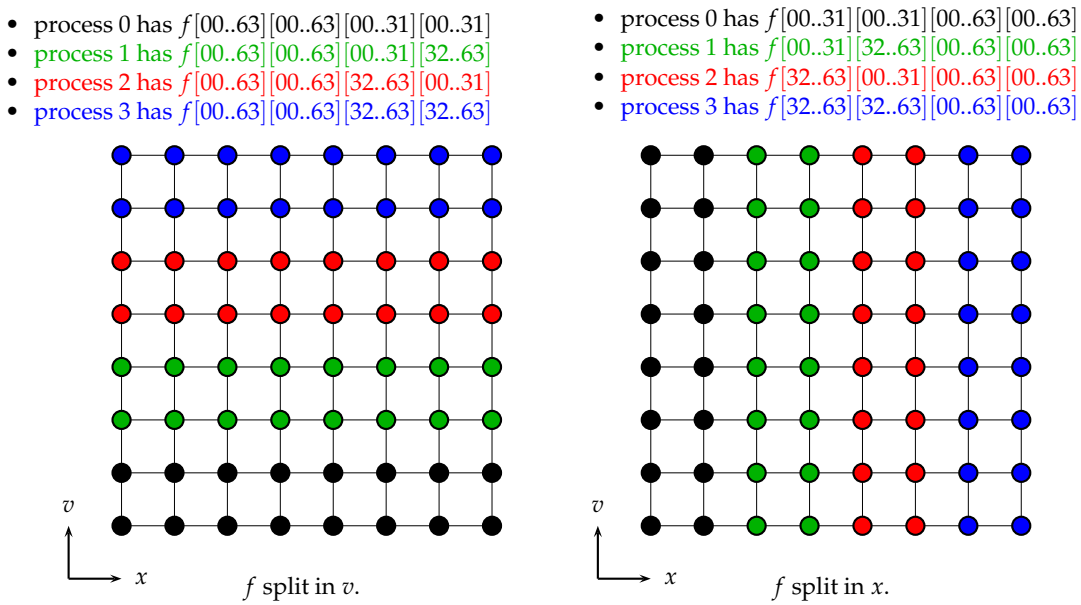
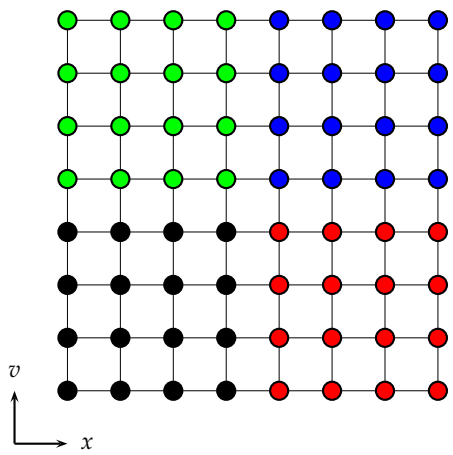


Figure 6.6 – Transposition: change in process memory with a 64^4 grid and $P = 4$ processes.

0 $k\_min, k\_max, l\_min, l\_max \leftarrow local\_MPI\_indices()$
1 **For** k from k_min to k_max
2 **For** l from l_min to l_max
3 Compute the displacement (d_x, d_y) and the interpolation coefficients $coeff$.
4 **For** i from 0 to $ncx-1$
5 **For** j from 0 to $ncy-1$
6 $new\_f[i][j][k][l] \leftarrow$ interpolate$(coeff, i-\lfloor d_x \rfloor, j-\lfloor d_y \rfloor, k, l)$

Figure 6.7 – Parallel advection on $\vec{x}$ pseudo-code (modifications from Figure 6.5 in red).



- process 0 has $f[00..31][00..63][00..31][00..63]$
- process 1 has $f[00..31][00..63][32..63][00..63]$
- process 2 has $f[32..63][00..63][00..31][00..63]$
- process 3 has $f[32..63][00..63][32..63][00..63]$

Figure 6.8 – Domain decomposition with a 64^4 grid and 4 processes.

6.2.2 Other Implementation Choices

The Poisson solver that computes E from ρ uses a Fourier Transform. As discussed in Section 2.4, in practice this is really fast, so this computation is done redundantly by the P processes. Hence, each process needs to have the full E, thus the full ρ: it has to be broadcast just after an advection on x.

The two-dimensional advections are split again leading to one-dimensional advections; this does not introduce additional errors since it concerns linear advection for which this sub-splitting is exact.

The one-dimensional interpolations are performed using high-order Lagrange polynomials (*e.g.*, of order 17).

6.3 A New Domain-Decomposition Implementation

The domain decomposition [96] is another parallel strategy for semi-Lagrangian implementations. The 4d domain is split among the P processes into regular hyper-rectangles, as pictured on Figure 6.8. We will here only consider *regular* decompositions. The ncx values of x, the ncy values of y, the $ncvx$ values of v_x and the $ncvy$ values of v_y are each decomposed into $P^{1/4}$ intervals. The i-th process is in charge of a sub-domain denoted as:

$$[x[i][0]; x[i][1]] \times [y[i][0]; y[i][1]] \times [vx[i][0]; vx[i][1]] \times [vy[i][0]; vy[i][1]].$$

This method has been implemented in various recent works, *e.g.*, [97, 98, 102, 100]. These implementations all share a common drawback: they require assumptions over the number of cells per sub-domain and/or over the time step, to ensure that communications are only done with the neighboring sub-domain (or at most two consecutive sub-domains in the flow direction in the case of [97]).

In [98], it is reported that this constraint is so restrictive that transposition is preferred over domain decomposition. However, in [100, Section 3], it is reported that the constraint $|\alpha| \leq \frac{q}{2}\Delta x$ (where α is the displacement, q the degree of the Lagrange interpolation and Δx the cell size) is rather manageable, and that this strategy is more efficient than the transposition strategy.

The algorithm we sketch in this chapter gets rid of all assumptions, and is shown to be efficient. It makes use of the `MPI 3.0` asynchronous data exchanges.

6.3.1 Naive Domain Decomposition Implementation

Figure 6.9 is a pseudo-code of the original transposition strategy. It is essentially a refined version from Figure 6.5, using the fact that in practice, the 2d advection is split into two 1d

```
Local variable:
buffer[max(ncx, ncy)].

Algorithm:
0  Transpose f (Figure 6.6) Now f[full][full][local][local]
1  Foreach vx
2      Foreach vy
3          Compute the displacement on the x-axis
4          Foreach y
5              buffer ← f[:][y][vx][vy]
6              Foreach x
7                  Interpolate on the x-axis from buffer
8          Compute the displacement on the y-axis
9          Foreach x
10             buffer ← f[x][:][vx][vy]
11             Foreach y
12                 Interpolate on the y-axis from buffer
```

Figure 6.9 – Transposition.

```
Local variable:
buffer[max(ncx, ncy) + Lagrange degree].

Algorithm:
   Recall that f[local][local][local][local]
1  Foreach vx
2      Foreach vy
3          Compute the displacement on the x-axis
4          Foreach y
5              Communicate the needed points
6              Foreach x
7                  Interpolate on the x-axis from buffer
8          Compute the displacement on the y-axis
9          Foreach x
10             Communicate the needed points
11             Foreach y
12                 Interpolate on the y-axis from buffer
```

Figure 6.10 – Domain Decomposition: v1.

advections. It allows to explain how the domain decomposition strategy shown in Figure 6.10 has been derived from this original transposition strategy.

As reported in the previous section, the algorithm has no constraint on the displacement whatsoever. This means that there must be some magic operating behind "Communicate the needed points" (lines 5 and 10 of Figure 6.10). Indeed, there are some technical details behind these communications.

Let us focus on the communication line 5 (the one line 10 is similar). At line 5, we need values from other processes, but we know that those values are on processes whose intervals for y, v_x and v_y in the decomposition are the same. We might have a lot of ISend/IRecv to execute, but not among all the processes — which would lead to $\mathrm{O}\left(P^2\right)$ ISend and IRecv —, only among processes that have common intervals for y, v_x and v_y — which leads to $\mathrm{O}\left(\left(P^{1/4}\right)^2\right) = \mathrm{O}\left(\sqrt{P}\right)$ ISend and IRecv.

Even with this property in mind, the communication could still be time-consuming. Maybe do we need a global communication among these $\mathrm{O}\left(\sqrt{P}\right)$ processes in order to tell each process which values to send to which process? In fact, we do not need it. For all processes needing to communicate, the displacement computed line 3 is the same. It is just the velocity v_x multiplied by the time step Δt. Thus, each process not only knows to which other process he has to ask values... but already to which other processes he has to send values:

- the process in charge of the value x must ask values to processes that are in charge of the values in the range $[x + v_x \cdot \Delta t - d; x + v_x \cdot \Delta t + d + 1]$, where d is the degree of the Lagrange interpolation.
- conversely, for each value x, if a process is in charge of a value in the range $[x + v_x \cdot \Delta t - d; x + v_x \cdot \Delta t + d + 1]$, then it has to send values to the process in charge of x

The algorithm for the data exchange is thus given in Figure 6.11.

There is one last subtlety if we want to handle all the possible cases. As we have seen, the i-th process that handles the values $[x[i][0]; x[i][1]]$ needs the values $[x[i][0] + v_x \cdot \Delta t - d; x[i][1] + v_x \cdot \Delta t + d + 1]$. A special case might happen if the number of values needed is greater than ncx (which will happen if we have not enough processes and the x values have not been split into multiple intervals: if we look back at Figure 6.8, this happens for the y values). In that case, each process has to send multiple times its values to the same process. This is handled by having a different identifier for each communication, and by having loops lines 4 and 9 in

```
1  Foreach process index i_recv on the x communicator (processes that share the same y, vx and vy intervals)
2      Compute the index i_send_first of the process that has x[i_recv][0] + v_x · Δt − d
3      Compute the index i_send_last of the process that has x[i_recv][1] + v_x · Δt + d + 1
4      For i_send from i_send_first to i_send_last
5          If my_index = i_send and my_index ≠ i_recv, then
6              Mark the need for a send to i_recv
7              Launch an ISend to i_recv
8      If my_index = i_recv, then
9          For i_send from i_send_first to i_send_last
10             Mark the need for a receive from i_send
11             If my_index ≠ i_send, then
12                 Launch an IRecv from i_send
13 If there is a need for a receive from my_index, then
14     Remove the need for a receive from my_index
15     Make a local copy of the needed values
16 Wait for completion of the launched ISend/ IRecv
```

Figure 6.11 – Communication algorithm for lines 5 and 10 of Figure 6.10. A given process looks from the point of view of each process, to see if it needs to send data to that process. When it is looking from its own point of view, he also sees from which processes he needs to receive data.

Figure 6.11 be indexed in practice not on the process identifier but on the number of f values to send. As a consequence, it is possible to handle any number of communications between the same couple of processes.

6.3.2 First Communication Optimization

As we have seen, we were able to mimic the algorithm with transposition, by introducing some communications allowing to make copies inside the buffer useful for interpolation.

But when P grows, those communications will involve really small messages. For each communication, we need at most $\frac{ncx}{P^{1/4}}$ values, which can be a really low number. We thus fill our network with `MPI` messages of really small size. This is bad because of the following property:

Property of `MPI` Communication
Smaller `MPI` messages lead to smaller memory bandwidth.
(See, *e.g.*, https://computing.llnl.gov/tutorials/mpi_performance/#MessageSize).

The next idea thus becomes really natural. Because the displacement does not depend on y, we can communicate all the needed values for all the y values at once. This increases the size of the buffer, but is better suited if we want to run our algorithm on a lot of processes. This version is given in Figure 6.12.

Figure 6.13 compares different versions of the algorithm with transposition (one that always stores f as $f[x][y][vx][vy]$, one that always stores f as $f[vx][vy][x][y]$ and one that changes from one representation to the other at each transposition, to minimize strides during the interpolation) to our two algorithms with domain decomposition (the one from Figure 6.10 that communicates 1d sub-arrays of f, and the one from Figure 6.12 that communicates 2d sub-arrays of f).

We did not implement the standard domain decomposition algorithm with a time step restriction: this algorithm would have exactly the same communication patterns as our algorithm and thus similar results when choosing a time step that allows it to work.

If our first naive algorithm was not competitive against the transposition algorithm, when applying this first optimization we obtain runs almost twice as fast as with the transposition algorithm.

```
Local variable:
buffer[max(ncx * (ncy + Lagrange degree),
           ncy * (ncx + Lagrange degree))].

Algorithm:
   Recall that f[local][local][local][local]
1  Foreach vx
2      Foreach vy
3          Compute the displacement on the x-axis
4          Communicate the needed points
5          Foreach y
6              Foreach x
7                  Interpolate on the x-axis from buffer
8          Compute the displacement on the y-axis
9          Communicate the needed points
10         Foreach x
11             Foreach y
12                 Interpolate on the y-axis from buffer
```

Figure 6.12 – Domain Decomposition: v2 (modifications from Figure 6.10 in red).

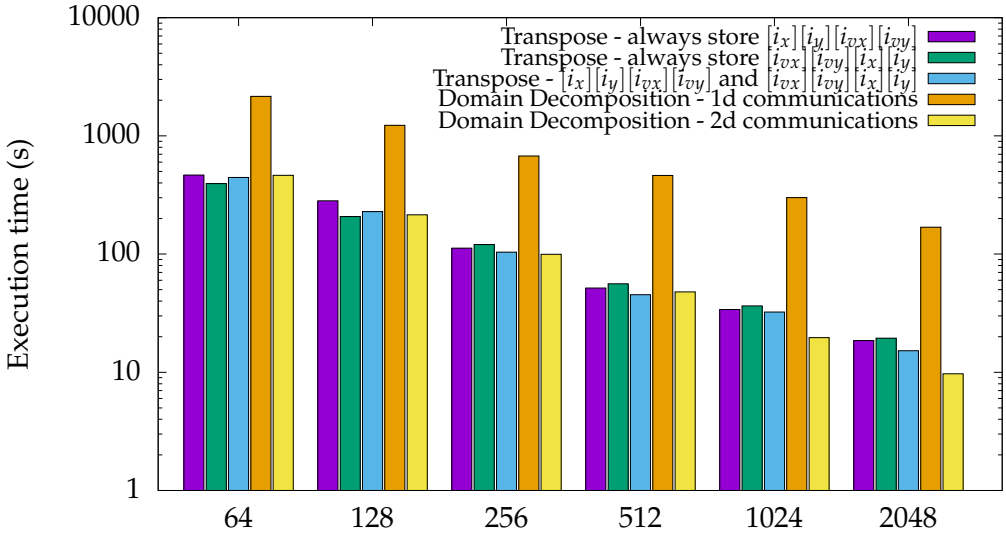


Figure 6.13 – Strong scaling on 2 048 cores (1 process per core): $128^2 \times 512^2$ grid. Two-stream instability test case, 13 iterations with $\Delta t = 0.1$. Architecture: Intel Broadwell EP (2016).

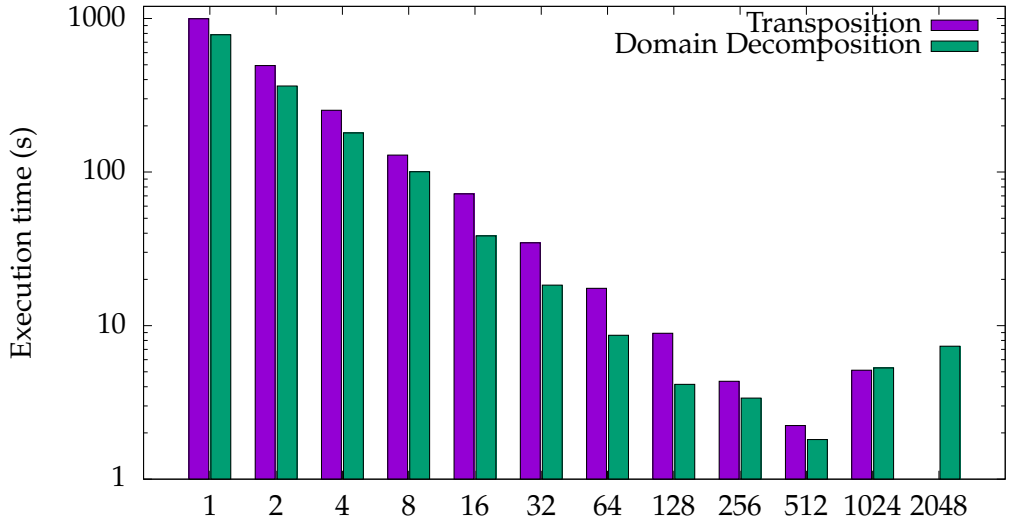


Figure 6.14 – Strong scaling on 2 048 cores (1 process per core): $32^2 \times 512^2$ grid. Two-stream instability test case, 10 iterations with $\Delta t = 0.1$. Architecture: Intel Broadwell EP (2016).

6.3.3 Second Communication Optimization (Future Work)

From now on, we will consider the transposition algorithm that switches from one representation to the other, because it is the fastest one on more processes, and we will consider the domain decomposition algorithm with 2d communications.

One good theoretical property of the domain decomposition is that it can make use of more processes. On a $ncx \times ncy \times ncvx \times ncvy$ grid, an algorithm with transposition cannot use more than $min(ncx \cdot ncy, ncvx \cdot ncvy)$ processes. This is due to the fact that each process needs to have all the $\vec{v}$ values for the advection on $\vec{x}$ (and all the $\vec{x}$ values for the advection on $\vec{v}$). When using domain decomposition, we can use up to $ncx \times ncy \times ncvx \times ncvy$ processes. We will try to see if, in practice, the use of more processes than $min(ncx \cdot ncy, ncvx \cdot ncvy)$ is really profitable. We will thus here use a grid with fewer points.

Figure 6.14 compares the algorithm with transposition to our algorithm with domain decomposition. These results show that up to 512 processes, the implementation with domain decomposition is twice faster. Unfortunately, when using more processes, we face the same problem we already faced before: `MPI` messages become too small, and the communications become the main bottleneck. We also have to acknowledge that for the Poisson solver, there are more communications when using domain decomposition than when using the transposition algorithm. This is due to the fact that there is an additional reduction step when using domain decomposition (among processes that have same the same intervals for x and y but different ones for vx and vy), which is not needed when using the transposition algorithm (a process always has all the values for vx and vy when f is split in $\vec{x}$). Compare Listing 6.1 and Listing 6.2.

It is not straightforward to optimize the communications needed for the Poisson solve, but the communications needed for the advections can be optimized like before, by having 3d communications instead of 2d ones when we want to use more processes. This version is given in Figure 6.15.

If you compare Figure 6.12 and Figure 6.15, what has been done can be summarized in the following set of loop transformations:

- first, a loop splitting, see Figure 6.16. It is not obvious that this transformation is legal. It is due to the fact that the loops on vx and vy (lines 1–2) carry no dependence: each iteration with $vx = i$ and $vy = j$ modifies only the values $f[.][.][i][j]$. We can thus safely

```
1  if (!is_par_x)
2      exchange_parallelizations();
3  // Integration of f_local in v to generate spatial_density_local
4  pos = 0;
5  for (i = 0; i < size_x_par_x; i++) {
6      for (j = 0; j < size_y_par_x; j++) {
7          send_buf[pos] = 0;
8          for (k = 0; k < size_vx_par_x; k++)
9              for (l = 0; l < size_vy_par_x; l++)
10                 send_buf[pos] += f_parallel_in_x[i][j][k][l];
11         send_buf[pos++] *= velocity_mesh.delta_x * velocity_mesh.delta_y;
12     }
13 }
14 // Gather the concatenation of the spatial_density_locals
15 MPI_Allgatherv(send_buf, size_x_par_x * size_y_par_x, MPI_DOUBLE_PRECISION,
       recv_buf, recv_counts, displs, MPI_DOUBLE_PRECISION, MPI_COMM_WORLD);
16 // Rebuild spatial_density from the concatenation of the spatial_density_locals
17 pos = 0;
18 for (int process = 0; process < mpi_world_size; process++)
19     for (i = layout_par_x[process].i_min; i <= layout_par_x[process].i_max; i++)
20         for (j = layout_par_x[process].j_min; j <= layout_par_x[process].j_max;
               j++)
21             spatial_density[i][j] = recv_buf[pos++];
```

Listing 6.1 – 2d Poisson solver with transposition.

```
1  // Integration of f_local in v to generate spatial_density_local
2  pos = 0;
3  for (i = 0; i < size_x_local; i++) {
4      for (j = 0; j < size_y_local; j++) {
5          recv_buf[pos] = 0;
6          for (k = 0; k < size_vx_local; k++)
7              for (l = 0; l < size_vy_local; l++)
8                  recv_buf[pos] += f_parallel[i][j][k][l];
9          recv_buf[pos++] *= velocity_mesh.delta_x * velocity_mesh.delta_y;
10     }
11 }
12 // Reduce for processes that have same (x,y) but different (vx,vy)
13 MPI_Reduce(recv_buf, send_buf, size_x_local * size_y_local, MPI_DOUBLE_PRECISION
       , MPI_SUM, 0, comm_reduce);
14 // Gather the concatenation of the spatial_density_locals
15 if (mpi_comm_reduce_rank == 0)
16     MPI_Gatherv(send_buf, size_x_local * size_y_local, MPI_DOUBLE_PRECISION,
           recv_buf, recv_counts, displs, MPI_DOUBLE_PRECISION, 0, comm_gather);
17 // Broadcast the spatial_density
18 MPI_Bcast(recv_buf, spatial_mesh.num_cell_x * spatial_mesh.num_cell_y,
       MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD);
19 // Rebuild spatial_density from the concatenation of the spatial_density_locals
20 pos = 0;
21 for (int process = 0; process < mpi_comm_gather_size; process++)
22     for (i = layout_4d[world_rank_from_comm_gather_rank[process]].i_min; i <=
           layout_4d[world_rank_from_comm_gather_rank[process]].i_max; i++)
23         for (j = layout_4d[world_rank_from_comm_gather_rank[process]].j_min; j
               <= layout_4d[world_rank_from_comm_gather_rank[process]].j_max; j++)
24             spatial_density[i][j] = recv_buf[pos++];
```

Listing 6.2 – 2d Poisson solver with domain decomposition. Note that the actions reduce (comm_reduce), gather (comm_gather) and broadcast (comm_world) can be replaced by the actions allreduce (comm_reduce) and allgather (comm_gather).

```
Local variable:
buffer[max(ncvx * ncx * (ncy + Lagrange degree),
           ncvy * ncy * (ncx + Lagrange degree))].

Algorithm:
1  Foreach vx
2        Compute the displacement on the x-axis
3        Communicate the needed points
4        Foreach vy
5              Foreach y
6                    Foreach x
7                          Interpolate on the x-axis from buffer
8  Foreach vy
9        Compute the displacement on the y-axis
10       Communicate the needed points
11       Foreach vx
12             Foreach x
13                   Foreach y
14                         Interpolate on the y-axis from buffer
```

Figure 6.15 – Domain Decomposition: v3.

reorder all the iterations, as long as for a given (i, j), all instructions are performed in the same order, which is the case here.

- then, a loop interchange, see Figure 6.17. It is quite clear that this is legal because nothing depends on vx inside the loop body (lines 8–12).
- finally, we apply the same transformation that transformed Figure 6.10 to Figure 6.12: when performing a displacement on the x-axis, nothing depends on vy so we may move the loop-invariant code that computes the displacement up one loop level, and we may also communicate more points all at once, and move this communication up one loop level, see Figure 6.18.

6.3.4 Third and Fourth Communication Optimizations (Future Work)

One last optimization in the communications lies in the fact that, when performing an advection on $\vec{x}$, the displacement is proportional to the velocity. It becomes clear that a process that handles big values of velocities will have a bigger displacement, and thus will probably need to ask a lot of values to the other processes, whereas a process which handles values of velocities near 0 will probably already have all the needed values and will thus need to communicate a lot less. Because we have a barrier at the end of each advection, we thus have to wait a lot for the processes in charge of big velocity values, because they will require more time for communication.

Figure 6.19 illustrates this problem. What is shown is the location of processes to which the processes in the fourth column have to ask values for the displacement on the x-axis. The size of the rectangles is proportional to the number of values a process has to ask. For example (a) on the top row, we see that the process of the fourth column has only a tiny number of values already present locally. It has to ask all the values from its right neighbor (shown in red), and also a lot of values on the next neighbor on the right; (b) on the fourth row, we see that the process of the fourth column already has the majority of values it needs for its computations (shown in green). It only needs to ask some values to the left and right processes.

Thus, the processes on the top and bottom rows will have to wait for a lot of communications before they will be able to compute, whereas the processes on the center rows will be able to compute a lot faster, because they need to ask less values to their neighbors.

```
1  Foreach vx
2      Foreach vy
3          Compute the displacement on the x-axis
4          Communicate the needed points
5          Foreach y
6              Foreach x
7                  Interpolate on the x-axis from buffer
8          Compute the displacement on the y-axis
9          Communicate the needed points
10         Foreach x
11             Foreach y
12                 Interpolate on the y-axis from buffer
```

```
1  Foreach vx
2      Foreach vy
3          Compute the displacement on the x-axis
4          Communicate the needed points
5          Foreach y
6              Foreach x
7                  Interpolate on the x-axis from buffer
7.1 Foreach vx
7.2     Foreach vy
8          Compute the displacement on the y-axis
9          Communicate the needed points
10         Foreach x
11             Foreach y
12                 Interpolate on the y-axis from buffer
```

Figure 6.16 – From Figure 6.12 to Figure 6.15; first transformation: loop splitting.

```
1  Foreach vx
2      Foreach vy
3          Compute the displacement on the x-axis
4          Communicate the needed points
5          Foreach y
6              Foreach x
7                  Interpolate on the x-axis from buffer
7.1 Foreach vx
7.2     Foreach vy
8          Compute the displacement on the y-axis
9          Communicate the needed points
10         Foreach x
11             Foreach y
12                 Interpolate on the y-axis from buffer
```

```
1  Foreach vx
2      Foreach vy
3          Compute the displacement on the x-axis
4          Communicate the needed points
5          Foreach y
6              Foreach x
7                  Interpolate on the x-axis from buffer
7.1 Foreach vy
7.2     Foreach vx
8          Compute the displacement on the y-axis
9          Communicate the needed points
10         Foreach x
11             Foreach y
12                 Interpolate on the y-axis from buffer
```

Figure 6.17 – From Figure 6.12 to Figure 6.15; second transformation: loop interchange.

```
1  Foreach vx
2      Foreach vy
3          Compute the displacement on the x-axis
4          Communicate the needed points
5          Foreach y
6              Foreach x
7                  Interpolate on the x-axis from buffer
7.1 Foreach vy
7.2     Foreach vx
8          Compute the displacement on the y-axis
9          Communicate the needed points
10         Foreach x
11             Foreach y
12                 Interpolate on the y-axis from buffer
```

```
1  Foreach vx
2      Compute the displacement on the x-axis
3      Communicate the needed points
4      Foreach vy
5          Foreach y
6              Foreach x
7                  Interpolate on the x-axis from buffer
7.1 Foreach vy
8          Compute the displacement on the y-axis
9          Communicate the needed points
9.1        Foreach vx
10         Foreach x
11             Foreach y
12                 Interpolate on the y-axis from buffer
```

Figure 6.18 – From Figure 6.12 to Figure 6.15; third transformation.

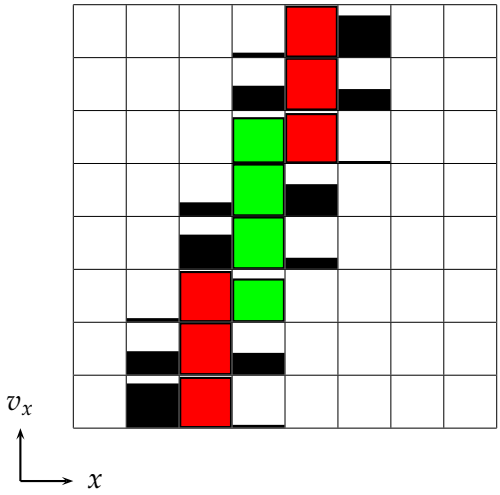


Figure 6.19 – Domain Decomposition: 64^4 grid on $8^4 = 4\,096$ processes. Location of the f values needed for processes who hold x values in the fourth column (sub-timestep of 0.5, velocities in $[-5; 5]$).

Can we avoid this problem? We can think of dynamic domain decomposition. The processes of the top rows of Figure 6.19 could all switch their local domains one tile to the right, in order to be in the same good position as the processes on the middle rows. And processes on the bottom rows could all switch their local domain one tile to the left.

However, we can see on rows 3 and 6 that the gain will not necessarily be tremendous: the green data and the red data are almost equivalent, thus the processes on those rows will probably take longer time even if we switch their tile.

Figure 6.20 shows that it is possible to refine the domain decomposition. In addition to the tile switch idea, we might decompose the domain in velocities with intervals containing more or less values. Processes that have almost as much data locally as they need to receive from neighbors would handle less velocities than the average (velocities for which the green or red rectangle is next to an almost black one), and processes that already have locally most of the data would handle more velocities (velocities for which the green or red rectangle is next to almost white ones). This decomposition can be done at the beginning of the simulation and does not need to be adapted dynamically because the advections are always with the same coefficient. However, we must acknowledge that this solution only works with the Strang splitting. When using other splittings, the coefficients are not always the same at each sub-step, hence, we cannot decompose the domain efficiently for all the sub-steps. It is probable that there is a best configuration anyway, but this best configuration will not be that easy to find.

6.4 Takeaways

In this chapter, we tested various parallel algorithms for the 2d2v semi-Lagrangian method. We designed an algorithm with domain decomposition that gets rid of limitations found in the state-of-the-art. This algorithm is shown to be twice faster than the transposition algorithm.

We identified three ways to further improve our implementation to make it scale better on more processes. It would be interesting to test them, and also to apply our ideas in a 3d3v implementation.

Last but not least, it would be interesting to compare the efficiency of the different algorithms to other implementations, as we did for the Particle-in-Cell method in Table 3.3. An interesting metric for semi-Lagrangian implementations is the number of cells updated by second [207, Section 6].

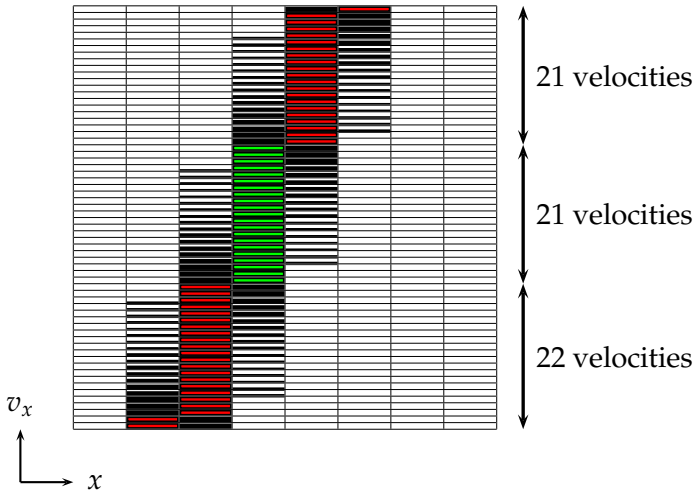


Figure 6.20 – Domain Decomposition: 64^4 grid on 4 096 processes. Location of the f values needed for processes who hold x values in the fourth column (sub-timestep of 0.5, velocities in $[-5; 5]$).

Chapter 7

Numerical Results

This chapter presents numerical results from all the test cases studied during this thesis. It also introduces a framework, called "PICSL" (pronounced "pixel"), which allows to run simulations with an arbitrary number of species, each species being simulated by a PIC or a SL method. Depending on the number of dimensions of the simulation, of the test case considered, and sometimes depending on the species inside a simulation, one or the other of these methods can be better. It is thus comfortable to have a common framework for these two methods.

First, Section 7.1 presents test cases from Chapters 4–6. All those test cases simulate only one species.

Then, Section 7.2 presents the PICSL framework. We introduced this framework in [207]. We first use it to simulate a two-species 1d1v test case from the literature, then we simulate a new one-species 2d2v test case. Those two test cases are validated with mathematical tools, but the details of this validation are left to the interested reader inside this article and not presented here, as it is not our own contribution. A third two-species 2d2v test case is also simulated.

7.1 Test Cases from Previous Chapters

7.1.1 Some References on Landau Damping in 1d

A test case which was the most studied during this thesis is the Landau damping, in 1d, 2d and 3d. As said in the introduction, this test case, when choosing parameters which lead to a linear damping, can be explained theoretically. It is possible to prove (see, *e.g.*, [35, Section 4.4.2]) that the electric field $E(x,t)$ will behave, after a short time, like:

$$4Are^{\gamma t}\sin(kx)\cos(\omega t-\phi).$$

where A is the perturbation, and r, γ, k, ω and ϕ will be explained hereafter. We can then, *e.g.*, match the electric energy $\frac{1}{2}\int_0^L E(x,t)^2\,\mathrm{dx}$ against (7.1).

$$\frac{1}{2}(4Ar)^2\frac{\pi}{k}e^{2\gamma t}(0.5+0.5\cos(2(\omega t-\phi))). \tag{7.1}$$

Vlasov gave an approximation of ω, the frequency of the waves that appear in a plasma (known as space-charge waves, electrostatic waves or Langmuir waves), in [178, Equation 49]:

$$\omega^2=1+3k^2$$

We can then deduce the following formula[1], which can be found in [157, Equation 16]:

$$\omega=1+\frac{3}{2}k^2$$

[1] By performing a Taylor expansion: when x is small, $(1+x)^a=1+ax+o(x)$.

k	ω	γ	r	ϕ
0.2	1.0640	-0.00005510	1.129664	0.00127377
0.3	1.1598	-0.012623	0.63678	0.114267
0.4	1.2850	-0.066133	0.424666	0.3357725
0.5	1.4156	-0.15336	0.3677	0.536245

Table 7.1 – Impact of k on Landau damping.

k	ω	γ	r	ϕ
$\frac{\pi}{11}$	1.14329890862	-0.00846641513031	0.682419563231	0.0856428742970
$\frac{\sqrt{2}}{2}$	1.68289327433	-0.402080551005	0.296396520576	0.857725164340
$\frac{\sqrt{3}}{2}$	1.88197206516	-0.634778971007	0.274195506626	1.04424333451

Table 7.2 – Impact of k on Landau damping.

Landau then gave an approximation of γ, the damping, in [157, Equation 17]:

$$\gamma = -\sqrt{\frac{\pi}{8}}\frac{1}{k^3}\mathrm{e}^{-\frac{1}{2k^2}}$$

This formula can also be found, *e.g.*, in [5, Section 5-15, Equation 2]. A little refinement of the computation gives us the following formula, which can be found in [26, Equation 1.9.7]:

$$\gamma = -\sqrt{\frac{\pi}{8}}\frac{1}{k^3}\mathrm{e}^{-\frac{1}{2k^2}-\frac{3}{2}}$$

Some more refinements of these approximations (more accurate when $k < 0.3$) can be found in [163, Equations 21 & 24]:

$$\begin{cases} \omega = 1 + \dfrac{3}{2}k^2 + \dfrac{15}{8}k^4 + \dfrac{147}{16}k^6 \\ \gamma = -\sqrt{\dfrac{\pi}{8}}\left(\dfrac{1}{k^3} - 6k\right)\mathrm{e}^{-\frac{1}{2k^2}-\frac{3}{2}-3k^2-12k^4} \end{cases}$$

Some further refinements (even more accurate when $k < 0.6$) can be found in [80, Equation 53][2]:

$$\begin{cases} \omega = 1 + \dfrac{3}{2}k^2 + \dfrac{15}{8}k^4 + \dfrac{147}{16}k^6 + 736.437k^8 - 14729.3k^{10} \\ \qquad + 105429k^{12} - 370151k^{14} + 645538k^{16} - 448190k^{18} \\ \gamma = -\sqrt{\dfrac{\pi}{8}}\left(\dfrac{1}{k^3} - 6k - 40.7173k^3 + 3900.23k^5 - 2462.25k^7 - 274.99k^9\right) \\ \qquad \mathrm{e}^{-\frac{1}{2k^2}-\frac{3}{2}-3k^2-12k^4-575.516k^6+3790.16k^8-8827.54k^{10}+7266.87k^{12}} \end{cases}$$

Those formulas only bring approximate results, and only for small values of k. Therefore, the best way of obtaining ω and γ values is to numerically find them, by finding the zeros of a particular function. For example, using formulas from [26] for $k = 0.1$ give 3.3% error for ω and 61% error for γ, see [163]. Computing the zeros numerically has been done, *e.g.*, in [140] (for values of k from 0.25 to 2.0, every 0.05) and in [35, Section 4.4.2] (more accurate values for $k = 0.2, 0.3, 0.4$ and 0.5 together with values for r and ϕ, see Table 7.1).

During this thesis, we computed some more values, see Table 7.2. Those zeros were computed thanks to the library ZEAL [156] and thanks to Maple [188].

[2] With a typo in the formula for ω_i: one should read $+3900.23k^5$ instead of $-3900.23k^5$.

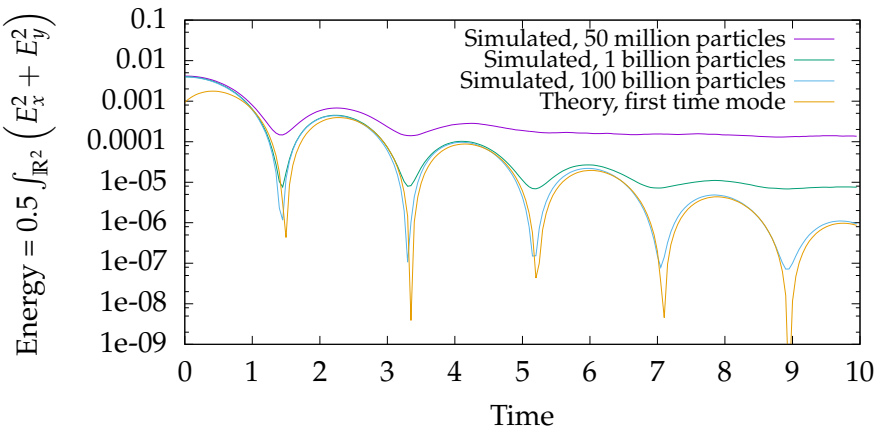


Figure 7.1 – Time evolution of the electric energy, Landau damping 2d2v test case in (7.2) and (7.3).

7.1.2 A 2d2v test case: Landau damping

Description of the equations and initial conditions

We consider a classical Landau damping test case [5, 23]. We look for f satisfying

$$\begin{cases} \partial_t f + \mathbf{v} \cdot \nabla_{\mathbf{x}} f - \mathbf{E} \cdot \nabla_{\mathbf{v}} f = 0, \\ -\Delta_{\mathbf{x}} \Phi = 1 - \int_{\mathbb{R}^2} f \, \mathrm{d}v_x \, \mathrm{d}v_y, \\ -\nabla_{\mathbf{x}} \Phi = \mathbf{E}, \end{cases} \tag{7.2}$$

with initial function as in [148] (the spatial domain is $\Omega = [0, 4\pi)^2$):

$$f(0, \mathbf{x}, \mathbf{v}) = \left(1 + 0.01 \cos\left(\frac{x}{2}\right) \cos\left(\frac{y}{2}\right)\right) \frac{1}{2\pi} \mathrm{e}^{-\frac{|\mathbf{v}|^2}{2}}. \tag{7.3}$$

Numerical results

Figure 7.1 represents the evolution of the electric energy. A grid size of 256×256 is chosen, with a time step of 0.05 and three different values for the number of particles (50 million particles, 1 billion particles and 100 billion particles). With our notations, this test case corresponds to $A = 0.01$ and has a dominant mode $k = \frac{\sqrt{2}}{2}$. We see on the figure that the decay slope of the electric energy is in accordance with the theoretical values $\omega = 1.68289$ and $\gamma = -0.402080$ obtained from the dispersion analysis, see Table 7.2. Here, we fitted the values for r and ϕ.

We see on the figure the influence of the number of particles on the simulation: to accurately retrieve the values of the electric energy, we need 100 billion particles. On this kind of test cases, another mean of reducing the noise should probably be used.

Figure 7.2 represents the conservation of the total energy. We plot the relative error across time and see that the total error is well conserved, as expected.

7.1.3 A 3d3v test case: Landau damping

Description of the equations and initial conditions

We consider a classical Landau damping test case [5, 23]. We look for f satisfying

$$\begin{cases} \partial_t f + \mathbf{v} \cdot \nabla_{\mathbf{x}} f - \mathbf{E} \cdot \nabla_{\mathbf{v}} f = 0, \\ -\Delta_{\mathbf{x}} \Phi = 1 - \int_{\mathbb{R}^3} f \, \mathrm{d}v_x \, \mathrm{d}v_y \, \mathrm{d}v_z, \\ -\nabla_{\mathbf{x}} \Phi = \mathbf{E}, \end{cases} \tag{7.4}$$

with initial function as in [78] (the spatial domain is $\Omega = [0, 22)^3$):

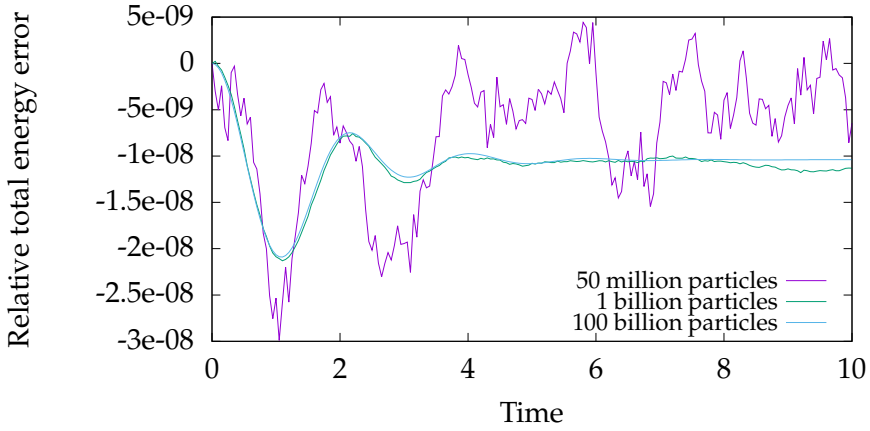


Figure 7.2 – Conservation of the total energy, Landau damping 2d2v test case in (7.2) and (7.3).

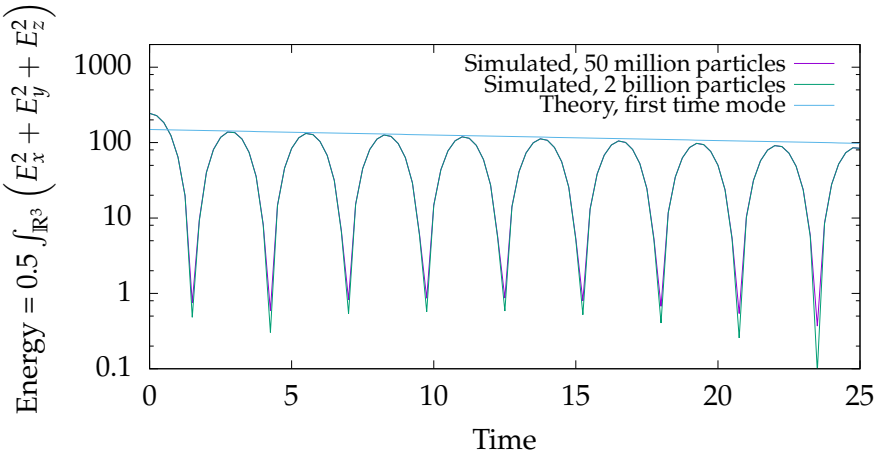


Figure 7.3 – Time evolution of the electric energy, Landau damping 3d3v test case in (7.4) and (7.5).

$$f_0(x,y,z,\mathbf{v}) = \frac{1}{(2\pi)^{3/2}} \mathrm{e}^{-\frac{|\mathbf{v}|^2}{2}} L(x)L(y)L(z) \quad \text{with } L(w) = 1 + 0.05\cos\left(w\frac{\pi}{11}\right) \tag{7.5}$$

Numerical results

Figure 7.3 represents the evolution of the electric energy. A grid size of $64 \times 64 \times 64$ is chosen, with a time step of 0.05 and two different values for the number of particles (50 million particles and 2 billion particles). With our notations, this test case corresponds to $A = 0.05$ and has a dominant mode $k = \frac{\pi}{11}$. We see on the figure that the decay slope of the electric energy is in accordance with the theoretical value $\gamma = -0.008466$ obtained from the dispersion analysis, see Table 7.2.

We see on the figure that contrary to the 2d2v Landau damping test case, a low number of particles suffices to accurately retrieve the values of the electric energy. It is because the initial function is mostly the superposition of 1d functions, and not a "real" 3d one.

Figure 7.4 represents the conservation of the total energy. We plot the relative error across time and see that the total error is well conserved, as expected.

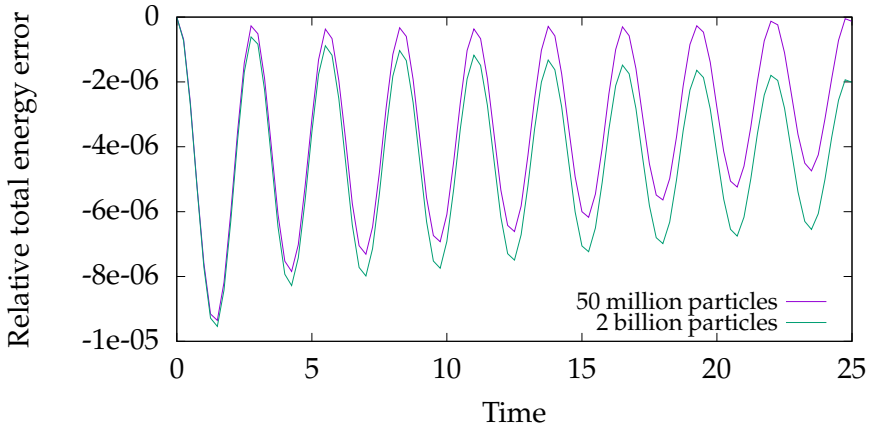


Figure 7.4 – Conservation of the total energy, Landau damping 3d3v test case in (7.4) and (7.5).

7.1.4 A 2d3v test case: Electron hole

Description of the equations and initial conditions

We consider a more complex test case proposed by Muschietti *et al.* [165]. We look for f satisfying

$$\begin{cases} \partial_t f + \mathbf{v}\cdot\nabla_{\mathbf{x}} f - (\mathbf{E}+\mathbf{v}\times\mathbf{B})\cdot\nabla_{\mathbf{v}} f = 0, \\ -\Delta_{\mathbf{x}}\Phi = 1 - \int_{\mathbb{R}^3} f \,\mathrm{d}v_x\,\mathrm{d}v_y\,\mathrm{d}v_z, \\ -\nabla_{\mathbf{x}}\Phi = \mathbf{E}, \end{cases} \tag{7.6}$$

The external magnetic field $\mathbf{B} = B_0\mathbf{e_x}$ is aligned with the x-axis and has amplitude $B_0 = 0.2$. We simulate 64 billion particles on a 512×512 grid. Time step is 0.1 and spatial domain is $[0, L)^2$, with $L = 32$. The initial function is:

$$\begin{aligned} &f(x,y,\mathbf{v}) = F_1(v_x^2 - 2\phi(x,y))\mathrm{e}^{-50(v_y^2+v_z^2)} \\ &\text{with potential } \phi(x,y) = \mathrm{e}^{-0.5\left((x-L/2)/\Delta_\parallel - 0.3\cos(0.39y)\right)^2}, \\ &\Delta_\parallel = 3 \text{and } F_1 \text{ defined as} \\ &F_1(w) = \begin{cases} \frac{\sqrt{-w}}{\pi\Delta_\parallel^2}\left(1 + 2\ln(\frac{\psi}{-2w})\right) + \frac{6+(\sqrt{2}+\sqrt{-w})(1-w)\sqrt{-w}}{\pi(\sqrt{2}+\sqrt{-w})(4-2w+w^2)}, \text{ for } -2\psi \le w < 0, \\ \frac{6\sqrt{2}}{\pi(8+w^3)}, \text{ for } w > 0. \end{cases} \end{aligned} \tag{7.7}$$

Numerical results

Figure 7.5 shows the charge density $\rho(t,x,y) = 1 - \int_{\mathbb{R}^3} f(t,x,y,\mathbf{v})\,\mathrm{d}v_x\,\mathrm{d}v_y\,\mathrm{d}v_z$, on the left at time $t = 20$, and on the right at time $t = 40$. These results are qualitatively similar to those from Muschietti *et al.* [165].

In addition, we studied the convergence of the simulation with respect to the number of particles and to the grid size. To that end, we compare, for different settings of these two parameters, the time evolution of a quantity representative of the instability[3]. Results appear in Figure 7.6. They show that using a small 32×32 grid with 200 million particles as considered by Muschietti *et al.* exhibits the correct qualitative behavior up to $t = 50$, but diverges beyond this point.

For a quick simulation, it appears preferable to use a 64×64 grid with only 20 million particles, as it gives quantitatively accurate results up to $t = 50$. For longer simulations, our results show that using a 128×128 or a 256×256 grid with 200 million particles suffices to

[3]This quantity, which we call "y part of electric field norm", is defined as half of the square root of the electric energy $\int_{\mathbb{R}^2}(E_x^2 + E_y^2)\,\mathrm{d}x\,\mathrm{d}y$ minus the part of that energy corresponding to the modes in x (here, the first 20 modes).

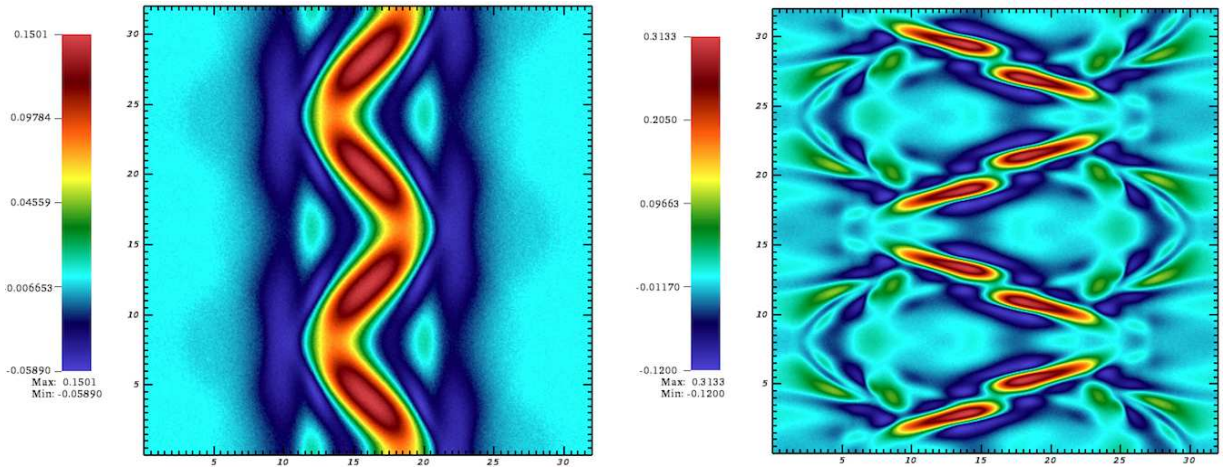


Figure 7.5 – Time evolution of $\rho(t,x,y) = 1 - \int_{\mathbb{R}^3} f \, dv_x \, dv_y \, dv_z$ at $t = 20$ (left) and $t = 40$ (right), 2d3v electron hole test case in (7.6) and (7.7).

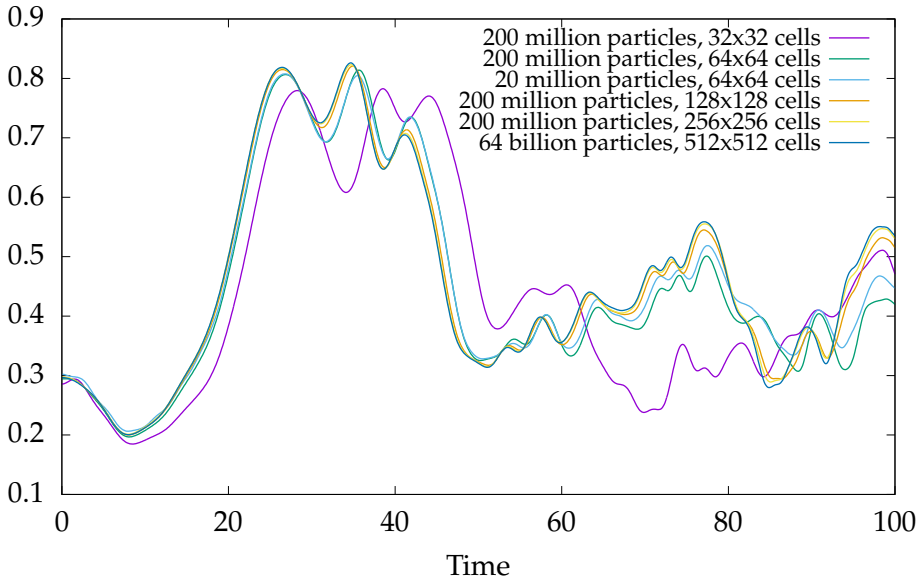


Figure 7.6 – Time evolution of the y part of electric field norm for different values of the number of particles and of the grid size, 2d3v electron hole test case in (7.6) and (7.7).

give accurate results up to $t = 100$. Indeed, the two corresponding curves are close to that of our large-scale simulation, which uses a 512×512 grid with 64 billion particles (the top-most curve at $t = 100$).

Figures 7.7–7.9 represent the conservation of the total energy. We plot the relative error across time and see the influence of the grid size and of the number of particles on the conservation on this quantity.

7.2 The PICSL Framework

7.2.1 Introduction

We consider the two-species Vlasov–Poisson system in 2d2v. We look for ion and electron distribution functions $f_s = f_s(t, \mathbf{x}, \mathbf{v})$, with $s \in \{e, i\}$ and electric field $E = E(t, \mathbf{x})$, satisfying

$$\begin{cases} \partial_t f_i + \mathbf{v} \cdot \nabla_{\mathbf{x}} f_i + \frac{q}{m_i} \mathbf{E} \cdot \nabla_{\mathbf{v}} f_i = 0, \\ \partial_t f_e + \mathbf{v} \cdot \nabla_{\mathbf{x}} f_e - \frac{q}{m_e} \mathbf{E} \cdot \nabla_{\mathbf{v}} f_e = 0, \\ -\varepsilon_0 \Delta_{\mathbf{x}} \Phi = q \int_{\mathbb{R}^2} (f_i - f_e) \, \mathrm{dv_x} \, \mathrm{dv_y}, \\ -\nabla_{\mathbf{x}} \Phi = \mathbf{E}. \end{cases} \tag{7.8}$$

and subject to initial distributions $f_i(t = 0, \mathbf{x}, \mathbf{v})$ and $f_e(t = 0, \mathbf{x}, \mathbf{v})$. Here $q = 1$ is the charge, m_s is the mass of the species s and $\varepsilon_0 = 1$ is the dielectric constant, $t \in \mathbb{R}^+$ is the time, $\mathbf{x} = (x, y) \in \Omega = \mathbb{R}^2/(L_x\mathbb{Z} \times L_y\mathbb{Z})$ the position and $\mathbf{v} \in \mathbb{R}^2$ the velocity. $\Phi = \Phi(t, \mathbf{x})$ is the electric potential. The original aim of the PICSL Cemracs project was to develop an implementation that works both for Particle-in-Cell (PIC) and semi-Lagrangian (SL) method and that is able to solve system (7.8). We focus here on some of the difficulties of kinetic simulations that are the multi-dimensionality (here 2d2v instead of 1d1v), multi-species (ions and electrons) and multi-methods (both PIC and SL) aspects. Extensions to higher dimensions, Vlasov–Maxwell (see [175] for such a recent work, that discusses also the pros/cons between PIC and semi-Lagrangian methods) and gyrokinetics (that includes the issue of using more complex geometries) are out of the scope of this work.

In the literature, works on single species 1d1v Vlasov–Poisson solvers are abundant. We refer here to [169, 166, 148, 146, 147, 141, 92, 136, 133, 99, 160, 170] for works on 2d2v Vlasov–Poisson simulations and to [132, 159] on multi-species simulations. This list is far from being exhaustive; there is a huge number of papers in plasma physics on the subject. Validation with respect to the dispersion relation is often performed for 1d1v simulations (see for example [80], where comparison of different implementations is also performed). The dispersion relation analysis, even if less used, permits also to study multi-dimensional and multi-species simulations.

Using such an analysis, our first aim is to justify the two-species simulations of [132], following [35], and consisting in the linearization of the equations around the Maxwellian equilibrium. Note that the dispersion analysis dates back to Landau [157]. With respect to the usual single species case, two main modes play here a role and their relative weights have an importance, in order to catch the right behavior. A finer study permits to exhibit relevant nonlinear effects, thanks to a second order expansion as done in [143], for example. Details of this analysis are in [207].

We then had in mind to extend the results to the multi-dimensional case. We focus there first on the single species case and are able to show a *true* multi-dimensional effect solely visible from a second order expansion. Note that in the literature, usual 2d2v test cases reduce to 1d1v (see previous references based on Landau and two-stream instabilities test cases; note that in [160], an effort has been put to get a two dimensional character). We then could study the two-species case, in the 2d2v setting, but such analysis is basically the superposition of the two previous analyses and we prefer here to focus on performing nonlinear simulations, where the analysis coming from the dispersion relation is anyway no more valid.

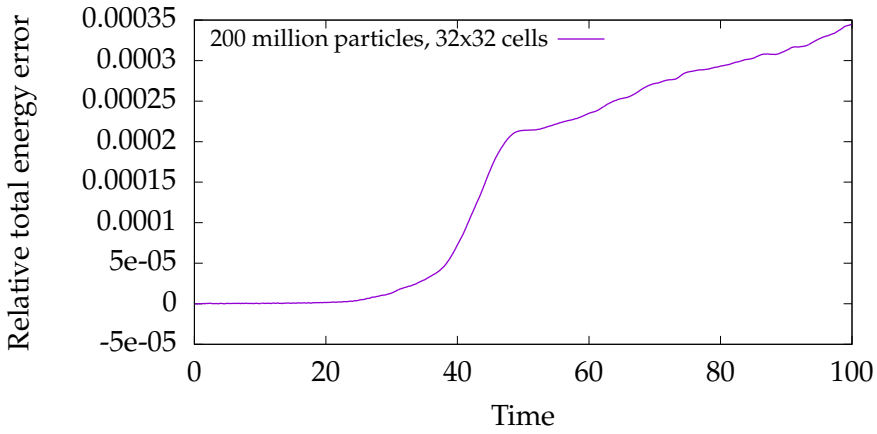


Figure 7.7 – Conservation of the total energy, 2d3v electron hole test case in (7.6) and (7.7).

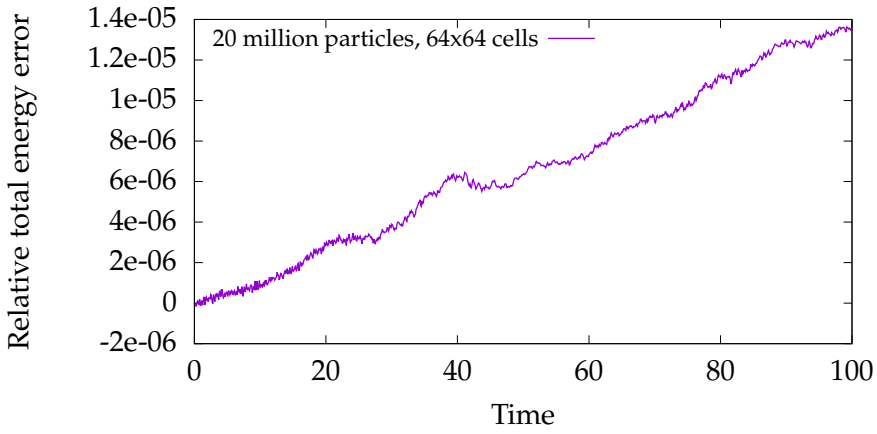


Figure 7.8 – Conservation of the total energy, 2d3v electron hole test case in (7.6) and (7.7).

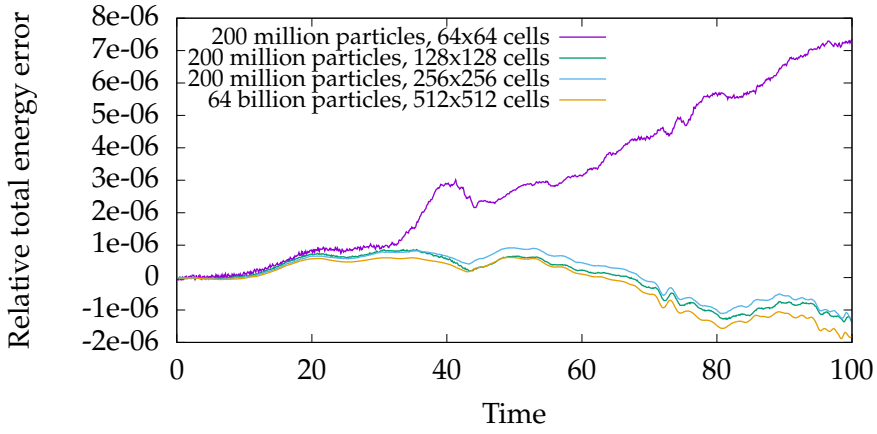


Figure 7.9 – Conservation of the total energy, 2d3v electron hole test case in (7.6) and (7.7).

If a subsequent part of the work relies on the verification of the implementation through the dispersion relation, another part of the work is based on the comparison and the mixing of semi-Lagrangian and PIC methods. As the two methodologies are quite different, getting the same answers from both methods is a further validation. On the other hand, defining a common implementation that works for both of them is a step to allow for a wider application range, as each method has his own benefits and drawbacks. We could for example consider in the future, the PIC method when a species is well localized (which permits to prevent from the discretization of the whole phase space) and the semi-Lagrangian method for other species. Note that the coupling of the two methods seems not to have been considered; at least, such approach is rare, as generally one method is privileged for a given simulation. The difficulties rely here on the definition of a common framework and on the need of expertise in both methods. In the numerical results, we will see that the present approach does not generate extra problems due to the coupling of the methods, which is encouraging for further developments.

Section 7.2.2 presents the numerical method and implementation details that permit to deal with both PIC and SL methods. We then describe the equations and test cases analyzed, together with numerical results. Section 7.2.3 studies the two-species 1d1v test case proposed in [132]. Section 7.2.4 proposes a new *true* 2d2v one species test case. Section 7.2.5 finally presents full non linear two-species results in 2d2v (an extension to two-species of a test case presented in [160]).

7.2.2 Numerical Method

In order to solve the Vlasov equation, we develop an implementation that is able to use both Particle-in-Cell (PIC) and semi-Lagrangian methods, in the framework of the SeLaLib[4] library. For the Poisson solver, we classically use the FFT; time and space (semi-Lagrangian or PIC) will be further detailed thereafter. The framework is such that we can use PIC for the two species, semi-Lagrangian for the two species, or PIC for one species and semi-Lagrangian for the other species.

Time discretization

We consider two types of time discretizations. The first one is based on a splitting by direction, as in [132] and the second one is a splitting by species.

Splitting first by direction: The algorithm can be sketched as in Figures 7.10 and 7.11 for the classical Strang splitting. This scheme can be generalized to higher order splitting; we will here use the classical 6^{th} order splitting of Blanes and Moan [137], as in [93].

Splitting first by species: In order to have the possibility of dealing with the species differently, we developed another scheme based on a splitting by species. It is depicted in Figure 7.12. Once more, this scheme can be generalized to higher order splitting. Nevertheless, we cannot use the splitting coefficients from the classical 6^{th} order splitting, because they are not suitable for a splitting by species.

Semi-Lagrangian discretization

We use a classical backward semi-Lagrangian (BSL) method like in Chapter 6, consisting here in solving successive constant advection equations on a uniform 1d periodic mesh [94]. Centered Lagrange interpolation of degree 9 is used for the interpolation; see for example [101]. Concerning the splitting by species, we use Strang splitting on each species for the corresponding solving of the Vlasov–Poisson equation.

[4] http://selalib.gforge.inria.fr/

Parameters :
Δt, the time step.
$ncx \times ncy$, the size of the spatial grid.

Variables :
$f_{electrons}$, the distribution function (4d array for SL, array of particles for PIC).
$\rho[ncx][ncy]$, a 2d array containing the charge distribution.
$E[ncx][ncy]$, a 2d array containing the self-induced electric field.

Initialization :
1 Initialize $f_{electrons}$

Algorithm :
2 **Foreach** time iteration
3 Advection of $f_{electrons}$ in x over $\Delta t/2$ $\partial_t f_{electrons} + v \cdot \nabla_x f_{electrons} = 0$
4 Compute ρ from $f_{electrons}$ Integration in v for SL, deposit for PIC
5 Compute E from ρ Poisson solver
6 Advection of $f_{electrons}$ in v over Δt $\partial_t f_{electrons} - E \cdot \nabla_v f_{electrons} = 0$
7 Advection of $f_{electrons}$ in x over $\Delta t/2$ $\partial_t f_{electrons} + v \cdot \nabla_x f_{electrons} = 0$
8 **End Foreach**

Figure 7.10 – One species (electrons) pseudo-code.

Parameters :
Δt, the time step.
$ncx \times ncy$, the size of the spatial grid.
$\varepsilon = \sqrt{\frac{m_{electrons}}{m_{ions}}}$, the square root of the mass ratio.

Variables :
$f_{electrons}$ and f_{ions}, the distribution function for electrons and ions (4d arrays for SL, arrays of particles for PIC).
$\rho_{electrons}[ncx][ncy]$, $\rho_{ions}[ncx][ncy]$ and $\rho[ncx][ncy]$, 2d arrays containing the charge distribution.
$E[ncx][ncy]$, a 2d array containing the self-induced electric field.

Initialization :
1 Initialize $f_{electrons}$ and f_{ions}

Algorithm :
2 **Foreach** time iteration
3 Advection of $f_{electrons}$ in x over $\Delta t/2$ $\partial_t f_{electrons} + (1/\varepsilon) v \cdot \nabla_x f_{electrons} = 0$
4 Advection of f_{ions} in x over $\Delta t/2$ $\partial_t f_{ions} + v \cdot \nabla_x f_{ions} = 0$
5 Compute $\rho_{electrons}$ from $f_{electrons}$ and ρ_{ions} from f_{ions} Integration in v for SL, deposit for PIC
6 Compute E from $\rho = \rho_{ions} - \rho_{electrons}$ Poisson solver
7 Advection of $f_{electrons}$ in v over Δt $\partial_t f_{electrons} - (1/\varepsilon) E \cdot \nabla_v f_{electrons} = 0$
8 Advection of f_{ions} in v over Δt $\partial_t f_{ions} + E \cdot \nabla_v f_{ions} = 0$
9 Advection of $f_{electrons}$ in x over $\Delta t/2$ $\partial_t f_{electrons} + (1/\varepsilon) v \cdot \nabla_x f_{electrons} = 0$
10 Advection of f_{ions} in x over $\Delta t/2$ $\partial_t f_{ions} + v \cdot \nabla_x f_{ions} = 0$
11 **End Foreach**

Figure 7.11 – Two-species pseudo-code, splitting by direction.

Parameters :
Δt, the time step.
$ncx \times ncy$, the size of the spatial grid.
$\varepsilon = \sqrt{\frac{m_{electrons}}{m_{ions}}}$, the square root of the mass ratio.

Variables :
$f_{electrons}$ and f_{ions}, the distribution function for electrons and ions (4d arrays for SL, arrays of particles for PIC).
$\rho_{electrons}[ncx][ncy]$, $\rho_{ions}[ncx][ncy]$ and $\rho[ncx][ncy]$, 2d arrays containing the charge distribution.
$E[ncx][ncy]$, a 2d array containing the self-induced electric field.

Initialization :
1 Initialize $f_{electrons}$ and f_{ions}

Algorithm :
2 **Foreach** time iteration
3 Solve Vlasov–Poisson (ions, $\Delta t/2$)
4 Solve Vlasov–Poisson (electrons, Δt)
5 Solve Vlasov–Poisson (ions, $\Delta t/2$)
6 **End Foreach**

Subroutine Solve Vlasov–Poisson (*species*, *time_step*) :
7 Advection of $f_{species}$ in x over $time\_step/2$
8 Compute $\rho_{species}$ from f
9 Compute E from $\rho = \rho_{ions} - \rho_{electrons}$
10 Advection of $f_{species}$ in v over $time\_step$
11 Advection of $f_{species}$ in x over $time\_step/2$
12 Compute $\rho_{species}$ from $f_{species}$

Figure 7.12 – Two-species pseudo-code, splitting by species.

PIC discretization

A Particle-in-Cell (PIC) method consists in discretizing (sampling) the distribution function by a collection of N macro-particles that move in the phase space following the characteristics of the Vlasov equation. We use the classical PIC method, explained in Chapter 2, with linear or cubic splines for the deposition of the charge and for the interpolation of the electric field. The macro-particles are initialized randomly, which ensures a stochastic convergence in $\frac{1}{\sqrt{N}}$.

Time schemes presented in Figures 7.10, 7.11 and 7.12 are still valid for the PIC method. These schemes are used when running simulations using PIC for one species and BSL for the other. However, to ensure efficiency when running simulations only with the PIC method, a leap-frog scheme is used (second order in time).

7.2.3 A 1d1v two-species test case

Description of the equations and initial conditions

We first consider a test case studied by [132]. We look for f_i, f_e satisfying

$$\begin{cases} \partial_t f_i + v\,\partial_x f_i + E\,\partial_v f_i = 0, \\ \partial_t f_e + \frac{1}{\varepsilon} v\,\partial_x f_e - \frac{1}{\varepsilon} E\,\partial_v f_e = 0, \\ \partial_x E = \int_{\mathbb{R}} (f_i - f_e)\,\mathrm{d}v, \end{cases} \tag{7.9}$$

with $\varepsilon = \sqrt{\frac{m_e}{m_i}}$, the root of the mass ratio between ions and electrons, and with initial functions

$$\begin{cases} f_e(0,x,v) = \frac{1}{\sqrt{2\pi}} \mathrm{e}^{-\frac{v^2}{2}}, \\ f_i(0,x,v) = \frac{v^2}{\sqrt{2\pi\sigma^3}} \mathrm{e}^{-\frac{v^2}{2\sigma^2}} (1 + A\cos(kx)), \end{cases} \tag{7.10}$$

with $k = \frac{2\pi}{L}$, and A the amplitude of the perturbation. The phase-space domain is $[0, L) \times [-v_{\max}, v_{\max})$. We will take here $\sigma = \frac{1}{2}$ and $L = 21$, as in [132], and $v_{\max} = 6$.

This is a first example of two-species simulation. Our goal is to reproduce these results [132] from the literature with our commonly used methods (as in [101] for example) and also

to provide a dispersion analysis, which permits to further validate the implementation. Note that this test is 1d1v, but it will be simulated in the 2d2v implementation; this enables to have a first check of the implementation.

Numerical results

We take here $\varepsilon = 1$, as we first want here to validate the two-species feature; this permits to have a first example taken from the literature [132] that is here justified with the dispersion relation analysis and that can be cheaply reproduced in this one dimensional context. On Figure 7.13 (left, logarithmic scale; right, standard scale), we represent, for the perturbation $A = 0.0001$, the electric energy defined by $\sqrt{\frac{1}{2}\int_0^L |E|^2 \,\mathrm{dx}}$ versus time t and also the absolute value of the first and second Fourier modes multiplied by $\sqrt{\frac{1}{2}}$, in order to be comparable to the electric energy. We represent also *theoretical* results, coming from the study of the dispersion analysis developed in [207, Section 3]. The theoretical first mode is here the expression

$$\mathrm{E}_1 = A\,|-0.1\exp(0.089t) - 6.9\cos(1.5t))|\,. \tag{7.11}$$

It comes from the first order dispersion relation whose more precise expression, using the two first relevant zeros, is

$$A\,|a_1\exp(\gamma_1 t) + a_2\exp(\gamma_2 t)\cos(\omega t) + a_3\exp(\gamma_2 t)\sin(\omega t)|\,,$$

with

$$a_1 = -0.098626662403769140798,\ a_2 = -6.9231540740080643228,\ a_3 = -0.015835049471186903442,$$
$$\gamma_1 = 0.089001301682640372604,\ \gamma_2 = -0.00015911724084755207863,\ \omega = 1.5006859732648583225.$$

We remark that this analytical expression permits to describe precisely, up to time $t = 80$, the behavior of the first mode that is simulated and also the whole electric energy, as this first mode is dominant. For the simulation, we have used the BSL method on a 1024×2048 grid, with $\Delta t = 0.1$. The study of the linear analysis at order 2 developed in [207, Section 3.2] permits to explain the behavior of the electric energy up to time $t = 90$, and the behavior of the second Fourier mode from initial time to time $t = 90$. We have used here the following analytical expression for the second Fourier mode

$$\mathrm{E}_2 = A^2\left(0.3\exp(2\cdot 0.089t) + 0.7\exp(0.145t)\right). \tag{7.12}$$

Here the coefficients 0.3 and 0.7 are chosen to fit the numerical results, 0.089 is an approximation of γ_1 and 0.145 is the rounding of 0.144982725814, coming from the dispersion analysis of [207, Section 3.2]. The theoretical electric energy is then given in the figures by $\sqrt{\mathrm{E}_1^2 + \mathrm{E}_2^2}$. Note that after time 100, we are in the non linear phase and the dispersion relation analysis is no more valid.

On Figure 7.14, we take $A = 0.01$, as in [132]. We take here as parameters, the BSL method on a grid 128×256 with $\Delta t = 0.02$. The behavior is similar. As the perturbation is bigger, the non linear phase appears sooner. We can note also that the first mode does not have time to develop and that the instability is essentially explained by the second order expansion.

Then, we study the influence of the numerical parameters, on Figure 7.15. We see that for $A = 0.0001$, the grid 64×256 is quite good, as the difference with the refined run on a grid 1024×2048 (similar to 2048×4096) is only visible at the end of the simulation, around $T = 150$. For $A = 0.01$, we get converged results until T around $80 - 100$; then for longer times, we see that the results start to differ, and the grid 64×256 seems not fine enough. For the time step, it seems that $\Delta t = 0.1$ is a good choice, as the results are very similar between $\Delta t = 0.02$ or $\Delta t = 0.1$.

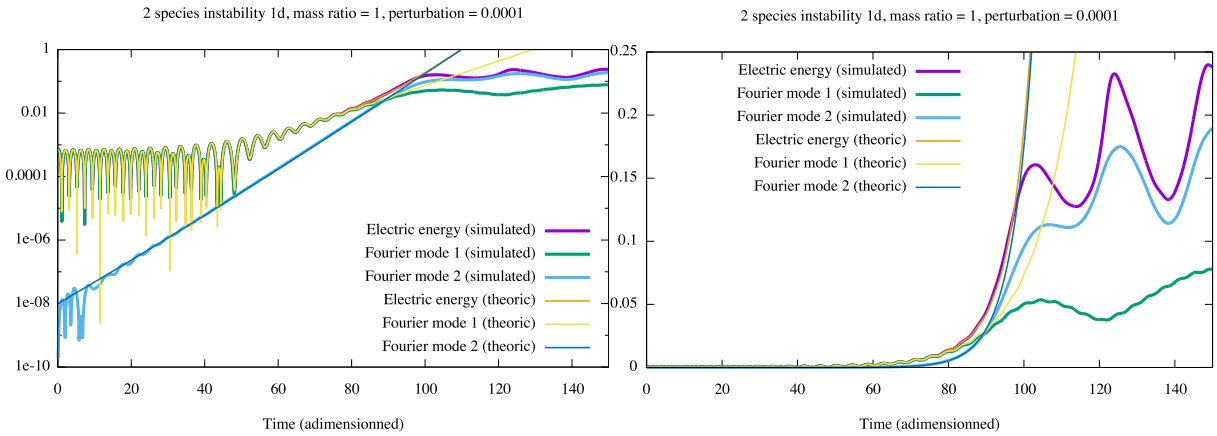


Figure 7.13 – $A = 0.0001$, 1d1v test case in (7.9) and (7.10)

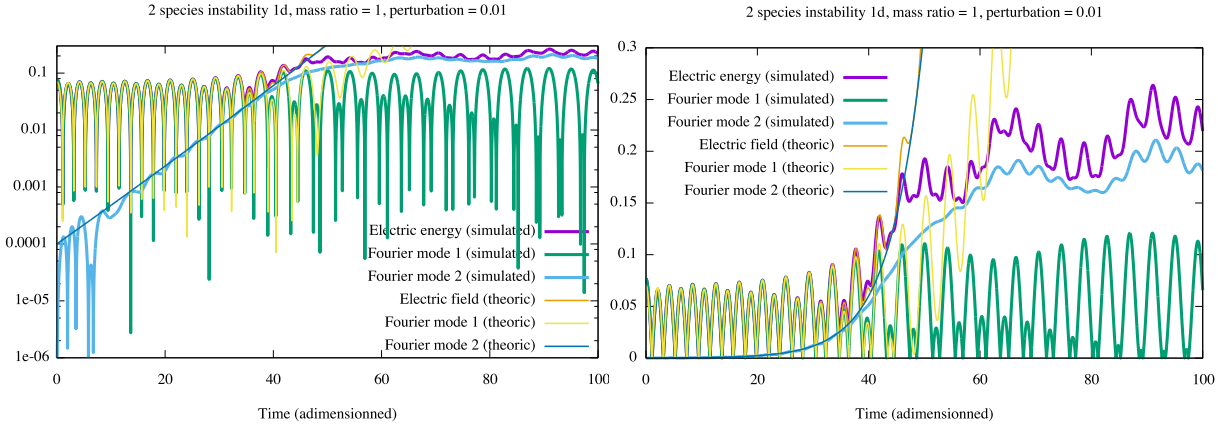


Figure 7.14 – $A = 0.01$, 1d1v test case in (7.9) and (7.10)

On Figures 7.22–7.26, we can appreciate the convergence on the diagnostics of conservation of L^1, L^2 norms; the mass is conserved up to machine precision.

$x - vx$ cut permits to measure the structures and the filaments, here for $A = 0.01$ (see Figures 7.16–7.19). It is confirmed that at time $T = 80$, the mesh 64×256 correctly describes the ions (see Figures 7.20 and 7.21). At time $T = 150$, however, as already seen on the electric energy (Figure 7.15, right), we see the differences between the fine run (512×2048 grid) and the coarse one (64×256) for the ions (see Figures 7.18 and 7.19); for the electrons the differences are smaller.

7.2.4 A new *true* 2d2v one-species test case

Description of the equations and initial conditions

We focus then on 2d2v phase space. We look for f satisfying

$$\begin{cases} \partial_t f + \mathbf{v} \cdot \nabla_{\mathbf{x}} f - \mathbf{E} \cdot \nabla_{\mathbf{v}} f = 0, \\ -\Delta_{\mathbf{x}} \Phi = 1 - \int_{\mathbb{R}^2} f \, \mathrm{d}v_x \, \mathrm{d}v_y, \\ -\nabla_{\mathbf{x}} \Phi = \mathbf{E}, \end{cases} \tag{7.13}$$

with initial function

$$f(0, \mathbf{x}, \mathbf{v}) = \left(1 + A\left(\cos\left(\frac{y}{2}\right) + \cos\left(\frac{x+y}{2}\right)\right)\right) \frac{v_x^2}{2\pi} \mathrm{e}^{-\frac{|\mathbf{v}|^2}{2}}. \tag{7.14}$$

We take $L_x = L_y = 4\pi$, the phase-space domain is $[0, 4\pi)^2 \times [-v_{\max}, v_{\max})^2$ and $v_{\max} = 10$.

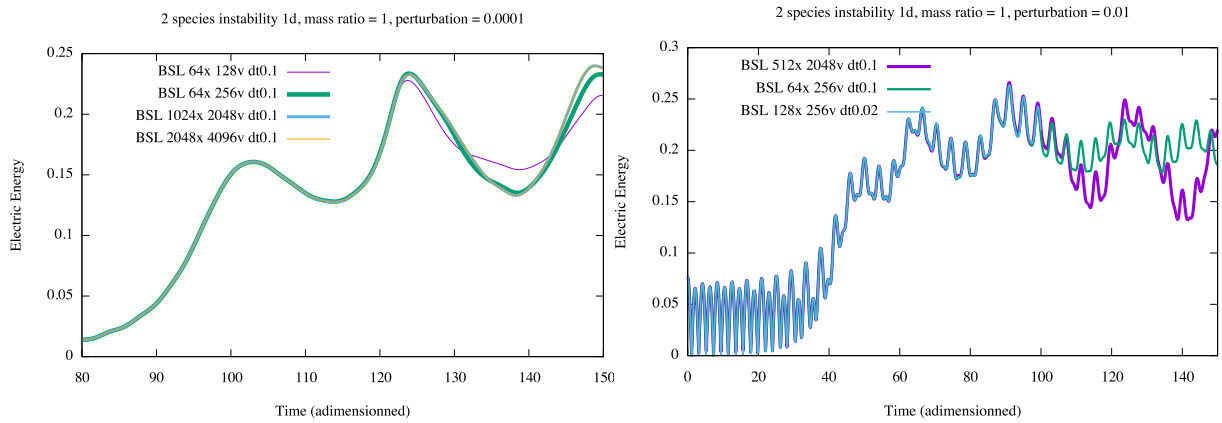


Figure 7.15 – Convergence study of the electric energy: $A = 0.0001$ (left) and $A = 0.01$ (right), 1d1v test case in (7.9) and (7.10)

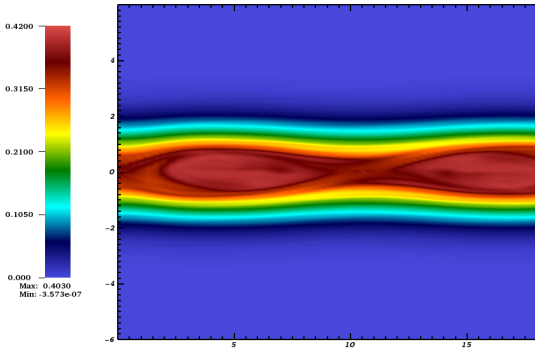


Figure 7.16 – $x - vx$ cut (electrons distribution) for $A = 0.01$, BSL $\Delta t = 0.1$ on 64×256 grid, at time $T = 150$, 1d1v test case in (7.9) and (7.10)

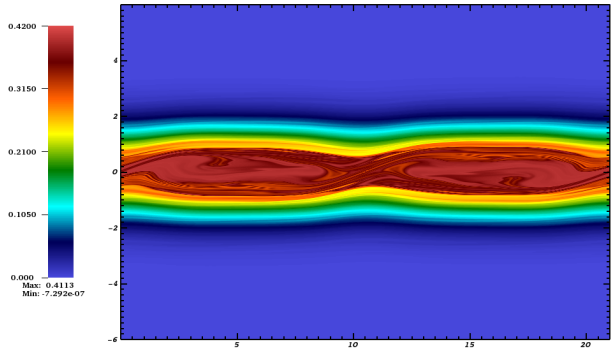


Figure 7.17 – $x - vx$ cut (electrons distribution) for $A = 0.01$, BSL $\Delta t = 0.1$ on 512×2048 grid, at time $T = 150$, 1d1v test case in (7.9) and (7.10)

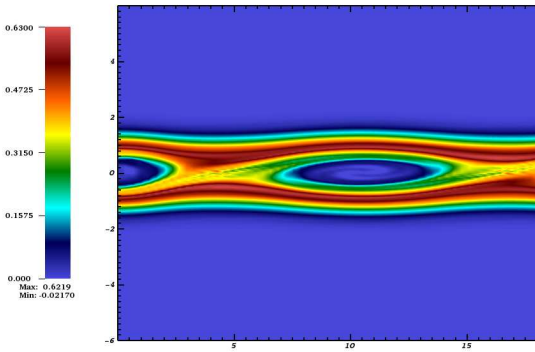


Figure 7.18 – $x - vx$ cut (ions distribution) for $A = 0.01$, BSL on 64×256 grid, at time $T = 150$, 1d1v test case in (7.9) and (7.10)

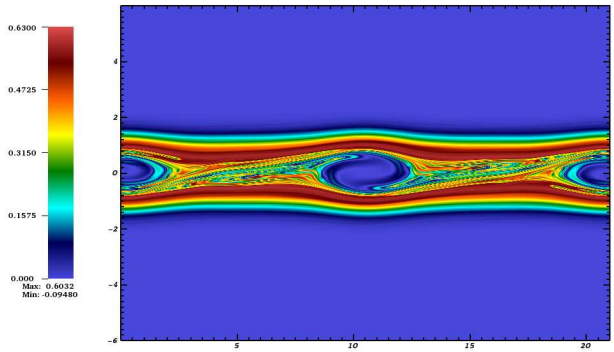


Figure 7.19 – $x - vx$ cut (ions distribution) for $A = 0.01$, BSL $\Delta t = 0.1$ on 512×2048 grid, at time $T = 150$, 1d1v test case in (7.9) and (7.10)

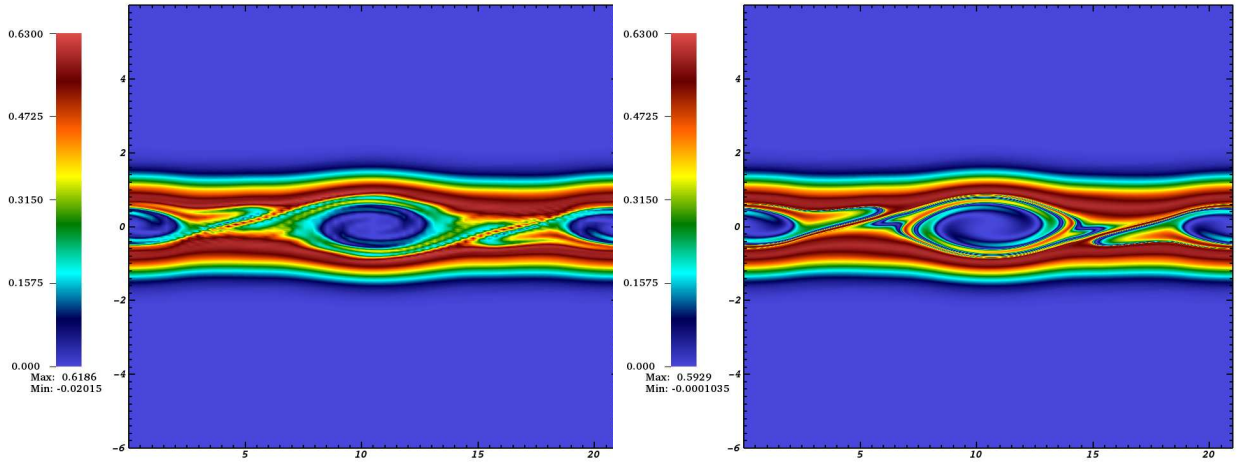


Figure 7.20 – $x - vx$ cut (ions distribution) for $A = 0.01$, BSL $\Delta t = 0.1$ on 64×256 grid, at time $T = 80$, 1d1v test case in (7.9) and (7.10)

Figure 7.21 – $x - vx$ cut (ions distribution) for $A = 0.01$, BSL $\Delta t = 0.1$ on 512×2048 grid, at time $T = 80$, 1d1v test case in (7.9) and (7.10)

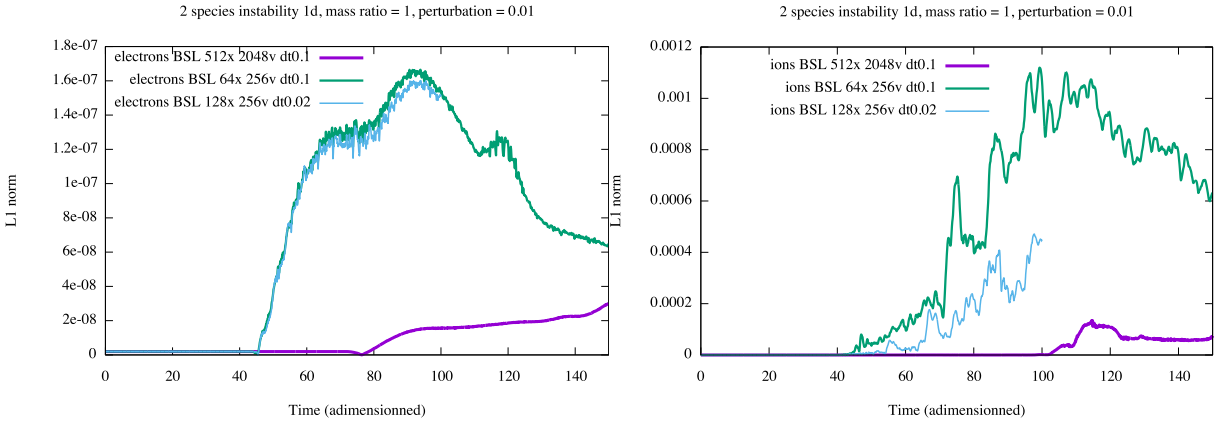


Figure 7.22 – Relative error of L^1 norm, $A = 0.01$, 1d1v test case in (7.9) and (7.10)

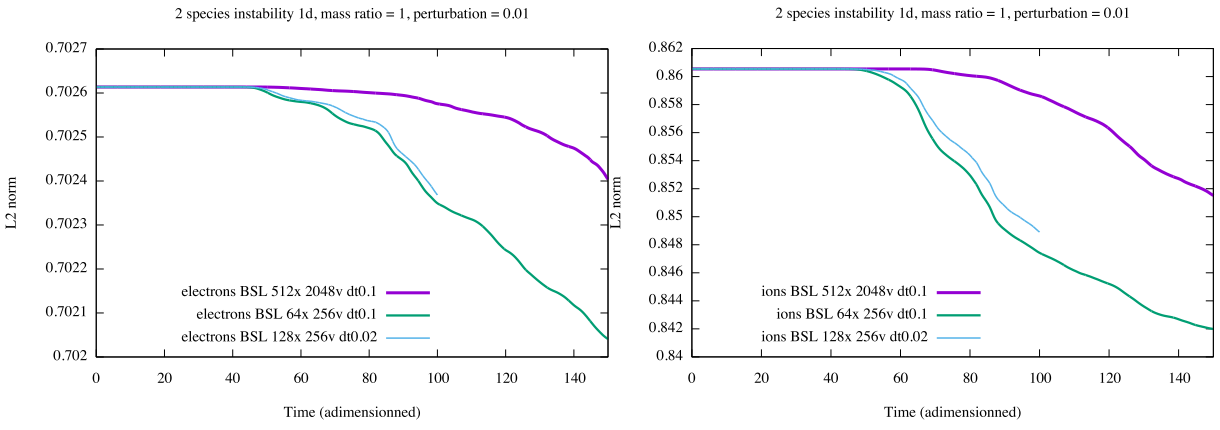


Figure 7.23 – L^2 norm, $A = 0.01$, 1d1v test case in (7.9) and (7.10)

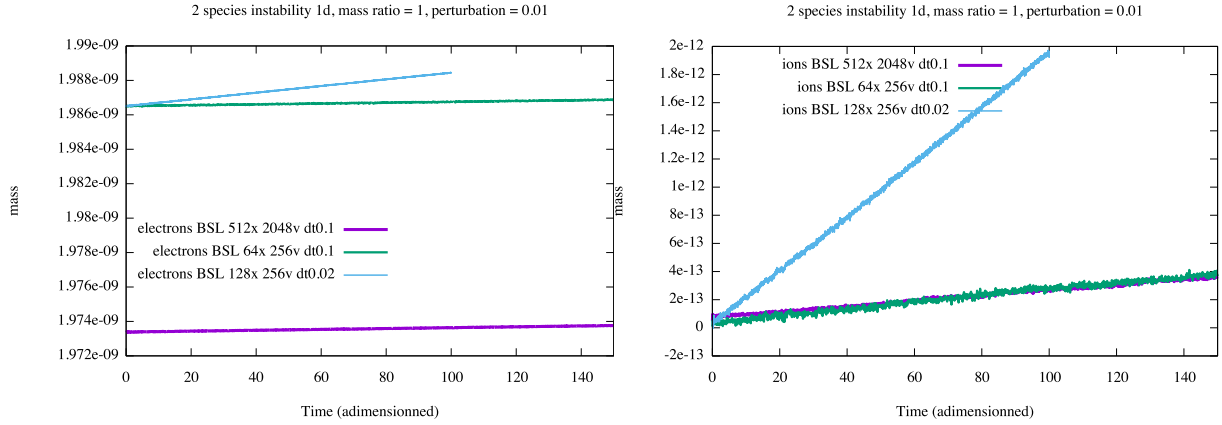


Figure 7.24 – Relative mass error, $A = 0.01$, 1d1v test case in (7.9) and (7.10)

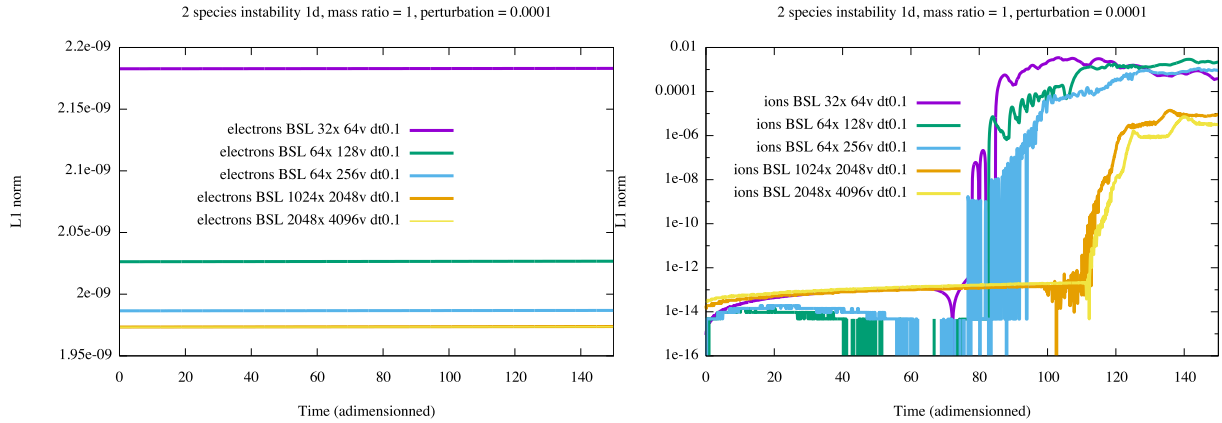


Figure 7.25 – Relative error of L^1 norm, $A = 0.0001$, 1d1v test case in (7.9) and (7.10)

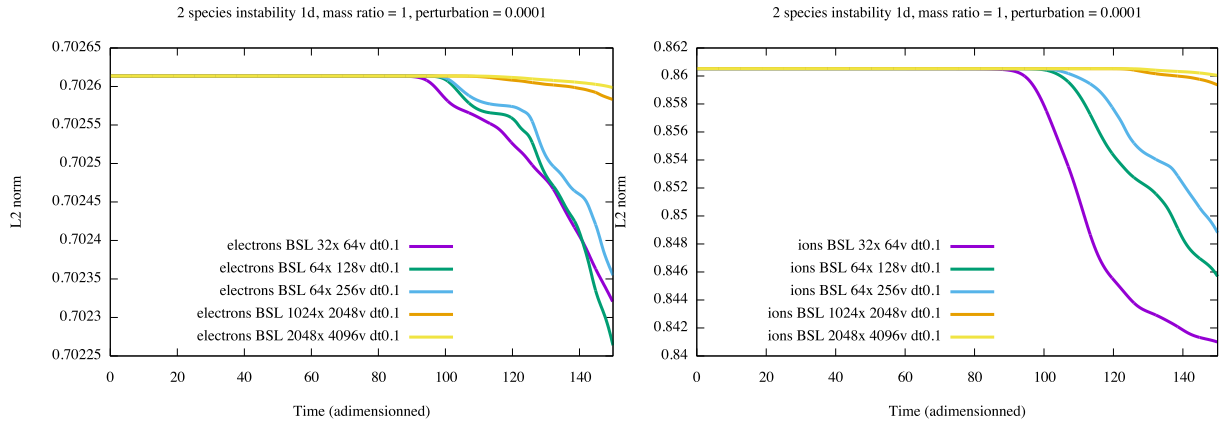


Figure 7.26 – L^2 norm, $A = 0.0001$, 1d1v test case in (7.9) and (7.10)

This test permits to capture the interaction of different modes and reveals 2d space features, which would not be visible by 1d1v implementations. We are not aware of such a test case in the literature; it seems not to be standard.

Numerical results

On Figure 7.27 (left, logarithmic scale; right, standard scale), we represent, for the perturbation $A = 0.1$, the electric energy defined by $\sqrt{\frac{1}{2}\int_0^{L_x}\int_0^{L_y}|E|^2\,\mathrm{dx}\,\mathrm{dy}}$ versus time t and also the absolute value of the first and second Fourier modes multiplied by $\sqrt{\frac{1}{2}}$, in order to be comparable to the electric energy. We represent also *theoretical* results, coming from the study of the dispersion analysis developed in [207, Section 3]. The theoretical first mode is here the expression

$$\mathrm{E}_1 = 0.89\,|\cos(1.416t + 2.6)\exp(-0.1533x))|\,. \tag{7.15}$$

We have here only used the theoretical values 1.416, -0.1533 and fitted the two other coefficients.

We remark that this analytical expression permits to describe precisely, up to time $t = 12$, the behavior of the first mode that is simulated and also the whole electric energy, as this first mode is dominant. For the simulation, we have used the BSL method on a $32 \times 32 \times 256 \times 256$ grid, with $\Delta t = 0.1$. The study of the linear analysis at order 2 developed in [207, Section 3.2] permits to explain the behavior of the electric energy up to time $t = 25$, and the behavior of the second Fourier mode from initial time to time $t = 25$. We have used here the following analytical expression for the second Fourier mode

$$\mathrm{E}_2 = 0.0028\exp(0.259t) \tag{7.16}$$

Here the coefficient 0.0028 is chosen to fit the numerical results, 0.259 is coming from the dispersion analysis of [207, Section 3.2]. The theoretical electric energy is then given in the figures by $\sqrt{\mathrm{E}_1^2 + \mathrm{E}_2^2}$. Note that after time $25 - 30$, we are in the non linear phase and the dispersion relation analysis is no more valid.

We then study the convergence on the diagnostic of the electric energy on Figures 7.28 and 7.29 (left). We notice that both PIC and BSL methods converge to the same state in the non linear phase, which permits to validate the results, from this cross comparison.

We see on Figure 7.29 (right) the time evolution of the L^2 norm; we notice that the conservation is clearly improved by refining the grid in space.

On Figures 7.30 and 7.31, we see $x - vx$ and $y - vy$ cuts; the first looks similar to two-stream instability and the second to Landau damping simulations. The filaments seem to be well resolved thanks to a relatively high number of points in the velocity directions.

On Figure 7.32, we see the contour plots of ρ at different times; we clearly see the behavior of the modes: first the mode $(0,1)$ dominates and then it is the mode $(1,0)$.

7.2.5 A 2d2v two-species test case

Description of the equations and initial conditions

We look for f_i, f_e satisfying

$$\begin{cases} \partial_t f_i + \mathbf{v}\cdot\nabla_{\mathbf{x}} f_i + \mathbf{E}\cdot\nabla_{\mathbf{v}} f_i = 0, \\ \partial_t f_e + \frac{1}{\varepsilon}\mathbf{v}\cdot\nabla_{\mathbf{x}} f_e - \frac{1}{\varepsilon}\mathbf{E}\cdot\nabla_{\mathbf{v}} f_e = 0, \\ -\Delta_{\mathbf{x}}\Phi = 1 - \int_{\mathbb{R}^2}(f_i - f_e)\,\mathrm{dv_x}\,\mathrm{dv_y}, \\ -\nabla_{\mathbf{x}}\Phi = \mathbf{E}, \end{cases} \tag{7.17}$$

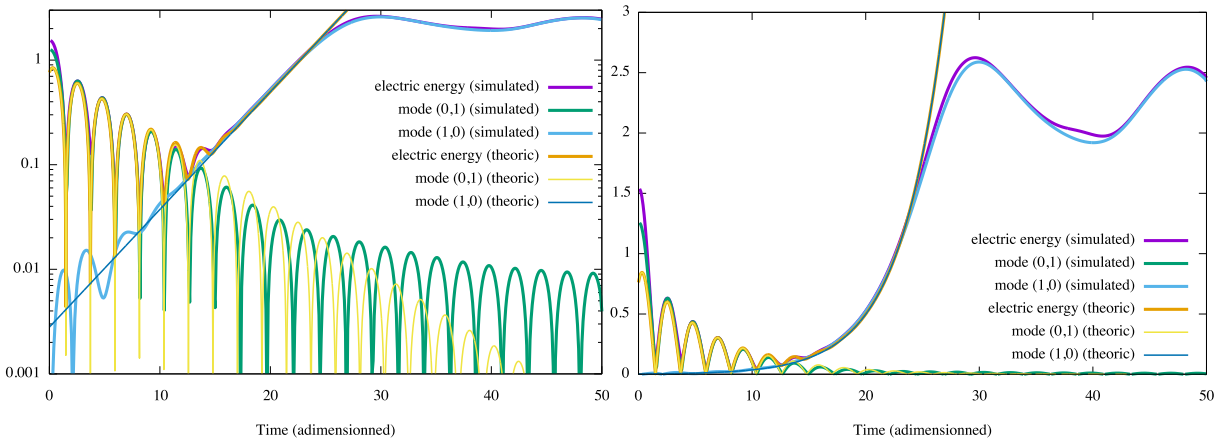


Figure 7.27 – Comparison between simulation and analytical results from dispersion relation, on the electric energy and the relevant modes. $A = 0.1$, 2d2v test case in (7.13) and (7.14)

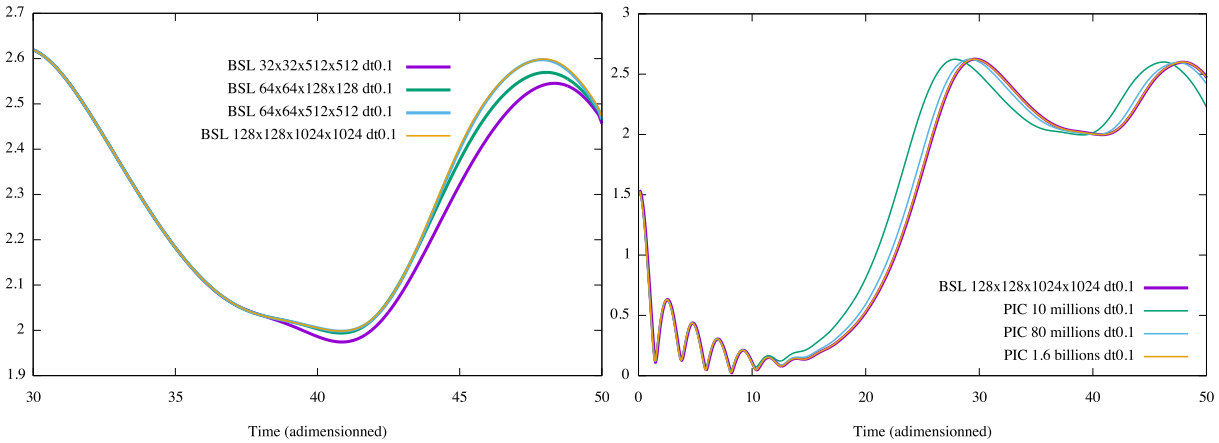


Figure 7.28 – Time evolution of electric energy, with convergence of BSL (left) and PIC (right). $A = 0.1$, 2d2v test case in (7.13) and (7.14). Different grid sizes for BSL and different numbers of particle for PIC are used. Time step is $\Delta t = 0.1$. The reference solution is here BSL with grid size $128 \times 128 \times 1024 \times 1024$.

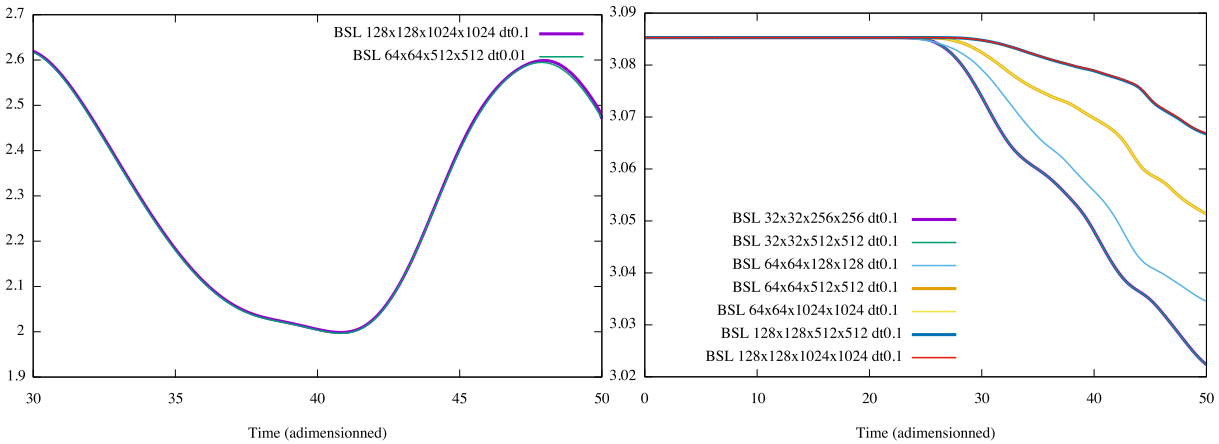


Figure 7.29 – Left: time evolution of electric energy with convergence in time for BSL. The solution with BSL with grid size $128 \times 128 \times 1024 \times 1024$ and $\Delta t = 0.1$ is similar to the solution with $64 \times 64 \times 512 \times 512$ and $\Delta t = 0.1$, which is also similar to the solution with $64 \times 64 \times 512 \times 512$ and $\Delta t = 0.01$. Right: time evolution of L^2 norm of f. $A = 0.1$, 2d2v test case in (7.13) and (7.14).

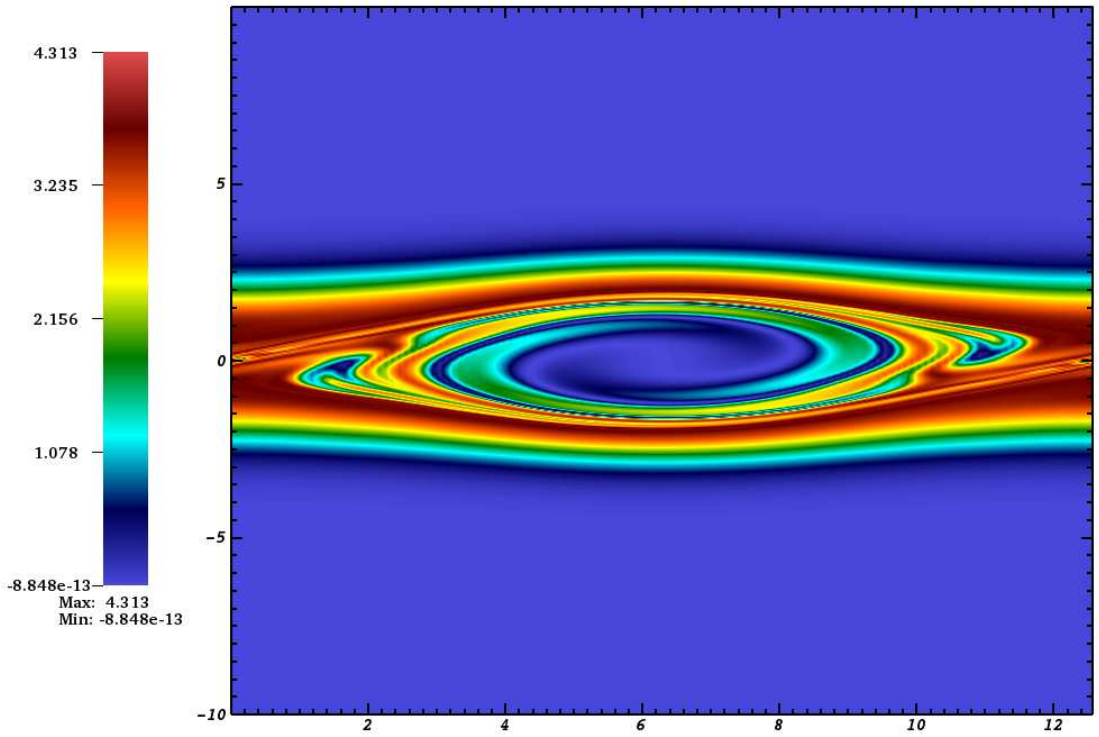


Figure 7.30 – $x - vx$ cut for $A = 0.1$, 2d2v test case in (7.13) and (7.14); BSL $128 \times 128 \times 1024 \times 1024$, $\Delta t = 0.05$ at final time $t = 50$.

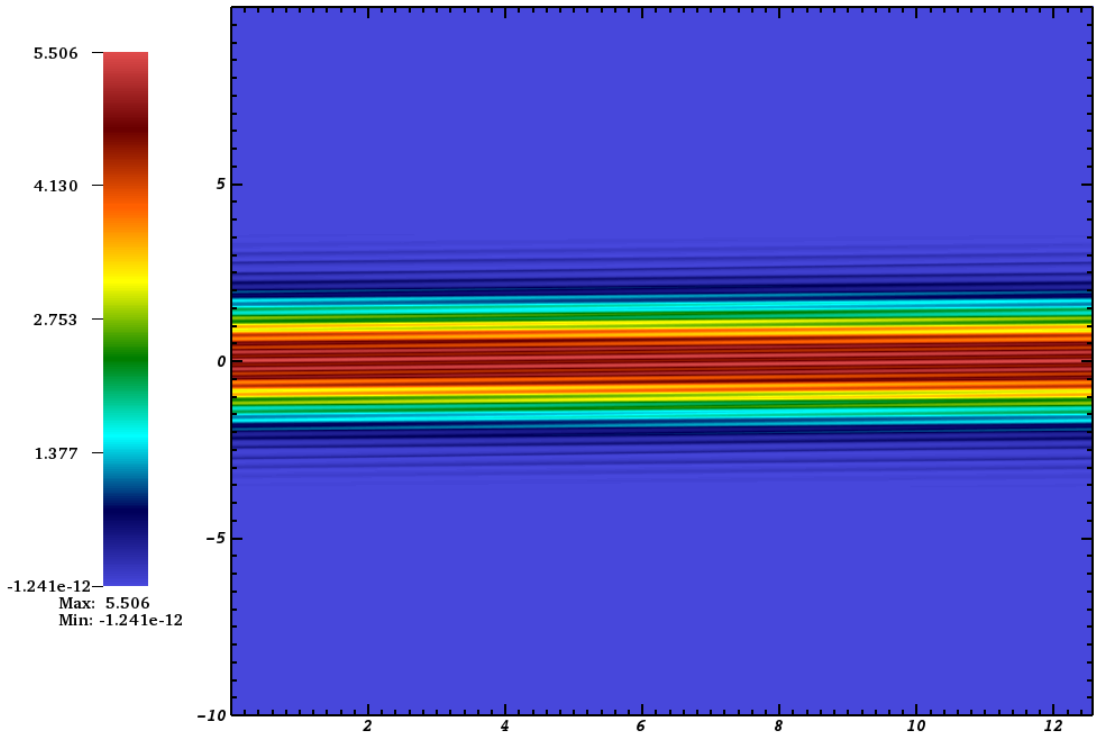


Figure 7.31 – $y - vy$ cut for $A = 0.1$, 2d2v test case in (7.13) and (7.14); BSL $128 \times 128 \times 1024 \times 1024$, $\Delta t = 0.05$ at final time $t = 50$.

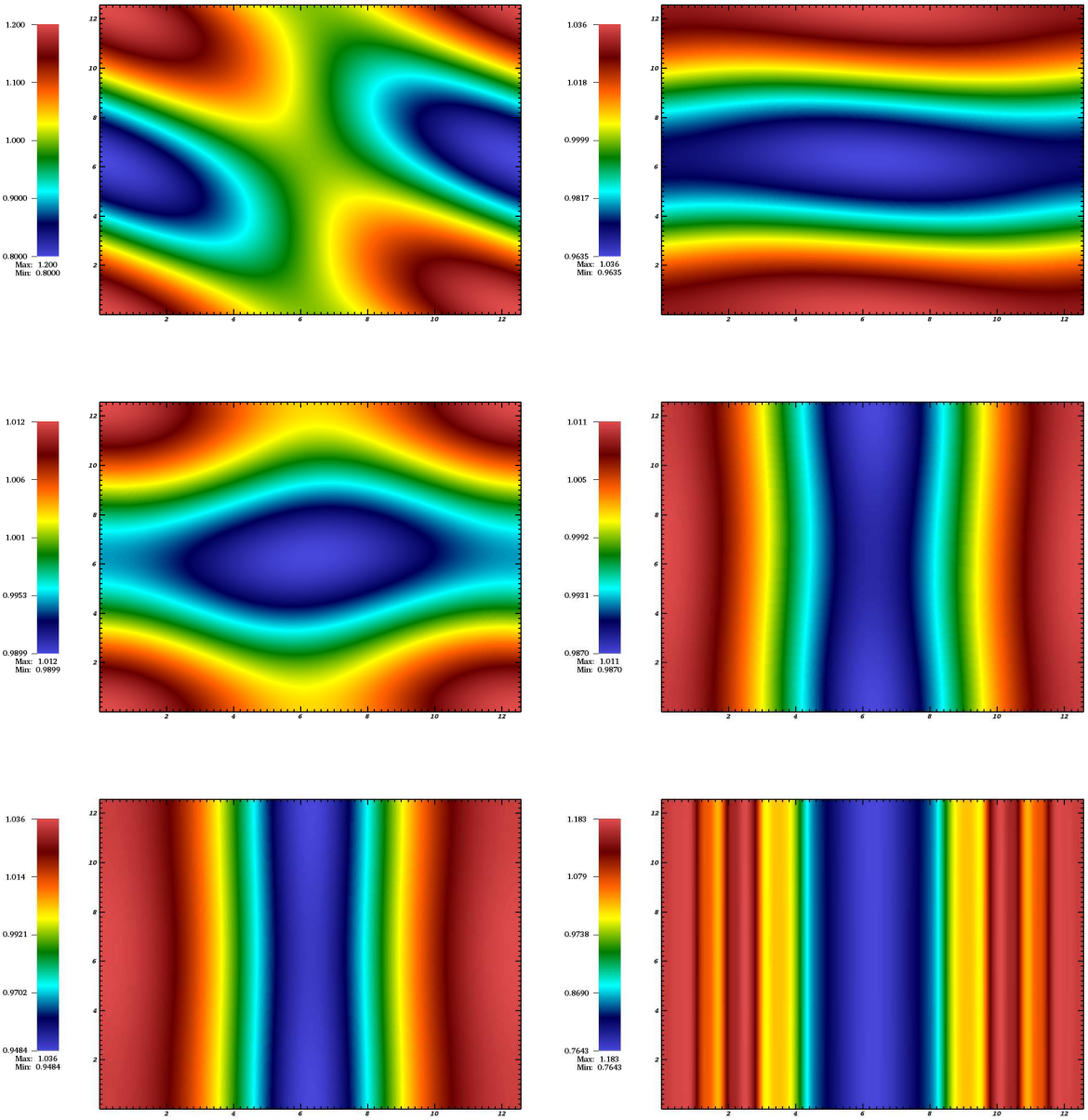


Figure 7.32 – ρ at different times ($t = 0, 5, 10, 15, 20, 50$); BSL $128 \times 128 \times 1024 \times 1024$, $\Delta t = 0.05$; $A = 0.1$, 2d2v test case in (7.13) and (7.14).

with $\varepsilon = \sqrt{\frac{m_e}{m_i}}$, the root of the mass ratio between ions and electrons and with initial functions

$$\begin{cases} f_e(0,\mathbf{x},\mathbf{v}) = \frac{1}{2\pi}\mathrm{e}^{-\frac{|\mathbf{v}|^2}{2}}, \\ f_i(0,\mathbf{x},\mathbf{v}) = \frac{1}{4\pi\sigma_x\sigma_y}\left(1 - A_x\sin(k_x x) - A_y\sin(k_y y)\right)\left(\mathrm{e}^{-\frac{(v_x-v_d)^2}{2\sigma_x^2}} + \mathrm{e}^{-\frac{(v_x+v_d)^2}{2\sigma_x^2}}\right)\mathrm{e}^{-\frac{v_y^2}{2\sigma_y^2}}, \end{cases} \tag{7.18}$$

with $k_x = \frac{2\pi}{L_x}$, $k_y = \frac{2\pi}{L_y}$, the perturbation amplitudes A_x, A_y, the velocity drift v_d and the thermal velocities σ_x, σ_y. The domain is $[0, L_x) \times [0, L_y) \times [-v_{\max}, v_{\max})^2$.

Our aim is to develop 2d2v two-species simulations; so, here is such an example. It is a generalization of the first test to the 2d2v framework; we use a 2d2v initial function for the ions that was developed in [160]. We will take here $v_d = 2.4$, $A_x = 0.005$, $A_y = 0.25$, $\sigma_x = 0.5$, $\sigma_y = 1$, $k_x = k_y = 0.2$ together with $v_{\max} = 10$.

Numerical results

We take here $\varepsilon = \sqrt{0.01}$. This leads to a more oscillatory behavior. We focus here on the electric energy. On Figure 7.33, we give the electric energy for BSL using the 6-th order scheme, for $\Delta t = 0.02$ and $\Delta t = 0.01$. We remark that there is a lot of oscillations. We see that the results are very similar, which is a mark of the fact that the scheme is converged in time. We then do the comparison with other methods and numerical parameters. The same quantity is plotted for other numerical parameters on Figures 7.34–7.38.

On Figure 7.34, we see that the result is equivalent with using the Strang scheme with $\Delta t = 0.0025$.

On Figure 7.35, we see that the convergence is not complete when passing from a grid $32 \times 256 \times 32 \times 512$ to a grid $32 \times 512 \times 32 \times 1024$, which means that high resolution in $y - vy$ is needed.

On Figure 7.36, we see that on the contrary, 32 points in x seem sufficient, as the curve for the $32 \times 512 \times 32 \times 1024$ and $64 \times 512 \times 32 \times 1024$ well match, and we see that going to $\Delta t = 0.005$ in the Strang splitting case changes more the solution; so that it seems to be a little better to stick to $\Delta t = 0.0025$.

On Figure 7.37, we see that more clearly that high resolution in $y - vy$ is needed: the grid 128 in y and 512 in v_y is clearly not sufficient.

On Figure 7.38, we see simulations using a splitting first by species. The time step for the ions is $\Delta t_i = 0.1$; for the electrons the time step is $\Delta t_e = 0.01$; BSL (resp. PIC) is used for the electrons on the left (resp. right) figure. The results are converged (they are compared to a "reference" solution: BSL with 6-th order time scheme and $\Delta t = 0.01$ on grid $32 \times 512 \times 32 \times 2048$). Thus, we validate the splitting by species using BSL for ions and BSL or PIC for electrons, with sub-steps for the electrons. This opens the door to use specific PIC (or BSL) schemes that are designed for capturing high oscillations (see [51]). On Figure 7.39, we compare the total energy conservation between Strang and the 6-th order splitting; we remark that the conservation is really improved with the 6-th order splitting, which is coherent with [93], where such a splitting is also used for a single species. Then, on Figures 7.40–7.44, we give some 2d plots.

On Figure 7.40, we see the $x - vx$ cut for the electrons (left) and the ions (right). We note that this picture does not change much with time; in particular, luckily, a two-stream instability is here not developed, which permits to keep a resolution small in these directions.

On Figure 7.41, we see on the contrary, that for the $y - vy$ cut for the electrons, very fine structures appear; this confirms the fact that high resolution is here needed.

On Figure 7.42, we see a Landau damping behavior in for the $y - vy$ cut for the ions.

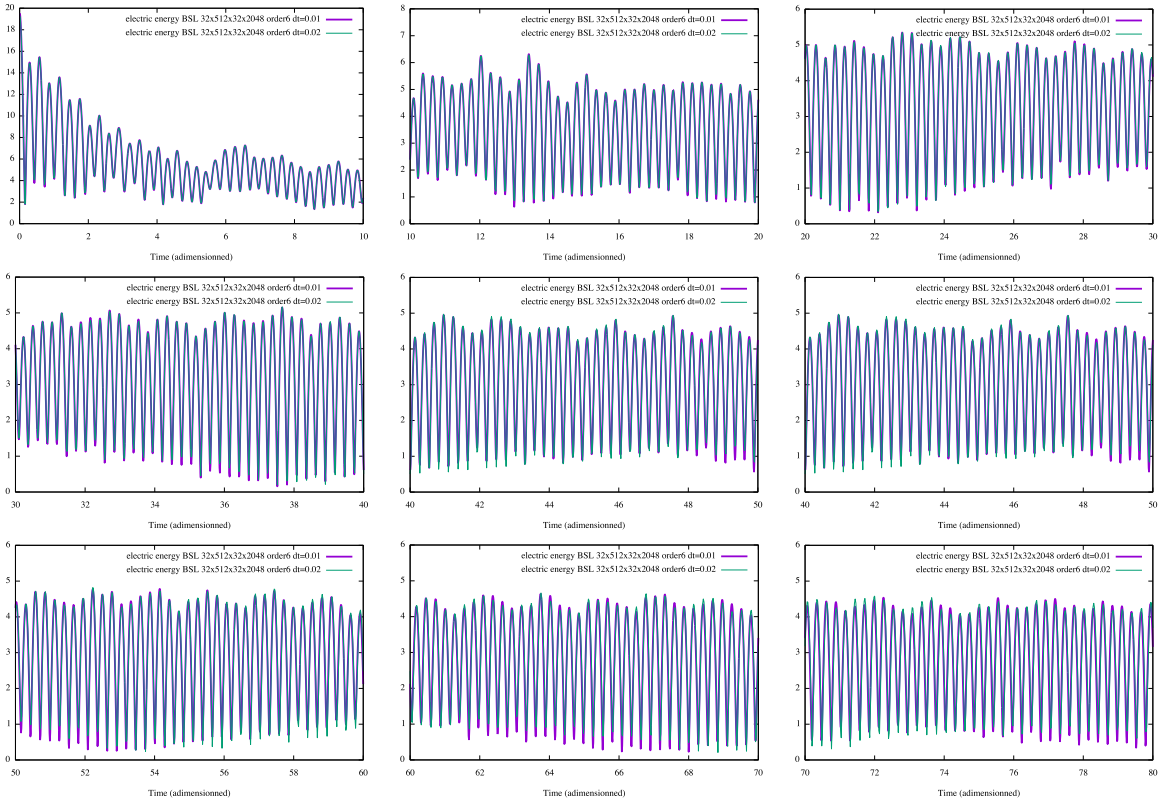


Figure 7.33 – Electric energy, 2d2v test case in (7.17) and (7.18): comparisons for 6-th order scheme and $\Delta t = 0.01$ versus $\Delta t = 0.02$, on $32 \times 512 \times 32 \times 2048$ grid with BSL.

On Figure 7.43, we see the time evolution of $\rho_e = \int_{\mathbb{R}^2} f_e(x, y, v_x, v_y)\, \mathrm{dv_x}\, \mathrm{dv_y}$, and on Figure 7.44, the time evolution of $\rho_i = \int_{\mathbb{R}^2} f_i(x, y, v_x, v_y)\, \mathrm{dv_x}\, \mathrm{dv_y}$. We see the rapid change of ρ_e with respect to time; we remark also some structures in x and the amplitude of $\rho - 1$ is small.

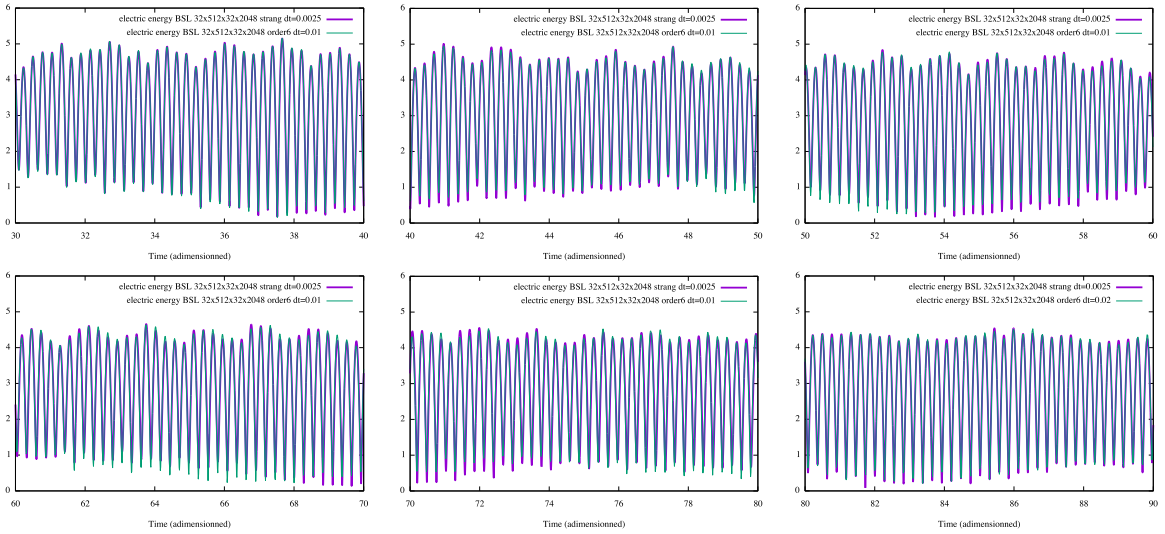


Figure 7.34 – Electric energy, 2d2v test case in (7.17) and (7.18): comparisons for 6-th order splitting and $\Delta t = 0.01$ (or $\Delta t = 0.02$) Strang splitting with $\Delta t = 0.0025$, on $32 \times 512 \times 32 \times 2048$ grid with BSL.

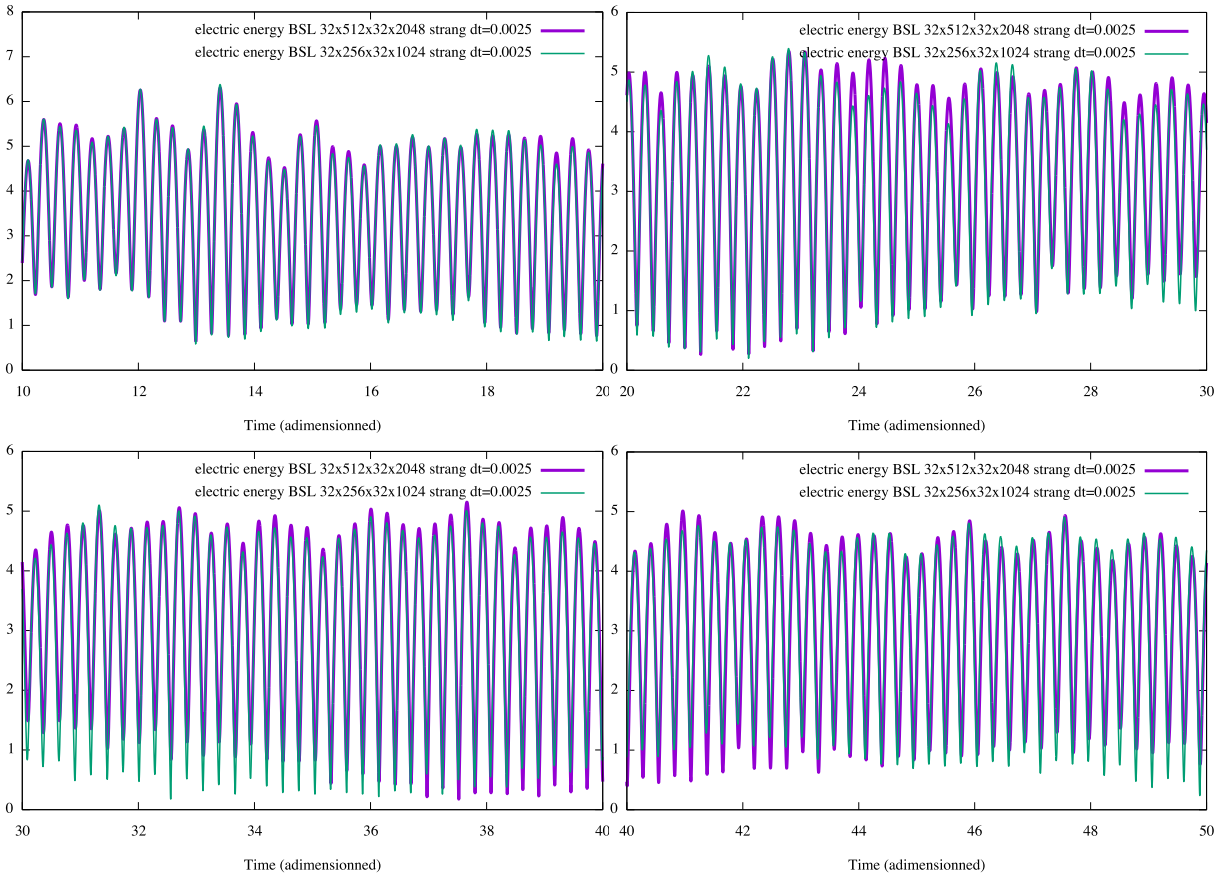


Figure 7.35 – Electric energy, 2d2v test case in (7.17) and (7.18): comparisons between $32 \times 512 \times 32 \times 2048$ grid and $32 \times 256 \times 32 \times 1024$ grid with BSL, using Strang splitting with $\Delta t = 0.0025$.

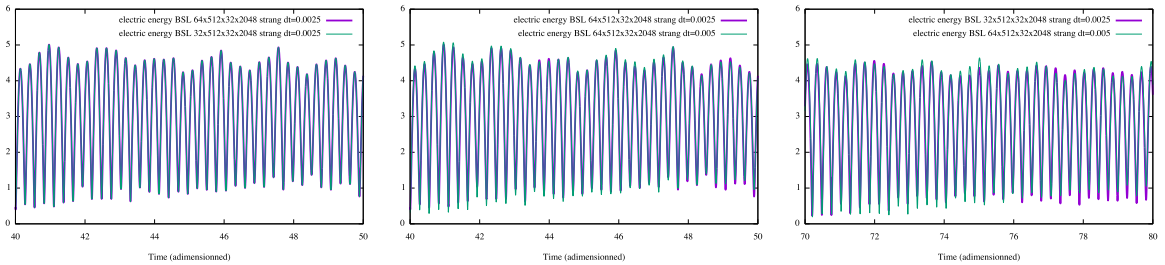


Figure 7.36 – Electric energy, 2d2v test case in (7.17) and (7.18): comparisons between grids $32 \times 512 \times 32 \times 2048$ and $64 \times 512 \times 32 \times 2048$, with Strang splitting and $\Delta t = 0.005$ or $\Delta t = 0.0025$.

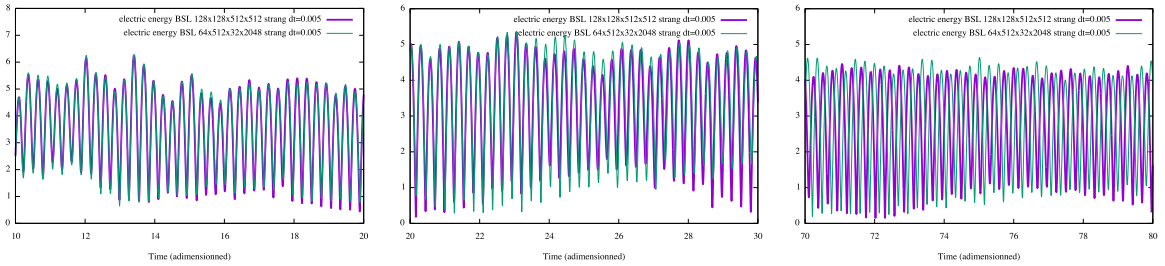


Figure 7.37 – Electric energy, 2d2v test case in (7.17) and (7.18): comparisons between $64 \times 512 \times 32 \times 2048$ grid and $128 \times 128 \times 512 \times 512$ grid with BSL, using Strang splitting with $\Delta t = 0.005$.

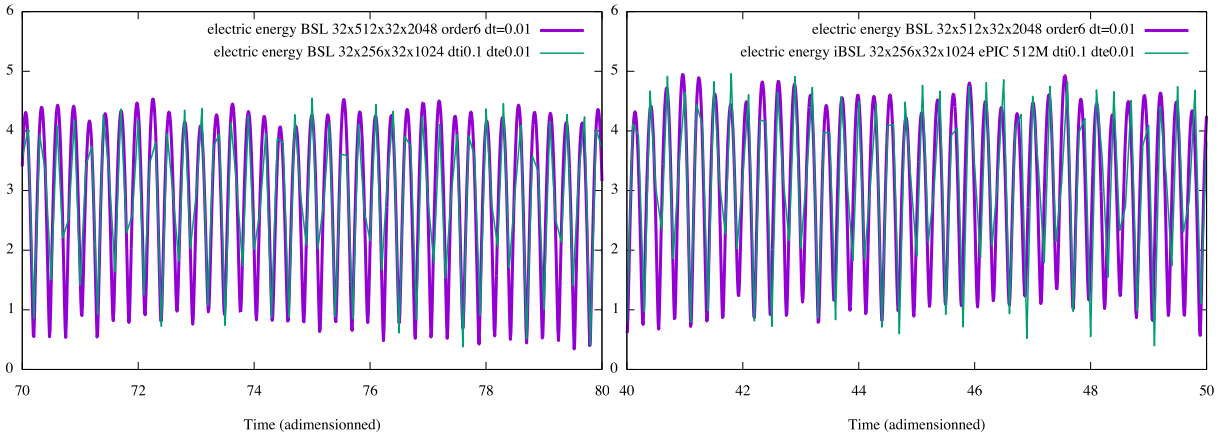


Figure 7.38 – Electric energy, 2d2v test case in (7.17) and (7.18); splitting first by species $\Delta t_{\text{ions}} = 0.1$, $\Delta t_{\text{electrons}} = 0.01$ (for the ions: BSL; for the electrons: BSL, left; PIC right), on grid $32 \times 512 \times 32 \times 2048$

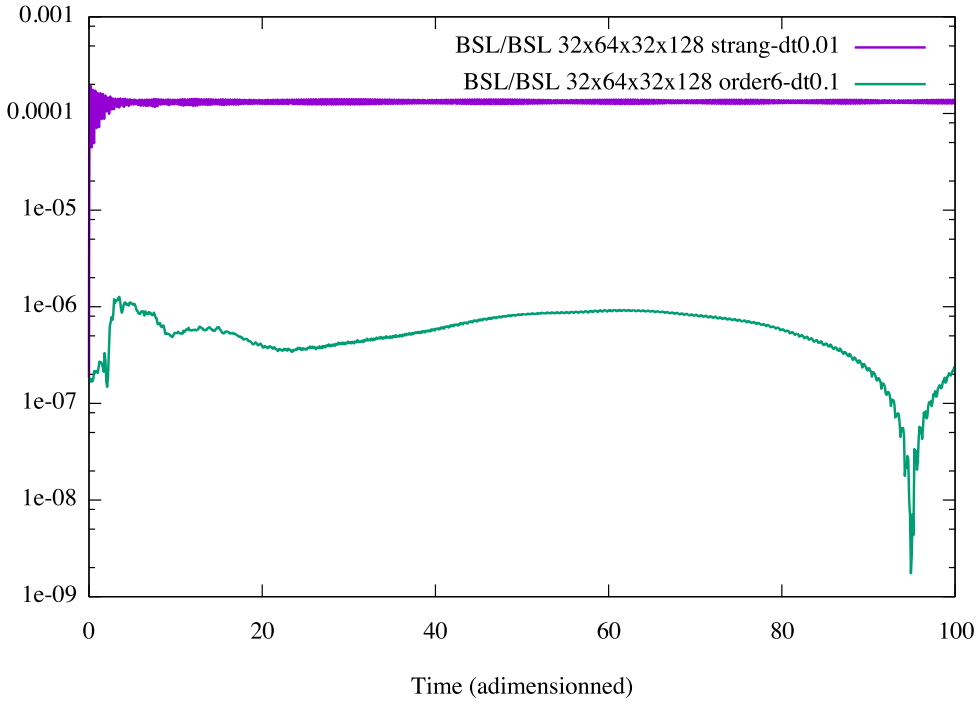


Figure 7.39 – Total energy, 2d2v test case in (7.17) and (7.18): comparison between order 6, with $\Delta t = 0.1$ and Strang, with $\Delta t = 0.01$ on $32 \times 64 \times 32 \times 128$ grid, with BSL.

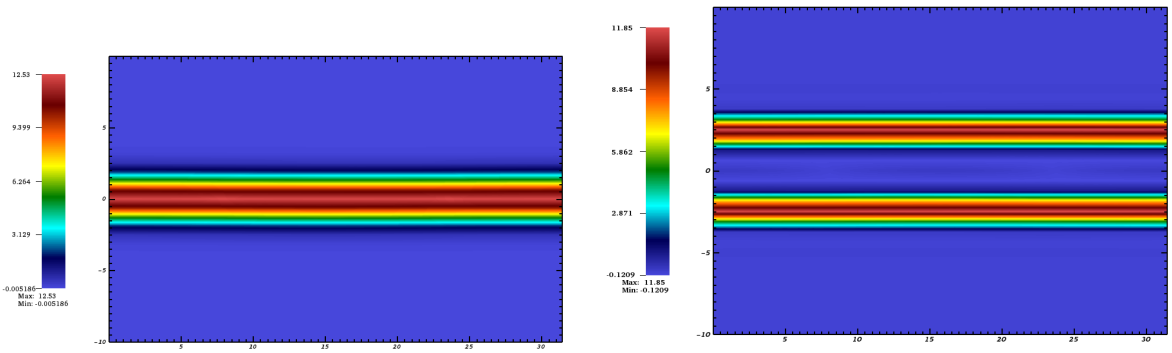


Figure 7.40 – x-vx cut electrons (left) and ions (right), 2d2v test case in (7.17) and (7.18), BSL method, at final time $t = 100$.

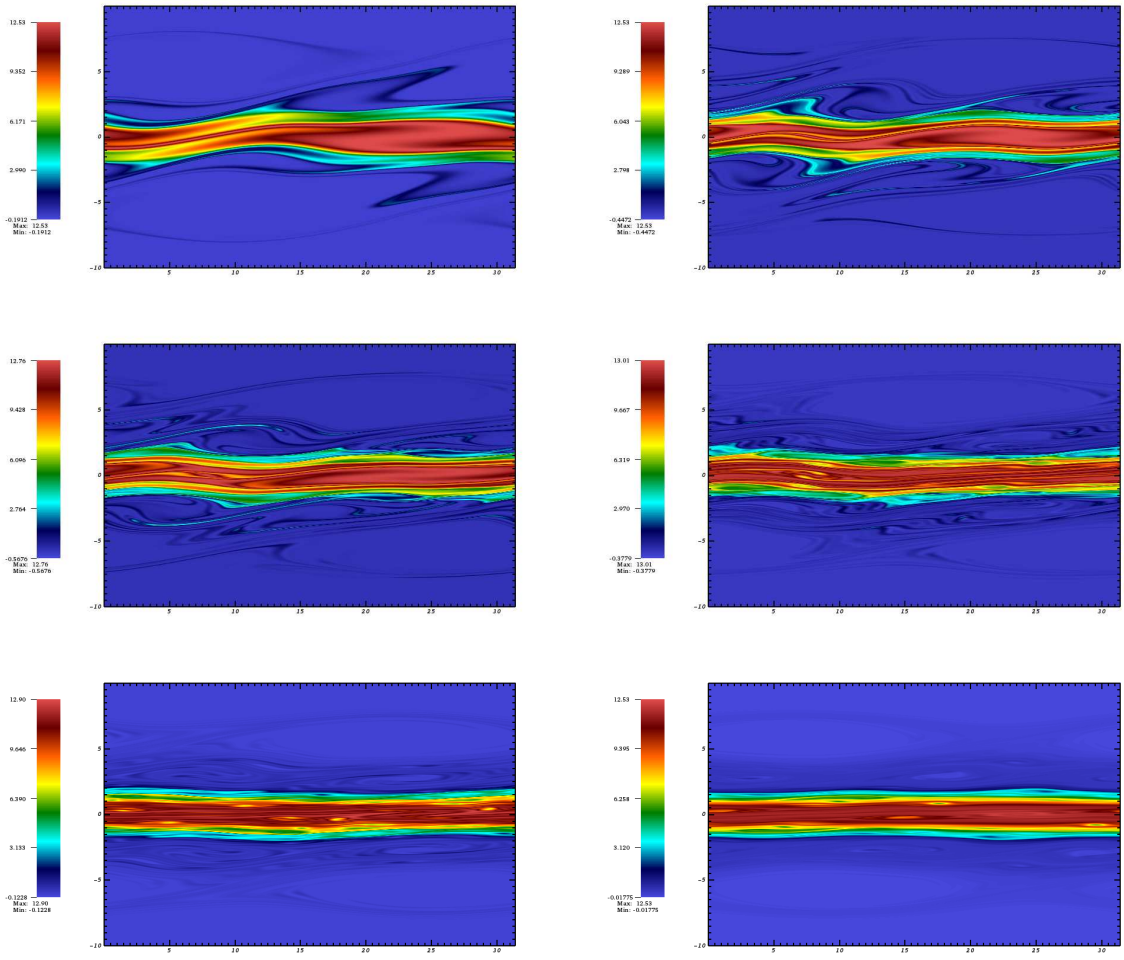


Figure 7.41 – y-vy cut electrons at times $2, 4, 5, 10, 20, 50$, 2d2v test case in (7.17) and (7.18), BSL method.

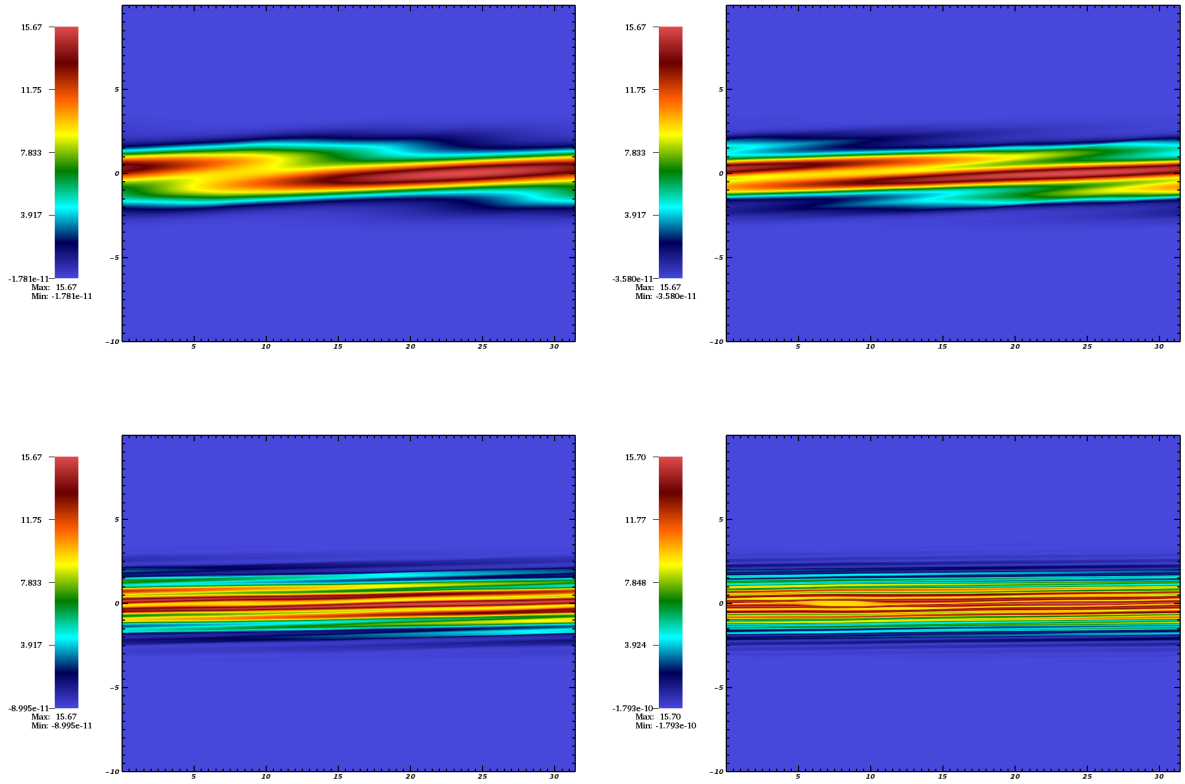


Figure 7.42 – y-vy cut ions at times $10, 20, 50, 100$, 2d2v test case in (7.17) and (7.18), BSL method.

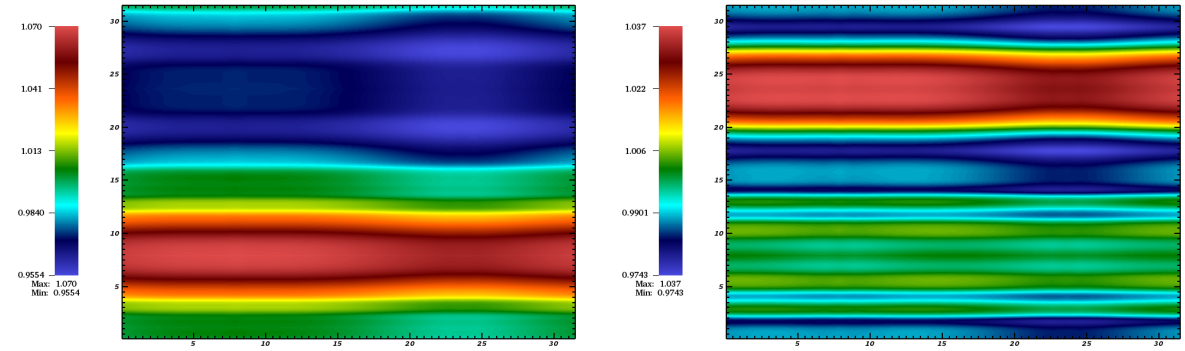


Figure 7.43 – ρ for electrons at time $44.3, 44.4$, 2d2v test case in (7.17) and (7.18), BSL method.

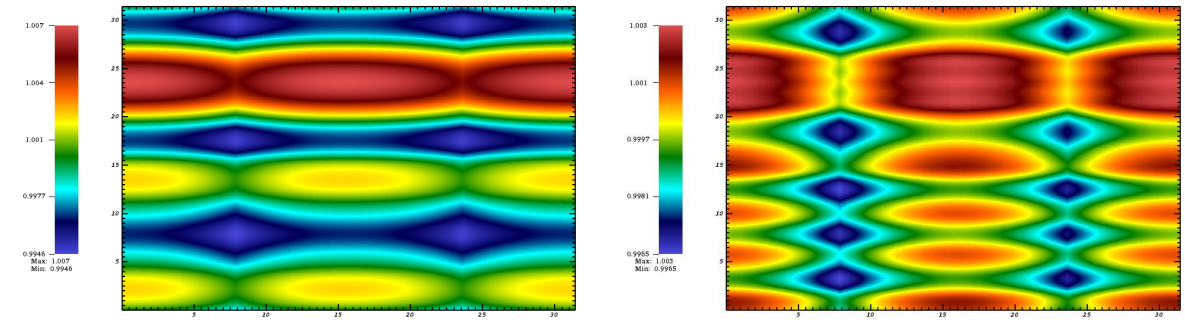


Figure 7.44 – ρ for ions at time $50, 54$, 2d2v test case in (7.17) and (7.18), BSL method.

Chapter 8

The End

This chapter will now conclude the work shown in this thesis and show some possible future directions.

First, Section 8.1 will show performance results of our different algorithms on a same architecture.

Then, Section 8.2 will give a summary of some important points from this thesis.

Finally, Section 8.3 will give some possible future work for `Pic-Vert` (future work for the semi-Lagrangian implementation was already given in Chapter 6).

8.1 Three Years in Four Graphs

This section compares all the results from Chapter 4 and Chapter 5 on a same architecture. The results presented in this section come from simulations run on different computers.

Our Inria team machine "icps-gc-6". This machine features 2 sockets, and each of those sockets is an Intel Xeon E5-2650 v3 @2.3 GHz (Haswell) with 16 GB of RAM, 2 memory channels, and 10 cores. Its theoretical memory bandwidth peak is 34 GB/s (only 2 memory channels installed on a maximum of 4[1]), its theoretical single precision floating-point operation peak is 736 GFlops/s. On this machine, we had access to `gcc 6.2` and `icc 17.0.0`[2].

The Haswell nodes of the CINES supercomputer "Occigen"[3] (2 106 nodes). Each node features 2 sockets, and each of those sockets is an Intel Xeon E5-2690 v3 @ 2.6 GHz (Haswell) with 64 GB of RAM, 4 memory channels, and 12 cores. Its theoretical memory bandwidth peak is 68 GB/s, its theoretical single precision floating-point operation peak is 998 GFlops/s. On this machine, we had access to `icc 17.0`.

Table 8.1 summarizes the architectural parameters of those machines. Because our PIC code is memory-bound, the parameter that matters most is the memory bandwidth. In this chapter, we used the hyper-threading capabilities of our architectures. On those two architectures, each core is able to host two threads.

Table 8.2 presents the test case simulated to compare our algorithms in 3d, and Table 8.3 presents the one in 3d. The 2d test case is similar to the one presented in the previous chapters (here we chose $v_{th} = 0.1$), and the 3d one is exactly the same one as used previously.

The number of particles chosen for the simulations is the maximum possible for all our algorithms on each architecture. Table 8.4 summarizes the memory requirements of our different algorithms in 2d and 3d. For our simulations with chunks, we used a value of chunkSize that enables to use as many particles as with SoA. As detailed in Chapter 5, we have two possible algorithms with chunks. For the one which uses less memory (two bags per cell), a chunkSize of 256 was always possible. For the one which uses more memory (one bag per thread per cell),

[1] `http://ark.intel.com/products/81705`

[2] Thanks to `https://software.intel.com/en-us/qualify-for-free-software/student`

[3] `https://www.cines.fr/calcul/materiels/occigen/configuration/`

we were obliged to use chunkSize = 128 in 2d on icps-gc-6, to use chunkSize = 64 in 3d on Occigen, and we were not able to use this algorithm in 3d on icps-gc-6 (chunkSize = 16 would have worked but it is too low to have good performance). This table illustrates that the memory gains thanks to our last algorithm are not worthless.

Figures 8.1–8.4 show roofline models of our algorithm on our test architectures, in 2d and 3d. Let us now analyze the data shown on those graphs.

Hyper-threading (HT). On those graphs, when hyper-threading is not shown for an algorithm, it is because the performance do not change much (or is decreased) by using it. When using SoA with 3 loops, with or without strip-mining, hyper-threading does not improves efficiency. However, when using SoA with 1 loop or when using chunks, this technology becomes useful.

Operational intensity. On those graphs, we see three different operational intensities for our algorithms. The number of memory operations associated with our "SoA 3 loops" algorithm was explained in Chapter 4, in Listings 4.27–4.30 on pages 85–87. The number of memory operations needed for our algorithms with chunks was explained in Chapter 5, in Figure 5.12 on page 116 and in footnote 16 on page 125. For our algorithms with only 1 loop or with strip-mining, the explanations are similar. The detail of those numbers is given in Table 8.5.

Figures 8.1 and 8.3 show the performance of our 2d algorithms. We see that in 2d, the SoA layout for particles gives the best performance, if we use the strip-mining on the core loops. The best operational intensity that can be attained with the particle representation chosen is 1.3. For this intensity:

- on icps-gc-6, the maximum number of Flops/s is 39 GFlops/s. We reach 64% of this maximum.
- on Occigen, the maximum number of Flops/s is 75 GFlops/s. We reach 65% of this maximum.

Figures 8.2 and 8.4 show the performance of our 3d algorithms. We see that in 3d, the chunk bags gives the best performance, if we use the algorithm with colors and with atomics (Variant 2 in Table 5.3). The best operational intensity that can be attained with the particle representation chosen is 2.9. For this intensity:

- on icps-gc-6, the maximum number of Flops/s is 87 GFlops/s. We reach 47% of this maximum.
- on Occigen, the maximum number of Flops/s is 169 GFlops/s. We reach 44% of this maximum.

Furthermore, many scientific articles show the efficiency of their implementations on a "cold plasma" test case. It is, *e.g.*, a test case where the time step is set to 0. This test case gets rid of almost all the cache misses, because particles do not move at all: they always stay in the same cell during the simulation. This is thus a test case which exhibits the maximum performance of an implementation. As argued in Chapter 5, our performance results are relatively independent from the particle velocities. The associated results are not pictured on the graphs. However, they are of course a little better.

`Pic-Vert` performance on a "cold plasma" test case.
- on icps-gc-6, we reach 69% of the maximum performance in 2d and 52% in 3d.
- on Occigen, we reach 69% of the maximum performance in 2d and 46% in 3d.

The more we approach peak performance, the more difficult it becomes to grasp the last percents that separate us from this maximum. We have seen in this thesis how to get performance improvements thanks to loop transformation, data layouts, data structures and algorithms. Grasping the last percents seems at least very challenging.

	icps-gc-6	Occigen HSW
Processor	Intel Xeon E5-2650 v3 (Haswell)	Intel Xeon E5-2690 v3 (Haswell)
RAM	16 GB	64 GB
#mem. channels	2	4
Memory bandwidth	34 GB/s	68 GB/s
#cores (#threads)	10 (20)	12 (24)
Clock frequency	2.3 GHz	2.6 GHz
Floating-point	736 GFlops/s	998 GFlops/s

Table 8.1 – Architectural parameters of one socket of our test machines.

Physical test case	Linear Landau damping [5, Section 5.15], initial distribution $f(x,y,v_x,v_y,t=0) = \left(1+0.01\cos\left(\frac{x}{2}\right)\cos\left(\frac{y}{2}\right)\right)\frac{1}{0.02\pi}\exp\left(-\frac{v_x^2+v_y^2}{0.02}\right)$
Spatial grid	$[0;4\pi)^2$ decomposed in 256^2 cells, periodic boundaries
Other parameters	Cloud-in-cell model [42], 100 iterations and $\Delta t = 0.1$
Particle crossing: averaged, per iteration	30% of the particles move 1 cell away, 0.0000082% move 2 cells away

Table 8.2 – 2d test case for comparison.

Physical test case	Linear Landau damping [5, Section 5.15], initial distribution $f(x,y,z,v_x,v_y,v_z,t=0) = \left(1+0.01\cos\left(\frac{x}{2}\right)\cos\left(\frac{y}{2}\right)\cos\left(\frac{z}{2}\right)\right)\frac{1}{(2\pi)^{3/2}}\exp\left(-\frac{v_x^2+v_y^2+v_z^2}{2}\right)$
Spatial grid	$[0;4\pi)^3$ decomposed in 64^3 cells, periodic boundaries
Other parameters	Cloud-in-cell model [42], 100 iterations and $\Delta t = 0.05$
Particle crossing: averaged, per iteration	49% of the particles move 1 cell away, 0.0015% of the particles move 2 cells away

Table 8.3 – 3d test case for comparison.

Algorithm	Memory usage, in bytes	Largest N, in millions	
		icps-gc-6	Occigen
SoA, out of place except for i_{cell}	$(28+24)\cdot N$	250	1 100
One chunk bag / thread / cell (size 128)	$\left(24+\frac{64}{\mathsf{chunkSize}}\right)\cdot N + C_1$	280	2 000
(size 256)		0	1 600
Two chunk bags / cell (size 256)	$\left(24+\frac{64}{\mathsf{chunkSize}}\right)\cdot N + C_2$	530	2 300
SoA, out of place except for i_{cell}	$(40+36)\cdot N$	190	750
One chunk bag / thread / cell (size 32)	$\left(36+\frac{64}{\mathsf{chunkSize}}\right)\cdot N + C_1$	70	1 200
(size 64)		0	830
(size 128)		0	120
Two chunk bags / cell (size 256)	$\left(36+\frac{64}{\mathsf{chunkSize}}\right)\cdot N + C_2$	270	1 500

Table 8.4 – Memory usage of our PIC implementations. N denotes the number of particles, $C_2 \approx 4\cdot\mathsf{nbCells}\cdot\mathsf{memoryOf(chunk)}$, enhanced from $C_1 \approx 2\cdot\mathsf{nbThreads}\cdot\mathsf{nbCells}\cdot\mathsf{memoryOf(chunk)}$. Top: 2d with a grid of size 256×256. Bottom: 3d with a grid of size $64\times64\times64$.

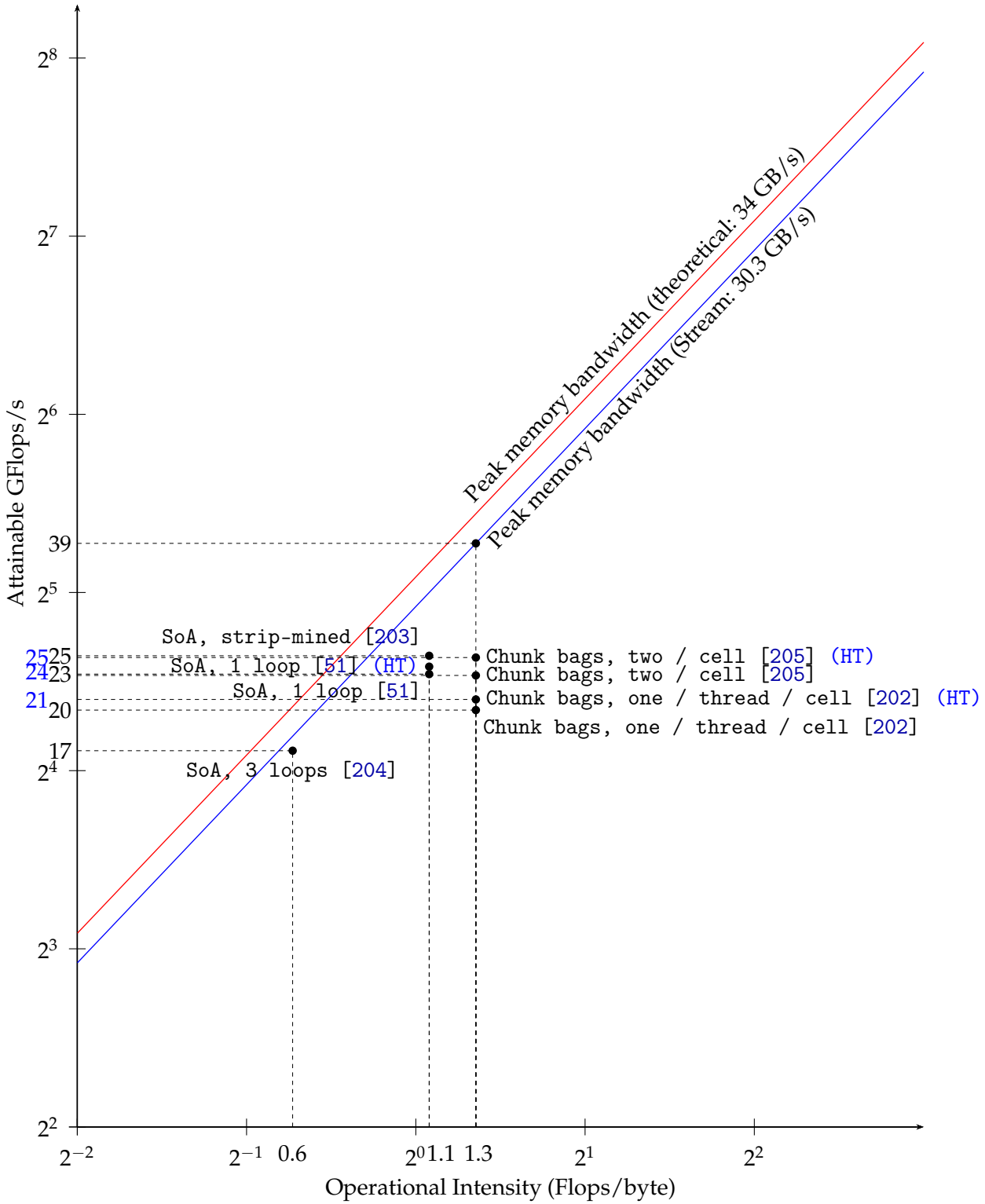


Figure 8.1 – Analysis of performance in the roofline model. 2d test case in Table 8.2 with 250 million particles, on icps-gc-6. Results with hyper-threading in blue.

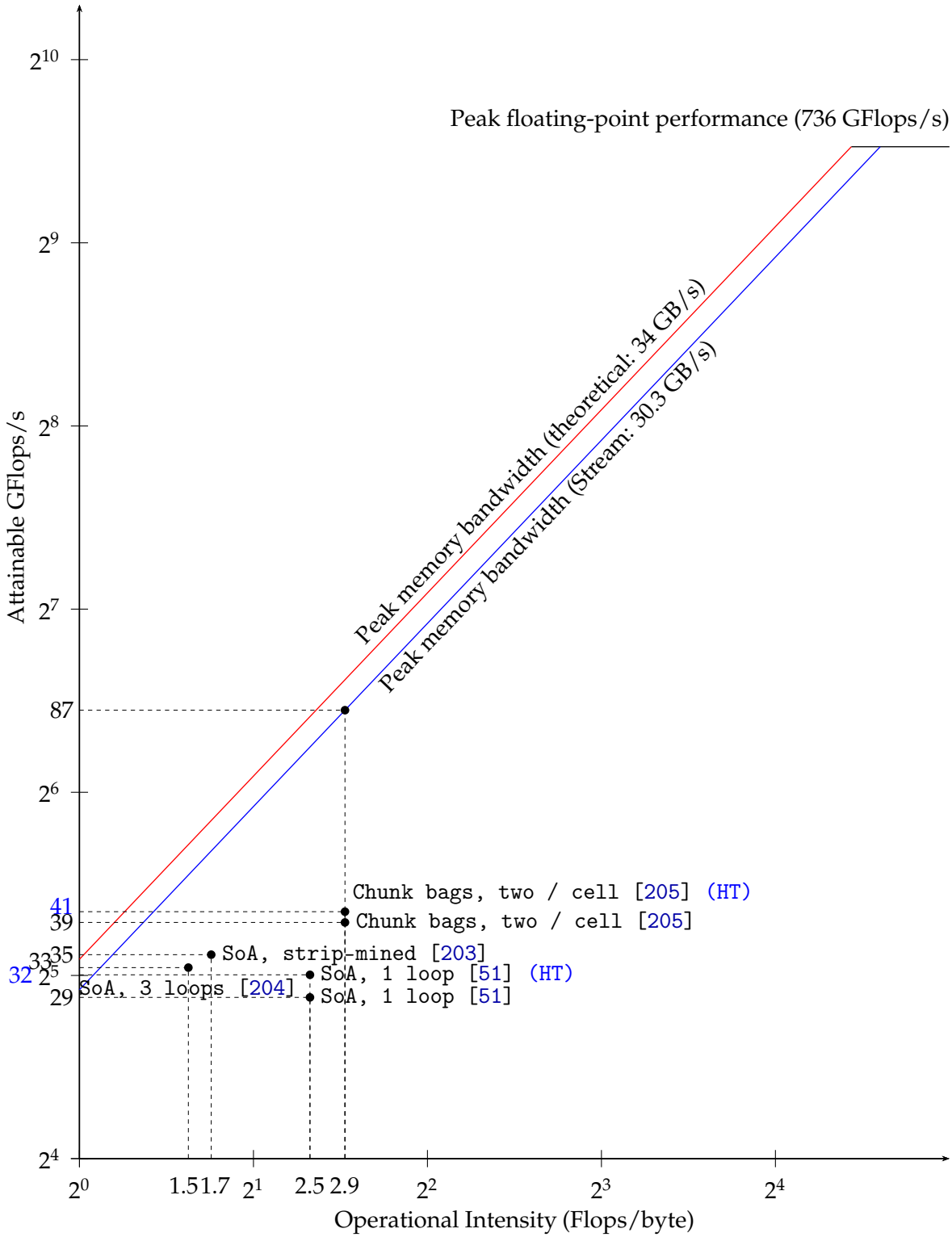


Figure 8.2 – Analysis of performance in the roofline model. 3d test case in Table 8.3 with 175 million particles, on icps-gc-6. Results with hyper-threading in blue.

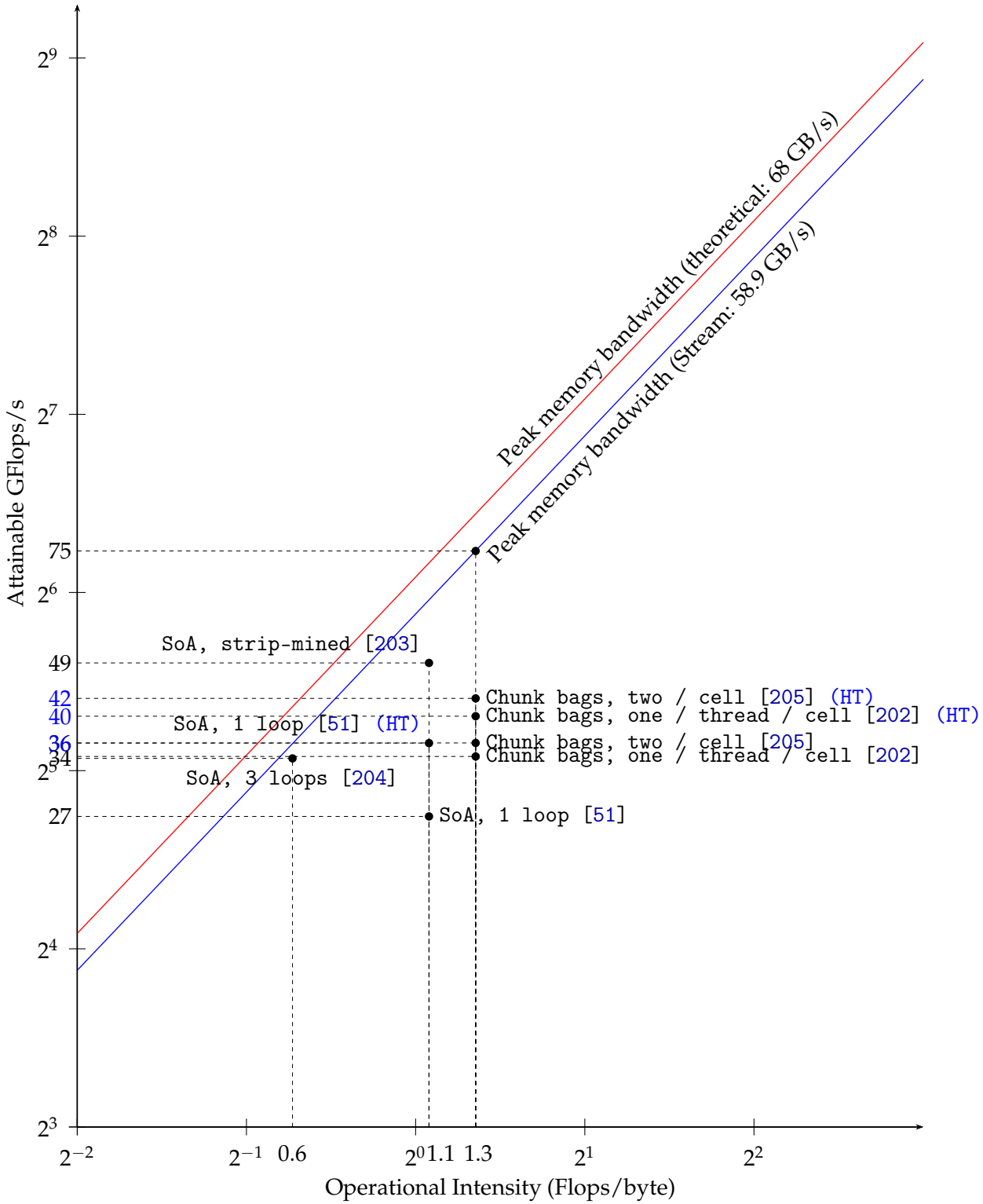


Figure 8.3 – Analysis of performance in the roofline model. 2d test case in Table 8.2 with 1 billion particles, on Occigen. Results with hyper-threading in blue.

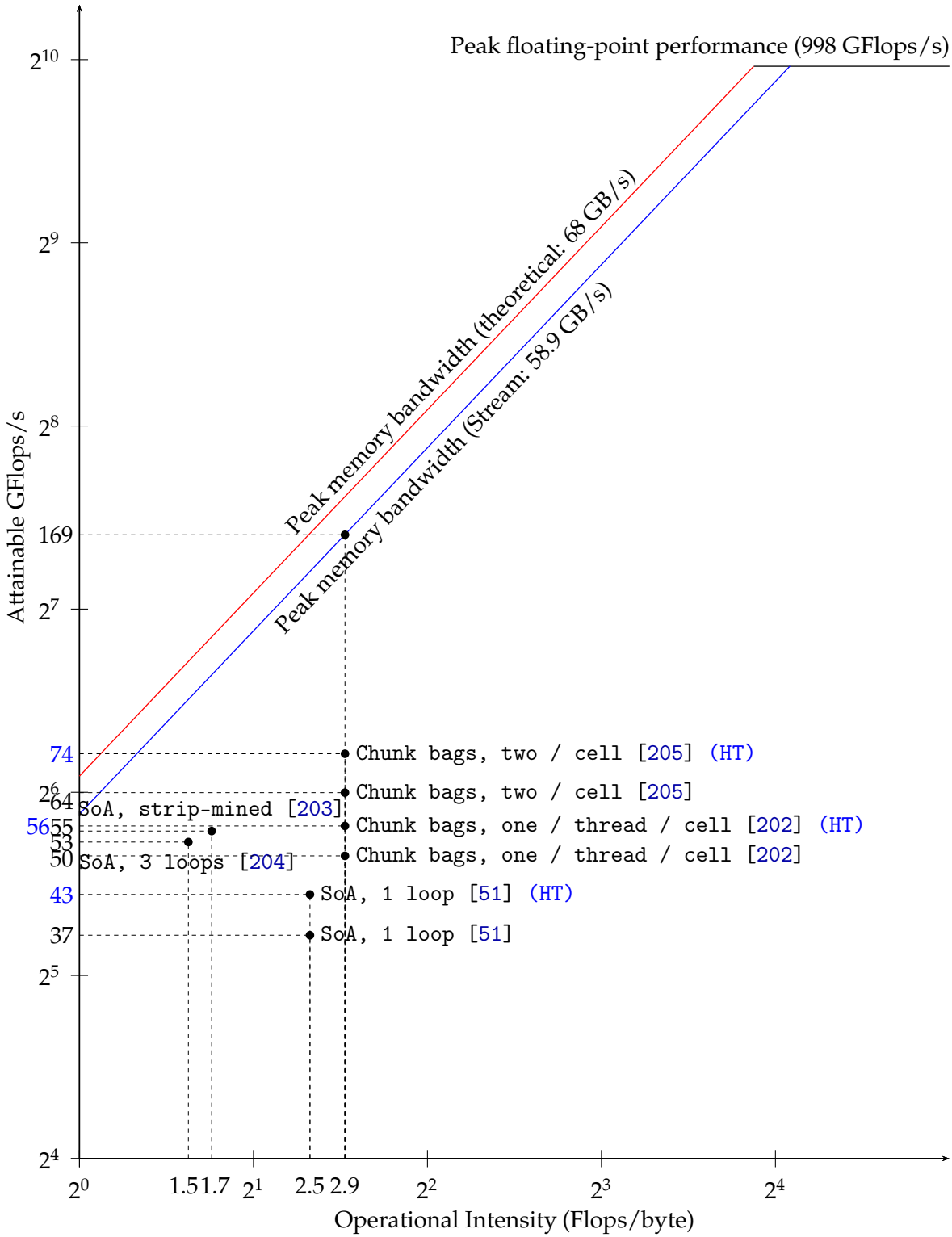


Figure 8.4 – Analysis of performance in the roofline model. 3d test case in Table 8.3 with 700 million particles, on Occigen. Results with hyper-threading in blue.

Algorithm	# floating-point operations	Memory moved, in bytes	Operational intensity
SoA, 3 loops	62	$44 + 44 + 12 + 68 \cdot \frac{4}{100}$	0.6
SoA, strip-mining	62	$28 \cdot 2 + 68 \cdot \frac{4}{100}$	1.1
SoA, 1 loop	62	$28 \cdot 2 + 68 \cdot \frac{4}{100}$	1.1
Chunk bags	62	$\left(24 + \frac{64}{\text{chunkSize}}\right) \cdot 2$	1.3
SoA, 3 loops	209	$64 + 52 + 16 + 84 \cdot \frac{4}{100}$	1.5
SoA, strip-mining	209	$64 + 56 + 84 \cdot \frac{4}{100}$	1.7
SoA, 1 loop	209	$40 \cdot 2 + 84 \cdot \frac{4}{100}$	2.5
Chunk bags	209	$\left(36 + \frac{64}{\text{chunkSize}}\right) \cdot 2$	2.9

Table 8.5 – Operational intensities of our PIC implementations. The numbers in the second and third column are given per particle per time step. The SoA implementations perform a sorting every 20 iterations, hence 4 sortings on 100 iterations. In 2d, the strip-mining is on the 3 loops, and in 3d on the two last loops (update-positions and accumulate). Top: 2d. Bottom: 3d.

8.2 Takeaways

In Chapter 4, a lot of technical work was described. Suppose now that one is to optimize his or her implementation. What lessons can be learned from our experiments?

> “ Traduttore, traditore.[4]
>
> Italian expression ”

Translation. One of our first works was to translate a `Fortran` code to a `C` code. Apart from the usual indices problems (`Fortran` arrays are indexed from 1 to n, `C` arrays are indexed from 0 to $n-1$), the bug which was the most annoying to identify was related to the pseudo-random generator. As shown in Section 2.3, the native `Fortran` generator `random_number` is “good enough” for PIC simulations, but not the native `C` generator `rand`. We thus recommand to be really careful when translating a code from one language to another. A lot of frustration can come when a library has different APIs in two languages (*e.g.*, HDF5), but it is a lot more frustrating to identify problems when the APIs are the same whereas the behaviors are not.

Data layouts. On modern computer architectures, the structure of arrays layout is required for efficient vectorization. This optimization is not hard to implement, but it may take some time without the appropriate search and replace tools. If an implementation would benefit from vectorization, we would highly recommend to implement this optimization. Practical ways to use vector instructions were presented in Section 4.3.4, which are generic enough to be used in many situations. Be aware that vectorization reports, which are extremely useful, highly depend on the compiler.

Performance metrics. To better understand the behavior of one’s implementation, a first useful step is to look at the roofline model [180]. To understand the floating-point operations performed and the memory moved, one possibility is to use tools such as Intel VTune Amplifier if using Intel architecture (this is what is done in, *e.g.*, PICADOR [81]) or the CUDA Metric API from CUDA 5.5 if using GPUs with CUDA code (this is what is done in, *e.g.*, PIConGPU [49]). Another possibility is to manually count the operations performed and the memory accesses inside the code (this is what is done in, *e.g.*, GTC [88] and in the Stream benchmark [162]). In this work, we also counted manually the operations and memory moved. In the process of doing it, we identified common subexpressions that, when factorized, improved performance

[4] Each translator is a traitor. (translation by this manuscript’s traitor)

($\approx$ 1.5%). We also better understood the behavior of our implementation. Thus, we have no choice but to advise others to do so as well.

Once the metrics are well understood, one should now know if his or her implementation is memory bound or compute bound. This is crucial to understand the scalability on multi-core architectures. Modern architectures usually have less memory channels than cores. Thus, if an implementation is memory bound and efficient, it is not expected to scale ideally with respect to the number of cores. When first looking at the scalability of `Pic-Vert` on icps-gc-6, we had a speedup of 3.4 on 10 cores with respect to 1 core. We had a hard time understanding this behavior until we saw that we had only 2 memory channels. For optimization matters, it is thus really important to understand it, of course, but also to understand the hardware that will run it.

Import our optimizations inside another PIC implementation. Let us note that the crucial optimizations presented in Chapter 4 (space-filling curves, strip-mining) are easy to implement. For the space-filling curves, the hard work was to design the correct curve and to test the different possible implementations. Once this work is finished, there is just a macro to copy/-paste – provided that you use the "index plus offset" representation (if not, this change is more time-consuming). For the strip-mining: as already stated, when looking at the performance results and analyzing them correctly, this transformation is really natural and it is really easy to implement, compare Listing 4.34 and Listing 4.35 on page 98. The only important thing was to test different ways of applying this transformation. Our first idea (apply it on the 3 steps) did not work in 3d, but it worked by applying it on the 2 last steps only. This was not expected at first, but it underlines the importance of experiments on top of an accurate theoretical analysis.

Debugging. Most of our performance results are obtained thanks to `icc`. However, for debugging purposes, we used only `gcc`. We present here what we used for debugging, and in which situations it was useful:

- Test parts of code independently.
- Add `printfs` in the code. This was our main debugging tool, in every situation.
- Compile with `-fsanitize=undefined` or `-fsanitize=address` to get out of bounds error on arrays.
- Compile with `-Wall` to get all usual warnings. And remove them.
- Compile with `-Wfloat-conversion`. This helps locating errors from automatic casts. An example of such errors is using `abs` from `<math.h>` instead of `fabs` for real numbers (there would be a hidden cast to an integer without this compilation option).
- Compile with `-Wmissing-field-initializers`. This helps locating errors coming from fields not initialized in structures. It is perfectly legal to omit field structures, and they will be set to 0. But sometimes, it is not what was intended when writing the code.
- When everything else fails to an ennoying `segfault` whose line cannot be located:
  - compile with `-g`;
  - run the executable and locate in the standard error the offset producing the `segfault` (in hexadecimal, *e.g.*, `0x418287`);
  - execute `addr2line -e /path/to/buggy_executable.out offset`.

8.3 Perspectives

Particle representation. During this work, we sticked to the "index plus offset" representation of particles. This representation is a little less precise than using plain `doubles` for positions, but uses `doubles` for velocities. It would be very interesting to compare the precision of simulations using `doubles` to the precision of simulations using `floats`. Whenever `floats` are sufficiently precise for velocities, it could be more efficient to use them, to reduce the memory footprint of a particle, hence to reduce the memory bandwidth.

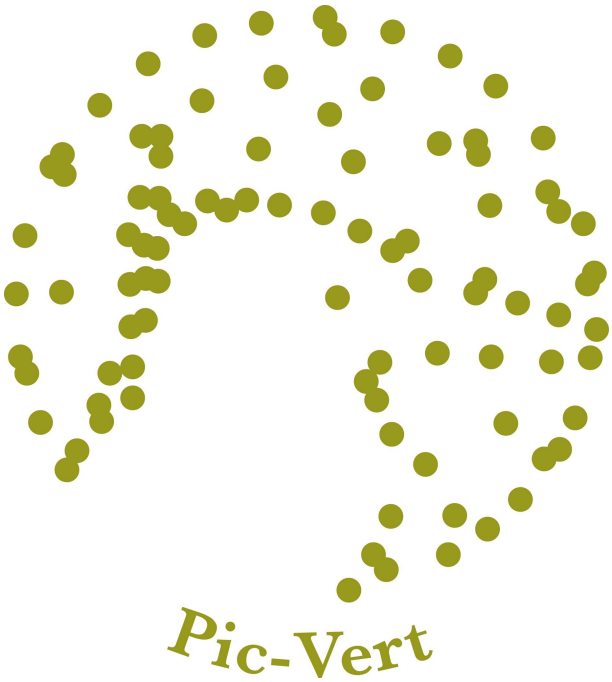
Particle shapes. As shown in the previous section, `Pic-Vert` achieves close to optimal performance results on modern CPU architectures. But those results were obtained with the Cloud-in-Cell model. It would be interesting to see what can be reached with higher interpolation levels. When the interpolation level rises, it means more computation but also more accesses in the E and ρ arrays.

Distributed memory parallelism. In this work, we focused on multi-core and SIMD parallelism. In future work, it would be great to extend our algorithm with a layer of domain decomposition, using `MPI` communications. One critical aspect is the exchange of particles crossing domain boundaries. We speculate that chunks could be used as buffers for emission and reception of particles reaching the cells at the frontier of a domain. These chunks could then be merged, at the end of the time step, with the locally-processed chunks. The flexibility offered by chunks might be helpful for dealing with dynamically-sized domains.

Architectures. Furthermore, it would be interesting to adapt our algorithm to target architectures with larger number of cores, such as Graphics Processing Unit (GPU) or Many Integrated Core (MIC). We think that the organization in chunks could help addressing the issue of load balancing, which is critical on these architectures (*e.g.*, [68]).

Language. During this work we used `C` for the implementation. We tried our best to make our code readable and easy to maintain, and we hope that the result, which can be downloaded at `http://www.barsamian.am/Pic-Vert/`, is not too far from our goal. However, we reached a point where, because of a lack of modularity in the language, we copied/pasted a lot of code when testing the different data structures in different dimensions. If our implementation is to be used in the long term, it would probably be a good idea to port it, *e.g.*, to `C++` and use templates to factorize more code.

Other home trees for `Pic-Vert`? Last but not least, we hope that (parts of) our implementation will be used by other scientists. We devoted a lot of time to this implementation, proved that it is more efficient that other PIC implementation (see Tables 3.1–3.3 in Section 3.3), and we thus think that it would be good to use some of our ideas in other PIC implementations.



Chapter 9

References

Books and Thesis Manuscripts

[1] U. A. Acar and G. E. Blelloch. *Algorithm Design: Parallel and Sequential*. 2017 (cit. on pp. 24, 219).
URL: http://www.parallel-algorithms-book.com.

[2] M. Bader. *Space-Filling Curves*. Vol. 9. Texts in Computational Science and Engineering. Springer, Berlin, Heidelberg, 2013 (cit. on p. 47).
DOI: 10.1007/978-3-642-31046-1.

[3] R. Barthelmé. "Le problème de conservation de la charge dans le couplage des équations de Vlasov et de Maxwell". PhD thesis. Université de Strasbourg, 2005 (cit. on p. 37).
URL: http://scd-theses.u-strasbg.fr/998/01/barthelme.pdf.

[4] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation*. 1st edition. McGraw-Hill, New York, 1985 (cit. on p. 51).

[5] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation*. 2nd edition. The Adam Hilger Series on Plasma Physics. IOP Publishing Ltd, 1991 (cit. on pp. 21, 33, 37, 49, 62, 89, 113, 114, 150, 151, 179, 216).

[6] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation*. 3rd edition. Series in Plasma Physics. CRC Press, 2004 (cit. on pp. 37, 49).
URL: https://www.crcpress.com/Plasma-Physics-via-Computer-Simulation/Birdsall-Langdon/p/book/9780750310253.

[7] M. Bonitz and D. Semkat. *Introduction to Computational Methods in Many Body Physics*. Rinton Press, 2006 (cit. on p. 49).
URL: http://www.rintonpress.com/books/mbonitz.html.

[8] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer, New York, NY, 1987 (cit. on p. 38).
DOI: 10.1007/978-1-4419-8724-2.

[9] O. Buneman. "TRISTAN The 3-D Electromagnetic Particle Code". In: *Computer Space Plasma Physics: Simulation Techniques and Software*. Ed. by H. Matsumoto and Y. Omura. Terra Scientific Publishing Company (TERRAPUB), Tokyo, 1993, pp. 67–84 (cit. on p. 50).
URL: https://www.terrapub.co.jp/e-library/cspp/.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 3rd edition. The MIT Press, 2009 (cit. on pp. 40, 44, 76, 206).

[11] P. A. M. Dirac. *The Principles of Quantum Mechanics*. 4th edition. The International Series of Monographs on Physics. Oxford University Press, 1958 (cit. on p. 32).
URL: https://archive.org/details/DiracPrinciplesOfQuantumMechanics.

[12] M. Durand. "PaVo. An Adaptative Parallel Sorting Algorithm." PhD thesis. Université de Grenoble, 2013 (cit. on pp. 102, 110, 121).
URL: https://tel.archives-ouvertes.fr/tel-01137944.

[13] L. Euler. *Institutionum Calculi Integralis*. (Translation available in English by I. Bruce http://www.17centurymaths.com/contents/integralcalculusvol1.htm). 1768 (cit. on pp. 21, 216).
DOI: 10.3931/e-rara-29427.

[14] L. C. Evans. *Partial Differential Equations*. Vol. 19. Graduate Studies in Mathematics. American Mathematical Society (AMS), 1998 (cit. on pp. 22, 216).
URL: https://bookstore.ams.org/gsm-19-r.

[15] J. Fritz. *Partial differential equations*. 3rd edition. Vol. 1. Applied Mathematical Sciences. Springer, New York, NY, 1978 (cit. on pp. 22, 216).
DOI: 10.1007/978-1-4684-0059-5.

[16] J. E. Gentle. *Random Number Generation and Monte Carlo Methods*. 2nd edition. Statistics and Computing. Springer, New York, NY, 2003 (cit. on pp. 43, 44).
DOI: 10.1007/b97336.

[17] J. Gustedt. *Modern C*. 2018 (cit. on p. 207).
URL: http://icube-icps.unistra.fr/index.php/File:ModernC.pdf.

[18] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I*. 2nd edition. Vol. 8. Springer Series in Computational Mathematics. Springer, Berlin, Heidelberg, 2008 (cit. on pp. 21, 216).
DOI: 10.1007/978-3-540-78862-1.

[19] E. Hairer, G. Wanner, and C. Lubich. *Geometric Numerical Integration*. 2nd edition. Vol. 31. Springer Series in Computational Mathematics. Springer, Berlin, Heidelberg, 2006 (cit. on p. 135).
DOI: 10.1007/3-540-30666-8.

[20] J. M. Hammersley and D. C. Handscomb. *Monte Carlo Methods*. Monographs on Applied Probability and Statistics. Springer, Dordrecht, 1964 (cit. on pp. 32, 45).
DOI: 10.1007/978-94-009-5819-7.

[21] M. Harris, S. Sengupta, and J. D. Owens. "Parallel Prefix Sum (Scan) with CUDA". In: *GPU Gems 3*. Ed. by H. Nguyen. Addison Wesley, 2007, pp. 851–876 (cit. on p. 79).
URL: https://www.mimuw.edu.pl/~ps209291/kgkp/slides/scan.pdf.

[22] R. G. Hemker. "Particle-In-Cell Modeling of Plasma-Based Accelerators in Two and Three Dimensions". PhD thesis. University of California, 2000 (cit. on p. 49).
URL: https://arxiv.org/abs/1503.00276.

[23] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. Taylor & Francis, Inc., 1988 (cit. on pp. 46, 51, 62, 89, 101, 113, 151).
DOI: 10.1201/9781439822050.

[24] M. Kaku. *Physics of the future. How science will shape human destiny and our daily lives by the year 2100.* Doubleday, 2011 (cit. on pp. 15, 17, 209, 210).

[25] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd edition. Addison Wesley Longman Publishing Co., Inc., 1997 (cit. on p. 107).

[26] N. A. Krall and A. W. Trivelpiece. *Principles of Plasma Physics*. McGraw-Hill, New York, 1973 (cit. on p. 150).

[27] H. Matsumoto and Y. Omura. "Particle Simulations of Electromagnetic Waves and its Applications to Space Plasmas". In: *Computer Simulations of Space Plasmas*. Ed. by H. Matsumoto and T. Sato. Terra Scientific Publishing Company (TERRAPUB) and Reidel Co., 1985, pp. 43–102 (cit. on p. 49).

[28] M. Mehrenberger. "Ingham inequalities and semi-Lagrangian schemes for the Vlasov equation". PhD thesis. Université de Strasbourg, 2012 (cit. on p. 136).
URL: https://tel.archives-ouvertes.fr/tel-00735678.

[29] M. Melzani. "Collisionless magnetic reconnection in relativistic plasmas with particle-in-cell simulations". PhD thesis. École Normale Supérieure de Lyon, 2014 (cit. on p. 50).
URL: https://tel.archives-ouvertes.fr/tel-01126912.

[30] Y. Omura. "One-dimensional Electromagnetic Particle Code: KEMPO1 A Tutorial on Microphysics in Space Plasmas". In: *Advanced Methods for Space Simulations*. Ed. by H. Usui and Y. Omura. Terra Scientific Publishing Company (TERRAPUB), Tokyo, 2007, pp. 1–21 (cit. on p. 49).
URL: http://www.terrapub.co.jp/e-library/amss/.

[31] G. Pólya. *How to Solve It*. Princeton University Press, 1945 (cit. on pp. 69, 97).

[32] B. D. Ripley. *Stochastic Simulation*. Wiley Series in Probability and Statistics. John Wiley & Sons, Inc., 1987 (cit. on p. 37).
DOI: 10.1002/9780470316726.

[33] X. Sáez Pous. "Particle-in-Cell Algorithms for Plasma Simulations on Heterogeneous Architectures". PhD thesis. Universitat Politècnica de Catalunya, 2016 (cit. on pp. 26, 49, 221).
URL: http://hdl.handle.net/10803/381258.

[34] C. Severance and K. Dowd. *High Performance Computing*. OpenStax CNX, 2010 (cit. on pp. 59, 63).
URL: http://cnx.org/content/col11136/1.5/.

[35] E. Sonnendrücker. *Numerical Methods for the Vlasov–Maxwell equations*. (Book in preparation) (cit. on pp. 19, 22, 41, 149, 150, 155, 213, 217).

[36] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on pp. 62, 63, 93, 97).

Articles - Particle-in-Cell

[37] W. An, V. K. Decyk, W. B. Mori, and T. M. Antonsen. "An improved iteration loop for the three dimensional quasi-static particle-in-cell algorithm: QuickPIC". In: *Journal of Computational Physics* 250 (2013), pp. 165–177 (cit. on p. 49).
DOI: 10.1016/j.jcp.2013.05.020.

[38] M. A. Asgarian, J. P. Verboncoeur, A. Parvazian, and R. Trines. "Kinetic simulation of the O-X conversion process in dense magnetized plasmas". In: *Physics of Plasmas* 20.10 (2013), p. 102516 (cit. on p. 50).
DOI: 10.1063/1.4826977.

[39] A. Y. Aydemir. "A unified Monte Carlo interpretation of particle simulations and applications to non-neutral plasmas". In: *Physics of Plasmas* 1.4 (1994), pp. 822–831 (cit. on p. 32).
DOI: 10.1063/1.870740.

[40] S. Bastrakov, R. Donchenko, A. Gonoskov, E. Efimenko, A. Malyshev, I. Meyerov, and I. Surmin. "Particle-in-cell plasma simulation on heterogeneous cluster systems". In: *Journal of Computational Science* 3.6 (2012), pp. 474–479 (cit. on p. 49).
DOI: 10.1016/j.jocs.2012.08.012.

[41] J. T. Beale and A. Majda. "Vortex Methods. II: Higher Order Accuracy in Two and Three Dimensions". In: *Mathematics of Computation* 39.159 (1982), pp. 29–52 (cit. on p. 32).
DOI: 10.1090/S0025-5718-1982-0658213-7.

[42] C. K. Birdsall and D. Fuss. "Clouds-in-Clouds, Clouds-in-Cells Physics for Many-Body Plasma Simulation". In: *Journal of Computational Physics* 3 (1969), pp. 494–511 (cit. on pp. 47, 62, 89, 114, 179).
DOI: 10.1006/jcph.1997.5723.

[43] K. J. Bowers. "Accelerating a Particle-in-Cell Simulation Using a Hybrid Counting Sort". In: *Journal of Computational Physics* 173.2 (2001), pp. 393–411 (cit. on pp. 60, 100, 110, 121).
DOI: 10.1006/jcph.2001.6851.

[44] K. J. Bowers. "Petascale Kinetic Plasma Simulation with VPIC and Roadrunner". In: Prospects in Theoretical Physics (PiTP). 2009 (cit. on pp. 35, 59, 65).
URL: https://video.ias.edu/Bowers.

[45] K. J. Bowers. "Speed optimal implementation of a fully relativistic particle push with charge conserving current accumulation on modern processors". In: *Proceedings of the 18th Int. Conf. Numerical Simulation of Plasmas (ICNSP)*. 2003, pp. 383–386 (cit. on pp. 35, 50).
URL: http://web.mit.edu/ned/ICNSP/ICNSP_BookofAbstracts.pdf.

[46] K. J. Bowers, B. J. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, and T. J. T. Kwan. "Advances in petascale kinetic plasma simulation with VPIC and Roadrunner". In: *Journal of Physics: Conference Series* 180.1 (2009), p. 012055 (cit. on pp. 37, 50).
DOI: 10.1088/1742-6596/180/1/012055.
Slides: http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-09-06299

[47] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation". In: *Physics of Plasmas* 15.5 (2008), p. 055703 (cit. on pp. 34, 54, 56, 76, 94, 100, 105, 110, 121, 124).
DOI: 10.1063/1.2840133.

[48] H. Burau, R. Widera, W. Honig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T. E. Cowan, R. Sauerbrey, and Bussmann M. "PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster". In: *IEEE Transactions on Plasma Science* 38.10 (2010), pp. 2831–2839 (cit. on pp. 48, 49).
DOI: 10.1109/TPS.2010.2064310.

[49] M. Bussmann, H. Burau, T. E. Cowan, A. Debus, A. Huebl, G. Juckeland, T. Kluge, W. E. Nagel, R. Pausch, F. Schmitt, U. Schramm, J. Schuchart, and R. Widera. "Radiative Signatures of the Relativistic Kelvin–Helmholtz Instability". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2013, 5:1–5:12 (cit. on pp. 37, 56, 65, 100, 101, 105, 110, 121, 184).
DOI: 10.1145/2503210.2504564.

[50] J. A. Byers and M. Grewal. "Perpendicularly Propagating Plasma Cyclotron Instabilities Simulated with a One-Dimensional Computer Model". In: *The Physics of Fluids* 13.7 (1970), pp. 1819–1830 (cit. on p. 45).
DOI: 10.1063/1.1693160.

[51] E. Chacon-Golcher, S. A. Hirstoaga, and M. Lutz. "Optimization of Particle-In-Cell simulations for Vlasov–Poisson system with strong magnetic field". In: *ESAIM: Proceedings and Surveys* 53 (2016), pp. 177–190 (cit. on pp. 37, 55, 59, 76, 169, 180–183).
DOI: 10.1051/proc/201653011.

[52] G.-H. Cottet and P. A. Raviart. "Particle Methods for the One-Dimensional Vlasov–Poisson Equations". In: *SIAM Journal on Numerical Analysis* 21.1 (1984), pp. 52–76 (cit. on p. 32).
DOI: 10.1137/0721003.

[53] V. K. Decyk and T. V. Singh. "Particle-in-Cell algorithms for emerging computer architectures". In: *Computer Physics Communications* 185.3 (2014), pp. 708–719 (cit. on pp. 55, 65, 75, 101).
DOI: 10.1016/j.cpc.2013.10.013.

[54] V. K. Decyk, S. R. Karmesin, A. de Boer, and P. C. Liewer. "Optimization of particle-in-cell codes on reduced instruction set computer processors". In: *Computers in Physics* 10.3 (1996), pp. 290–298 (cit. on pp. 35, 73, 76, 100).
DOI: 10.1063/1.168571.

[55] J. Denavit and J. M. Walsh. "Nonrandom initializations of Particle Codes". In: *Comments on Plasma Physics and Controlled Fusion* 6.6 (1981), pp. 209–223 (cit. on p. 45).

[56] J. Derouillat, A. Beck, F. Pérez, T. Vinci, M. Chiaramello, A. Grassi, M. Flé, G. Bouchard, I. Plotnikov, N. Aunai, J. Dargent, C. Riconda, and M. Grech. "Smilei : A collaborative, open-source, multi-purpose particle-in-cell code for plasma simulation". In: *Computer Physics Communications* 222 (2018), pp. 351–373 (cit. on pp. 37, 50).
DOI: 10.1016/j.cpc.2017.09.024.

[57] R. A. Fonseca, J. Vieira, F. Fiuza, A. Davidson, F. S. Tsung, W. B. Mori, and L. O. Silva. "Exploiting multi-scale parallelism for large scale numerical modelling of laser wakefield accelerators". In: *Plasma Physics and Controlled Fusion* 55.12 (2013), p. 124011 (cit. on pp. 47, 49, 56).
DOI: 10.1088/0741-3335/55/12/124011.

[58] R. A. Fonseca, L. O. Silva, F. S. Tsung, V. K. Decyk, W. Lu, C. Ren, W. B. Mori, S. Deng, S. Lee, T. Katsouleas, and J. C. Adam. "OSIRIS: A Three-Dimensional, Fully Relativistic Particle in Cell Code for Modeling Plasma Based Accelerators". In: *Computational Science — International Conference on Computational Science (ICCS)*. Vol. 2331. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2002, pp. 342–351 (cit. on p. 46).
DOI: 10.1007/3-540-47789-6_36.

[59] K. Germaschewski, W. Fox, S. Abbott, N. Ahmadi, K. Maynard, L. Wang, H. Ruhl, and A. Bhattacharjee. "The Plasma Simulation Code: A modern particle-in-cell code with patch-based load-balancing". In: *Journal of Computational Physics* 318 (2016), pp. 305–326 (cit. on pp. 47, 49, 55, 65, 76, 100, 105, 124).
DOI: 10.1016/j.jcp.2016.05.013.

[60] C. Huang, V. K. Decyk, C. Ren, M. Zhou, W. Lu, W. B. Mori, J. H. Cooley, T. M. Antonsen, and T. Katsouleas. "QUICKPIC: A highly efficient particle-in-cell code for modeling wakefield acceleration in plasmas". In: *Journal of Computational Physics* 217.2 (2006), pp. 658–679 (cit. on p. 49).
DOI: 10.1016/j.jcp.2006.01.039.

[61] A. Jocksch, F. Hariri, T.-M. Tran, S. Brunner, C. Gheller, and L. Villard. "A Bucket Sort Algorithm for the Particle-In-Cell Method on Manycore Architectures". In: *Parallel Processing and Applied Mathematics: 11th International Conference (PPAM)*. Vol. 9573. Lecture Notes in Computer Science. Springer, Cham, 2016, pp. 43–52 (cit. on pp. 56, 65, 94, 95, 100, 101, 110, 121).
DOI: 10.1007/978-3-319-32149-3_5.

[62] A. Jocksch, N. Ohana, E. Lanti, A. Scheinberg, S. Brunner, C. Gheller, and L. Villard. "Prediction of the Inter-Node Communication Costs of a New Gyrokinetic Code with Toroidal Domain". In: *Parallel Processing and Applied Mathematics: 12th International Conference (PPAM)*. Vol. 10777. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 370–380 (cit. on p. 49).
DOI: 10.1007/978-3-319-78024-5_33.

[63] G. Jost, T. M. Tran, K. Appert, W. A. Cooper, and L. Villard. "Development of a Global Linear Gyrokinetic PIC Code in 3D magnetic configurations". In: *Proceedings of the Joint Varenna-Lausanne International Workshop on Theory of Fusion Plasmas*. Società Italiana di Fisica (SIF), Bologna, 1999, pp. 419–425 (cit. on p. 49).
URL: https://infoscience.epfl.ch/record/121151/files/lrp_617_98_hq.pdf.

[64] C. C. Kim and S. E. Parker. "Massively Parallel Three-Dimensional Toroidal Gyrokinetic Flux-Tube Turbulence Simulation". In: *Journal of Computational Physics* 161.2 (2000), pp. 589–604 (cit. on p. 48).
DOI: 10.1006/jcph.2000.6518.

[65] X. Kong, M. C. Huang, C. Ren, and V. K. Decyk. "Particle-in-cell simulations with charge-conserving current deposition on graphic processing units". In: *Journal of Computational Physics* 230.4 (2011), pp. 1676–1685 (cit. on pp. 55, 102, 121).
DOI: 10.1016/j.jcp.2010.11.032.

[66] L. Lancia, A. Giribono, L. Vassura, M. Chiaramello, C. Riconda, S. Weber, A. Castan, A. Chatelain, A. Frank, T. Gangolf, M. N. Quinn, J. Fuchs, and J.-R. Marquès. "Signatures of the Self-Similar Regime of Strongly Coupled Stimulated Brillouin Scattering for Efficient Short Laser Pulse Amplification". In: *Physical Review Letters* 116 (7 2016), p. 075001 (cit. on p. 50).
DOI: 10.1103/PhysRevLett.116.075001.

[67] G. Lapenta and J. U. Brackbill. "Control of the number of particles in fluid and MHD particle in cell methods". In: *Computer Physics Communications* 87.1 (1995), pp. 139–154 (cit. on p. 32).
DOI: 10.1016/0010-4655(94)00180-A.

[68] A. Larin, S. Bastrakov, A. Bashinov, E. Efimenko, I. Surmin, A. Gonoskov, and I. Meyerov. "Load Balancing for Particle-in-Cell Plasma Simulation on Multicore Systems". In: *Parallel Processing and Applied Mathematics: 12th International Conference (PPAM)*. Vol. 10777. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 145–155 (cit. on p. 186).
DOI: 10.1007/978-3-319-78024-5_14.

[69] Z. Lin, T. S. Hahm, W. W. Lee, W. M. Tang, and R. B. White. "Turbulent Transport Reduction by Zonal Flows: Massively Parallel Simulations". In: *Science* 281.5384 (1998), pp. 1835–1837 (cit. on p. 49).
DOI: 10.1126/science.281.5384.1835.

[70] G. Marin, G. Jin, and J. Mellor-Crummey. "Managing locality in grand challenge applications: a case study of the gyrokinetic toroidal code". In: *Journal of Physics: Conference Series* 125 (2008), p. 012087 (cit. on pp. 61, 100).
DOI: 10.1088/1742-6596/125/1/012087.

[71] Y. Miyake and H. Usui. "New electromagnetic particle simulation code for the analysis of spacecraft-plasma interactions". In: *Physics of Plasmas* 16.6 (2009), p. 062904 (cit. on p. 49).
DOI: 10.1063/1.3147922.

[72] H. Nakashima, Y. Summura, K. Kikura, and Y. Miyake. "Large Scale Manycore-Aware PIC Simulation with Efficient Particle Binning". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2017, pp. 202–212 (cit. on pp. 35, 49, 56, 102, 110, 117, 121, 230).
DOI: 10.1109/IPDPS.2017.65.

[73] H. Nakashima, Y. Miyake, H. Usui, and Y. Omura. "OhHelp: A Scalable Domain-decomposing Dynamic Load Balancing for Particle-in-cell Simulations". In: *Proceedings of the 23rd International Conference on Supercomputing (ICS)*. ACM, 2009, pp. 90–99 (cit. on p. 47).
DOI: 10.1145/1542275.1542293.

[74] R. K. Narayanan and K. Madduri. "Parallel Particle-in-Cell Performance Optimization: A Case Study of Electrospray Simulation". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, 2017, pp. 1158–1167 (cit. on p. 46).
DOI: 10.1109/IPDPSW.2017.160.

[75] C. Nieter and J. R. Cary. "VORPAL: a versatile plasma simulation code". In: *Journal of Computational Physics* 196.2 (2004), pp. 448–473 (cit. on p. 50).
DOI: 10.1016/j.jcp.2003.11.004.

[76] S. E. Parker and W. W. Lee. "A fully nonlinear characteristic method for gyrokinetic simulation". In: *Physics of Fluids B: Plasma Physics* 5.1 (1993), pp. 77–86 (cit. on p. 32).
DOI: 10.1063/1.860870.

[77] P. A. Raviart. "An analysis of particle methods". In: *Numerical Methods in Fluid Dynamics*. Vol. 1127. Lecture Notes in Mathematics. Springer Berlin Heidelberg, 1985, pp. 243–324 (cit. on p. 32).
DOI: 10.1007/BFb0074532.

[78] L. F. Ricketson and A. J. Cerfon. "Sparse grid techniques for particle-in-cell schemes". In: *Plasma Physics and Controlled Fusion* 59.2 (2017), p. 024002 (cit. on p. 151).
DOI: 10.1088/1361-6587/59/2/024002.

[79] 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". In: *Computer Physics Communications* 182.9 (2011), pp. 2047–2051 (cit. on p. 48).
DOI: 10.1016/j.cpc.2010.12.038.

[80] M. Shalaby, A. E. Broderick, P. Chang, C. Pfrommer, A. Lamberts, and E. Puchwein. "SHARP: A Spatially Higher-order, Relativistic Particle-in-cell Code". In: *The Astrophysical Journal* 841.1 (2017), p. 52 (cit. on pp. 47, 49, 150, 155).
DOI: 10.3847/1538-4357/aa6d13.

[81] I. Surmin, S. Bastrakov, Z. Matveev, E. Efimenko, A. Gonoskov, and I. Meyerov. "Co-design of a Particle-in-Cell Plasma Simulation Code for Intel Xeon Phi: A First Look at Knights Landing". In: *Proceedings of the 16th International Conference on Algorithms and Architectures for Parallel Processing Collocated Workshops (ICA3PP, SCDT)*. Vol. 10049. Lecture Notes in Computer Science. Springer, Cham, 2016, pp. 319–329 (cit. on pp. 49, 56, 65, 101, 184).
DOI: 10.1007/978-3-319-49956-7_25.

[82] I. Surmin, A. Bashinov, S. Bastrakov, E. Efimenko, A. Gonoskov, and I. Meyerov. "Dynamic Load Balancing Based on Rectilinear Partitioning in Particle-in-Cell Plasma Simulation". In: *Parallel Computing Technologies: 13th International Conference (PaCT)*. Vol. 9251. Lecture Notes in Computer Science. Springer, Cham, 2015, pp. 107–119 (cit. on p. 47).
DOI: 10.1007/978-3-319-21909-7_12.

[83] W. Tang, B. Wang, S. Ethier, G. Kwasniewski, T. Hoefler, K. Z. Ibrahim, K. Madduri, S. Williams, L. Oliker, C. Rosales-Fernandez, and T. Williams. "Extreme Scale Plasma Turbulence Simulations on Top Supercomputers Worldwide". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Press, 2016, pp. 502–513 (cit. on pp. 27, 37, 49, 51, 56, 100, 101, 222).
DOI: 10.1109/SC.2016.42.

[84] T. M. Tran, K. Appert, M. Fivaz, G. Jost, J. Vaclavik, and L. Villard. "Global gyrokinetic simulation of ion-temperature-gradient-driven instabilities using particles". In: *Proceedings of the Joint Varenna-Lausanne International Workshop on Theory of Fusion Plasmas*. Società Italiana di Fisica (SIF), Bologna, 1999, pp. 45–58 (cit. on p. 49).
URL: https://infoscience.epfl.ch/record/121151/files/lrp_617_98_hq.pdf.

[85] D. Tskhakaya and R. Schneider. "Optimization of PIC codes by improved memory management". In: *Journal of Computational Physics* 225.1 (2007), pp. 829–839 (cit. on pp. 101, 105, 110, 121).
DOI: 10.1016/j.jcp.2007.01.002.

[86] J. P. Verboncoeur, A. B. Langdon, and N. T. Gladd. "An object-oriented electromagnetic PIC code". In: *Computer Physics Communications* 87.1 (1995), pp. 199–211 (cit. on p. 50).
DOI: 10.1016/0010-4655(94)00173-Y.

[87] H. Vincenti, M. Lobet, R. Lehe, R. Sasanka, and J.-L. Vay. "An efficient and portable SIMD algorithm for charge/current deposition in Particle-In-Cell codes". In: *Computer Physics Communications* 210 (2016), pp. 145–154 (cit. on pp. 36, 37, 49, 65, 101, 123).
DOI: 10.1016/j.cpc.2016.08.023.

[88] E. Wang, S. Wu, Q. Zhang, J. Liu, W. Zhang, Z. Lin, Y. Lu, Y. Du, and X. Zhu. "The Gyrokinetic Particle Simulation of Fusion Plasmas on Tianhe-2 Supercomputer". In: *Proceedings of the 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. IEEE Press, 2016, pp. 25–32 (cit. on p. 184).
DOI: 10.1109/ScalA.2016.8.

[89] T. Weinzierl, B. Verleye, P. Henri, and D. Roose. "Two particle-in-grid realisations on spacetrees". In: *Parallel Computing* 52 (2016), pp. 42–64 (cit. on p. 47).
DOI: 10.1016/j.parco.2015.12.007.

[90] E. Zenker, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, and M. Bussmann. "Performance-Portable Many-Core Plasma Simulations: Porting PIConGPU to OpenPower and Beyond". In: *International Conference on High Performance Computing (ISC)*. Vol. 9945. Lecture Notes in Computer Science. Springer, Cham, 2016, pp. 293–301 (cit. on pp. 49, 56).
DOI: 10.1007/978-3-319-46079-6_21.

Articles - Semi-Lagrangian

[91] M. L. Bégué, A. Ghizzo, and P. Bertrand. "Two-Dimensional Vlasov Simulation of Raman Scattering and Plasma Beatwave Acceleration on Parallel Computers". In: *Journal of Computational Physics* 151.2 (1999), pp. 458–478 (cit. on p. 136).
DOI: 10.1006/jcph.1999.6193.

[92] N. Besse, J. Segré, and E. Sonnendrücker. "Semi-Lagrangian Schemes for the Two-Dimensional Vlasov–Poisson System on Unstructured Meshes". In: *Transport Theory and Statistical Physics* 34.3-5 (2005), pp. 311–332 (cit. on p. 155).
DOI: 10.1080/00411450500274592.

[93] F. Casas, N. Crouseilles, E. Faou, and M. Mehrenberger. "High-order Hamiltonian splitting for the Vlasov–Poisson equations". In: *Numerische Mathematik* 135.3 (2017), pp. 769–801 (cit. on pp. 133, 135, 157, 169).
DOI: 10.1007/s00211-016-0816-z.

[94] C. Z. Cheng and G. Knorr. "The Integration of the Vlasov Equation in Configuration Space". In: *Journal of Computational Physics* 22.3 (1976), pp. 330–351 (cit. on pp. 22, 134, 135, 157, 216).
DOI: 10.1016/0021-9991(76)90053-X.

[95] O. Coulaud, E. Sonnendrücker, E. Dillon, P. Bertrand, and A. Ghizzo. "Parallelization of semi-Lagrangian Vlasov codes". In: *Journal of Plasma Physics* 61.3 (1999), pp. 435–448 (cit. on p. 136).
URL: https://www.cambridge.org/core/journals/journal-of-plasma-physics/article/parallelization-of-semilagrangian-vlasov-codes/BDB63B842A067B1AADD87EF0DC93D218.

[96] N. Crouseilles, G. Latu, and E. Sonnendrücker. "A parallel Vlasov solver based on local cubic spline interpolation on patches". In: *Journal of Computational Physics* 228.5 (2009), pp. 1429–1446 (cit. on p. 138).
DOI: 10.1016/j.jcp.2008.10.041.

[97] L. Einkemmer. "High performance computing aspects of a dimension independent semi-Lagrangian discontinuous Galerkin code". In: *Computer Physics Communications* 202 (2016), pp. 326–336 (cit. on p. 138).
DOI: 10.1016/j.cpc.2016.01.012.

[98] V. Grandgirard, J. Abiteboul, J. Bigot, T. Cartier-Michaud, N. Crouseilles, G. Dif-Pradalier, Ch. Ehrlacher, D. Esteve, X. Garbet, Ph. Ghendrih, G. Latu, M. Mehrenberger, C. Norscini, Ch. Passeron, F. Rozar, Y. Sarazin, E. Sonnendrücker, A. Strugarek, and D. Zarzoso. "A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations". In: *Computer Physics Communications* 207 (2016), pp. 35–68 (cit. on p. 138).
DOI: 10.1016/j.cpc.2016.05.007.

[99] K. Kormann. "A semi-Lagrangian Vlasov solver in tensor train format". In: *SIAM Journal on Scientific Computing* 37.4 (2015), B613–B632 (cit. on p. 155).
DOI: 10.1137/140971270.

[100] K. Kormann, K. Reuter, and M. Rampp. "A massively parallel semi-Lagrangian solver for the six-dimensional Vlasov–Poisson equation". In: Submitted to The international journal of high performance computing applications. 2018 (cit. on p. 138).

[101] M. Mehrenberger, C. Steiner, L. Marradi, N. Crouseilles, E. Sonnendrücker, and B. Afeyan. "Vlasov on GPU (VOG project)". In: *ESAIM: Proceedings and Surveys* 43 (2013), pp. 37–58 (cit. on pp. 157, 159).
DOI: 10.1051/proc/201343003.

[102] M. Sarrat, A. Ghizzo, D. Del Sarto, and L. Serrat. "Parallel implementation of a relativistic semi-Lagrangian Vlasov–Maxwell solver". In: *The European Physical Journal D* 71.11 (2017), p. 271 (cit. on p. 138).
DOI: 10.1140/epjd/e2017-80188-4.

[103] E. Sonnendrücker, J. Roche, P. Bertrand, and A. Ghizzo. *The Semi-Lagrangian Method for the Numerical Resolution of Vlasov Equations*. Tech. rep. RR-3393. INRIA, 1998 (cit. on p. 136).
URL: https://hal.inria.fr/inria-00073296.

Articles - (Pseudo-, Quasi-) Random Numbers

[104] G. E. P. Box and M. E. Muller. "A Note on the Generation of Random Normal Deviates". In: *The Annals of Mathematical Statistics* 29.2 (1958), pp. 610–611 (cit. on p. 44).
DOI: 10.1214/aoms/1177706645.

[105] J. G. van der Corput. "Verteilungsfunktionen. (Erste Mitteilung, Zweite Mitteilung)". In: *Proceedings of the Koninklijke Akademie van Wetenschappen te Amsterdam* 38.8, 10 (1935), pp. 813–821, 1058–1066 (cit. on p. 45).
URL: http://www.dwc.knaw.nl/DL/publications/PU00014607.pdf.

[106] D. T. Jones. *Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications*. Tech. rep. Department of Computer Science, University College London, 2010 (cit. on pp. 38, 41).
URL: http://www0.cs.ucl.ac.uk/staff/D.Jones/GoodPracticeRNG.pdf.

[107] S. T. Lavavej. "rand() Considered Harmful". In: Going Native. 2013 (cit. on p. 38).
URL: https://channel9.msdn.com/Events/GoingNative/2013/rand-Considered-Harmful.

[108] P. L'Ecuyer. "Efficient and Portable Combined Random Number Generators". In: *Communications of the ACM* 31.6 (1988), pp. 742–751 (cit. on p. 38).
DOI: 10.1145/62959.62969.

[109] P. L'Ecuyer. "Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators". In: *Operations Research* 47.1 (1999), pp. 159–164 (cit. on p. 37).
DOI: 10.1287/opre.47.1.159.

[110] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. "An Object-Oriented Random-Number Package with Many Long Streams and Substreams". In: *Operations Research* 50.6 (2002), pp. 1073–1075 (cit. on p. 37).
DOI: 10.1287/opre.50.6.1073.358.

[111] G. Marsaglia. "Random numbers for C: The END?" In: The electronic billboard sci.crypt.random-numbers. 1999 (cit. on pp. 37, 38).
URL: http://www.ciphersbyritter.com/NEWS4/RANDC.HTM#36A5FC62.17C9CC33@stat.fsu.edu.

[112] G. Marsaglia. "Xorshift RNGs". In: *Journal of Statistical Software, Articles* 8.14 (2003), pp. 1–6 (cit. on p. 37).
DOI: 10.18637/jss.v008.i14.

[113] G. Marsaglia and A. Zaman. "Monkey tests for random number generators". In: *Computers & Mathematics with Applications* 26.9 (1993), pp. 1–10. ISSN: 0898-1221 (cit. on pp. 37, 38).
DOI: 10.1016/0898-1221(93)90001-C.

[114] M. Matsumoto and T. Nishimura. "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pp. 3–30 (cit. on pp. 37, 38).
DOI: 10.1145/272991.272995.

[115] F. Panneton, P. L'Ecuyer, and M. Matsumoto. "Improved Long-period Generators Based on Linear Recurrences Modulo 2". In: *ACM Transactions on Mathematical Software (TOMS)* 32.1 (2006). (Source Code: http://www.iro.umontreal.ca/~panneton/WELLRNG.html), pp. 1–16 (cit. on p. 38).
DOI: 10.1145/1132973.1132974.

[116] S. K. Park and K. W. Miller. "Random Number Generators: Good Ones Are Hard to Find". In: *Communications of the ACM* 31.10 (1988), pp. 1192–1201 (cit. on p. 37).
DOI: 10.1145/63039.63042.

[117] B. D. Ripley. "Thoughts on pseudorandom number generators". In: *Journal of Computational and Applied Mathematics* 31.1 (1990), pp. 153–163 (cit. on pp. 37, 38).
DOI: 10.1016/0377-0427(90)90346-2.

[118] G. G. Rose. "KISS: A bit too simple". In: *Cryptography and Communications* 10 (2018), pp. 123–137 (cit. on pp. 37, 38).
DOI: 10.1007/s12095-017-0225-x.

[119] I. M. Sobol'. "On the distribution of points in a cube and the approximate evaluation of integrals". In: *USSR Computational Mathematics and Mathematical Physics* 7.4 (1967). (Also in Russian: "О Распределении Точек В Кубе и Приближенном Вычислении Интегралов", Журнал Вычислительной Математики и Математической Физики **7**, 4), pp. 86–112 (cit. on p. 45).
DOI: 10.1016/0041-5553(67)90144-9.

[120] M. J. Wichura. "Algorithm AS 241: The Percentage Points of the Normal Distribution". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 37.3 (1988). (Source Code: https://people.sc.fsu.edu/%7Ejburkardt/f_src/asa241/asa241.html), pp. 477–484 (cit. on p. 43).
DOI: 10.2307/2347330.

Articles - Space-Filling Curves

[121] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. "Nonlinear Array Layouts for Hierarchical Memory Systems". In: *Proceedings of the 13th International Conference on Supercomputing (ICS)*. ACM, 1999, pp. 444–453 (cit. on pp. 65, 67).
DOI: 10.1145/305138.305231.

[122] D. DeFord and A. Kalyanaraman. "Empirical Analysis of Space-Filling Curves for Scientific Computing Applications". In: *42nd International Conference on Parallel Processing (ICPP)*. IEEE Computer Society, 2013, pp. 170–179 (cit. on p. 65).
DOI: 10.1109/ICPP.2013.26.

[123] D. Hilbert. "Über die stetige abbildung einer linie auf ein flächenstück". In: *Mathematische Annalen* 38 (1891), pp. 459–460 (cit. on p. 67).
URL: https://eudml.org/doc/157555.

[124] J. Mellor-Crummey, D. Whalley, and K. Kennedy. "Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings". In: *International Journal of Parallel Programming* 29.3 (2001), pp. 217–247 (cit. on p. 65).
DOI: 10.1023/A:1011119519789.

[125] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. Tech. rep. IBM Ltd, 1966 (cit. on p. 67).
URL: https://domino.research.ibm.com/library/cyberdig.nsf/0/0dabf9473b9c86d48525779800566a39?OpenDocument.

[126] R. Raman and D. S. Wise. "Converting to and from Dilated Integers". In: *IEEE Transactions on Computers* 57.4 (2008), pp. 567–573 (cit. on pp. 67, 68).
DOI: 10.1109/TC.2007.70814.

[127] J. Skilling. "Programming the Hilbert curve". In: *AIP Conference Proceedings*. Vol. 707. 2004, pp. 381–387 (cit. on p. 67).
DOI: 10.1063/1.1751381.

Articles - Other

[128] U. A. Acar, A. Charguéraud, and M. Rainey. "Theory and Practice of Chunked Sequences". In: *European Symposium on Algorithms (ESA)*. Vol. 8737. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 25–36 (cit. on p. 104).
DOI: 10.1007/978-3-662-44777-2_3.

[129] Nuclear Energy Agency. *Preservation of Records, Knowledge and Memory across Generations (RK&M): Markers – Reflections on Intergenerational Warnings in the Form of Japanese Tsunami Stones*. Tech. rep. Organisation for Economic Cooperation and Development (OECD), 2014 (cit. on pp. 17, 210).
URL: https://www.oecd-nea.org/rwm/docs/2014/rwm-r2014-4.pdf.

[130] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the American Federation of Information Processing Societies Spring Joint Computer Conference (AFIPS)*. ACM, 1967, pp. 483–485 (cit. on pp. 24, 219).
DOI: 10.1145/1465482.1465560.

[131] J. D. Annan and J. C. Hargreaves. "A new global reconstruction of temperature changes at the Last Glacial Maximum". In: *Climate of the Past* 9.1 (2013), pp. 367–376 (cit. on pp. 16, 211).
DOI: 10.5194/cp-9-367-2013.

[132] M. Badsi and M. Herda. "Modelling and simulating a multispecies plasma". In: *ESAIM: Proceedings and Surveys* 53 (2016), pp. 22–37 (cit. on pp. 155, 157, 159, 160).
DOI: 10.1051/proc/201653002.

[133] J. W. Banks, R. L. Berger, S. Brunner, B. I. Cohen, and J. A. F. Hittinger. "Two-dimensional Vlasov simulation of electron plasma wave trapping, wavefront bowing, self-focusing, and sideloss". In: *Physics of Plasmas* 18.5 (2011), p. 052102 (cit. on p. 155).
DOI: 10.1063/1.3577784.

[134] J. Barnes and P. Hut. "A hierarchical $O(NlogN)$ force-calculation algorithm". In: *Nature* 324.3 (1986), pp. 446–449 (cit. on p. 47).
DOI: 10.1038/324446a0.

[135] M. A. Bender, E. D. Demaine, and M. Farach-Colton. "Cache-Oblivious B-Trees". In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 2000, pp. 399–409 (cit. on p. 102).
DOI: 10.1109/SFCS.2000.892128.

[136] R. L. Berger, S. Brunner, J. W. Banks, B. I. Cohen, and B. J. Winjum. "Multi-dimensional Vlasov simulations and modeling of trapped-electron-driven filamentation of electron plasma waves". In: *Physics of Plasmas* 22.5 (2015), p. 055703 (cit. on p. 155).
DOI: 10.1063/1.4917482.

[137] S. Blanes and P. C. Moan. "Practical symplectic partitioned Runge–Kutta and Runge–Kutta–Nyström methods". In: *Journal of Computational and Applied Mathematics* 142.2 (2002), pp. 313–330 (cit. on pp. 135, 157).
DOI: 10.1016/S0377-0427(01)00492-7.

[138] G. E. Blelloch. *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, 1990 (cit. on p. 79).
URL: http://www.cs.cmu.edu/~scandal/papers/CMU-CS-90-190.html.

[139] E. Buks, R. Schuster, M. Heiblum, D. Mahalu, and V. Umansky. "Dephasing in electron interference by a 'which-path' detector". In: *Nature* 391 (1998), pp. 871–874 (cit. on pp. 27, 222).
DOI: 10.1038/36057.

[140] J. Canosa. "Numerical solution of Landau's dispersion equation". In: *Journal of Computational Physics* 13.1 (1973), pp. 158–160 (cit. on p. 150).
DOI: 10.1016/0021-9991(73)90131-9.

[141] N. Crouseilles, M. Gutnic, G. Latu, and E. Sonnendrücker. "Comparison of two Eulerian solvers for the four-dimensional Vlasov equation: Part II". In: *Communications in Nonlinear Science and Numerical Simulation* 13.01 (2008), pp. 94–99 (cit. on p. 155).
DOI: 10.1016/j.cnsns.2007.03.017.

[142] J.-B. Delambre. "De l'usage du calcul différentiel dans la construction des tables astronomiques". In: *Mémoires de l'Académie royale des Sciences, années MDCCIX–XCI* (1793), pp. 143–180 (cit. on pp. 21, 215, 216).
URL: https://books.google.be/books?id=sHMIizih0WQC.

[143] J. Denavit. "First and Second Order Landau Damping in Maxwellian Plasmas". In: *The Physics of Fluids* 8.3 (1965), pp. 471–478 (cit. on p. 155).
DOI: 10.1063/1.1761247.

[144] M. Durand, B. Raffin, and F. Faure. "A Packed Memory Array to Keep Moving Particles Sorted". In: *Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS)*. The Eurographics Association, 2012 (cit. on pp. 35, 102, 110, 121).
DOI: 10.2312/PE/vriphys/vriphys12/069-077.

[145] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer. *A Fortran to C Converter*. Tech. rep. 149. AT&T Bell Laboratories, 1990 (cit. on p. 60).
URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.6209&rep=rep1&type=pdf.

[146] F. Filbet and E. Sonnendrücker. "Comparison of Eulerian Vlasov solvers". In: *Computer Physics Communications* 150.3 (2003), pp. 247–266 (cit. on p. 155).
DOI: 10.1016/S0010-4655(02)00694-X.

[147] F. Filbet and E. Sonnendrücker. "Modeling and numerical simulation of space charge dominated beams in the paraxial approximation". In: *Mathematical Models and Methods in Applied Sciences* 16.05 (2006), pp. 763–791 (cit. on p. 155).
DOI: 10.1142/S0218202506001340.

[148] F. Filbet, E. Sonnendrücker, and P. Bertrand. "Conservative Numerical Schemes for the Vlasov Equation". In: *Journal of Computational Physics* 172.1 (2001), pp. 166–187 (cit. on pp. 151, 155).
DOI: 10.1006/jcph.2001.6818.

[149] M. Frigo and S. G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231 (cit. on pp. 46, 57, 61, 113, 124).
DOI: 10.1109/JPROC.2004.840301.
URL: http://www.fftw.org.

[150] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. "Cache-Oblivious Algorithms". In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 1999, pp. 285–299 (cit. on p. 69).
DOI: 10.1109/SFFCS.1999.814600.

[151] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Proceedings of the 11th European Parallel Virtual Machine / Message Passing Interface Users' Group Meeting (EuroPVM/MPI)*. Vol. 3241. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2004, pp. 97–104 (cit. on p. 56).
DOI: 10.1007/978-3-540-30218-6_19.

[152] D. R. Hanson. "Fast Allocation and Deallocation of Memory Based on Object Lifetimes". In: *Software - Practice & Experience* 20.1 (1990), pp. 5–12 (cit. on pp. 106, 107).
DOI: 10.1002/spe.4380200104.

[153] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. "MAO - an Extensible Micro-Architectural Optimizer". In: *Proceedings of the 8th International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2011, pp. 1–10 (cit. on p. 62).
DOI: 10.1109/CGO.2011.5764669.

[154] V. F. Ialenti. "Adjudicating Deep Time: Revisiting the United States' High-Level Nuclear Waste Repository Project at Yucca Mountain". In: *Science & Technology Studies* 27.2 (2014), pp. 27–48 (cit. on pp. 17, 210).
URL: https://ssrn.com/abstract=2457896.

[155] W. Kahan. "Pracniques: Further Remarks on Reducing Truncation Errors". In: *Communications of the ACM* 8.1 (1965), p. 40 (cit. on p. 35).
DOI: 10.1145/363707.363723.

[156] P. Kravanja, M. Van Barel, O. Ragos, M. N. Vrahatis, and F. A. Zafiropoulos. "ZEAL: A mathematical software package for computing zeros of analytic functions". In: *Computer Physics Communications* 124.2 (2000). Code repository: http://cpc.cs.qub.ac.uk/summaries/ADKW, pp. 212–232 (cit. on p. 150).
DOI: 10.1016/S0010-4655(99)00429-4.

[157] L. Landau. "On the vibrations of the electronic plasma". In: *Journal of Physics (USSR)* 10 (1946). (Translated from Russian: Л. Ландау, "О колебаниях электронной плазмы", Журнал Экспериментальной и Теоретической Физики **16**, pp. 574–586 (1946); reproduced in "Collected Papers of L.D. Landau", https://doi.org/10.1016/C2013-0-01806-3, pp. 445-460 (1965)), pp. 25–34 (cit. on pp. 22, 149, 150, 155, 217).

[158] F. Le Gall. "Powers of Tensors and Fast Matrix Multiplication". In: *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC)*. ACM, 2014, pp. 296–303 (cit. on pp. 28, 224).
DOI: 10.1145/2608628.2608664.

[159] M. Lesur. "Method and scheme-independent entropy production in turbulent kinetic simulations". In: *Computer Physics Communications* 200 (2016), pp. 182–189 (cit. on p. 155).
DOI: 10.1016/j.cpc.2015.12.001.

[160] E. Madaule, M. Restelli, and E. Sonnendrücker. "Energy conserving discontinuous Galerkin spectral element method for the Vlasov–Poisson system". In: *Journal of Computational Physics* 279 (2014), pp. 261–288 (cit. on pp. 155, 157, 169).
DOI: 10.1016/j.jcp.2014.09.010.

[161] S. A. Marcott, J. D. Shakun, P. U. Clark, and A. C. Mix. "A Reconstruction of Regional and Global Temperature for the Past 11,300 Years". In: *Science* 339.6124 (2013), pp. 1198–1201 (cit. on pp. 16, 211).
DOI: 10.1126/science.1228026.

[162] J. D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture Newsletter (TCCA)* (1995), pp. 19–25 (cit. on pp. 28, 85, 86, 95, 96, 115, 116, 125, 184, 224).
URL: https://www.cs.virginia.edu/stream/.

[163] C. J. McKinstrie, R. E. Giacone, and E. A. Startsev. "Accurate formulas for the Landau damping rates of electrostatic waves". In: *Physics of Plasmas* 6.2 (1999), pp. 463–466 (cit. on p. 150).
DOI: 10.1063/1.873212.

[164] H. M. Mott-Smith. "History of "Plasmas"". In: *Nature* 233 (1971), p. 219 (cit. on pp. 18, 212).
DOI: 10.1038/233219a0.

[165] L. Muschietti, I. Roth, C. W. Carlson, and R. E. Ergun. "Transverse Instability of Magnetized Electron Holes". In: *Physical Review Letters* 85.1 (2000), pp. 94–97 (cit. on pp. 51, 153).
DOI: 10.1103/PhysRevLett.85.94.

[166] T. Nakamura and T. Yabe. "Cubic interpolated propagation scheme for solving the hyper-dimensional Vlasov–Poisson equation in phase space". In: *Computer Physics Communications* 120.2 (1999), pp. 122–154 (cit. on p. 155).
DOI: 10.1016/S0010-4655(99)00247-7.

[167] D. M. Nicol. "Rectilinear Partitioning of Irregular Data Parallel Computations". In: *Journal of Parallel and Distributed Computing* 23.2 (1994), pp. 119–134 (cit. on p. 47).
DOI: 10.1006/jpdc.1994.1126.

[168] J. D. Shakun, P. U. Clark, F. He, S. A. Marcott, A. C. Mix, Z. Liu, B. Otto-Bliesner, A. Schmittner, and E. Bard. "Global warming preceded by increasing carbon dioxide concentrations during the last deglaciation". In: *Nature* 484 (2012), pp. 49–54 (cit. on pp. 16, 211).
DOI: 10.1038/nature10915.

[169] M. M. Shoucri and R. R. T. Gagné. "A multistep technique for the numerical solution of a two-dimensional Vlasov equation". In: *Journal of Computational Physics* 23.3 (1977), pp. 242–262 (cit. on p. 155).
DOI: 10.1016/0021-9991(77)90093-6.

[170] D. A. Silantyev, P. M. Lushnikov, and H. A. Rose. "Langmuir wave filamentation in the kinetic regime. II. Weak and strong pumping of nonlinear electron plasma waves as the route to filamentation". In: *Physics of Plasmas* 24.4 (2017), p. 042105 (cit. on p. 155).
DOI: 10.1063/1.4979290.

[171] N. Smith and S. van der Walt. "A Better Default Colormap for Matplotlib". In: SciPy 2015. Code repository: https://bids.github.io/colormap/. 2015 (cit. on p. 207).
URL: https://www.youtube.com/watch?v=xAoljeRJ3lU.

[172] C. Störmer. "Sur les trajectoires des corpuscules électrisés dans l'espace sous l'action du magnétisme terrestre avec application aux aurores boréales". In: *Archives des sciences physiques et naturelles* 24 (1907), pp. 5–18, 113–158, 221–247, 317–364 (cit. on pp. 21, 216).
URL: http://gallica.bnf.fr/ark:/12148/bpt6k2991282/f222.

[173] G. Strang. "On the Construction and Comparison of Difference Schemes". In: *SIAM Journal on Numerical Analysis* 5.3 (1968), pp. 506–517 (cit. on p. 134).
DOI: 10.1137/0705041.

[174] V. Strassen. "Gaussian elimination is not optimal". In: *Numerische Mathematik* 13.4 (1969), pp. 354–356 (cit. on pp. 28, 224).
DOI: 10.1007/BF02165411.

[175] T. Umeda, S. Ueno, and T. K. M. Nakamura. "Ion kinetic effects on nonlinear processes of the Kelvin–Helmholtz instability". In: *Plasma Physics and Controlled Fusion* 56.7 (2014), p. 075006 (cit. on p. 155).
DOI: 10.1088/0741-3335/56/7/075006.

[176] L. Verlet. "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard–Jones Molecules". In: *Physical Review* 159 (1 1967), pp. 98–103 (cit. on pp. 21, 216).
DOI: 10.1103/PhysRev.159.98.

[177] B. Videau, K. Pouget, L. Genovese, T. Deutsch, D. Komatitsch, F. Desprez, and J.-F. Méhaut. "BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications". In: *The International Journal of High Performance Computing Applications* 32.1 (2018), pp. 28–44 (cit. on p. 61).
DOI: 10.1177/1094342017718068.

[178] A. A. Vlasov. "The Vibrational Properties of an Electron Gas". In: *Soviet Physics Uspekhi* 10.6 (1968). (Translated from Russian: А. А. Власов, "О Вибрационных Свойствах Электронного Газа", Журнал Экспериментальной и Теоретической Физики **8** (3), 291 (1938)), pp. 721–733 (cit. on pp. 19, 149, 213).
DOI: 10.1070/PU1968v010n06ABEH003709.

[179] R. C. Whaley and A. Petitet. "Minimizing development and maintenance costs in supporting persistently optimized BLAS". In: *Software: Practice and Experience* 35.2 (2005), pp. 101–121 (cit. on p. 61).
DOI: 10.1002/spe.v35:2.

[180] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76 (cit. on pp. 28, 51, 125, 184, 224).
DOI: 10.1145/1498765.1498785.

[181] M. Winkel, R. Speck, H. Hübner, L. Arnold, R. Krause, and P. Gibbon. "A massively parallel, multi-disciplinary Barnes–Hut tree code for extreme-scale N-body simulations". In: *Computer Physics Communications* 183.4 (2012), pp. 880–889 (cit. on p. 47).
DOI: 10.1016/j.cpc.2011.12.013.

Tools

[182] OpenMP ARB (Architecture Review Boards). *Open Multi-Processing*. 1997 (cit. on pp. 26, 46, 56, 207, 221).
URL: http://www.openmp.org/.

[183] French Atomic Energy Commision (CEA Cadarache), INRIA TONUS Team (University of Strasbourg), and Max-Planck-Institut für Plasmaphysik (IPP Garching). *Semi-Lagrangian Library*. 2010 (cit. on pp. 59, 133).
URL: http://selalib.gforge.inria.fr.

[184] MPI Forum. *Message Passing Interface*. 1993 (cit. on pp. 47, 56, 207).
URL: https://www.mpi-forum.org/.

[185] The HDF Group. *Hierarchical Data Format 5*. 2007 (cit. on p. 57).
URL: https://www.hdfgroup.org/.

[186] Intel. *Cilk Plus*. 2010 (cit. on p. 207).
URL: https://www.cilkplus.org/.

[187] Lawrence Livermore National Laboratory. *Visualize It*. 2002 (cit. on p. 207).
URL: https://wci.llnl.gov/simulation/computer-codes/visit/.

[188] Maplesoft. *Maple*. 1982 (cit. on pp. 44, 150).
URL: https://www.maplesoft.com/products/maple/.

[189] D. Markauskas. *Code::Blocks IDE for Fortran*. 2015 (cit. on p. 60).
URL: http://cbfortran.sourceforge.net.

[190] Innovative Computing Laboratory at the University of Tennessee. *Performance Application Programming Interface*. 2000 (cit. on p. 57).
URL: http://icl.cs.utk.edu/papi.

[191] T. Williams and C. Kelley. *Gnuplot: An Interactive Plotting Program*. 1986 (cit. on p. 207).
URL: http://www.gnuplot.info/.

Miscellaneous

[192] B. Barré. *Pourquoi le nucléaire*. DeBoeck Supérieur. 2017 (cit. on pp. 17, 212).

[193] C. Baudelaire. *Les Fleurs du mal*. Poulet-Malassis et De Broise. 1861 (cit. on p. 105).
URL: https://fleursdumal.org/.

[194] L. Carroll. *Through the Looking-Glass*. Macmillan. 1872 (cit. on p. 50).
URL: http://www.gutenberg.org/ebooks/12.

[195] C. Chaplin. *Modern Times*. 1936 (cit. on pp. 24, 219).

[196] L. Faull, S. Sole, S. Brümmer, and S. Shipanga. *French nuclear frontrunner's toxic political dealings in SA*. Mail & Guardian, August 03, 2012 (cit. on pp. 17, 210).
URL: https://mg.co.za/article/2012-08-03-00-nuclear-frontrunner-arevas-toxic-political-dealings.

[197] Juvenal. *Satires*. (Translation by G. G. Ramsay: https://sourcebooks.fordham.edu/ancient/juvenal-satvi.asp). 90–127 (cit. on p. 61).
URL: http://www.thelatinlibrary.com/juvenal/6.shtml.

[198] Bureau international des poids et mesures. *Comptes rendus des séances de la 22e Conférence générale des poids et mesures*. (English unofficial title "Proceedings of the 22nd General Conference on Weights and Measures"). 2003 (cit. on p. 205).
URL: https://www.bipm.org/utils/common/pdf/CGPM/CGPM22.pdf.

[199] G. Pitron. *La guerre des métaux rares. La face cachée de la transition énergétique et numérique.* Les liens qui libèrent. 2018 (cit. on pp. 17, 210).

[200] J. de Salisbury. *Metalogicus*. Documenta Catholica Omnia. 1159 (cit. on pp. 9, 11).
URL: http://www.documentacatholicaomnia.eu/04z/z_1115-1180__Joannis_Saresberiensis__Metalogicus_%5BMetalogicum%5D__MLT.pdf.html.

[201] W. Shakespeare. *The Tragedy of Hamlet, Prince of Denmark*. 1599–1602 (cit. on p. 101).
URL: http://shakespearestudyguide.com/Hamlet%20Text.html.

Self-Citations

[202] Y. Barsamian, A. Charguéraud, and A. Ketterlin. "A Space and Bandwidth Efficient Multicore Algorithm for the Particle-in-Cell Method". In: *Parallel Processing and Applied Mathematics: 12th International Conference (PPAM)*. Vol. 10777. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 133–144 (cit. on pp. 53, 55, 99, 180, 182, 183).
DOI: 10.1007/978-3-319-78024-5_13.
Slides: http://www.barsamian.am/Slides/slides_2017-09-11.pdf

[203] Y. Barsamian, S. A. Hirstoaga, and É. Violard. "Efficient Data Layouts for a Three-Dimensional Electrostatic Particle-in-Cell Code". In: *Journal of Computational Science* 27 (2018), pp. 345–356 (cit. on pp. 53, 59, 121, 180–183).
DOI: 10.1016/j.jocs.2018.06.004.

[204] Y. Barsamian, S. A. Hirstoaga, and É. Violard. "Efficient Data Structures for a Hybrid Parallel and Vectorized Particle-in-Cell Code". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, 2017, pp. 1168–1177 (cit. on pp. 27, 28, 52, 55, 59, 70, 110, 180–183, 223).
DOI: 10.1109/IPDPSW.2017.74.
Slides: http://www.barsamian.am/Slides/slides_2017-06-02.pdf

[205] Y. Barsamian, A. Charguéraud, S. A. Hirstoaga, and M. Mehrenberger. "Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors". In: *24th International Conference on Parallel and Distributed Computing (Euro-Par)*. Vol. 11014. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 749–763 (cit. on pp. 53, 55, 56, 99, 125, 180–183).
DOI: 10.1007/978-3-319-78024-5_33.
Slides: http://www.barsamian.am/Slides/slides_2018-08-30.pdf

[206] Y. Barsamian, A. Charguéraud, S. A. Hirstoaga, and M. Mehrenberger. *Software artifacts for Euro-Par 2018 paper: "Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors"*. Figshare. 2018 (cit. on pp. 13, 53, 117, 125, 126).
URL: https://doi.org/10.6084/m9.figshare.6391796.

[207] Y. Barsamian, J. Bernier, S. A. Hirstoaga, and M. Mehrenberger. "Verification of $2D \times 2D$ and two-species Vlasov–Poisson solvers". In: *ESAIM: Proceedings and Surveys* 63 (2018), pp. 78–108 (cit. on pp. 22, 44, 51, 53, 146, 149, 155, 160, 165, 217).
DOI: 10.1051/proc/201863078.
Slides: http://www.barsamian.am/Slides/slides_2016-08-25.pdf

Appendix A

Notations

A.1 Some Useful Mathematical Notations

Differentiation. In this manuscript, we are considering two or three dimensions. When we need to derive a function with respect to a vector, and not a scalar, we do not use $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$, but we use the ∇ operator instead (pronounce "nabla"): $\nabla_{\vec{x}} f(\vec{x})$. It is the same concept, but in higher dimensions.

Numbers. Throughout this manuscript, we use the dot (.) for the decimal marker (to separate the integer part from the decimal part of numbers). In French, the comma (,) is commonly used, while it can be confusing because in English the comma is commonly used to separate groups of three digits to facilitate reading. To make this manuscript clear in both languages, following [198, Résolution 10], we do not use commas in numbers — neither between groups of three digits (we use spaces instead), neither as a decimal marker (we use the dot instead). In some published articles, we used the comma between groups of three digits. We hope our French compatriots will forgive us.

Symbols. When reading mathematical documents, it can happen that the same symbol is used for multiple reasons. With enough knowledge, the meaning of the symbol will be straightforward given the context. But who can say that he was never a beginner? It also happens in theater that a same actor plays different roles (theater companies have similar problems than researchers regarding fundings, which could explain that). In this manuscript, the letter Δ will play different roles. When encountering $\Delta\phi$, it should be read as "Laplacian of phi", because it represents the Laplace operator:

$$\Delta\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} + \frac{\partial^2\phi}{\partial z^2}$$

When encountering Δt, it should be read as "delta tee", because it represents a variation (here, in time).

Vector product (or cross product). In the French part of this manuscript, we use the French notation $\vec{a} \wedge \vec{b}$ for the vector product in 3d. In the English part of this manuscript, we use the English notation $\vec{a} \times \vec{b}$ instead. Too bad this product is called "cross product" and not "reverse vee product" :-)

A.2 Some Useful Computer Science Notations

Array notation. Throughout this manuscript, whenever we have to refer to indices of objects in an array of N objects, those indices will be in $\{0, 1, \ldots, N-1\}$. This is the standard `C` notation (which is the notation we have in buildings, where we have the ground floor of index 0, then

first floor, etc.). If you come from `Fortran`, where indices start at 1 rather than 0, this might puzzle you a little bit at first (which is the notation in buildings *e.g.*, in Russia, where the floors start at 1 — what we call ground floor is called first floor in Russia, what we call first floor is called second floor in Russia, and so on).

Complexity. To give an idea of the time needed by a specific algorithm, there exist specific notations [10, Sect. 3.1]. Suppose that an algorithm takes as input data of size N (*e.g.*, sort a list of N elements; find the shortest path among N cities...). With f being a function, this algorithm is said to have complexity:

- $o(f(N))$ if, when N grows, the number of operations needed for the algorithm is really small compared to $f(N)$.
- $\mathrm{O}(f(N))$ if, when N grows, the number of operations needed for the algorithm is smaller than $c \cdot f(N)$, where c is a constant.
- $\Omega(f(N))$ if, when N grows, the number of operations needed for the algorithm is greater than $d \cdot f(N)$, where d is a constant.
- $\Theta(f(N))$ if this algorithm has complexity both $\mathrm{O}(f(N))$ and $\Omega(f(N))$.

Memory. In the French part of this manuscript, we use the French notation Go (gigaoctet). In the English part of this manuscript, we use the English notation GB (gigabyte). The most little piece of information on a computer is a bit (binary digit, whose value can thus be 0 or 1). A byte is 8 of those bits (this definition is more natural in French with the prefix "oct"). The prefix giga means we take 10^9 of those bytes. Both notations are commonly used to represent in fact a GiB (gibibyte). A gibibyte is 2^{30} bytes. This approximation is common in computer science, because $2^{10} = 1\,024 \approx 1\,000$.

The memory is always measured for the RAM (Random Access Memory). We do not care about ROM (Read Only Memory), which is the size of the data that can be stored on the disk, but we care only about the size of the dynamic memory.

Appendix B

Technical Corner

Supercomputers. In this thesis, we had access to the supercomputers Curie[1], Marconi[2], and Occigen[3]. We are grateful to the French and European grants that enabled us to use them. Part of this work was granted access to the HPC resources of TGCC under the allocation 2015-T2016067580 made by GENCI. Part of this work was granted access to the HPC resources of CINES under the allocation 2017-A0030510318 made by GENCI. Part of this work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom Research and Training Program 2014–2018 under Grant Agreement No. 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

There exist a worldwide ranking of supercomputers: `https://www.top500.org/`.

Parallel languages. Following the work of many other colleagues in the scientific computing community, we focused here on the `OpenMP` [182] language extension (for `C`, `C++` and `Fortran`) for shared memory parallelism. There are other alternatives, *e.g.*, POSIX Threads (`man pthreads`), C11 threads [17, Section 19], Cilk (Plus) [186].

We also used `MPI` [184] for distributed memory parallelism.

Design of this manuscript. For the rendering of this manuscript, we used:

- LaTeX for the general rendering,
- Palatino Roman (regular text), Computer Modern Typewriter (code examples), Pazo Math (mathematical equations) and Baskerville (`Pic-Vert` logo) for the fonts,
- gnuplot [191] for most graphs,
- the colormap "plasma" from [171] for Figures 4.3–4.6 and 4.25–4.28 on pages 68 and 90,
- VisIt [187] to output 2d visualizations of the particle density and of the charge density in Chapter 7.

[1] `http://www-hpc.cea.fr/fr/complexe/tgcc-curie.htm`
[2] `https://www.cineca.it/en/content/marconi`
[3] `https://www.cines.fr/calcul/materiels/occigen/`

Annexe C

Introduction (en français)

Tout au long de cette thèse, notre travail a consisté en des implémentations efficaces de simulations numériques dans le domaine de la physique des plasmas. Ce chapitre d'introduction va donner une vue d'ensemble des notions physiques et mathématiques qui sont au cœur de ces implémentations. Certaines des notations utilisées peuvent être difficiles à comprendre, n'hésitez pas à vous reporter au Chapitre A.

Tout d'abord, la Section C.1 motive le besoin en simulations dans le domaine de la physique des plasmas, explique ce qu'est un plasma, et donne un aperçu de la physique qui s'y déroule.

La Section C.2 présente ensuite les équations mathématiques qui gouvernent ces simulations, ainsi que des méthodes pour les résoudre numériquement.

La Section C.3 introduit enfin ce qui est nécessaire en termes informatiques pour comprendre les optimisations qui ont été apportées tout au long de cette thèse.

C.1 Quelques notions de physique

C.1.1 Énergie

> “Today our planet is thoroughly wedded to fossil fuels in the form of oil, natural gas, and coal. Altogether, the world consumes about 14 trillion watts of power, of which 33 percent comes from oil, 25 percent from coal, 20 percent from gas, 7 percent from nuclear, 15 percent from biomass and hydroelectric, and a paltry .5 percent from solar and renewables.[1]
>
> M. Kaku [24, Chapter 5] ”

La maîtrise de l'énergie n'est pas un phénomène nouveau dans l'histoire. Nous avons fait travailler d'autres animaux pour nous, exploité des premières sources d'énergie renouvelable (les moulins à vent, les roues à aube…), avant de découvrir de nouvelles manières d'utiliser les ressources énergétiques de notre planète : le charbon, puis le pétrole.

Cela dit, ces nouvelles sources d'énergie que nous utilisons directement ou indirectement tous les jours ont une nouveauté par rapport aux autres : elles ont un impact massif sur notre planète. Les rencontres internationales (par exemple, le protocole de Kyoto en 1997), les groupes de recherche internationaux (par exemple, le GIEC — Groupe d'experts Intergouvernemental sur l'Évolution du Climat, http://www.ipcc.ch) nous expliquent les désastres climatiques vers lesquels on se dirige si l'on continue à se comporter de la sorte, et essayent de

[1] “Aujourd'hui, notre planète est très fortement liée aux combustibles fossiles sous la forme de pétrole, de gaz naturel, et de charbon. En tout, le monde consomme à peu près 14 billions de watts d'énergie, dont 33 pourcents proviennent du pétrole, 25 pourcents du charbon, 20 pourcents du gaz, 7 pourcents du nucléaire, 15 pourcents de la biomasse et de l'hydroélectrique, et une quantité dérisoire de 0.5 pourcent du solaire et des renouvelables.” M. Kaku (traduction par l'auteur de ce manuscrit)

montrer des chemins à suivre si l'on veut éviter un réchauffement climatique intenable, voir par exemple la Figure C.1.

Aujourd'hui, de nombreuses personnes vantent les mérites de l'énergie "verte" définie comme celle donnée par le vent et le soleil : les éoliennes, les panneaux solaires... Ces énergies pourraient être notre futur, parce qu'elles relâchent moins de dioxyde de carbone (CO_2). Cela dit, pour ces énergies il existe encore des problèmes de disponibilité (nous ne pouvons pas contrôler le vent et avons quand même besoin d'énergie quand il ne souffle pas) et en stockage (les batteries actuelles ne sont pas assez puissantes pour stocker l'énergie produite ; le pompage-turbinage est une technique de stockage sans batteries, mais en très petite quantité). Réduire les émissions de CO_2 avec les énergies renouvelables n'est possible que si l'on réduit en parallèle notre consommation de combustibles fossiles. Malheureusement, des centrales à charbon (qui relâchent beaucoup de CO_2) sont couramment utilisées pour produire de l'énergie lorsque les éoliennes et les panneaux solaires n'en produisent pas (par exemple, la nuit).

Une source d'énergie qui existe aujourd'hui et qui a également la propriété de relâcher moins de CO_2 que les combustibles fossiles est l'énergie produite par les centrales nucléaires, grâce à la fission atomique. Nous devons cela dit souligner que les centrales électriques sont un sujet polémique : les accidents nucléaires de Three Mile Island (1979), de Tchernobyl (1986) et de Fukushima (2011) ont changé notre point de vue sur cette énergie. Certaines erreurs politiques n'ont pas non plus aidé à rendre cette énergie acceptable. On peut par exemple citer les nombreux pots-de-vin à des hommes politiques dont les pays contiennent des ressources d'uranium [196], ou bien les grandes quantités d'argent dépensées pour acheter des mines d'uranium desquelles rien ne peut être extrait[2], etc.

Bien que ce soit probablement de nos jours l'une des énergies qui relâche le moins de CO_2, il y a tout de même un problème de taille : en produisant de l'énergie avec la fission, des "déchets nucléaires" sont créés, dont les durées de demi-vie peuvent dépasser 200 000 ans. Des recherches sont en cours pour expliquer avec des symboles clairs, aux endroits où on les stocke, que ces déchets sont dangereux, afin que nos descendants dans plusieurs milliers d'années comprennent ces symboles [129]. Un autre défi est de réellement construire de tels lieux capables de stocker ces déchets nucléaires pendant tant de temps [154].

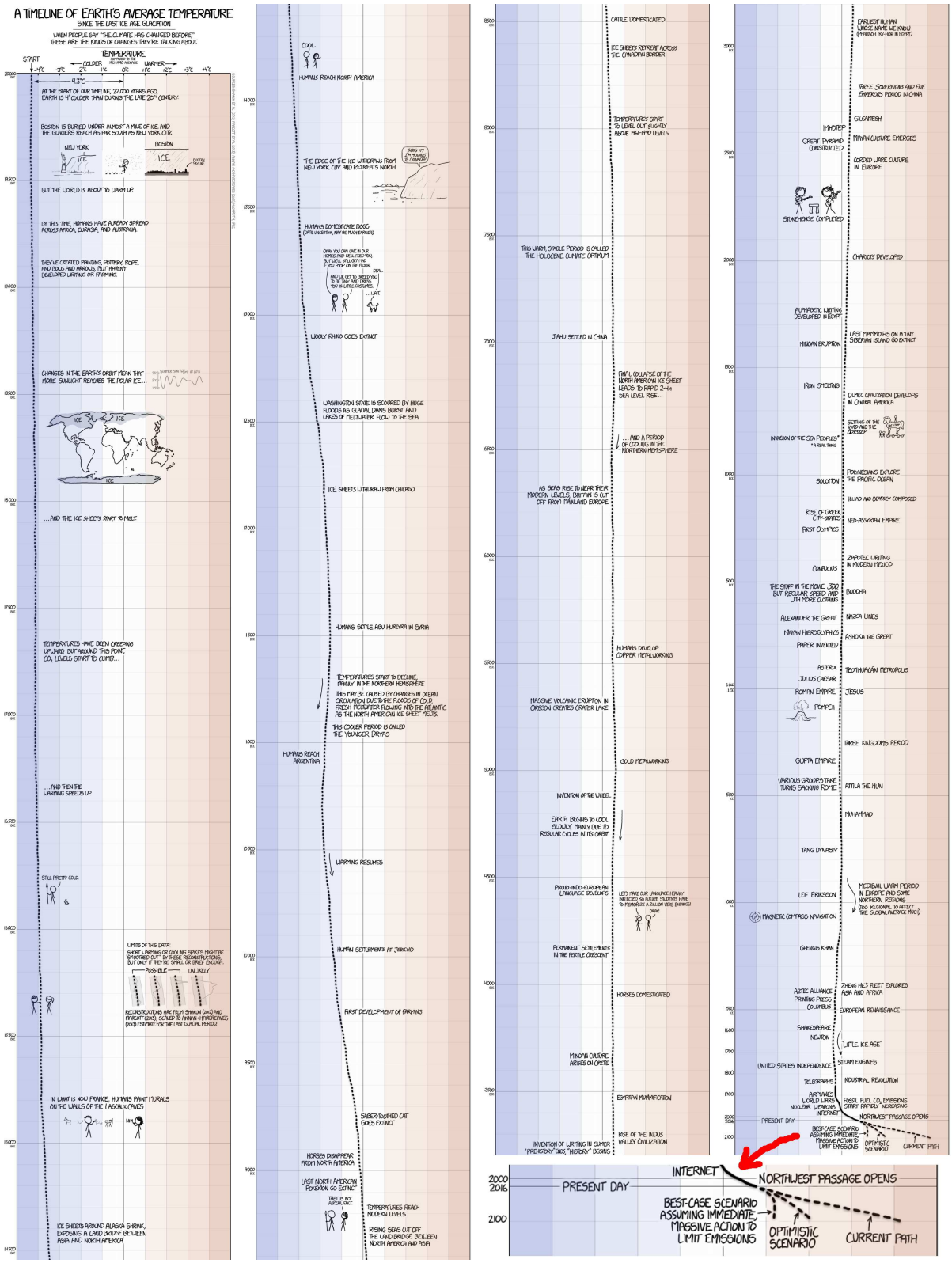
> "Les scientifiques annoncent ainsi l'avènement des piles à combustible, de la fusion par laser ou par confinement magnétique, des véhicules à hydrogène ou à sustentation magnétique, et même des centrales solaires placées en orbite autour de la Terre. [24, Chapter 5]
>
> G. Pitron [199] "

Face à ces problèmes d'énergie, les chercheurs essayent de mettre à jour de nouveaux moyens de produire de l'énergie. L'un de ces moyens est une application majeure de nos travaux : la *fusion thermonucléaire contrôlée*. La fusion est une réaction qui peut être vue comme l'inverse de la fission. La fission crée de l'énergie en "cassant" de gros noyaux (par exemple, d'uranium) pour en créer de plus petits. La fusion crée de l'énergie en "fusionnant" de petits noyaux (par exemple, d'hydrogène) pour en créer de plus gros. Ce point est déjà une amélioration par rapport à la fission, car il est bien plus simple de se procurer de l'hydrogène que de l'uranium : on peut produire du deutérium et du tritium (les isotopes d'hydrogène dont on a besoin) au lieu d'avoir besoin de l'extraire de notre sous-sol.

En produisant de l'énergie avec la fusion, aucun déchet nucléaire direct n'est créé. Cela dit, les neutrons massivement chargés d'énergie qui sont créés vont irradier la structure environnante, et c'est toujours un sujet de recherche de réussir à apprivoiser ce phénomène. Aujourd'hui, créer de l'énergie grâce à la fusion est donc un défi majeur. Le projet international ITER[3], situé à Cadarache (France), s'est fixé ce but.

[2] https://fr.wikipedia.org/wiki/UraMin

[3] International Thermonuclear Experimental Reactor, "Le chemin" (en latin) : http://www.iter.org



"[After setting your car on fire] Listen, your car's temperature has changed before."

Crédits — Randall Munroe, https://xkcd.com/1732/ ; Sources — Shakun *et al.* (2012) [168], Marcott *et al.* (2013) [161], Annan and Hargreaves (2013) [131], HadCRUT4 (https://www.metoffice.gov.uk/hadobs/hadcrut4/), GIEC (http://www.ipcc.ch/).

FIGURE C.1 – Chronologie de la température moyenne terrestre.

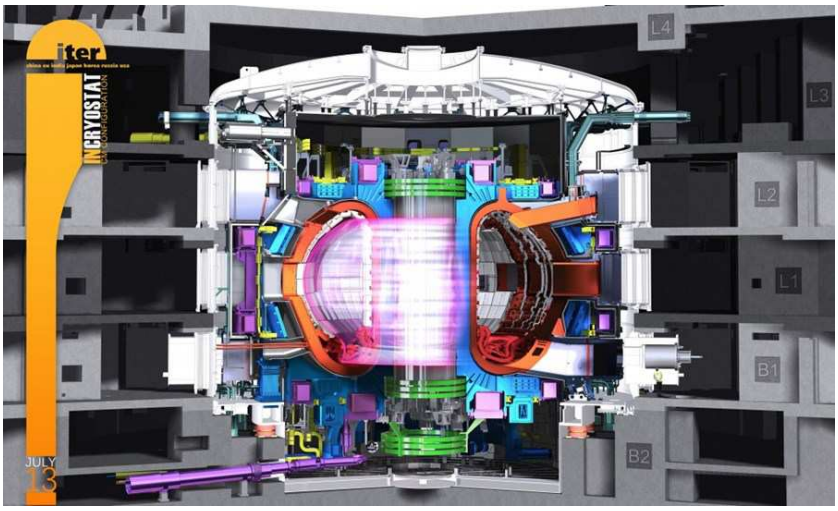


FIGURE C.2 – Le tokamak d'ITER — vue d'artiste. *Image utilisée avec l'aimable accord d'ITER*

Les projets comme ITER sont très coûteux. Avant d'utiliser une machinerie aussi complexe qu'un *tokamak*[4] de 29 m × 28 m — voir la Figure C.2 — nous devons savoir ce qu'il y a à l'intérieur et comment cela se comporte.

> « La fusion est prometteuse, mais ne pourra guère être industrialisée avant la fin du XXI^e siècle.
>
> B. Barré [192, Chapter 13] »

Avant d'expliquer ce qui se trouve dans le gros doughnut de la Figure C.2, nous devons reconnaître que la fusion ne pourra probablement pas nous fournir d'énergie pour une utilisation grand public avant la fin du siècle. Ainsi, même si nous pensons que c'est une bonne idée d'investir notre temps et notre argent dans cette source prometteuse d'énergie, nous devons également changer notre mode de vie — au lieu de le considérer "non négociable" — afin d'utiliser moins d'énergie jusqu'alors. Finissons cette section sur une note optimiste et parions sur le fait que nous ferons des choix politiques courageux dans la direction d'un monde plus raisonnable, où la croissance économique ne sera plus notre but principal.

C.1.2 Plasma

> « This reminds [Langmuir] of [...] the way blood plasma carries around red and white corpuscles and germs. So he proposed to call our 'uniform discharge' a 'plasma'.[5]
>
> H. M. Mott-Smith [164] »

Pour produire de l'énergie grâce à la fusion, l'idée du projet ITER est de créer un plasma à l'intérieur d'un tokamak, puis de contenir ce plasma à l'intérieur avec un champ magnétique élevé. Le plasma est le quatrième état de la matière, et on pense qu'il forme 99% de la masse de l'univers visible. La matière atteint cet état à de très hautes températures (> 10 000 K). À ces températures, les *particules* (ions et électrons) se comportent différemment. Il y a du plasma,

[4] Токамак: тороидальная камера с магнитными катушками (une chambre toroïdale avec des bobines magnétiques)

[5] "Cela rappela [à Langmuir] [...] la manière dont le plasma sanguin conduit les globules blancs et rouges ainsi que les germes. Il proposa donc d'appeler notre 'décharge uniforme' un 'plasma'." H. M. Mott-Smith (traduction par l'auteur de ce manuscrit)

par exemple, dans un éclair, dans un tube de néon, dans le soleil. Maîtriser la fusion thermonucléaire contrôlée n'est donc rien d'autre que de mettre le soleil dans un doughnut. On peut trouver plus d'informations sur le plasma et ses applications sur http://www.plasmas.org.

Pour comprendre comment se comporte un plasma, nous devons étudier les particules qui y résident, et donc nous devons traquer leurs positions spatiales $\vec{x} = (x, y, z) \in \mathbb{R}^3$ et leurs vitesses $\vec{v} = (v_x, v_y, v_z) \in \mathbb{R}^3$ au cours du temps. Pour traquer les particules, il y a essentiellement trois modèles en physique des plasmas :

- le *modèle à N corps* : dans ce modèle, nous suivons toutes les N particules, en prenant en compte toutes les interactions créées par chaque couple de particules, ce qui implique donc $\Theta(N^2)$ interactions. Ce modèle est le plus précis. Il existe des formules pour $N = 2$, mais ce problème requiert des approximations dès que $N = 3$. Malheureusement, à cause de la complexité de ce modèle, nous devrons peut-être attendre les prochains ordinateurs quantiques pour pouvoir utiliser cette méthode efficacement (dans un plasma avec $N > 10^{10}$ particules, il y a plus de $10^{20} = 100\,000\,000\,000\,000\,000\,000$ interactions).
- le *modèle cinétique* : plutôt que de suivre chaque particule, nous étudions la densité de particules f (une fonction de sept variables : trois pour les positions, trois pour les vitesses, et une pour le temps), qui donne la probabilité de présence de particules autour d'un certain temps, d'une certaine position et d'une certaine vitesse. Bien sûr les différentes *espèces* de particules dans le plasma ont différents impacts sur le comportement, donc nous devons suivre une fonction de densité par espèce. Habituellement, nous avons des électrons et un type d'ions, donc nous devons suivre f_e (pour les électrons) et f_i (pour les ions).
- le *modèle fluide* : ce modèle peut être utilisé lorsque la fonction de densité f respecte certaines propriétés. Nous pouvons alors résoudre des équations un peu moins coûteuses, qui donnent une vue macroscopique du plasma, plutôt qu'une vision microscopique.

Dans cette thèse, nous nous plaçons dans le modèle cinétique : nous résolvons numériquement le système d'équations Vlasov–Poisson, voir la Section C.2.1. Ces équations sont résolues dans *l'espace des phases* (un espace à six dimensions : trois pour les positions, trois pour les vitesses). Parfois il est possible d'utilise des modèles simplifiés avec moins de dimensions. Par exemple dans notre thèse, nous avons simulé des cas tests dans un espace des phases 1d1v (1 dimension pour les positions, 1 dimension pour les vitesses : 2 dimensions au total), mais aussi en 2d2v, 2d3v, et 3d3v.

C.2 Quelques notions mathématiques

C.2.1 Les équations à résoudre

> “[...] вследствие большой массы ионов в сравнении с электронами можно их перемещением пренебречь, т.е. считать ионы фактически неподвижными[6]”
>
> А. А. Власов [178, Section 2]

L'équation principale que nous devons résoudre est due à Vlasov [178, Equation II]. Elle peut être reformulée de la sorte : la fonction de distribution f est conservée le long des trajectoires des particules qui sont déterminées par le champ électrique moyen [35, Section 2.1]. Dans le cas non-relativiste, cela s'exprime par le fait que $\frac{\mathrm{d}f(\vec{x}, \vec{v}, t)}{\mathrm{d}t} = 0$, ce qui donne l'équation suivante :

[6] “[...] étant donnée la masse importante des ions comparée à celle des électrons, nous pouvons négliger leurs déplacements, i.e., nous pouvons quasiment considérer que les ions sont immobiles” A. A. Vlasov (traduction par l'auteur de ce manuscrit)

$$\underbrace{\frac{\mathrm{d}\vec{x}}{\mathrm{dt}}}_{\vec{v}} \cdot \nabla_{\vec{x}} f + \underbrace{\frac{\mathrm{d}\vec{v}}{\mathrm{dt}}}_{\vec{a}} \cdot \nabla_{\vec{v}} f + \underbrace{\frac{\mathrm{dt}}{\mathrm{dt}}}_{1} \frac{\partial f}{\partial t} = 0 \tag{C.1}$$

Vlasov a également remarqué que la seule force qui agit sur les particules est la force électromagnétique, aussi appelée la *force de Lorentz* :

$$\vec{F}(\vec{x},t) = q(\vec{E}(\vec{x},t) + \vec{v}(t) \wedge \vec{B}(\vec{x},t)) \tag{C.2}$$

Avec les notations suivantes :

- $\vec{E}(\vec{x},t)$: le champ électrique
- $\vec{B}(\vec{x},t)$: le champ magnétique

La seconde loi de Newton indique que, dans le cas non-relativiste, la somme des forces est égale à la masse multipliée par l'accélération :

$$\sum \overrightarrow{\text{forces}} = m \cdot \vec{a}$$

Cette loi mise ensemble avec (C.2) implique que $\vec{a} = \frac{q}{m}(\vec{E} + \vec{v} \wedge \vec{B})$. En remplaçant $\vec{a}$ par cette valeur dans (C.1), nous obtenons l'équation de Vlasov que nous allons utiliser tout au long de cette thèse, voir (C.3).

$$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla_{\vec{x}} f + \frac{q}{m}(\vec{E} + \vec{v} \wedge \vec{B}) \cdot \nabla_{\vec{v}} f = 0 \tag{C.3}$$

Cette équation est couplée avec les quatre équations de Maxwell qui permettent de déduire les champs auto-cohérents $\vec{E_s}(x,t)$ et $\vec{B_s}(x,t)$, voir (C.4).

$$\begin{cases} \operatorname{div} \vec{B_s} = \vec{0} & \text{Maxwell–Thomson} \\ \operatorname{rot} \vec{E_s} = -\dfrac{\partial \vec{B_s}}{\partial t} & \text{Maxwell–Faraday} \\ \operatorname{div} \vec{E_s} = \dfrac{\rho}{\epsilon_0} & \text{Maxwell–Gauss} \\ \operatorname{rot} \vec{B_s} = \mu_0 \left(\vec{J} + \epsilon_0 \dfrac{\partial \vec{E_s}}{\partial t} \right) & \text{Maxwell–Ampère} \end{cases} \tag{C.4}$$

Avec les notations :

- ϵ_0, μ_0 : permittivité du vide et perméabilité magnétique du vide, liées à la vitesse de la lumière : $c = \dfrac{1}{\sqrt{\epsilon_0 \cdot \mu_0}}$
- $\rho(\vec{x},t) = q \int f(\vec{x},\vec{v},t)\,\mathrm{d}\vec{v}$: densité volumique de charge électrique
- $\vec{J}(\vec{x},t) = q \int f(\vec{x},\vec{v},t)\vec{v}\,\mathrm{d}\vec{v}$: densité de courant

Dans cette thèse, il n'y a aucun champ électrique externe. Dans certains cas, il n'y a aucun champ externe, ce qui veut dire que les champs sont exactement les champs auto-cohérents : $\vec{E}(\vec{x},t) = \vec{E_s}(\vec{x},t)$ et $\vec{B}(\vec{x},t) = \vec{B_s}(\vec{x},t)$. Dans d'autres cas (comme dans le tokamak du projet ITER), il y a un champ magnétique externe $\vec{B_e}(\vec{x},t)$, ce qui veut dire que $\vec{B}(\vec{x},t) = \vec{B_s}(\vec{x},t) + \vec{B_e}(x,t)$.

Dans certains cas, nous pouvons simplifier les équations de Maxwell : nous nous plaçons dans l'hypothèse simplificatrice que $||\vec{v}|| \ll c$, qui implique que nous pouvons négliger la densité de courant et le champ magnétique. Il ne nous reste qu'une seule équation à la place de quatre, une équation de Poisson :

$$\boxed{-\Delta_{\vec{x}}\phi = \frac{\rho}{\epsilon_0}} \quad \text{où} \quad \vec{E}(\vec{x},t) = -\nabla_{\vec{x}}\phi(\vec{x},t)$$

La plupart du temps, cette thèse se focalisera sur le système Vlasov–Poisson, dans les cas où :

- il n'y a aucun champ externe
- le champ magnétique auto-cohérent est négligé
- les ions sont supposés immobiles (et neutralisent la charge), donc nous ne simulons que des électrons de charge $q = -\mathrm{e}$ (où e est la charge élémentaire)

Cela donne le système d'équations (C.5).

$$\begin{cases} \dfrac{\partial f}{\partial t} + \vec{v}\cdot\nabla_{\vec{x}} f + \dfrac{q}{m}\vec{E}\cdot\nabla_{\vec{v}} f = 0 & \text{Vlasov} \\ -\Delta_{\vec{x}}\phi = \dfrac{\rho}{\epsilon_0}\left(= \dfrac{q}{\epsilon_0}\left(\displaystyle\int f(\vec{x},\vec{v},t)\,\mathrm{d}\vec{v} - 1\right)\right) & \text{Poisson} \end{cases} \tag{C.5}$$

De plus, dans cette thèse nous travaillons dans un monde adimensionné où $\mathrm{e} = m = \epsilon_0 = 1$ (donc $q = -1$). Parfois, nous aurons besoin d'ajouter un champ magnétique externe constant, parfois nous simulerons à la fois les électrons et les ions, et nous devrons donc mettre à jour le système en fonction.

C.2.2 Approximations numériques

> « On pourroit aussi construire une table des différences premières & l'employer seule. [...] [L]e procédé est aussi exact que la méthode des secondes différences, mais il est moins commode. Ce qui vient en partie de ce que [...] pour les différences premières il faut les prendre dans la table subsidiaire avec $(u + \frac{1}{2}du)$.
>
> J.-B. Delambre [142, Article X] »

Nous rappelons qu'il est difficile de prouver l'existence de solutions au système général de manière rigoureuse. Utilisons la notation $f(t) = \{f(\vec{x},\vec{v},t) \mid (\vec{x},\vec{v}) \in \text{espace des phases}\}$, qui représente toutes les valeurs de f au temps t. Nous n'avons donc pas de formule donnant $f(t)$ pour tout t.

Schémas numériques

Nous utilisons une méthode numérique pour obtenir des approximations de la solution :

- $f(0)$ est connue (c'est la condition initiale)
- on choisit un Δt "petit" (le pas de temps) et n (le nombre de pas de temps)
- on déduit $f^*(\Delta t)$, une approximation de $f(\Delta t)$, à partir de $f(0)$
- puis $f^*(2\Delta t)$, une approximation de $f(2\Delta t)$, à partir de $f^*(\Delta t)$
- puis $f^*(3\Delta t)$, une approximation de $f(3\Delta t)$, à partir de $f^*(2\Delta t)$
- ...
- et enfin $f^*(n\Delta t)$, une approximation de $f(n\Delta t)$, à partir de $f^*((n-1)\Delta t)$

Euler a proposé un schéma numérique [13, Part I, Section II, Chapter VII, Problem 85] dont nous donnons ici l'idée si on travaille avec des fonctions de $\mathbb{R}$ dans $\mathbb{R}$. L'inconnue est f ; nous connaissons g, une équation différentielle $f'(t) = g(t)$, et $f(a)$; nous voulons connaître $f(b)$. "Suffisamment proche" de n'importe quelle valeur x, f se comporte comme une fonction affine (sa dérivée), c'est-à-dire que pour $x \in [a;b]$ et pour de petites valeurs de h,

$$f(x+h) = f(x) + h \cdot f'(x) + O(h^2) \qquad \text{(Série de Taylor au premier ordre)} \qquad \text{(C.6)}$$

Avec n étapes de taille $\Delta t = \frac{b-a}{n}$, on peut obtenir une bonne approximation de $f(b)$, en calculant :

$$\begin{cases} f^*\left(a + \frac{b-a}{n}\right) = f(a) + \frac{b-a}{n} \cdot g(a) \\ \forall 2 \leq k \leq n, f^*\left(a + k \cdot \frac{b-a}{n}\right) = f^*\left(a + (k-1) \cdot \frac{b-a}{n}\right) + \frac{b-a}{n} \cdot g\left(a + (k-1) \cdot \frac{b-a}{n}\right) \end{cases}$$

Le schéma d'Euler est une méthode de *premier ordre* : l'erreur est au plus proportionnelle à Δt[7]. Un schéma numérique plus précis (de *second ordre*) est la méthode saute-mouton, qui a été (re)découverte de nombreuses fois, voir par exemple, [142, Article X], [172, Chapter III], [176], d'où son nom, l'algorithme de Verlet–Störmer–Delambre. Son application aux méthodes particulaires ou PIC (pour *Particle-in-Cell*) est expliquée dans [5, Section 2.4].

D'autres schémas numériques souvent utilisés sont les méthodes Runge–Kutta [18, Chapter 2], qui existent pour n'importe quel ordre[8]. Enfin, il y a également les méthodes par *scindage* (*splitting*) qui sont spécifiques à certains types d'équations différentielles, par exemple, [94] pour l'équation de Vlasov, qui est utilisée dans la méthode semi-Lagrangienne.

Méthode des caractéristiques

Il n'est pas évident d'utiliser l'un de ces schémas numériques pour résoudre le système de Vlasov–Poisson, parce que l'équation de Vlasov est une équation aux dérivées partielles. Il est plus simple de résoudre des équations aux dérivées ordinaires. Passer des premières aux secondes peut se faire grâce à la méthode des caractéristiques [15, Section 1.4][14, Section 3.2].

L'idée est d'utiliser des fonctions paramétriques à la place des variables t, $\vec{x}$ and $\vec{v}$. Nous introduisons donc trois fonctions $T : \mathbb{R} \to \mathbb{R}$, $\vec{X} : \mathbb{R} \to \mathbb{R}^3$ et $\vec{V} : \mathbb{R} \to \mathbb{R}^3$. Nous cherchons une équation différentielle qui contienne $f(\vec{X}(s), \vec{V}(s), T(s))$. Nous calculons maintenant $\frac{\mathrm{d}f(\vec{X}(s), \vec{V}(s), T(s))}{\mathrm{d}s}$ ce qui donne :

$$\frac{\mathrm{d}\vec{X}(s)}{\mathrm{d}s} \cdot \nabla_{\vec{x}} f(\vec{X}(s), \vec{V}(s), T(s)) + \frac{\mathrm{d}\vec{V}(s)}{\mathrm{d}s} \cdot \nabla_{\vec{v}} f(\vec{X}(s), \vec{V}(s), T(s)) + \frac{\mathrm{d}T(s)}{\mathrm{d}s} \frac{\partial f}{\partial t}(\vec{X}(s), \vec{V}(s), T(s))$$

Maintenant si l'on fixe $\frac{\mathrm{d}\vec{X}(s)}{\mathrm{d}s} = \vec{V}(s)$, $\frac{\mathrm{d}\vec{V}(s)}{\mathrm{d}s} = \frac{q}{m}\vec{E}(\vec{X}(s), s)$ et $\frac{\mathrm{d}T(s)}{\mathrm{d}s} = 1$, nous reconnaissons le membre de gauche de l'équation de Vlasov (C.5). C'est donc ce que l'on fait. On choisit $T(s) = s$ comme solution naturelle de $\frac{\mathrm{d}T(s)}{\mathrm{d}s} = 1$ et nous trouvons donc que les caractéristiques de l'équation de Vlasov sont solution du système d'équations aux dérivées ordinaires (C.7).

[7] Nous avons n étapes, et à chaque étape l'erreur est au plus proportionnelle à $(\frac{b-a}{n})^2$, voir (C.6). La constante dépend donc de la longueur de l'intervalle $|b-a|$ et de $sup_{x\in[a;b]}|g'(x)|$.

[8] En fait, la méthode Runge–Kutta de premier ordre est la méthode d'Euler.

$$\begin{cases} \dfrac{\mathrm{d}\vec{X}(s)}{\mathrm{d}s} = \vec{V}(s) \\ \dfrac{\mathrm{d}\vec{V}(s)}{\mathrm{d}s} = \dfrac{q}{m}\vec{E}(\vec{X}(s), s) \qquad \text{(Seconde loi de Newton)} \end{cases} \tag{C.7}$$

Nous voyons ici une propriété très importante de f, qui provient de l'équation de Vlasov (C.5). Cette équation nous dit que $\dfrac{\mathrm{d}f(\vec{X}(t), \vec{V}(t), t)}{\mathrm{d}t} = 0$ ce qui implique la propriété suivante :

Propriété du système de Vlasov–Poisson
Si f est solution du système de Vlasov–Poisson, alors f est constante le long des caractéristiques de l'équation de Vlasov.

Cette propriété nous permet de suivre f en temps simplement en suivant les caractéristiques.

Vérification de l'implémentation

Afin de vérifier notre implémentation et de comprendre l'erreur que nous commettons avec la simulation numérique, nous avons à notre disposition des cas tests pour lesquels il existe des solutions théoriques presque exactes, grâce à une analyse de dispersion, par exemple, [157]. Certains de ces cas tests et leurs solutions théoriques sont disponibles dans [35, Chapter 4]. Nous avons également mis au point de nouveaux cas tests avec leurs solutions théoriques pendant cette thèse [207].

Discrétisation spatiale

En plus des erreurs inhérentes au schéma de discrétisation en temps, nous commettons également des erreurs avec la discrétisation spatiale. Nos ordinateurs ont une mémoire finie, et nous ne pouvons donc pas avoir accès à toutes les "valeurs de $\vec{E}$ au temps t" : pour un t fixé, un ordinateur ne peut pas stocker $\vec{E}(\vec{x}, t)$ pour toutes les valeurs de $\vec{x}$. Nous stockons alors les valeurs sur une grille, par exemple pour l'axe des x :

- on peut toujours choisir de définir l'espace physique dans lequel les particules évoluent comme l'intervalle $[x_{\min}; x_{\max}]$:
  - lorsque cet espace est périodique (par exemple, un tore), on choisit $x_{\max} - x_{\min} =$ période (conditions aux bords périodiques)
  - lorsque cet espace est fermé (les particules ne vont pas plus loin qu'une limite donnée), on sait que f vaut 0 au-delà de cette limite (conditions aux bords libres)
  - dans d'autres cas, l'espace physique évolue à chaque pas de temps, i.e. $x_{\min}$ et $x_{\max}$ dépendent du temps (fenêtre mobile)
- on choisit un Δx "petit" et on ne stocke que $\dfrac{x_{\max} - x_{\min}}{\Delta x}$ valeurs différentes sur l'axe des x
- pour chaque pas de temps, on ne stocke les valeurs de $\vec{E}$ que sur la grille

C.3 Quelques notions d'informatique

Notre but dans cette thèse est d'écrire des algorithmes parallèles efficaces. C'est exactement la même chose qu'il faut faire lorsque vous voulez cuisiner efficacement avec vos amis. L'algorithme sera votre recette de cuisine, et le parallélisme viendra des différentes personnes qui travailleront ensemble à réaliser la recette.

- Beurrez et farinez un moule à soufflé d'environ 20 cm de diamètre.
- Mettez le moule au réfrigérateur.
- Préchauffez le four à 200 °C (thermostat 6-7).
- Lavez et équeutez 300 g d'épinards frais.
- Faites fondre les épinards avec 10 g de beurre dans une poêle pendant 4 ou 5 minutes.
- Égouttez les épinards dans une passoire, et hachez-les au couteau.
- Cassez 4 œufs en séparant les blancs des jaunes.
- Préparez 300 mL de béchamel.
- Ajoutez à la béchamel les jaunes d'œufs, les épinards et une pincée de curry.
- Montez les blancs d'œufs en neige ferme avec une pincée de sel.
- Incorporez-les délicatement à la béchamel à l'aide d'une spatule.
- Remplissez le moule de cette préparation.
- Enfournez pour 30 minutes et n'ouvrez pas la porte du four durant la cuisson.
- Servez aussitôt.

FIGURE C.3 – Recette pour un soufflé aux épinards.

La Figure C.3 contient une recette de cuisine pour un plat que j'adore préparer : un soufflé aux épinards — adaptée de https://cuisine.larousse.fr/recette/souffle-aux-epinards. Premièrement, la recette contient de nombreux ordres que tout le monde comprend (beurrer, fariner, mettre, laver, équeuter, fondre, égoutter, hacher, casser, séparer, ajouter, monter, incorporer, remplir, enfourner, servir). Ils correspondent à ce que l'on appelle les *instructions* d'un algorithme. Puis, il y a une ligne spéciale que l'on ne comprend peut-être pas (préparer une béchamel). Cela correspond à ce que l'on appelle une *fonction*. Si nous ne savons pas ce que cette fonction fait, nous pouvons toujours aller voir les instructions dans sa définition. On peut probablement trouver ailleurs une recette pour préparer une sauce béchamel. Enfin, il y a un ordre dans lequel ces instructions doivent être exécutées. Suivre l'ordre qui est donné dans la recette fonctionnera toujours, mais il est parfois possible de faire certains changements sans changer le résultat final. Par exemple pour cette recette, vous pouvez séparer les œufs au tout début de la recette sans changer le résultat (par exemple, si vous avez peur de casser les jaunes, qui ont tendance à se casser plus facilement lorsque les œufs sont plus chauds).

Savoir quand réorganiser les instructions est l'une des clefs si l'on veut cuisiner avec des amis. Essayer de faire coopérer de multiples ordinateurs sur un programme donné (le but étant de calculer plus rapidement) est le *parallélisme*. Vous avez de nombreuses manières de cuisiner en parallèle :

- *Vous pouvez effectuer différentes tâches en parallèle les unes des autres.* Une solution classique lorsque vous voulez cuisiner avec des amis est que chaque participant s'occupe d'une partie différente du repas (l'entrée, le plat, le dessert). Et même à l'intérieur de ces parties, on peut appliquer récursivement cette solution (la préparation des épinards et de la sauce béchamel peuvent être faites en parallèle).

- *Vous pouvez choisir une tâche, et la diviser parmi les participants.* Ici, laver et équeuter les épinards prend du temps, et vous pouvez le faire efficacement en parallèle : chaque personne prend une portion des épinards et s'en occupe en même temps que les autres.

- *Vous pouvez organiser les différentes tâches en un pipeline.* Cette option n'est pas vraiment réalisable dans notre exemple, mais vient naturellement si l'on doit cuisiner une tarte

aux pommes comme dessert après cet excellent soufflé. Effectivement, il est très efficace et assez courant que quelqu'un épluche les pommes pendant qu'une autre personne les coupe. Si on choisit ce mode de fonctionnement, au tout début la personne en charge de la découpe n'a aucune pomme à couper, puis dès que la première pomme est épluchée, elle a du travail jusqu'à la fin. Réciproquement, la personne en charge de l'épluchage a du travail depuis le début, mais n'a plus rien à faire pendant que l'autre personne coupe la dernière pomme (les choses peuvent varier légèrement si l'épluchage et la découpe des pommes ont des temps sensiblement différents). L'idée est de gagner du temps parce que chaque personne ne s'occupe que d'une seule chose. Il vaut mieux utiliser cette technique avec des êtres humains de manière raisonnée [195], mais c'est une technique excellente sans restriction quand utilisée sur des puces d'ordinateurs.

Dans notre thèse, nous mettrons l'accent sur le second point : diviser chaque tâche parmi les participants. Dans un monde parfait, si on demande à 2 personnes de se répartir une tâche qui demande 1 heure à 1 personne, nous devrions aboutir à une tâche de 30 minutes. L'*accélération* (*speedup* : le temps qu'il faut pour effectuer une tâche tout seul, divisé par le temps qu'il faut pour l'effectuer à plusieurs) optimale est de 2. Cela dit, dans certains programmes, cela ne se passe pas toujours idéalement. Par exemple, même si on divise le soufflé en ramequins et quel que soit le nombre de fours à notre disposition, il faudra toujours attendre 30 minutes pour la cuisson. Le reste de la recette prend à peu près 30 minutes également pour 1 personne. Nous réalisons donc que si nous sommes 2 à cuisiner cette recette, il n'est pas possible de la réaliser plus rapidement qu'en 45 minutes. Cela correspond à une accélération de $\frac{4}{3} \approx 1.33$. Nous venons de re-découvrir la loi d'Amdahl [130]. Amdahl prédisait que les parties séquentielles des programmes resteraient le goulot d'étranglement principal de la programmation en parallèle. Heureusement, sa prédiction ne s'est pas réalisée, et dans les programmes qui nous intéressent, ces parties séquentielles sont si petites que leur temps d'exécution reste négligeable, même dans une exécution parallèle — ce qui n'affecte donc pas l'accélération autant que dans notre exemple.

C.3.1 Le parallélisme

> “Today parallelism is available in all computer systems, and at many different scales starting with parallelism in the nano-circuits that implement individual instructions, and working the way up to parallel systems that occupy large data centers.[9]
>
> U. A. Acar & G. E. Blelloch [1, Section 1.1] ”

Les simulations réalistes nécessitent des billions d'octets de mémoire[10], et sont exécutées pendant des millions de pas de temps. Un ordinateur personnel typique possède entre 4 et 16 Go (milliards d'octets) de mémoire vive. Par exemple l'ordinateur que nous utilisons pour écrire cette thèse dispose de 16 Go de mémoire. Il a fallu faire une demande pour rajouter 8 Go de mémoire aux 8 Go habituellement disponibles, opération qui a coûté 100€. Cette mémoire est bien trop petite pour pouvoir espérer y lancer des simulations réalistes. Même les ordinateurs spécialisés en calcul scientifique n'ont pas assez de mémoire : ils ont souvent environ 100 Go de mémoire. Il est donc impossible de faire l'impasse sur des grappes de multiples ordinateurs, aussi connus sous le terme de *supercalculateurs*.

[9] “Aujourd'hui le parallélisme est disponible dans tous les systèmes composés d'ordinateurs, et à des échelles très variées en commençant par le parallélisme dans les nano-circuits qui implémentent des instructions individuelles, jusqu'aux systèmes parallèles qui occupent les gros centres de données.” U. A. Acar & G. E. Blelloch (traduction par l'auteur de ce manuscrit)

[10] Un octet est composé de 8 bits ; un bit vaut 0 ou 1, c'est l'unité de base des calculs sur un ordinateur.

Pour exécuter des programmes sur ces machines, il faut avoir au moins quelques notions de base sur la programmation en parallèle et sur l'architecture des ordinateurs modernes. Nous allons essayer de pallier toute lacune éventuelle à ces savoirs de base, si cela est nécessaire.

Nous travaillons avec des grappes d'ordinateurs, et il n'est donc pas possible d'éviter le paradigme de mémoire partagée. Coup de chance, c'est le modèle le plus simple et il se trouve même que tout peut être programmé dans ce modèle… même s'il faut également travailler dans le paradigme de la mémoire partagée si l'on veut vraiment profiter de toute la puissance des machines modernes.

La différence entre ces deux modèles est assez simple à comprendre. Supposons que vous soyez à la tête d'une entreprise de fleurs. Vous avez une boutique à Amiens (Somme, Hauts-de-France, France) et une autre au Touquet (Pas-de-Calais, Hauts-de-France, France). Un client rentre dans la première boutique pour acheter un bouquet de roses jaunes. Le vendeur vérifie la disponibilité, voit qu'il peut satisfaire le client, et lui vend le bouquet. Dans l'autre boutique, à peu près au même moment, le même scénario se déroule. Il n'y a aucun problème. À la fin de la semaine, les deux boutiques doivent se synchroniser pour connaître le nombre total de fleurs de chaque type, ainsi que les bénéfices totaux. Quand on travaille en mémoire distribuée, il y a donc besoin de distribuer les données (ici, les fleurs) parmi les différentes unités de calcul (ici, les boutiques), de s'assurer que tout le monde peut travailler avec sa partie des données totales, et enfin rassembler tous les calculs pour obtenir le résultat final et/ou remettre ensemble les différentes données (ici, obtenir le stock de fleurs et d'argent). À chaque fois qu'une unité veut accéder à une partie des données qu'elle n'a pas, elle peut la demander à un collègue, mais cela prendra probablement du temps (ici, envoyer des fleurs d'une boutique à l'autre prendra probablement plusieurs heures), et vous devez vous assurer que votre collègue s'attend à de telles demandes — sinon, il ne répondra jamais.

Supposons maintenant que nous ne sommes pas dans différentes boutiques, mais dans une seule boutique avec deux vendeurs. Envisageons maintenant le scénario suivant — pas très réaliste dans la vie de tous les jours, mais très probable sur un ordinateur — deux clients entrent dans la boutique. Le premier demande un bouquet de cinq roses jaunes, et le premier vendeur va dans l'arrière-boutique vérifier la disponibilité et le prix. Il voit sept roses jaunes, et revient dire à son client qu'il n'y a pas de problème. Son client est d'accord pour acheter les cinq roses au prix indiqué. Au même moment, un autre client veut un bouquet de trois roses jaunes. Le second vendeur se charge de cet autre client, va dans l'arrière-boutique juste après son collègue, y voit toujours les sept roses jaunes, et revient dire à son client qu'il n'y a aucun problème. Le premier client continue ses achats, tandis que le second ne souhaite que ce bouquet. Le second vendeur va chercher les trois fleurs, et son client s'en va après avoir payé, l'air satisfait. Quand le premier client a fini de choisir ses autres bouquets, le premier vendeur repart dans l'arrière-boutique… pour s'apercevoir qu'il n'y a plus que quatre roses jaunes. Bien sûr, pour éviter un tel scénario, chaque vendeur peut prendre avec lui les fleurs quand il part vérifier la disponibilité. En informatique, nous appellerions cela une opération *atomique*. Plutôt que de simplement vérifier un nombre, puis de plus tard changer sa valeur, on fait les deux en une seule passe. Si quelqu'un d'autre veut accéder à la donnée pendant ces deux opérations combinées, il ne peut pas car la donnée est verrouillée pendant une opération atomique. Cela évite le scénario décrit qui s'appelle un *accès concurrent* (deux acteurs différents accédant à la même donnée alors qu'au moins l'un d'entre eux essaye de la modifier). Bien sûr il n'y a aucun problème si les deux accès sont des lectures (les deux vendeurs peuvent vérifier le prix d'un même article sans problème). Nous avons vu ici les bénéfices de la mémoire partagée, mais aussi l'un de ses gros défauts. Quand ils partagent l'arrière-boutique, les deux vendeurs n'ont pas besoin de se synchroniser pour savoir combien de fleurs ils ont en stock. Par contre, ils doivent faire attention quand ils ont besoin de modifier le stock.

L'exemple précédent a présenté un bogue très commun quand on écrit un programme dans le modèle de mémoire partagée. Cela dit, nous ne sommes pas intéressés que par des simulations sans bogue, nous voulons également des simulations rapides. Supposons maintenant

que les deux vendeurs aient évité les accès concurrents en s'occupant des stocks de fleurs sur différents étages de l'étagère. Le premier vendeur pourrait avoir la responsabilité des roses, sur les deux étages du bas de l'étagère, et l'autre pourrait avoir la responsabilité des tournesols, sur les deux étages du haut de l'étagère. Il peut arriver qu'ils aient tous les deux besoin de fleurs sur cette même étagère en même temps. Quand cela arrive, ils vont probablement se gêner, parce qu'ils sont en face de la même étagère. Ce scénario se produit sur un ordinateur quand deux acteurs accèdent à deux données différentes qui sont "trop proches" en mémoire : cela s'appelle un *faux partage*. Premièrement, nous devons comprendre que les données ne sont pas organisées nombre par nombre, mais bloc de nombres par bloc de nombres. Si deux acteurs accèdent à deux nombres dans le même bloc, le système ne peut pas être certain qu'ils sont en train de manipuler deux nombres différents, et doit donc faire attention pour traiter le cas où ils accéderaient au même nombre à l'intérieur de ce bloc. Pour être certain que cela n'arrive jamais, nous pouvons mettre les roses et les tournesols sur deux étagères différentes. Voyant que les deux acteurs sont en train de calculer sur deux nombres dans deux blocs différents, le système peut maintenant être certain qu'aucun accès concurrent ne peut survenir.

Pour aller dans le même sens que les travaux de nombreux collègues dans la communauté du calcul scientifique, nous utiliserons dans cette thèse l'extension de langage `OpenMP` [182] (pour `C`, `C++` et `Fortran`) concernant le parallélisme en mémoire partagée. Les différents acteurs en `OpenMP` sont appelés des *fils d'exécution* (*threads*), ils sont les différents vendeurs dans notre exemple floral. Nous n'utiliserons qu'une petite partie des possibilités d'`OpenMP` dans cette thèse. Notons par exemple qu'`OpenMP` permet le *parallélisme par tâche* qui est utilisé dans certaines implémentations PIC, par exemple, OSIRIS [33, Section 8.6]. Dans cette thèse, nous ne nous sommes pas servis du mécanisme de tâches.

C.3.2 Vectorisation

> "Skill and knowledge of vectorization is absolutely ESSENTIAL to gain performance on the Intel® Xeon Phi™ product family.[11]
>
> https://software.intel.com/en-us/articles/vectorization-essential"

Nous avons vu le parallélisme en mémoire distribuée et en mémoire partagée. Il existe un troisième niveau de parallélisme, le parallélisme vectoriel.

La vectorisation est la transformation d'un code du genre de celui montré dans le Listing C.1 vers celui montré dans le Listing C.2. Dans cet exemple, `A`, `B` et `C` sont des tableaux de `doubles`, et nous faisons l'hypothèse qu'il est possible de calculer 4 opérations sur des nombres réels en double précision à la fois, ce qui nécessite des vecteurs de taille 256 bits (un `double` prend 64 bits de mémoire).

```
1 for (i = 0; i < 1024; i++)
2     C[i] = A[i] + B[i];
```

Listing C.1 – Code sans vectorisation.

```
1 for (i = 0; i < 1024; i+=4)
2     C[i:i+3] = A[i:i+3] + B[i:i+3];
```

Listing C.2 – Code vectorisé.

Le Listing C.1 utilise des instructions scalaires. L'ordinateur charge `A[0]` et `B[0]` dans deux registres scalaires, calcule `A[0] + B[0]`, stocke le résultat dans `C[0]`, puis charge `A[1]` et `B[1]`, calcule `A[1] + B[1]`, stocke le résultat dans `C[1]`... Sur les architectures "modernes", il existe la possibilité, si l'on effectue la même instruction sur de multiples données contiguës (un tableau), d'effectuer l'opération bloc d'éléments par bloc d'éléments, plutôt qu'élément par élément. L'ordinateur utilise alors des instructions vectorielles plutôt que des instructions scalaires. Dans

[11] "Les compétences et la connaissance de la vectorisation sont absolument ESSENTIELLES pour améliorer les performances sur les produits de la famille Intel® Xeon Phi™." https://software.intel.com/en-us/articles/vectorization-essential (traduction par l'auteur de ce manuscrit)

le Listing C.2, l'ordinateur charge `A[0]A[1]A[2]A[3]` dans un registre vectoriel, charge de la même manière `B[0]B[1]B[2]B[3]` dans un autre registre vectoriel, calcule les quatre additions en une seule instruction vectorielle, et stocke le vecteur résultat dans `C[0]C[1]C[2]C[3]`... Si les données ne sont pas contiguës en mémoire — vous pouvez avoir des accès indirects (par exemple, `A[f(i)]`), des pas non-unitaires (par exemple, en utilisant des tableaux de structures) — alors vous pourrez tout de même utiliser les opérations vectorielles mais vous aurez besoin d'opérations de *réunion* et/ou *dispersion* (*gather* et/ou *scatter*) pour charger et/ou stocker les données non contiguës en mémoire.

Ce type de parallélisme est connu sous le nom SIMD (une Seule Instruction sur de Multiple Données, *Single Instruction Multiple Data*), et pour atteindre les performances maximales données par les vendeurs de processeurs, il faut utiliser les instructions vectorielles.

Pour le travail présenté dans cette thèse, nous avons pris soin d'écrire notre code pour qu'il bénéficie de la mémoire distribuée, de la mémoire partagée ainsi que de la vectorisation. Voir la Section 4.3.4 pour des applications dans une implémentation PIC.

C.3.3 Efficacité

> “Note that metrics such as flop/s or percentage-of-peak are less relevant for the predominantly memory-bound gyrokinetic PIC methods, as modern architectures require 10 flops per byte moved from DRAM in order to be compute-limited.[12]
>
> W. Tang, B. Wang, S. Ethier, G. Kwasniewski, T. Hoefler, K. Z. Ibrahim, K. Madduri, S. Williams, L. Oliker, C. Rosales-Fernandez and T. Williams [83] ”

La majeure partie du travail fourni dans cette thèse concerne l'optimisation d'une implémentation PIC. Mais dès qu'on parle d'optimisation, de nombreuses questions émergent :

- Comment mesurer la performance d'une implémentation ?
- Comment lire et comprendre des résultats de performance ?
- Comment comparer les performances de deux implémentations différentes ?

Pour étudier la performance de notre implémentation, tout au long de cette thèse, les mesures de performances ont été obtenues en rajoutant quelques lignes de code pour obtenir des temps d'exécution. Observer un phénomène peut pourtant changer ce phénomène [139]. Nous avons donc pris soin de vérifier que ce n'était pas le cas ici. Il est possible d'obtenir des informations plus détaillées en utilisant des outils plus complexes, mais nous avons choisi la simplicité de cette approche.

Pour étudier la performance parallèle d'une implémentation, il y a deux tests.

Le premier test est le passage à l'échelle fort (*strong scaling*) : nous commençons avec un problème de taille N résolu par P processeurs. Puis, nous ajoutons quelques processeurs, tout en gardant la taille totale du problème N constante. Ce test évalue la capacité d'un programme à résoudre un problème donné sur de plus grosses machines.

[12] “Notez que les métriques telles que le nombre d'opérations en virgule flottante par seconde ou le pourcentage de la performance maximum sont moins pertinentes pour les méthodes PIC gyrocinétiques qui sont avant tout limitées par les accès mémoire, puisque les architectures modernes demandent 10 opérations en virgule flottante par octet transféré de la mémoire vive dynamique pour être limitées par les calculs.” W. Tang, B. Wang, S. Ethier, G. Kwasniewski, T. Hoefler, K. Z. Ibrahim, K. Madduri, S. Williams, L. Oliker, C. Rosales-Fernandez and T. Williams (traduction par l'auteur de ce manuscrit)

Parallélisme idéal (passage à l'échelle fort)
On dit qu'il y a parallélisme idéal si, quand on multiple le nombre de processeurs par k, le problème est résolu k fois plus vite[a].

[a] Remarque : parfois, des effets super-linéaires sont observés : le problème est résolu plus que k fois plus vite. Cela peut arriver par exemple quand le fait de découper le problème en sous-problèmes plus petits aboutit à une meilleure réutilisation des données dans le cache. Dans ce cas, cela veut dire que l'implémentation séquentielle peut probablement être améliorée pour également bénéficier d'une meilleure réutilisation du cache.

Le second test est le passage à l'échelle faible (*weak scaling*) : nous commençons avec un problème de taille N résolu par P processeurs. Puis, nous ajoutons quelques processeurs, tout en gardant la taille du problème par processeur N/P constante. Ce test évalue la capacité d'un programme à résoudre de plus gros problèmes sur de plus grosses machines.

Parallélisme idéal (passage à l'échelle faible)
Quand on considère un problème de complexité linéaire — $\Theta(N)$, comme l'algorithme PIC —, on dit qu'il y a parallélisme idéal si le temps d'exécution ne change pas alors qu'on augmente le nombre de processeurs.

Par exemple regardons la Figure C.5, en commençant à partir de 8 cœurs (le comportement de 1 à 8 cœurs sera expliqué plus tard). Nous voyons que nous avons un passage à l'échelle faible presque idéal jusqu'à 512 cœurs, et ensuite notre temps d'exécution augmente alors qu'il devrait continuer à être constant. C'est dû au fait que le schéma de parallélisation choisi ajoute un facteur logarithmique au temps d'exécution, comme discuté dans la Section 2.6.

Une fois que l'on sait combien de temps chaque partie de notre code prend, il est aisé de comparer deux implémentations différentes. Mais cela ne nous dit rien sur l'efficacité absolue de notre implémentation. Dans le résumé de notre premier article [204], nous indiquons que notre implémentation traite 65 millions de particules par seconde par cœur sur un processeur Intel Haswell (en n'utilisant qu'un seul fil d'exécution par cœur et pas deux comme le permettrait la technologie *hyper-threading* d'Intel). Comment savoir si cette performance est bonne ? Plus tard dans cet article, nous voyons qu'il y a un "saut" en efficacité quand on passe de 4 à 8 cœurs [204, Figure 7], voir la Figure C.5. Comment savoir si ce saut est une mauvaise chose ? Rentrons dans plus de détails pour le comprendre.

Pour extraire de l'information des résultats de performance, et avoir une intuition de ce qui devrait être un résultat bon ou mauvais, nous avons besoin d'avoir au moins des notions de base en architecture des ordinateurs, et de connaître quelques propriétés de notre implémentation. Nous avons pu éviter ces aspects techniques lorsque nous avons découvert les bases du parallélisme, mais maintenant que cette introduction touche à sa fin, nous devons les aborder.

À chaque fois qu'un calcul est effectué (par exemple, `C[i] = A[i] + B[i]`, ligne 2 du Listing C.1), une unité de calcul va se charger du calcul. Avant que le calcul ne puisse être effectué, les données (ici, `A[i]` et `B[i]`) doivent être chargées depuis la mémoire principale. Après le calcul, le résultat (ici, `C[i]`) doit être écrit dans la mémoire principale. Dans certains cas c'est un peu différent, mais si l'on effectue cette boucle sur de gros tableaux, c'est une bonne approximation de ce qui se passe. Il y a donc deux propriétés architecturales qui vont contribuer à l'efficacité de cette boucle :

- À quelle vitesse peut-on effectuer les calculs ? C'est la *fréquence* de notre processeur.
- À quelle vitesse peut-on accéder aux données ? C'est la *bande passante* de notre processeur.

Il y a un modèle simple mais très utile qui va nous permettre de savoir ce que l'on peut attendre d'une implémentation donnée, en connaissant ces paramètres architecturaux : le modèle

"ligne-toit" (*roofline*) [180]. Quand nous avons beaucoup d'accès mémoire et peu d'opérations, nous sommes limités par la bande passante. Nous sommes dans un cas où nous sommes *limités par la mémoire*. C'est le cas pour un algorithme PIC (nous avons $\Theta(N)$ données à lire et à écrire, et $\Theta(N)$ opérations à effectuer sur ces données). Quand on n'a pas beaucoup de mémoire à charger mais beaucoup d'opérations à effectuer, nous allons être limités par la performance des opérations en virgule flottante. Nous sommes dans un cas où nous sommes *limités par le calcul*. C'est le cas par exemple pour les multiplication de matrices denses (nous avons $\Theta(N^2)$ données à lire et à écrire, et $\Theta(N^3)$ opérations à effectuer sur ces données[13]). Le paramètre que nous devons regarder est donc *l'intensité opérationnelle* de notre implémentation. Combien d'opérations en virgule flottante avons-nous par octet que nous devons charger depuis la mémoire ou écrire en mémoire ? Dans le Listing C.1, si les tableaux contiennent des `doubles` (un `double` nécessite 8 octets pour être stocké), nous avons une opération pour 24 octets déplacés. Cette intensité opérationnelle est extrêmement faible sur les architectures modernes, et notre boucle sera donc limitée par la mémoire. Cette boucle sert en fait d'étalon (*benchmark*) pour tester la bande passante maximum qui peut être atteinte en pratique : l'étalon Stream [162]. En général, le pic atteignable en pratique est plus bas que le pic théorique.

Quelques paragraphes plus tôt, nous voulions savoir si 65 millions de particules par seconde était un bon résultat sur 1 cœur. Nous pouvons maintenant dessiner le modèle ligne-toit pour notre architecture et notre implémentation, voir la Figure C.4. Notre architecture peut atteindre 68 Go/s pour 4 canaux mémoire[14], donc 17 Go/s pour 1 cœur. La fréquence est de 2.3 GHz et le nombre maximum d'opérations en simple précision par cycle est de 32[15], donc le nombre maximum d'opérations par seconde est de 73.6 GFlops/s. Notre implémentation 2d atteint 62 opérations par particule, et nécessite $\approx$ 103 octets déplacés par particule. Cela nous donne une intensité opérationnelle de $\approx$ 0.60. Pour cette intensité, aucune implémentation ne peut dépasser 8.3 GFlops/s, donc notre implémentation qui atteint 4.5 GFlops/s est plutôt bonne. Remarquons néanmoins que ce graphique n'est pas suffisant pour analyser les performances d'une implémentation donnée. Même si l'on atteignait la ligne de l'étalon Stream, rien ne nous assure que la performance de l'implémentation ne peut pas être encore améliorée. Nous avons par la suite mis au point des algorithmes qui nécessitent moins de transferts mémoire. Ainsi, l'intensité opérationnelle augmente, et la performance maximum atteignable augmente elle aussi.

Ce modèle nous dit quelles performances peuvent être atteintes lorsque l'on a des propriétés fixées pour notre architecture et notre implémentation. Il reste une dernière étape à comprendre lorsque l'on étudie le passage à l'échelle d'une implémentation sur un processeur. Sur les architectures modernes, il y a plus de cœurs que de canaux mémoire. Quelques paragraphes plus tôt, nous voulions savoir s'il était attendu ou non d'avoir un passage à l'échelle "non parfait" pour notre implémentation, entre 4 et 8 cœurs. Notre architecture a 8 cœurs mais seulement 4 canaux mémoire. En utilisant 1 cœur, ce cœur peut utiliser un canal. Avec 2 ou 4 cœurs, chaque cœur peut utiliser son propre canal. Mais dès que l'on atteint 8 cœurs, les cœurs doivent partager les canaux mémoire. Parce que notre implémentation est bornée par la mémoire, ce n'est donc pas surprenant d'avoir ce genre de comportement, voir le passage à l'échelle faible (la taille du problème par processeur est constante, donc la taille totale du problème augmente avec le nombre de processeurs) de la Figure C.5.

Comparer les efficacités de différentes implémentations est une tâche ardue. La comparaison directe entre les résultats de performance de deux articles est difficile à mettre en œuvre, pour de multiples raisons :

[13] Des algorithmes de multiplications de matrices existent avec une complexité moindre, par exemple, l'algorithme de Strassen en $\Theta(N^{log_2(7)})$ [174] ou plus récemment $\Theta(N^{2.3728639})$ [158], mais pour des valeurs de N qui peuvent être traitées par un ordinateur classique, utiliser des variantes optimisées de l'algorithme naïf en $\Theta(N^3)$ aboutit aux meilleurs temps de calcul.

[14] http://ark.intel.com/products/81705

[15] https://en.wikipedia.org/wiki/FLOPS

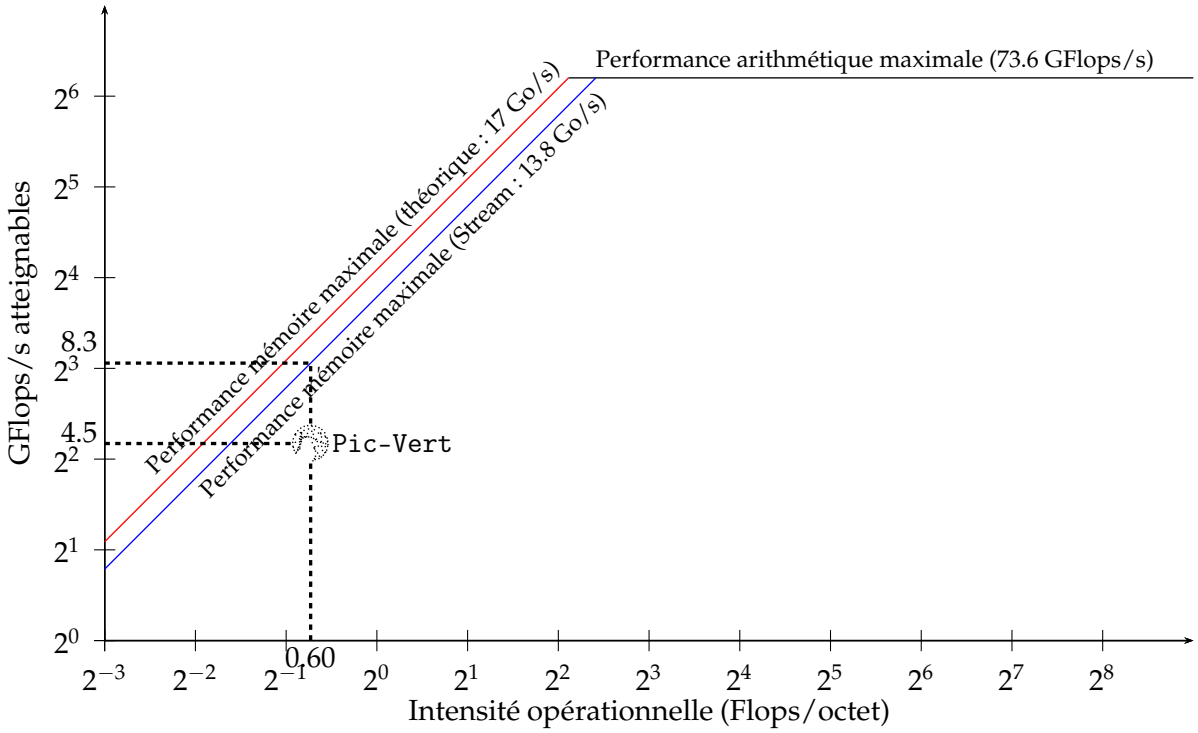


FIGURE C.4 – Modèle ligne-toit pour notre implémentation 2d sur 1 cœur Intel Haswell.

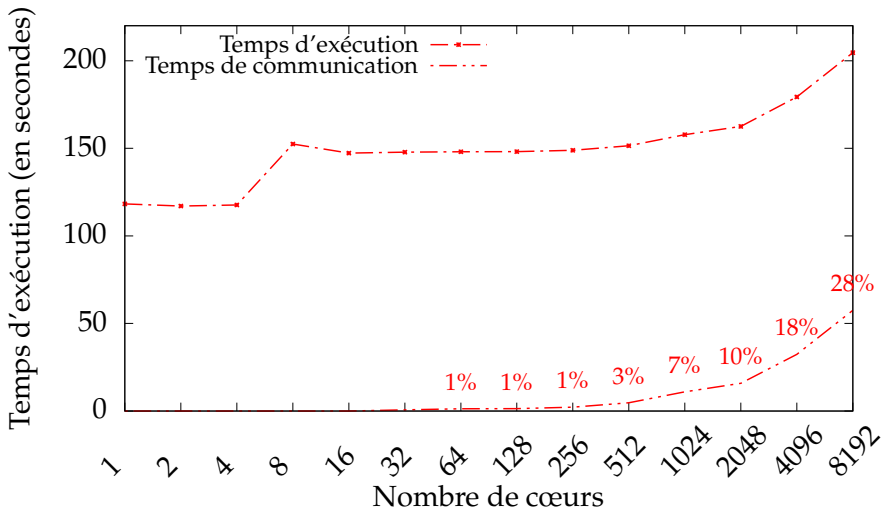


FIGURE C.5 – Passage à l'échelle faible sur le supercalculateur Curie (Intel Sandy Bridge) pour notre implémentation 2d. Cas test : grille de taille 128×128, 50 millions de particules par cœur, 100 itérations (tri toutes les 50 itérations). Le temps de communication est également donné en pourcentage du temps d'exécution.

- Les paramètres architecturaux sont très souvent différents dans les articles. Nous devons donc d'abord normaliser les résultats en fonction des paramètres architecturaux. Dans cette thèse, nous proposons une normalisation qui est utile pour les implémentations PIC ainsi que pour n'importe quel type d'implémentation limitée par la mémoire : nous normalisons les résultats en fonction de la bande passante mémoire maximale théorique des différentes architectures. Cela donne un premier aperçu du comportement d'une implémentation donnée par rapport à une autre, même si la bande passante mémoire maximale n'est pas le seul paramètre à prendre en compte.
- Les détails d'implémentation varient probablement (pour une implémentation PIC cela peut être les équations, les conditions initiales, la précision des calculs, les ordres d'interpolation et de discrétisation en temps...).

Pour vraiment comparer deux implémentations, nous devrions donc les exécuter sur la même architecture, avec les mêmes paramètres. C'est habituellement impossible parce que nous n'avons que très rarement accès au code source des autres articles, et même quand nous y avons accès, l'exécuter sur une architecture donnée pourrait ne pas lui rendre justice, puisque cette implémentation pourrait bien ne pas avoir été optimisée pour cette architecture cible. Les deux équipes qui ont écrit les deux codes devraient donc coopérer sensiblement, juste pour pouvoir comparer les performances de leurs deux implémentations. Les équipes n'ont pas autant de temps à consacrer à cela.

Annexe D

Contributions (en français)

D.1 Description

Dans cette thèse, nous nous intéressons à la résolution du système d'équations Vlasov–Poisson, utile dans le domaine de la physique des plasmas. Notre objectif principal est l'efficacité des implémentations qui résolvent ces équations, sur architectures multi-cœurs. Bien que l'application principale de notre implémentation soit la simulation de plasmas, il est également possible de modifier légèrement l'implémentation pour des simulations dans d'autres contextes, par exemple en astrophysique.

La contribution principale de notre thèse est un logiciel écrit dans le langage `C`, dont le nom est `Pic-Vert`. Il s'agit d'une implémentation de la méthode particulaire (*Particle-in-Cell*) pour la physique des plasmas. Trois étapes importantes sont indispensables pour une telle implémentation, étant donné son contexte pluridisciplinaire :

- (informatique) elle doit être efficace. La performance d'une telle implémentation est limitée par la bande passante de l'architecture considérée, ce qui revient à dire que les demandes en données pour résoudre le système numérique sont bien supérieures, proportionnellement aux ressources disponibles sur les ordinateurs modernes, aux demandes en calcul. Cette limitation peut être vérifiée dans le modèle "ligne-toit" (*roofline*). L'utilisation de la bande passante est donc une bonne métrique pour s'assurer de l'efficacité d'une implémentation donnée.

- (mathématiques) elle doit être vérifiée. Il existe dans la littérature de nombreux cas tests dont on connaît des solutions théoriques presque exactes. Il faut donc vérifier que l'implémentation se comporte correctement sur ces cas tests. De plus, pour d'autres cas tests, il existe des diagnostics pour s'assurer que les résultats sont cohérents, comme la conservation de l'énergie totale.

- (physique) elle doit être validée. Le modèle physique utilisé dans une implémentation a toujours des limites. Il faut donc valider le modèle choisi en comparant, quand cela est possible, les résultats de la simulation à des résultats tirés d'expériences physiques.

Pour s'assurer de la performance, nous proposons une implémentation qui (a) atteint un nombre quasi-minimal de transferts mémoires avec la mémoire principale, (b) exploite les instructions vectorielles (*SIMD*) pour les calculs numériques, et (c) expose une quantité suffisante de parallélisme pour occuper tous les cœurs d'un processeur moderne, en mémoire partagée. En plus de ces propriétés théoriques, nous montrons dans notre thèse des mesures de bande passante mémoire de notre implémentation en pratique. Nous avons en plus implémenté une parallélisation pour la mémoire distribuée, mais ce niveau de parallélisme n'est pas le cœur de notre travail. Pour mettre notre travail en perspective avec l'état de l'art, nous avons mis au point une nouvelle métrique permettant de comparer différentes implémentations de cette

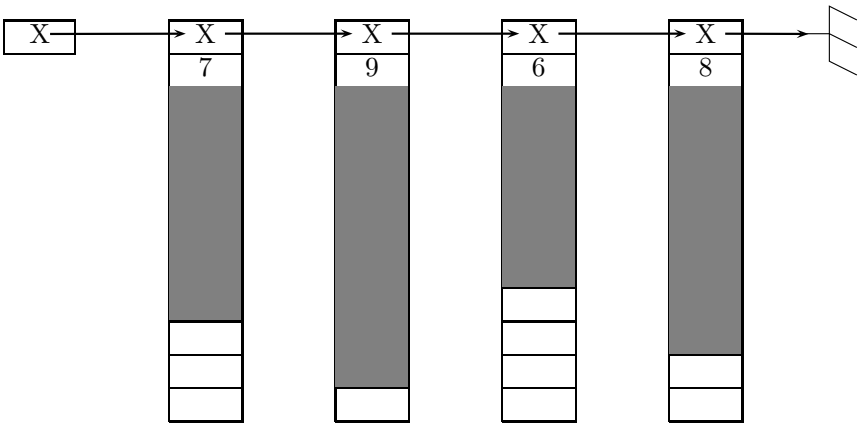


FIGURE D.1 – Structure de données en liste chaînée de tableaux de taille fixe : tableaux de taille 10, les particules sont stockées dans les cases grisées.

méthode, sur différentes architectures multi-cœurs. Nous donnons un aperçu de cette comparaison dans la suite de ce résumé.

Pour la vérification, nous simulons des cas tests classiques d'amortissement Landau, en deux et trois dimensions. Nous simulons également un nouveau cas test qui a été conçu par nos co-auteurs. La correction de notre implémentation sur ces cas tests est possible en comparant l'énergie électrique simulée à sa valeur théorique obtenue à partir d'une analyse de dispersion.

Pour la validation, nous simulons un cas test de trou d'électrons en deux dimensions spatiales et trois dimensions de vitesses. Notre implémentation reproduit les résultats d'un article présentant ce cas test, qui a été validé grâce à des mesures satellites de trous d'électrons dans différentes parties de la magnétosphère.

D.2 Organisation du manuscrit

Dans notre manuscrit, nous évoquons d'abord le contexte général de notre thèse et expliquons la méthode particulaire dans les deux premiers chapitres. Ensuite, nous expliquons nos différentes contributions à l'état de l'art.

Deux chapitres sont dédiés aux caractéristiques principales de `Pic-Vert`.

L'un de ces chapitres se concentre sur l'optimisation d'un algorithme relativement standard pour la méthode particulaire, où l'on utilise des organisations mémoires classiques pour représenter les particules, que l'on trie de manière périodique en temps pour réduire les défauts de cache. Plusieurs optimisations présentées dans ce chapitre sont classiques et d'autres sont, à notre connaissance, nouvelles. Dans nos tests en deux dimensions, cet algorithme donne lieu à l'implémentation la plus efficace, ce que l'on montre dans le dernier chapitre.

Le second de ces chapitres présente une heureuse combinaison entre les listes chaînées de tableaux de taille fixe, voir la Figure D.1, et la méthode particulaire. Les algorithmes présentés dans cette section utilisent en partie des optimisations du chapitre précédent, et aboutissent, dans nos tests en trois dimensions, à l'implémentation la plus efficace, ce que l'on montre également dans le dernier chapitre.

Un chapitre est ensuite dédié à l'implémentation d'une méthode semi-Lagrangienne, en deux dimensions, en utilisant la décomposition de domaine. Ce chapitre est indépendant des autres, à part si l'on prend en considération le chapitre introductif. Les structures de données pour cette méthode sont entièrement différentes de celles nécessaires pour la méthode particulaire, et le fait que l'on utilise la décomposition de domaine aboutit à un comportement radicalement différent en terme de mémoire distribuée. Nous présentons un algorithme qui s'affranchit des limitations trouvées dans l'état de l'art, et proposons des directions futures exploitables pour cette implémentation.

Un chapitre est ensuite dédié à la vérification et la validation de notre implémentation. Il décrit dans un second temps comment utiliser, dans une même implémentation, à la fois la méthode particulaire et la méthode semi-Lagrangienne.

Enfin, un dernier chapitre vient conclure cette thèse et présenter des travaux futurs.

D.3 Publications

Cette thèse se base sur les publications suivantes :

[i] Y. Barsamian, S. A. Hirstoaga, and É. Violard. "Efficient Data Structures for a Hybrid Parallel and Vectorized Particle-in-Cell Code". Dans : *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, 2017, pp. 1168–1177.
DOI : 10.1109/IPDPSW.2017.74
Transparents : http://www.barsamian.am/Slides/slides_2017-06-02.pdf.

[ii] Y. Barsamian, A. Charguéraud, and A. Ketterlin. "A Space and Bandwidth Efficient Multicore Algorithm for the Particle-in-Cell Method". Dans : *Parallel Processing and Applied Mathematics : 12th International Conference (PPAM)*. vol. 10777. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 133–144.
DOI : 10.1007/978-3-319-78024-5_13
Transparents : http://www.barsamian.am/Slides/slides_2017-09-11.pdf.

[iii] Y. Barsamian, S. A. Hirstoaga, and É. Violard. "Efficient Data Layouts for a Three-Dimensional Electrostatic Particle-in-Cell Code". Dans : *Journal of Computational Science* 27 (2018), pp. 345–356.
DOI : 10.1016/j.jocs.2018.06.004.

[iv] Y. Barsamian, J. Bernier, S. A. Hirstoaga, and M. Mehrenberger. "Verification of $2D \times 2D$ and two-species Vlasov–Poisson solvers". Dans : *ESAIM : Proceedings and Surveys* 63 (2018), pp. 78–108.
DOI : 10.1051/proc/201863078. Transparents : http://www.barsamian.am/Slides/slides_2016-08-25.pdf.

[v] Y. Barsamian, A. Charguéraud, S. A. Hirstoaga, and M. Mehrenberger. "Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors". Dans : *24th International Conference on Parallel and Distributed Computing (Euro-Par)*. Vol. 11014. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 749–763.
DOI : 10.1007/978-3-319-78024-5_33
Transparents : http://www.barsamian.am/Slides/slides_2018-08-30.pdf.

Cette publication est couplée aux fichiers suivants, permettant de reproduire nos résultats, qui ont reçu la distinction de "Meilleurs Artefacts" à la conférence Euro-Par 2018 :

[vi] Y. Barsamian, A. Charguéraud, S. A. Hirstoaga, and M. Mehrenberger. *Software artifacts for Euro-Par 2018 paper : "Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors"*. Figshare. 2018.
URL : https://doi.org/10.6084/m9.figshare.6391796.

En plus de ces publications, le chapitre sur la méthode semi-Lagrangienne est basé sur les travaux suivants, présentés à l'oral mais non encore publiés dans un article :

[vii] Y. Barsamian and M. Mehrenberger. "Semi-Lagrangian Simulations for Solving 2d2v Vlasov–Poisson Systems (one and two species)". Dans : *Platform for Advanced Scientific Computing (PASC), Minisymposium "Kinetic Simulations on HPC Platforms for Plasma Physics Applications (3/3) : Parallelization and New Hardware"*. 2017.
Transparents : http://www.barsamian.am/Slides/slides_2017-06-27.pdf.

Yann BARSAMIAN

Pic-Vert : Une implémentation de la méthode particulaire pour architectures multi-cœurs

Résumé

Cette thèse a pour contexte la résolution numérique du système de Vlasov–Poisson (modèle utilisé en physique des plasmas, par exemple dans le cadre du projet ITER) par les méthodes classiques particulaires (PIC pour "*Particle-in-Cell*") et semi-Lagrangiennes.

La contribution principale de notre thèse est une implémentation efficace de la méthode PIC pour architectures multi-cœurs, écrite dans le langage C, dont le nom est Pic-Vert. Notre implémentation (a) atteint un nombre quasi-minimal de transferts mémoires avec la mémoire principale, (b) exploite les instructions vectorielles (SIMD) pour les calculs numériques, et (c) expose une quantité suffisante de parallélisme, en mémoire partagée.

Pour mettre notre travail en perspective avec l'état de l'art, nous proposons une métrique permettant de comparer différentes implémentations sur différentes architectures. Notre implémentation est 3 fois plus rapide que d'autres implémentations récentes sur la même architecture (Intel Haswell).

Mots-clefs : Informatique · Parallélisme · Méthode particulaire · Méthode semi-Lagrangienne · Physique des plasmas · Multi-cœurs · Architecture SIMD · Mémoire partagée

Résumé en anglais

In this thesis, we are interested in solving the Vlasov–Poisson system of equations (useful in the domain of plasma physics, for example within the ITER project), thanks to classical Particle-in-Cell (PIC) and semi-Lagrangian methods.

The main contribution of our thesis is an efficient implementation of the PIC method on multi-core architectures, written in C, called Pic-Vert. Our implementation (a) achieves close-to-minimal number of memory transfers with the main memory, (b) exploits SIMD instructions for numerical computations, and (c) exhibits a high degree of shared memory parallelism.

To put our work in perspective with respect to the state-of-the-art, we propose a metric to compare the efficiency of different PIC implementations when using different multi-core architectures. Our implementation is 3 times faster than other recent implementations on the same architecture (Intel Haswell).

Keywords : Computer science · Parallelism · Particle-in-cell · Semi-Lagrangian · Plasma physics · Multi-core · SIMD architecture · Shared memory