Parallel algorithms for tracking of particles

Florent Bonnier

université PARIS-SACLAY

NNT : 2018SACLV080

Thèse de doctorat

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

Algorithmes Parallèles pour le Suivi de Particules.

Thèse de doctorat de l'Université Paris-Saclay
préparée à Université de Versailles-Saint-Quentin-en-Yvelines

Ecole doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Gif-sur-Yvette, le 12 Décembre 2018, par

Florent Bonnier

Composition du Jury :

Jean-Marc Delosme Professeur, Université d'Evry-Val d'Essonne (Laboratoire IBISC)	Président de jury
Sabine Roller Professeur, Université de Siegen (Zentrum für Informations- und Medientechnologie)	Rapporteur
Osni Marques Staff Scientist, Lawrence Berkeley National Laboratory(Computational Research Division)	Rapporteur
François-Xavier Roux Ingénieur de recherche, ONERA (MACI)	Examinateur
Nahid Emad Professeur, UVSQ/MDLS (Li-Parad)	Directrice de thèse
Xavier Juvigny Ingégnieur de recherche, ONERA (DAAA)	Co-directeur

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Algorithmes Parallèles pour le Suivi de Particules.

Mots clés: suivi de particules, Lagrangien, HPC, architecture par composants, calcul parallèle

Résumé:
Les méthodes de suivi de particules sont couramment utilisées en mécanique des fluides de par leur propriété unique de reconstruire de longues trajectoires avec une haute résolution spatiale et temporelle. De fait, de nombreuses applications industrielles mettant en jeu des écoulements gaz-particules, comme les turbines aéronautiques utilisent un formalisme Euler-Lagrange. L'augmentation rapide de la puissance de calcul des machines massivement parallèles et l'arrivée des machines atteignant le petaflops ouvrent une nouvelle voie pour des simulations qui étaient prohibitives il y a encore une décennie. La mise en œuvre d'un code parallèle efficace pour maintenir une bonne performance sur un grand nombre de processeurs devra être étudié. On s'attachera en particuliers à conserver un bon équilibre des charges sur les processeurs. De plus, une attention particulière aux structures de données devra être fait afin de conserver une certaine simplicité et la portabilité et l'adaptabilité du code pour différentes architectures et différents problèmes utilisant une approche Lagrangienne. Ainsi, certains algorithmes sont à repenser pour tenir compte de ces contraintes. La puissance de calcul permettant de résoudre ces problèmes est offerte par des nouvelles architectures distribuées avec un nombre important de cœurs. Cependant, l'exploitation efficace de ces architectures est une tâche très délicate nécessitant une maîtrise des architectures ciblées, des modèles de programmation associés et des applications visées. La complexité de ces nouvelles générations des architectures distribuées est essentiellement due à un très grand nombre de nœuds multi-cœurs. Ces nœuds ou une partie d'entre eux peuvent être hétérogènes et parfois distants. L'approche de la plupart des bibliothèques parallèles (PBLAS, ScalAPACK, P_ARPACK) consiste à mettre en œuvre la version distribuée de ses opérations de base, ce qui signifie que les sous-programmes de ces bibliothèques ne peuvent pas adapter leurs comportements aux types de données. Ces sous programmes doivent être définis une fois pour l'utilisation dans le cas séquentiel et une autre fois pour le cas parallèle. L'approche par composants permet la modularité et l'extensibilité de certaines bibliothèques numériques (comme par exemple PETSc) tout en offrant la réutilisation de code séquentiel et parallèle. Cette approche récente pour modéliser des bibliothèques numériques séquentielles/parallèles est très prometteuse grâce à ses possibilités de réutilisation et son moindre coût de maintenance. Dans les applications industrielles, le besoin de l'emploi des techniques du génie logiciel pour le calcul scientifique dont la réutilisabilité est un des éléments des plus importants, est de plus en plus mis en évidence. Cependant, ces techniques ne sont pas encore maÃőtrisées et les modèles ne sont pas encore bien définis. La recherche de méthodologies afin de concevoir et réaliser des bibliothèques réutilisables est motivée, entre autres, par les besoins du monde industriel dans ce domaine. L'objectif principal de ce projet de thèse est de définir des stratégies de conception d'une bibliothèque numérique parallèle pour le suivi lagrangien en utilisant une approche par composants. Ces stratégies devront permettre la réutilisation du code séquentiel dans les versions parallèles tout en permettant l'optimisation des performances. L'étude devra être basée sur une séparation entre le flux de contrôle et la gestion des flux de données. Elle devra s'étendre aux modèles de parallélisme permettant l'exploitation d'un grand nombre de cœurs en mémoire partagée et distribuée.

Title: Parallel Algorithms for Particle Tracking.

Keywords: particle tracking, Lagrangian, HPC, Component based software, parallel computation.

Abstract: Particle tracking is often used in the domain of fluid dynamics because it ables the reconstruction of long trajectories with high spatial and temporal accuracy. Thus, lots of applications in the industry related to gas-particles as in aeronautic engines, use the Euler-Lagrange method. The increase of the computation power of high massively parallel machines and the arrival of petaflop systems begin a new approach for simulations that were prohibited a decade ago. The implementation of an efficient parallel code to keep high performances on a large number of processors must be studied. One especially tries to keep a good work balancing on processes. In addition, a special attention must be paid to data structures in order to keep a kind of simplicity, to keep the code portable and adaptive to multiple architectures and multiple problems using a Lagrangian method. Thus, some algorithms have to be thought again in order to respect these constraints.

The computational power capabled to solve such of these problems is given by modern distributed architectures with a large number of cores. However, exploiting these machines is difficult task that needs a lot of experience on the targeted architecture and associated programming models and adapted applications. The complexity of these new generations of distributed architectures is essencially due to a high number of multi-core nodes. Most of the nodes can be heterogeneous and sometimes remote. Today, nor the high number of nodes, nor the processes that compose the nodes are exploited by most of applications and numerical libraries. The approach of most of parallel libraries (PBLAS, ScalAPACK, P_ARPACK) consists in implementing the distributed version of its base operations, which means that the subroutines of these libraries can not adapt their behaviors to the data types. These subroutines must be defined once for use in the sequential case and again for the parallel case. The object-oriented approach allows the modularity and scalability of some digital libraries (such as PETSc) and the reusability of sequential and parallel code. This modern approach to modelize sequential/parallel libraries is very promising because of its reusability and low maintenance cost. In industrial applications, the need for the use of software engineering techniques for scientific computation, whose reusability is one of the most important elements, is increasingly highlighted. However, these techniques are not yet well defined. The search for methodologies for designing and producing reusable libraries is motivated by the needs of the industries in this field. The main objective of this thesis is to define strategies for designing a parallel library for Lagrangian particle tracking using a component approach. These strategies should allow the reuse of the sequential code in the parallel versions while allowing the optimization of the performances. The study should be based on a separation between the control flow and the data flow management. It should extend to models of parallelism allowing the exploitation of a large number of cores in shared and distributed memory.

"Wrong does not cease to be wrong because the majority share in it."

– Leo Tolstoy

"Vous ne serez jamais, et dans aucune circonstance, tout à fait malheureux si vous êtes bon envers les animaux."

– Victor Hugo

Acknowledgment

This study has become possible thanks to many people starting with Nahid Emad and Xavier Juvigny who were both extraordinarily patient and listening supervisors. I also thank Jean-Marc Delosme, Osni Marques and Sabine Roller who accepted to be part of the jury and showed some interest in the subject. I want to thank the ONERA laboratory and more precisly the DTIS-MACI team that proposed this thesis. Franǫis-Xavier Roux, Juliette Ryan and Ludomir Oteski were very good teamates and always proposed proper and smart solutions and ideas. Eric Quemerais and Alain Refloch who were supervisors of my master internship, nice colleagues and made me like the french space laboratory.
The environment of the ONERA laboratory was a very enthousiastic place to work in, the happiness of PhD students and other colleagues give the laboratory a unique charm.

I also wish to thank my whole family, thinking about my grand father who left us before the defense of this thesis. This family that was always there to support me at any moment regardless of my mood.

This thesis was emotionnally strong and I want to apologize to my friends. They were very patient too, supporting the difficulties I encountered and trying to help me as they can. I apologize have been so distant and sometimes incomprehensible. I encountered some new friends thanks again to the french laboratory where I met people with stong passions and nice hobbies. Discussions about History, ancient ages in front of a wood fire or video games and fiction in front of a board game was a real pleasure.

I thank the different clubs I have encountered that made me discover historical reenactment, free fights and historical combat. These activities made me feel stronger and confident. They made me discover multiple aspects from daily survival camping to the study of archeological pieces. These external activities made me think differently about the problem exposed in this thesis by meeting other scientists in different fields.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER

1

Context of the Study.

1.1 Introduction

In the simulation field, particle tracking is used to solve many kind of simulations. Simulated particle tracking is used by many scientists in multiple domains from aerospace to medical sciences. This operation consists in computing the movements and the positions of a set of corpuscles or droplets in the simulated environment. More precisely, it consists in observing the evolution of a large number of simulated objects in time and computing interactions the scientists want to simulate. Especially, particle tracking is used to simulate chemical reactions in a fluid like spray of diesel drops [1], solving gas-liquid interactions [2, 3] or to follow sand transport in a fluid [4, 5]. In another field, particle tracking is also used to draw streamlines for velocity field visualization [6].

Partial differential equation based problems are often solved using finite volume and finite element methods. This is concretely solved using meshes, the environment is discretized in smaller areas where solution is known in order to approximate the solutions of the equation. The finer the mesh, the more the solution is precise and close to the real solution. Particle movement can be simulated in a similar discretized environment and this study focuses on these cases.

People use several approaches depending on the targeted numerical precision, the machine's performance and compute capability.

The growth of size and complexity of the involved equations forced the scientists to search for more efficient computing systems. The need for parallelization and high performance computing

became obvious. Apart from the fact that simulations and meshes are more and more greedy in computation, they also occupy more memory space and more energy. The main reason for this is the need for more accurate simulations and computations in order to answer today's challenges. Today's machines and high performance computers try to combine performances, precision and also money saving as massively parallel machines cost a lot to maintain and are greedy in energy. The popularity of contests and score tables such as Green500 [7] or the popularity of workshops such as ENAHPC [8] proudly shows that scientists are aware and care about the actual climate change and the global warming trying to save energy consumption. The energy consumption is directly linked to the time spent to solve a problem. A simple way to reduce energy consumption of a system is to reduce computation time. The clock speed of single CPUs ceased to grow since several years and forces computer scientists to find new architectures to continue to increase the computation capabilities of computers by enabling MIMD and MPMD paradigms. Today, we can observe that CPUs use smaller clock rates and try to consume less power with highly parallel methods [9]. The consequences of all these parameters are the important growth of accelerators and GPUs usage in HPC, the use of multiple smaller clock rates together rather than a single high clock rate CPU and the study of new approaches of parallel algorithms [10, 11, 12, 13, 14] and software/hardware architectures [15].

The simulation field is not spared by these modern challenges as the need of accuracy always grows. Thus the parallelization of methods and the distribution of data on connected hardware become more and more important. Today meshes used in simulations are partitioned, distributed on several computing nodes and the equations are solved in parallel.
The particle tracking problem is no exception to this observation. In fact particle tracking can be parallelized as well as the velocity field and because particles are considered as independent in lagrangian formulation, particle load can be balanced as well as velocity fields using well known partitioning and domain decomposition methods [16, 17, 18] and distribution algorithms [19]. Some of these techniques work very well with limited number of processes or on specific kind of architectures which render today's hybrid and heterogeneous HPC machines hard to use. The challenge brought by particle tracking is the efficiency of the problem integration to the already parallelized problem represented by the meshed and distributed velocity field. To be more concrete, the problem lies in the fact that the mesh and particles can be parallelized with different methods that give two different distribution results with high consequences in terms of data proximity and relevance.

1.2 Problem to solve

The problem of particle tracking is an old challenge, many scientists use particles to simulates multiple granular and fluid behaviors. The fact is that recording particle paths requires a lot of computation and data access and the more particles there are, the more accurate is the simulation.
This makes the problem of particle tracking a good candidate for high performance computation. As particles may be followed independently, the paths computation is reduced to a SIMD (**S**ingle

Instruction **M**ultiple **D**ata) problem. It means that the parallelization capacity of the simulation highly depends on the parallelization and the distribution of the data.
As many scientists need the computation of particle tracking with high accuracy and high performances, there are many studies, algorithm, methods and techniques to control particle paths in a parallel context. Some of them perfectly distribute mesh and particle data in order to try to reduce the memory occupancy per node and to improve work balance [20], whereas others prefer to reduce the communications between computing nodes and do not take care about memory usage and workload [2, 21]. Most of these techniques try to balance both these points of view by communicating data packages reducing communications for some time steps.

The main goal of this thesis is to propose an efficient solution unifying these approaches to treat a very large number of particles on massively parallel and high performance systems. These approaches and strategies are adapted in order to be efficient on heterogeneous parallel systems. As the Lagrangian method allows the particles to be computed indepedently to the mesh and to other particles, a library based on this method higly depends on the algorithms used to track a single particle. Such a library can be adapted to modern architectures using full available computational power from GPUs, accelerators and dispersed NUMA nodes.
This goal is the first one. The second objective concerns the library's architecture. In order to be adapted and efficient on parallel machines and especially on higly heterogeneous systems, the implemented algorithms and methods must be independent as possible and such as the data used. To do so, the library is based on independent components, easily maintainable and more adapted to multiple architectures. In addition, the Lagrangian method implies independent particles which means that a set of instructions applied on a single particle is adapted to a set of particles. So these intructions and operations are almost ready to be reusable in sequential and parallel systems because the parallelization of lagrangian particle tracking lies in the particle distribution.
To summarize, the two objectives are: implement, optimize and adapt efficient algorithm to massively parallel systems to solve large number of particles, and develop a particle tracking library based on component-based architecture to improve maintenance, sequential-parallel reuse and adaptation to modern architectures, used in petascale computers and expected in exascale machines. The final library is called **ParOPTIC** for **Par**allel **O**bject **P**article **T**racking **I**nteroperable **C**omponent.

1.3 Plan of the study

The goal of this study is to design an efficient library gathering methods to track and record particle paths and study their interactions with the environment. These methods are efficient in a parallel system and must scale as much as possible on massively parallel machines close to exaflop machines.

This thesis consists in 8 chapters. Chapter 2 describes the methods used to traditionally track particles on parallel systems. These methods are used for several objectives from particle localization to memory management. Chapter 3 consists in the description of the particle tracking operation and the software architecture we have chosen to implement particle tracking and affiliated methods. This

chapter is the first key to determine the dependencies between internal and data conflicts possibly involved in parallel particle tracking. Chapter 4 describes the multiple algorithms we have chosen and the specificity of their implementation in order to record simulated particle tracking efficiently with a limited memory occupancy and execution time. This fourth chapter presents algorithms used on local data in order to compute particle tracking in a sequential run. Chapter 5 describes the implementation and the adaptation of the algorithms presented in chapter 4 to a parallel context. Chapter 6 presents a method to distribute tasks and in our case particles on massively parallel systems in order to balance the workload. This method is implemented, adapted and studied. Chapter 7 is about the library adaptation, and in a more general case, the adaptation of efficient algorithms to massively parallel machines. This chapter is the adaptation of particle tracking method to scale at high parallelism level but the used approach is relevant for many algorithms and methods. The last chapter 8 before the final conclusion is the experimentation and the study of the efficiency of the library on several generic cases.

Chapters 4, 5, 6 and 7 describe methods, discuss about the advantages of these methods and the encountered problems with a study of the efficiency brought by the various methods run in a general case.

CHAPTER

2

The State of the Art

Contents

2.1 Fluid resolution

The particle tracking problem often appears in a two phase problem [22, 23, 24]. This kind of problem consists in solving a fluid phase (a gas, a flow, a velocity field) and a solid phase, the particle phase (granular flow, droplets, ...). The particles need the velocity field given by the fluid phase in order to compute the direction of these particles and their next position. So the first computation to do before any particle movement is the determination of the particle's velocity by solving the local fluid equations.

In a computational fluid dynamics problem, most of the time the Navier-Stokes equations are solved [25, 26]. In order to adapt these equations to a computational form, several discretization methods are used [27, 28, 29]. The choice of the discretization methods highly impacts the computation performances and the accuracy of the simulation. This is due to the quality of the resulting mesh. Indeed, the mesh has a real impact on the convergence of the solution, the accuracy and the CPU time. This is the reason why the geometry, the boundary treatment and the refinement of the grid are very important in the computation of a fluid dynamic problem.

A mesh is defined by a set of vertices, cells, edges and faces that describe an environment using geometrical objects. This mesh describes a set of areas where the solution is known or computed. The methods used to solve a fluid equation at a particular point depends on the way that the mesh describes the physics quantities. For example, figure 2.1 shows the fluid computation at a point in a cell using quantities stored in the vertices. This figure 2.1 shows the computation of the fluid

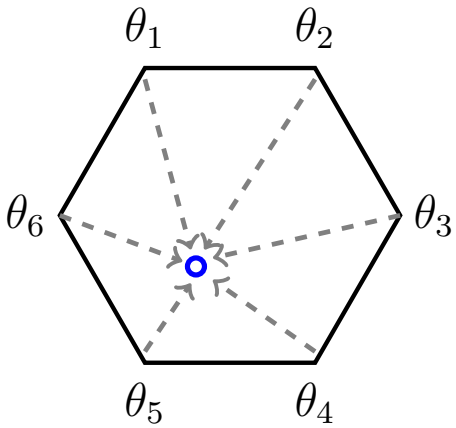


Figure 2.1: In-cell fluid computation using data on vertices.

velocity in the cell whereas the physics quantities (θ_1 to θ_6) are known at the vertices. Obviously the fluid data can be stored elsewhere than on vertices, depending on the mesh, the data can be stored on faces, edges, or cells. Beall and Shepard [30] gave a set of structures used to store and manage meshes. A way to compute the fluid vector using these quantities (whereas the quantities

inside the cell are not known because of the mesh discretization) is to compute the interpolation of these quantities and use this value to compute the fluid inside the cell.
The accuracy of a mesh and thus the accuracy of the fluid solution highly depends on the cells shape. The easiest manner is to use simple geometrical objects such as triangles, tetrahedron and prisms. There are many solvers that use simple objects [31, 32]. Especially with triangles and tetrahedron. The reason is the easier computations due to the simple shape of these objects and their properties. Indeed, a tetrahedron is always a convex object, with a static number of vertices, faces and edges and can be used to express very complex and curved areas. On the other hand, these objects are not well adapted to describe dead regions or less intensive regions. Some researches are focusing on hybrids grids that use multiple cell shapes regrouped in grid blocks [33].

In [34] the authors perform comparisons between two meshes of the same simulation. The difference between the two meshes is the shape of the cells, in one mesh, the cells are hexahedrons whereas in the other mesh, the cells are tetrahedrons. Both shapes are tested on the same two models which correspond to a NACA 0012 wing model [35] and a non-lifting rectangular helicopter rotor blade in the second case. According to the authors, the tetrahedral schemes lead to some limitations such as the loss of the mesh quality due to disparate cell sizes, large face angles and high vertex degrees. The goal of this paper was to investigate the transformations of tetrahedral scheme to hexahedral scheme in order to improve the quality of the mesh. The authors observed that hexahedral meshes utilize computer resources more efficiently than tetrahedral meshes for the same level of solution accuracy. They also have shown that hexahedral meshes have half the storage requirements and run almost twice as fast as tetrahedral meshes.
This paper has shown the importance of the cells shape and that tetrahedral meshes do not necessary go hand in hand with mesh quality and performances.

For hybrid meshes with both tetrahedrons and hexahedrons, Vinchurkar et al. found that hybrid meshes showed no improvement in performance [33].

The final conclusion for this section is that the shape used to describe the geometry has a high impact on the mesh convergence and on the computing performance. But as multiple shapes of cell can be implemented in hybrid meshes it is clear that the developed algorithms for particle tracking must be implemented for multiple and non uniform shapes.

2.2 Eulerian and Lagrangian methods to track particles

2.2.1 Eulerian method

The eulerian approach consists in considering the particle tracking as the resolution of a flow field. Particles are considered as a velocity field representing the particle's concentration. This concen-

tration is a quantity integrated to the fluid phase. The eulerian approach considers the particles and the disperse phase as a statistic resolution. Indeed, the particles do not literally move but a concentration is computed in function of the local quantities of the fluid phase [36].
Briefly, the particle concentration computation is integrated to the fluid resolution. For each cell of the mesh, the particle concentration is calculated and each time the fluid is solved, the new concentration is determined in each cell according to the flow velocity.

This approach is very interesting as particle tracking is directly computed and linked to the fluid phase. The equations involved are similar to those of the velocity and the concentration field so that the integration is easily done.
On the other hand, particles are not measured with accuracy as only statistics represented by their concentration are computed. Particles interactions with the environment become quite problematic to compute. In fact boundary conditions are quite difficult to detect accurately and in specific cases where particles are launched in multiple directions, the tracking could become difficult.
If we look at parallelization capability, the eulerian approach is quite easy to parallelize as the method to describe the particles distribution is the same as the fluid phase. But the eulerian formulation is better used in cases where the particles, or more specifically the particle's concentrations, are close. Particle dispersion means that more concentration computation are needed whereas if particles are very close and located in the same area, less concentration equation have to be solved.

2.2.2 Lagrangian method

The Lagrangian approach considers the particles as independent entities. These particles are tracked independently following the equations of motion:

$$\frac{d\boldsymbol{x_i}}{dt} = \boldsymbol{u_i}(t) \tag{2.1}$$

where $\boldsymbol{x_i}$ is the i^{th} particle's coordinate, $\boldsymbol{u_i}$ is the velocity of the i^{th} particle and t represent the time step [37]. To be more precise, $\boldsymbol{u_i}$ represents the velocity vector of the particle which is determined by the addition of all forces applied to the particle i.
This lagrangian approach is the simplest and the more accurate formulation of a particle evolution. Indeed the accuracy of the particle movement only depends on the accuracy of the applied forces and the chosen time scheme. In other words, the accuracy of the track depends on the precision of the flow field with this formulation and the accuracy parameters. Another advantage with this formulation is that as the particles are considered as independent, the particles can be sent, received and distributed easily in a parallel system. On the other hand, the particles integration to the flow field is more complicated as the distribution method and the equations are different. In fact an operation of particle's localisation is needed to determine the forces and the quantities applied to the particle. This operation is described later in this manuscript.
It is to be noticed that a difference can be made between a stochastic and a deterministic Lagrangian approach [38]. The stochastic approach consists in tracking a smaller number of Lagrangian particles

that are the average quantities of neighbouring particles. The deterministic approach computes the path of all numerical particles.

2.2.3 Methods comparison

Z. Zhang and Q. Chen have evaluated the differences between Eulerian and Lagrangian methods to predict particle transport in enclosed spaces [39]. This evaluation compares the predicting particle concentration distribution in two models: first under steady-state conditions, then under unsteady-state conditions. To compare both methods, the authors compute the particle concentration in the same enclosed space obtained with both methods and the experimental results. The authors concluded that both methods yielded similar results close to experimental data. The Eulerian point of view simulates the dispersion faster than the Lagrangian one and with a smoother concentration under steady-state conditions. On the other hand, the Eulerian method has an important computation time increase under unsteady-state conditions whereas the Lagrangian method does not. The increased computing of the Lagrangian effort mainly came from the calculation of unsteady state airflow and turbulence.
This study gives pros and cons of the Eulerian and Lagrangian method to track particle concentrations. The authors have shown that the Lagrangian point of view achieves better efficiency in terms of execution time and precision under unsteady-state conditions and that the Eulerian point of view is better under steady-state conditions. They also proved that the Lagrangian point of view is very easy to adapt under both studied conditions.

The most common approach in simulation field is the Eulerian-Lagrangian [2, 4, 40, 41, 42, 21] approach which corresponds to the application of the Eulerian point of view to the flow field whereas the particles are tracked with a Lagrangian point of view. This allows the particles to be followed independently of the flow field. This coupled method is very appreciated as it reflects perfectly the two phases of the studied particle tracking simulation, one gaz phase to simulate the fluid modeled by the velocity field and the solid phase that correspond to the set of traveling particles through the fluid.
This coupled approach is quite efficient but brings up some difficulties and challenges such as the integration of the solid phase into the flow phase. In other words, in this approach as the particles do not have any relation with the eulerian phase, they have to be localized in the Eulerian phase to compute particle's motions. On the other hand, this approach, and the more general approach that consists in computing independently multiple phases, has the advantage of having a high parallel capacity.
M. Garcia [21] regrouped in a table the advantages and disadvantages of particle tracking using an eulerian or lagrangian approach for the disperse phase and using in both cases an eulerian representation of the fluid.

Table 2.1 presents the advantages and disadvantages M. Garcia regrouped for two-phases simulations. The Lagrangian approach is particularly well adapted to accurate simulations as this approach allows to compute multiple paths of dispersed particles. The Eulerian approach on the other hand

	Euler-Euler	Euler-Lagrange
Advantages	• Numerically straightforward treatment of dense zones, • Similarity with gaseous equations, • Direct transport of Eulerian quantities, • Similarity with gaseous parallelism.	• Numerically straightforward modeling of particle movements and interactions, • Robust and accurate for large number of particles, • Size distributions easy to describe, • Numerically straightforward to implement physical phenomena (e.g., heat and mass transfer, wall-particle interaction).
Disadvantages	• Difficult description of polydispersion, • Difficulty of droplet crossing treatment, • Limitation of the method in very dilute zones.	• Delicate coupling with combustion • Difficult parallel implementation and integration • Time spent in locating particles on unstructured grids.

Table 2.1: Advantages and disadvantages of Eulerian method and Lagrangian method in the disperse phase using Eulerain approach for the liquid phase.

is adapted to observe more global phenomena, like concentration controls and particles moving in a single localization, in the same main direction. As M. Garcia noticed, the Lagrangian approach is better used in polydispersion simulation, with particles moving in different directions in the same area. This is the approach to implement in order to simulate radiative transfers [43] for example.

The Lagrangian point of view is adopted for the rest of this study for the main reason of its parallel capacity. Another advantage is that this approach is not intrusive in the flow phase, so this approach allows to be independent from the kind of application and the method with which the velocity field is computed. The following chapters are about the methods to integrate the particle phase to the coupled fluid: particle localization, particle movement, particle-fluid interactions and particle and work distribution.

2.3 Methods to localize objects

2.3.1 Containment problem

In a volumic particle tracking simulation, the environment is discretized using volumes to approximate the solution of the simulation in 3D. It means that volumic entities contains a part of the solution and that is used in particle tracking. So the particles have to be localized and the volumic element where a particle is has to be determined.

During the implementation of their new algorithm, Haselbacher et al. use a specific algorithm to localize particles in the computing mesh [44]. This algorithm is a pretty traditional one and consists in determining for each face of a cell if the particle is on the right side of this face. This is called the

"in-cell test" by the authors and is also used by many others. This test consists in computing the outward unit normal of all faces of a cell and then checks whether:

$$(f_i - p) \cdot n_i \geqslant 0 \tag{2.2}$$

where f_i is the centroid of the i^{th} face of the cell, p is the particles coordinates and n_i is the outward unit normal of the i^{th} face. In other words, this test checks for each face of a cell if the particle is always on the same side of the face. If this test is passed for all faces, the particle is considered inside the cell. If the test is wrong for one or more cases, the particle is outside.
This algorithm is very simple to implement and to understand. It is also very fast as the complexity depends on the number of faces, the operation is done with the same number of vector operations.

There are two main problems with this algorithm. The first problem is related to the centroid of the faces. The "in-cell test" depends on the centre of the face. This algorithm works with faces with convex shape. Today, there are some research about cells fusion and mesh simplifications in order to save space and time to compute the flow field [45, 46]. But this problem can be solved by triangulation. The cell and its faces can be triangulated on the fly and the algorithm can be applied normally. The second problem is the need to compute or store the outward normal of each face. On moving meshes, this is not a good idea as the face equation has to be computed during each change of the mesh.
The fact is that this algorithm is still very efficient as it can be adapted to unstructured cells by transforming cells and faces into simplexes (faces into triangles and cells into tetrahedrons).

Kalay [47] discusses about two specific algorithms used to determine the position of a point according to a general polyhedra that are solved with a complexity of $O(n)$ where n is the number of faces of the polyhedra. Both algorithms use the same approach that consist in counting the number of intersections between the faces of the polyhedra and a ray with the tested point for origin.
The author proposes an algorithm that consists in launching a determined ray from the tested point and counts the number of faces of the polyhedra this ray intersects. If this number is odd it means that the tested point is inside the polyhedra. On the other hand, if the ray does not intersect any face or intersects an even number of faces, the point is considered to be outside of the polyhedra.
Both algorithms, called *projection method* and *intersection method* developed by Kalay are based on this approach. The first one, the *projection method* consists in projecting the faces of the polyhedra, onto a planar surface reducing the 3 dimensional problem into a 2D problem. The faces are projected onto an image-plane that is perpendicular to the ray.
During the computation of ray-face intersections, some singularities can appear: if the intersection point coincides with a vertex of the face, or is in an edge of the face, the position of the tested point can not be determined by counting the intersections. Another singularity appears if the ray is coplanar with the face. In this case too, the number of intersections can not be determined. By projecting the face on a plane that is perpendicular to the ray, the intersection computation and the detection of singularities is easier.

The second algorithm, the *intersection method* also reduces the dimension of the problem into a 2D one by generating planar polygons that contain the tested point. These new polygons are shaped according to the polyhedra. They are generated by identifying the intersection line segments between each face of the shape and the new plane of the polygon. The containment of the tested point with the polygon is then computed. If the tested point is inside the polygon (computed with ray-edge intersections), the point is inside the polyhedra. Otherwise, the point is outside.
In comparison with the first algorithm, the *intersection method* also encounters singularities.
The paper presents the approach of determining the position of a point compared to an object by computing intersections with the faces. This approach is robust if the singularities are treated. This method is particularly robust with polyhedrons that have very complex shapes, but on the other hand, it requires a lot of computation.
This is also considered as the most recommended algorithm in geometric books [48] and vizualisation guides.

Jing Li and Wencheng Wang implemented a fast and robust algorithm able to determine if a point is in a polyhedron or not [49]. They have adapted this algorithm to a GPU architecture enjoying the high number of available processes and the high parallelization of the algorithm.
This algorithm consists in performing local ray intersection tests in a general mesh. To do so, the environment is discretized into boxes that correspond to the cells of an overlapping grid. This grid fits with the boundaries of the cell. This is the bounding box of the general cell where the point probably is. The faces of the general polyhedron are then assigned to each cell of the bounding box. The authors then predetermine the position of the center point of the cuboids according to the cell of the computing mesh. To determine if the center of a cuboid is inside or outside the original cell, the idea is to count the number of intersected faces.
For each sub-box, the position is saved as *In, Out* or *Sin* if a singularity is encountered. The list of centroids are determined and a ray is built from the point the authors want to localize and the nearest centroid. If the centroid is noticed to have a singularity, the next nearest centroid is chosen. The algorithm parallelization is made on the number of centroids in the bounding box structure.

2.3.2 Localization by intersection computation

During the movement of a particle, it has to be located in case this particle leaves the current cell. The first and naive algorithm is to localize the particle's new location using traditional cell search. The method is quite accurate but is very greedy in terms of computation time and memory access. Haselbacher et al. proposed a smarter algorithm that consists in searching faces of the cell where the particle is passing through [44]. This algorithm is very efficient as it requests less computation than the traditional localization operation. In addition, this algorithm guarantees not to skip any cell in the path and that way, a more accurate particle's path can be computed.
The algorithm takes the advantage of the previous particle's location to deduce the next cell where the particle is localized. It consists in repeatedly computing ray/face intersections to find the neighbouring cell and move the particle until the next time step. The authors used this algorithm to localize particles during the movement but also to detect domain boundaries and compute boundary

conditions and interactions. Another advantage we found in this algorithm is its accuracy for very unusual cell shapes. Figure 2.2 presents an example of this particular case.

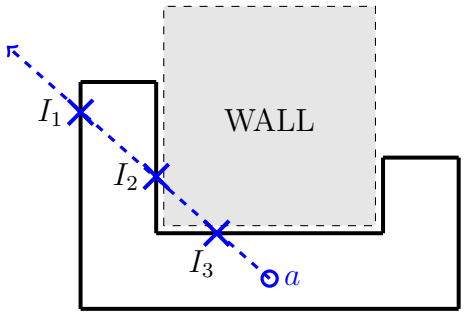


Figure 2.2: Particle intersecting multiple faces of its current cell. Faces are checked in this order : I_1, I_2 and I_3.

In figure 2.2, particle a intersects 3 faces at I_1, I_2 and I_3. In the loop that determines the face the particle is passing through, the intersection point I_1 is first revealed. If the algorithm stops here, when the first intersection point is detected, then the particle will go to this I_1. The problem is that this simulation is false because of the wall presence in figure 2.2. In this example a boundary condition has to be solved first at I_3 intersection point which has not been detected yet. This example demonstrates the importance of checking all faces of the current cell to determine the next intersection point.

The authors also made a very pertinent note related to the advantage figure 2.2 refers to: if the intersection point lies outside the cell, this intersection point will not be retained as only the smallest distance between the intersection point and the particle will be stored. This is a workaround that allows not to check if the intersection point is inside the face or not. If it is inside the face, the corresponding intersection point will be at the shortest distance from the particle whereas if it is outside the face, a closer intersection point will be found. The drawback of this algorithm is that all intersection must be computed to find the closest intersection.

2.3.3 Space discretization and geometric partitioning

Localizing points in space is a very difficult operation that requires many computations. The most efficient strategy to reduce computation time of this operation is to reduce the space to localize a point. Using grids is one application of this method, the space is reduced to a set of subspaces and an additional localization step is implemented in order to determine the subspace where the points are localized.

A smarter and more dynamic way to construct and determined subspaces is to use tree structures through geometric partitioning methods [16]. Numerical trees are structures used in graphs theory in order to describe a graph. A numerical tree is a connected graph where two nodes are connected with a single path. The path used to connect two nodes is unique.
Each node or group of nodes of the tree represent a subspace and the total tree represents the entire computing space of the simulation.

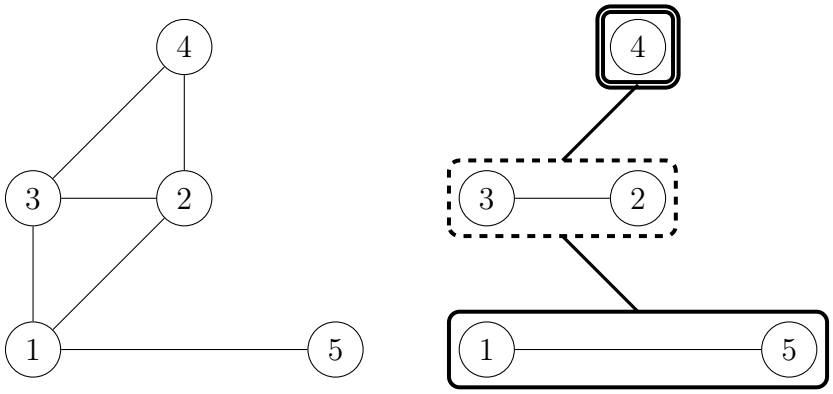


Figure 2.3: A graph and the associated tree

In this figure 2.3 a graph is represented with its associated tree. Each node has a value, that can represent a cell of a computing mesh. A node value is then a solution in a subspace of the simulated space.

The graph in figure 2.3 is discretized into multiple subspaces and each of these subspaces is represented by a node in the tree that is not a leaf.

The search in the tree is then reduced to a complexity equal to $O(n) = log_2(n)$, where n is the number of stages visited in the tree. The maximum number of stages is equal to the maximum height of the tree.

The ways to discretize space using trees are many, kd-trees, octrees, binary trees etc ... These different methods have multiple advantages and disadvantages, but these methods are called recursive bisections as it always consists in recursively building sub-domains of the spatial domain in order to reduce the area of computation.
A structured grid is a specific tree where the root of the tree is the whole simulated space and the leaves are the cells of the structured grid. On the other hand, the structured grid needs different methods as its construction is linear and does not depend on iterative schemes.

Binary trees and Kd-trees

Discretizing space using kd-trees consists in iteratively cutting the space following a dimension. In this tree, the nodes represent the different cuts. The nodes of a common level cut the space in the same direction. Figure 2.4 represent a mesh discretization and the related kd-tree.

The computation space represented by the unstructured mesh in figure 2.4 is discretized iteratively. At each iteration, a cut in each subspace is done in a direction according to the level in the tree. Each node that is not a leaf is a cut. The tree can be unbalanced as shown in the figure but this allows to regroup large areas into the same leaf, by example for poor areas in terms of computation.

A Kd-tree is a particular case of binary trees that are data structures where a non terminal node

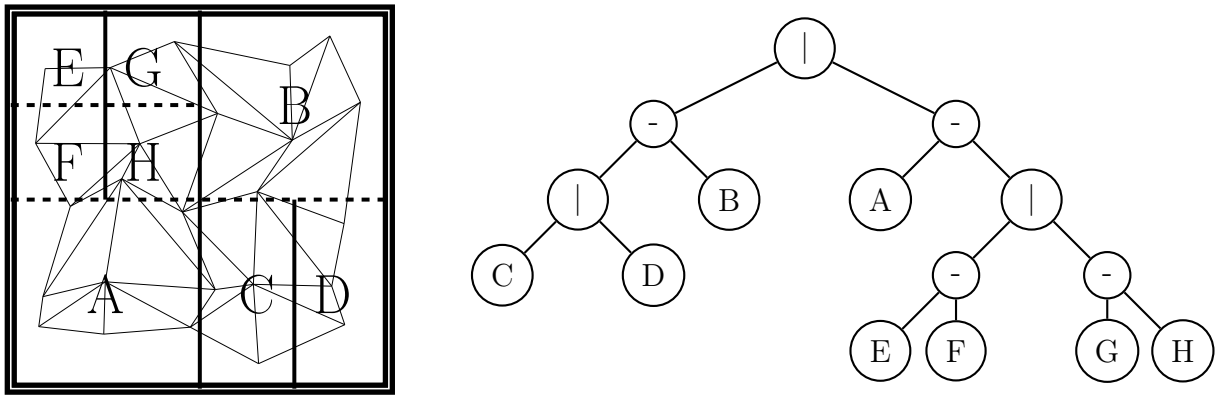


Figure 2.4: Spatial decomposition by Kd-tree.

has two children and an unique ancestor. W. Thibault from Georgia Institute of Technology studied operations on polyhedrons using binary trees [50]. The author proposes an original representation for polyhedrons using Binary Space Partitioning Trees. This representation is based on boundaries representation, using surface planes.

A position compared to a polyhedron can be determined by comparing the requested position according to the multiple surfaces of the polyhedron. The surfaces are stored in the form of a binary tree and by visiting the tree, it can be deduced that a point is inside or outside of the tested polyhedron. This representation uses the similar localization algorithm that consists in comparing the position of the requested point to the faces of the polyhedron.

Quadtrees and Octrees

Quadtrees and Octrees are other structures that are particularly adapted to geometric partitioning. Figure 2.5 gives an example of such a decomposition.

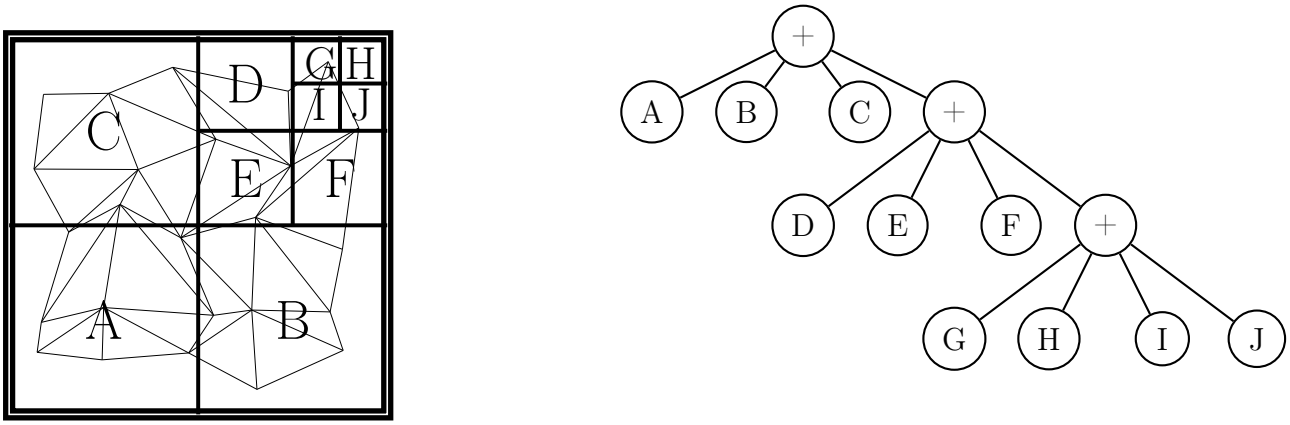


Figure 2.5: Spatial decompositon with Quad-tree.

Quadtrees are numerical trees where each node which is not a leaf, has 4 children. This is particularly well adapted to 2D and surface models. The computing space is decomposed into a grid of 4 cells which can very easily localize a position.

As numerical trees are dynamic objects (that can be augmented or reduced according to an arbitrary accuracy), quadtrees can fit to a more adapted shape than fixed structures such as cartesian grids.

An octree is the same object, built in the same manner but adapted for 3d models. In fact, a node

of an octree has 8 children and one ancestor. At each iteration, the subspace is decomposed into 8 new subspaces forming the shape of a 3d cartesian grid with 8 cells.

Structured grid

In the very special case where cells are cuboids, localization is one of the easiest problem to solve. In fact, determining the position of a point in relation to a cuboid consists in comparing maximum and minimum coordinates of the cuboid's vertices.
As a cuboid (or a rectangle in a 2D model) can be described with only 2 vertices, the point to localize is compared to these two vertices and if the point's coordinates are between the minimal and maximal coordinates of both vertices, then the point is inside the cell. This gives, compared to the previous techniques, a very fast and very easy algorithm to localize a particle as far as tree leaves and subspaces are simple shapes such as cuboids or rectangles.
The major problem of trees and iterative algorithms lies in the complexity of localization in the tree. This localization requires at the worst a visit of the deepest leaf.
Localizing a point in a set of cuboids of the same size, in other words in a regular grid, is a direct operation that does not have any intermediate operation of approximation. M. Garcia [21] remarks that localizing a point in a regular grid is done with a constant complexity that depends on the number of directions of the grid. The author uses regular grids to localize particles in unstructured meshes drawing a second mesh, a regular grid, of the size of the mesh partition. This method allows the author to first localize particles in the regular grid, with a constant complexity and then in a second time localize particles in cells of the unstructured mesh inside the area of the determined cuboid.
The advantage of this approach is that building a regular mesh is very fast to compute, the created object is light to store in memory and localization is very efficient. On the other hand, the size of the regular grid and the size of cuboids depend on the size of the computing mesh. In addition, it can happen that a cuboid cell of the overlapping regular grid, does not contain any cell of the computing mesh. It happens if the step discretization used to build the regular grid is too small compared to the average size of the computed mesh. One workaround consists in building another grid with higher steps. A more interesting workaround is to determine dynamically the steps according to the form of the computing mesh for example the size of a cuboid can not be smaller that the smallest edge of the computing mesh.

2.4 Methods to compute intersections

As some of the algorithms of localization require to compute particle-face intersections, this operation is one of the most important in the particle tracking instructions set. This operation is also used to compute boundary conditions when a particle intersects a wall for example. There is much work on the subject as the computation of a ray and a plane is used in graphics and in CFD.

Badouel [51] defined an algorithm to compute ray/polygon intersection efficiently. Earlier in 1987, Snyder and Barr [52] wrote a similar algorithm based on barycentric coordinates. Based on this algorithm Badouel proposed a faster algorithm that consists in 2 main steps : first, the algorithm determines if the ray intersects on the plane of a face of the tested polygon. This is done by computing a cross product with the ray and the face normal. The coordinates of the intersection point is also determined at the same time.
The second step determines if the intersection point is inside the face borders. To do so, the intersection point position is determined regarding with respect to the face normal. If the point is on the same "side" of each edge, then the intersection point is inside the face.
The faces are considered as sets of triangles if the number of vertices is greater than 3. For this reason, only intersections with convex polygons can be found with this algorithm. This problem is the same with many other algorithms that attempt to compute ray-polygon/face intersections. That is why faces are triangulated or simple and convex objects are used by scientists.

Möller and Trumbore [53] give an algorithm to determine whether a ray intersects a triangle. This algorithm published in 2005 is capable to determine the intersection point of a ray and a triangle in 3D. To do so, the triangle is translated to an origin and transformed into a unit triangle. The ray is then aligned with an axis and finaly the linear system 2.3 is solved where D is the ray direction, V_0, V_1 and V_2 are the triangle's vertices, O is the orign of the ray (the particle), u and v are the barycentric coordinates of the intersesction point and t is the distance from the ray origin to the intersection point.

$$[-D, V_1 - V_0, V_2 - V_0][t, u, v] = O - V_0 \tag{2.3}$$

The algorithm is really interesting to use because of its computing performances and its low need of memory storage. The algorithm does not need to compute or store plane equations and normal vectors of the triangles. The authors obtained execution time comparable to the ray/polygon intersection algorithm of Badouel [51] without computation and storage of normals and plane equations.

Shevtsov et al. [54] defined an algorithm to compute ray-triangle intersection for modern CPU architectures. The algorithm is tested using SIMD instructions. They noticed that ray intersections are computed using 2 main operations: computation of the intersection point between the ray and the triangle plane, then the aperture test that consists in determining if the intersection point is inside the triangle face or not. The authors also noticed that the aperture test exits more often than the distance test in ray-intersection operations. The idea is then to proceed to the aperture test before the distance test. The Plücker test and coordinates allow this operation. The tests have been implemented using precomputed data like edges and normal data and also using SSE instructions [55].
The advantage of this paper is that the Plücker test is presented and the aligned structures designed to use SSE instructions are presented.

The algorithms used are based on the same two steps that consist in computing the intersection

point and the distance to the origin of the ray, then determine if the intersection point is inside the polygon we are checking.
Some algorithms need more information about plane equations and plane normals, some others need more computation and a more specific formulation but all of them can be used with no loss of precision.

2.5 High Performance Computing and problem parallelization

2.5.1 Objectives of HPC

The High Performance Computing (HPC) is a branch of computer science that becomes more and more important since the advent of this field. Computers and serial processors rapidly showed their limitation in terms of computation capabilities and people show high interest in this field because of the theoretical capacity to reduce computation time without loosing computation's accuracy. Multiple processes working in parallel on the same discretized problem make reduce the number of tasks per process. The requiered amount of data is also reduced. As the number of tasks and the memory occupancy are reduced, the saved execution time and memory space can be used to improve the accuracy of the computation. The need for parallelization has become more and more important as the computation capacity of processors grows starting with vector processors.
Todays challenges are still the same with some additional ones, simulations and models are more accurate, and in addition, the need to reduce computation time reducing the energy cost of supercomputers is becoming more and more important due to environment challenges of our time.

2.5.2 HPC Technologies and execution models.

Flynn's Taxonomy

The models used in HPC are well known and based on the Flynn's Classification. This classification is a set of four classes that can describe a machine architecture.

Simple Instruction Simple Data This model is directly connected to a von Neumann architecture (Figure 2.6) which consists in setting a computing unit, a memory controller and a memory unit.

An instruction flow comes into the device with its data flow and a single output comes out. In this execution model, the flow works the same : a single set of instructions comes into the input of the computing unit, the associated data is stored in the memory unit of this computing unit and the computed data comes out of the computing unit.

Simple Instruction Multiple Data The SIMD model performs a single set of instructions or a single set of tasks at a time on multiple sets of data structures. This is the model exposed by the

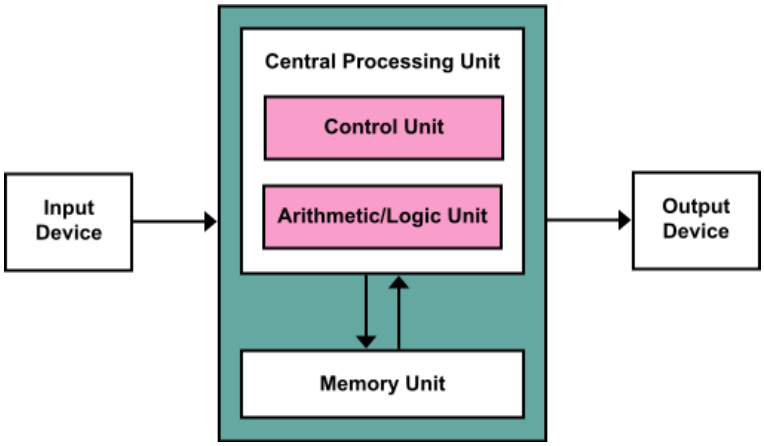


Figure 2.6: Von Neumann architecture (Diagram from *Wikipedia*).

particle tracking: the single instruction is the track of particles (localization, movement, interactions, ...) applied to multiple particles (multiple data sets). This is also the model that illustrates operations on vectors, a single operation is executed on multiple data stored in the form of a vector.

Multiple Instruction Simple Data The MISD model performs multiple instructions on a single set of data. Several processes uses the same data to compute different operations in parallel. This model is close to task-parallelism as different tasks are computed on the data flow. This architecture is represented by pipeline parallelism as multiple tasks are pushed into a pipeline with a data set and the tasks are run in parallel on the same data flow.

Multiple Instruction Multiple Data The MIMD is used to perform several instructions on several data sets. This is the model of today's complex solvers, the parallel processes execute different tasks on parallel data sets. This model is the most complex to implement as several problems caused by data synchronization, tasks stealing, and communication priorities.

As modern challenges are solved by multiple programs and libraries, we are not talking about instructions but programs. The same architectures are used but multiple programs are called on a set of data for a single simulation. This is the case for many CFD solvers that call in the same run a partitioner, a fluid solver, a solid solver, a task manager and a solution writer.

Memory Accesses and Communications.

The memory access, or the access of a datum, highly depends on multiple factors : the data distribution, the needs of the simulation, the shape of the network, and the machine architecture (Flynn's architectures). We have already seen that architectures and data distribution is part of this study. As there is as much data access needs as simulations, the last factor that impacts the performances of data access concerns the network and its topology.

The topology of a network is an application of graph theory, where the node of a network (the

multiple units of a network) is modeled by a node of a graph. Several network topologies are used in the design of massively parallel machines. We can cite star networks, rings and toruses, trees and meshes.
The topology of a network highly impacts the parallel performances as it determines the complexity of the data flow, this complexity depending on the length of the path needed to access a remote node. The more this path is long and complex, the less is the parallel efficiency. For example the longest path of a ring network is equal to $\frac{n_{nodes}}{2}$, where n_{nodes} is the total number of nodes in the network. Today, many HPC machines have custom network topologies that correspond most of the time to a hybrid topology, mixing several topologies for the same architecture.

Non **U**niform **M**emory **A**ccess (**NUMA**) [56] is a memory access design that is characterized by a set of *NUMA* nodes. Each *NUMA* node is connected to the other ones by a network. The specificity of a *NUMA* system is that all *NUMA* node see the global memory in the same way. As *NUMA* nodes do not share memory spaces in terms of hardware, these multiple memories are connected to a network. A defined process located in a *NUMA* node has access to all memory spaces of the *NUMA* system. To do so, to access a remote data stored on the memory of a remote *NUMA* node, the above process needs to see the memory. The requested data is then transfered to the local memory. This transfer is totally hidden from the requesting process. Transfers from remote memory spaces require latency due to the distance in the entire machine the data flow need to cross. Thus the time to access a data set in a *NUMA* design depends on the data location relative to the requesting processor.
NUMA nodes are the most used nowadays because of the production of multiprocessing units such as multi core processors, accelerators and graphic cards and now many-cores.

In fact, massively parallel machines can be sets of NUMA systems connected by a network topology. Each node of a machine has a memory address scheme (that can be based on NUMA design) and a memory policy. Accessing a data set stored on a remote memory space, on a remote NUMA system, requires a communication protocol which is often by **M**essage **P**assing **I**nterface for the majority of simulations and HPC softwares. Such a communication protocol allows to send and receive data sets though the network. Obviously the more the path of a dataflow is long, the longer is the communication.

2.5.3 Processes proximity and core affinity.

Balaj et al. [57] made a review of the 2009 MPI implementation. The authors evaluate the scalability of the current MPI implementation. MPI is an interface used to transfer data from a process to another one that does not share the same memory space. This communication is done using messages based on send and receive calls. The MPI implementation allows the user to call for collective messages, these are messages transfer from a group of processes to another group of processes. The authors measured that MPI is ready to scale to a million of processors with some deficiencies due to irregular collective communications that do not scale to this number of processes and also due to

the actual topology of processes virtual representation.
This paper advises then to avoid irregular collective communications in order to scale to a high number of processes. During the implementation of this library, this advice has been followed and the times where collective communications need to be called, efforts have been made to call regular collective communications, with a constant message size.

Placing processes can have a real impact on computation time. Indeed, the location of processes compared to the hardware placement and the numerical placement of data impacts the performances of the communications because of the size and the number of messages.
Emmanuel Jeannot and Guillaume Mercier [58] propose an algorithm *TreeMatch* able to map processes to available resources in order to reduce communication cost.
To do so, the authors use Hwloc [59], a software able to catch informations about hardware and architecture. This tool is then used to analyze these informations and manage or remap processes and numerical ranks in order to improve performances.
The strategy the authors developed is based on a tree structure that is a more reliable representation and a more adaptable representation than static structures as matrices. In addition, *Hwloc* also uses a tree structure to represent hardware informations.
The *TreeMatch* algorithm consists in ordering processes according to a communication matrix that stores communication speeds between processes. This matrix also corresponds to the processes proximity as far processes generate slow communications. The processes are then regrouped into multiple groups of processes.
According to the authors, this algorithm provides an optimal mapping as it outperforms other approaches.
This study shows the importance of process distance in case of communication bound applications.

2.5.4 Mesh Partitionning

Partitioning meshes is a NP-hard problem. In general the mesh is considered as a graph with connectivities and geometrical properties. There are many methods to partition a mesh on a parallel system with advantages and drawbacks.

Schloegel et al. summarized and wrote the state of the art of mesh and graph partitioning methods [16]. In this book, the authors summarize and explain the different methods to cut and partition meshed based problems. They studied several metrics to compare the different partitioning methods like the final partition quality, the required runtime to perform a partitioning scheme or the degree of parallelism.
To conclude the authors compared some partitioning tools related to their functionalities and the proposed schemes. It is important to specify that this study has been made in 2000, so the current tools may have evolved since that year. Generally this comparison shows that partitioners are specific tools. For example Metis [60] performs multilevel schemes, ParMetis [61] performs dynamic and parallel partitioning, Chaco [62] especially performs spectral schemes and Scotch [63] is specialized in combinatorial schemes.

A specific method to partition meshes consists in using space-filling curves to number mesh elements according to their spacial position. Curves like Hilbert curve, Peano curve and Lebesgue curve are famous ones and used to partition space.
Meng et al. [64] proposed to use the Hilbert curve to cut and partition spatial data. The algorithm is quite simple, it consists in building a Hilbert curve with the different spatial elements and sorting these element according to their position in the Hilbert curve. This sort determines the "disk" to which each element is assigned.
The Hilbert curve guarantees that close elements have higher chances to be contiguous in memory. The authors show that partitioning using Hilbert curve brings better response for element access compared to the Oracle Spatial algorithm. This is due to the spatial and temporal locality improvements brought by space filling curves.
Castro et al. [17] also confirmed performance improvements by using Hilbert space filling curve to partition space. The Hilbert partitioning method improves the execution time linearly and quadratically on a parallel run.
In the following, the Hilbert space filling curve is used to partition the mesh of the test cases.

2.5.5 Particle tracking in HPC.

Hiroshi Nishimura et al. [65] did a feasibility study of using GPU to track particles in a parallel code. It demonstrated that tracking particles on a GPU gives about one order of magnitude less time to solve even if only the tracking is done on the GPU. It shows that the track can be parallelized on shared memory accelerators and is the hotspot of the application.
The authors revealed a drawback of the adaptation of particle tracking which is connected to the size of data. A GPU, and more generally, an accelerator has its own memory. This memory is shared with the multiple cores of the accelerator and the access to this memory is done using a communication protocol.
Because the available memory space is reduced compared to HPC nodes, the total amount of particles and mesh data exportable to the accelerator's memory is limited.
Another problem the authors encountered is the huge number of pointers and data class they hardly managed on the GPU. The last obstacle they dealt with is the fact that the compilation of the entire particle tracking library failed. The reason is that some functions are too big to be optimized and so the compilation runs out of memory. Unfortunately the authors removed some computation from the GPU kernel to perform them in the CPU part.
On the other hand the speed-up obtained by porting particle tracking on GPUs is around 120 for optimized functions. In fact the functions ported on GPUs run 120 times faster than the same on CPUs.
This paper gives an overview of parallel and porting capacities of particle tracking on GPUs and other accelerators.

2.6 Parallelization strategies and Load Balancing

The particle localization is the most important step in order to compute particle tracking. This operation consists in localizing a set of particles in a given set of cells or a mesh partition. Localizing a particle allows the computation of the particle direction by giving the position of the particle and computing the sum of forces applied to the particle.

Plimpton et al. [20] have developed an algorithm to optimize load-balancing in a parallel electromagnetic particle-in-cell code. The field is partitioned and distributed to all processors in the form of blocks. These processors have to solve the fluid phase within the block it owns and push particles that are in this block. The algorithm describes a way to optimize load-balancing of particles in the case of a processor owning most of the particles. To do so, the authors have described field blocks and sub-blocks in the form of "windows". A window is a set of contiguous grid cells that can be assigned to a new processor. In this way particles can be sent to an other processor with its window, as long as they remain in the window.
Using the same serial multi-blocks approach of the original serial simulation, the authors developed a particle distribution algorithm that is independent of the partitioning method of the flow field. This method is based on the concentration of particles. Sub-domains are assigned to new processes as long as they contain particles to track and as long as the load-balancing is not good enough. The approach the authors developed has an impressive parallel efficiency as it scales close to the scaling of problems without load-balancing issues. It means that their distribution method is quite efficient. In the application of electromagnetics, their performances show that their distribution algorithm is close to the ideal particle balancing.
This study is very interesting and important in a parallel particle tracking code. The authors proved the efficiency of a well balanced particle code on parallel context and the efficiency of independent solves of the field and particle phases. This study is a very good inspiration to implement any approach for parallel particle tracking as the authors took care about particle balancing, flow partitioning and implicitly for particle localization and carry of cache usage.
On the other hand, the size of the window can be an entire block. Each time step they must send those values (Field) to the appropriate child processors to enable them (windows). The study shows that static imbalance of particles is 10 to 35% more efficient than a dynamic imbalance. But is about 20˜30% more efficient than a run without any particle unbalance. The particle balance is very close to the ideal. Building a window can be very complex due to its building method which depends on the number of particles. The complexity does not exists in a regular grid build.

Fonlupt et al. made the theoretical analysis of existing load-balancing algorithms for data parallel computation [19]. These algorithms were written and their computation complexity and communication pattern are studied.
This study is a summary of many load-balancing algorithms which are usable in a parallel context. These algorithms are compared using mathematical analysis. To do so, the authors defined two characters: the cost which is the complexity of one iteration of the algorithm and the quality which is the

product of one iteration cost by the number of iterations needed to reach a steady-state where each virtual processor owns the average load of the system. The method of evaluation the authors used is completely theoretical but it allows a first comparison and classification of the different algorithms. This study is relatively complete and provides many algorithms to optimize work and load balancing. The authors propose a classification of these algorithms and give some advice to decide what kind of algorithm to use on a specific architecture or in order to solve a specific problem.

The authors in [66] developed a scalable load-balancing technique for a massively parallel Monte Carlo particle transport code. Particle workload is distributed across processors using MPI. The final aim of the authors is to find a technique able to distribute workload over millions of computing units. The complexity of their previous algorithm was $O(N^2)$ where N is the total number of processors. Processors are paired and work is distributed between these two processors. A partner rank is chosen as the following :

$$\begin{cases} \text{rank} + 2^k & \text{if the } k^{th} \text{ binary digit of rank is 0} \\ \text{rank} - 2^k & \text{if the } k^{th} \text{ binary digit of rank is 1.} \end{cases}$$

The partner changes at each round of load balancing.
The authors test these algorithms on the Godiva critical assembly test problem from the Nuclear Energy Agency on the Sequoia super computer as a weak scaling problem (fixed work per process). The results of this test show that their algorithm maintains the parallel efficiency to 95% up to 2 millions processors whereas the application without load balancing brings the efficiency down to 68%. The results also show that the implemented algorithm impacts the tracking time, as it becomes constant compared to the same application without load balancing.
The authors give a load balance efficiency definition and measurement. They have developed a scalable load balancing algorithm that has a complexity of $O(log(N))$ where N is the number of processes. This algorithm is very interesting as it is very efficient for a high number of processes and take less than 12 seconds for millions of processes. The main drawback of this algorithm is that it is limited to a certain number of processes: the algorithm can be applied only to a number of processes equal to a power of 2. The reason is that the key of this algorithm lies in the coupling of processes.

Darmana et al. [2] developed a parallel algorithm to follow particles in a Euler-Lagrange model using a mirror domain technique. This technique consist in copying particles to follow on all processors. The authors have partitioned the flow field using PETSc. The particles and their data are scattered to all the system. For any process, it choses a subset of particles to track, computes their path and updates their position on all other processes. Particle's paths can then be computed completely independently.
The authors modeled the dispersed phase dynamics accounts for bubble-bubble collisions in the Poisson pressure equation. The fluid equation of the continuous phase is solved using decomposition domain with PETSc implementation. Concerning the bubbles (their particles to track), the authors want to simulate the interactions between bubbles, a very difficult operation often done in a second run once the theoretical paths are known which is very difficult to parallelize especially on very dense

regions, regions where the density of bubbles is very high.
The authors also measured the parallel efficiency of the application that reaches 20 for 32 processors. This means that the reached efficiency is around 62%.
This study is useful because the authors recall the importance of keeping particles independent. And that to keep a high level of independence between particles but also between processes, some data needs to be known by all the computing system. This way, the amount of communications and time spent in these communications is drastically lowered. This technique also allowed to not care about the particle's distribution as all processors have and update of all particles localization and velocity. A direct advantage of this strategy is that the theoretical parallelization of particle tracking is close to the ideal one. It can be noticed that subsets of bubbles are chosen by their location in memory. So that transfers and bubbles updates are very efficient as contiguous data are sent.

2.7 Methods to implement efficient libraries

2.7.1 Technology reuse.

Reusing softwares, codes and technologies has an old history in software engineering back in the 1970's. Developpers and scientists very often reused old codes, older methods to import in a new one in order to not invent repeatedly the wheel. In computer science and software engineering, reusing codes, applications and frameworks is unfortunately limited by the language, software patterns or conflicting structures.
According to Kang et al. [67], the development of a reuse-based software helps to identify the crucial activities and tasks to carry on during the software development. The whole paper proposed a general methodology to develop efficiently a software based on code and component reuse. This methodology includes multiple gears in the production and development chain but the most important thing to remember concerns the implementation of a component of the software: a new function, or component must follow this set of routines:

1. test locally (Coding and unit testing),
2. test and integrate the Computer Software Components (CSC),
3. test the Computer Software Configuration Item, and finally
4. test the whole system.

This simple plan can remove most singularities in the software components as all new component are drastically tested before its integration in the whole system.
In the methodology, another thing to note is that algorithms and solvers are constantly challenged with the idea of component reuse. To be clear, algorithms are encouraged to reuse already tested components in order to reduce maintenance costs and maximize the robustness of calling functions. The main principles have been followed during the development of the particle tracking library, using

independent components and calling the same reused and tested components.

Merijn de Jonge [68] gives further advices for components implementation, development of reusable components have two main objectives: increase the reuse level of the application increasing the pay-off in terms of development time and robustess and increase the reuse of inidividual components by implementing more general components.
In addition, the author specifies the different ways a component or a set of components can be connected: by file, system pipes or by other communication protocols. In our case in for a CFD solver, meshes, partitions and other data are often saved and exchange by files using specific data structures. A special attention is paid on the data structures to correspond to the expected structures and data types.

2.7.2 Component reuse adapted to HPC

Emad et al. [69, 70] apply the reuse politics and habits to parallel softwares. They presented a HPC software *LAKe* and its architecture that is designed for the use of the *same code for the sequential and the parallel* version.
The software architecture of *LAKe* is object oriented designed. There is a list of classes that describe an object or an operation from the most coarse grain to the most accurate and precise operator. The example of Arnoldi's method [71] is given and shows the need of a Matrix class, operators on matrices, a class for Arnoldi's blocks, a class that performs an iterative method and a class that performs the iterative block Arnoldi's method. Thus, this last class is a fine grain class that describes a precise operation with precise matrix storage, on the other hand it calls more coarse grain classes like Matrix class and Arnoldi class to compute the iterative block method.
The authors also make some remarks about the object oriented architecture and parallelism. A wrong design of classes can lead to inefficient parallelism. For example, accessing a single element of a matrix in parallel leads to numerous memory accesses that generate many accesses to class members.
In order to limit the number of accesses to members and to classes, a list of services is developed. These services are stored in a *Service Pattern* block that stores a state of arguments and performs multiple operation on the data base.
This study gives good advices and concrete examples of a well designed reuse library.

In another paper [72], the authors describe a high level design for reusable parallel library based on 3 main components that are the *Data Component* that stores and manage serial and parallel objects, the *Computation Component* that performs computations on an determined data-flow and the *Communication Component* that is used to manage communications between computation units using communication protocols like *MPI*[73] and accelerators management [74, 14, 75].
This architecture is very simple and allows to clearly define the role of each component of the library developed for particle tracking.

2.8 Conclusion

Parallel particle tracking calls for methods from several domains: the developpment of an efficient parallel library requires knowledges of HPC and parallel architectures, software engineering and network. In order to be as efficient as possible multiple techniques have to be implemented and the structures representation have to be well designed.
In addition, particle tracking requires the implementation of methods to localize particles in a defined area and other methods to compute the potential particle interactions. For example, a ray/face intersection algorithm is deployed in order to compute particle interactions and collisions with walls.
The current studies on particle tracking show that hybrid methods are used in simulations. Most of the time, the flowfield is solved using Euler equations whereas particle paths are computed using Langrangian equations. This technique allows to keep particles resolution independent from the flow field computation. On the other hand, an integration method have to be implemented in order to record particle paths using the flowfield. The most important impact of this integration is the fact that particles must be localized in the computing mesh.
This is an additional difficulty for particle tracking computation but gives in the same time multiple advantages as a better parallelisation and a more accurate particle localisation.
A last domain of study concerns the algorithms to distribute tasks on a parallel context. This is applied for particle tracking because the parallel efficiency of such a simulation depends on the number of particles per computing unit.
Studies on parallel particle tracking are many and many implementations have already been done. The scientists use and invented several strategies in order to compute particle tracking in parallel contexts in different operations. The main difficulty that rises in all studies lies in the particle distribution. Performances of particle tracking are correlated with the number of particles to track per process. Modern *NUMA* architectures such as accelerators and many-cores are well adapted for particle tracking that is a data-parallel problem.
As the final library is designed to compute parallel particle tracking, some of these methods are tested, implemented and optimized in order not to fit to a specific particle resolution but in order to compute particle tracking in a more general context. For this reason, the methods that are efficient for unstructured meshes and random cell shapes such as concave cells are selected and adapted to the software architecture.

CHAPTER

3

Design of a Particle Tracking Library.

Contents

3.1 Introduction

In this chapter, we describe the different operations needed to compute particle tracking and then deduce and justify the software architecture of the implemented library *ParOPTIC*. The general architecture is designed from an analysis of operations required for particle tracking. With this general architecture, general usages are detailed, as the strategies to access data structures, preconditioning methods and data preparation for transfers through a network, parallel strategies and good practices in order to efficiently adapt particle tracking to massively parallel machines.

Structures, classes and data types are detailed and justified to fit with parallel good practices and

some metrics are introduced in order to evaluate the sequential and parallel efficiency of the final library.

3.2 Description of the main operations to compute particle tracking.

As we have seen earlier in this manuscript, the Lagrangian approach describes the particle movement as the sum of forces applied to the particle. It means that, in order to know what forces are applied to the particle, what interactions the particle is going to create, this particle needs to be localized in the current flow field.

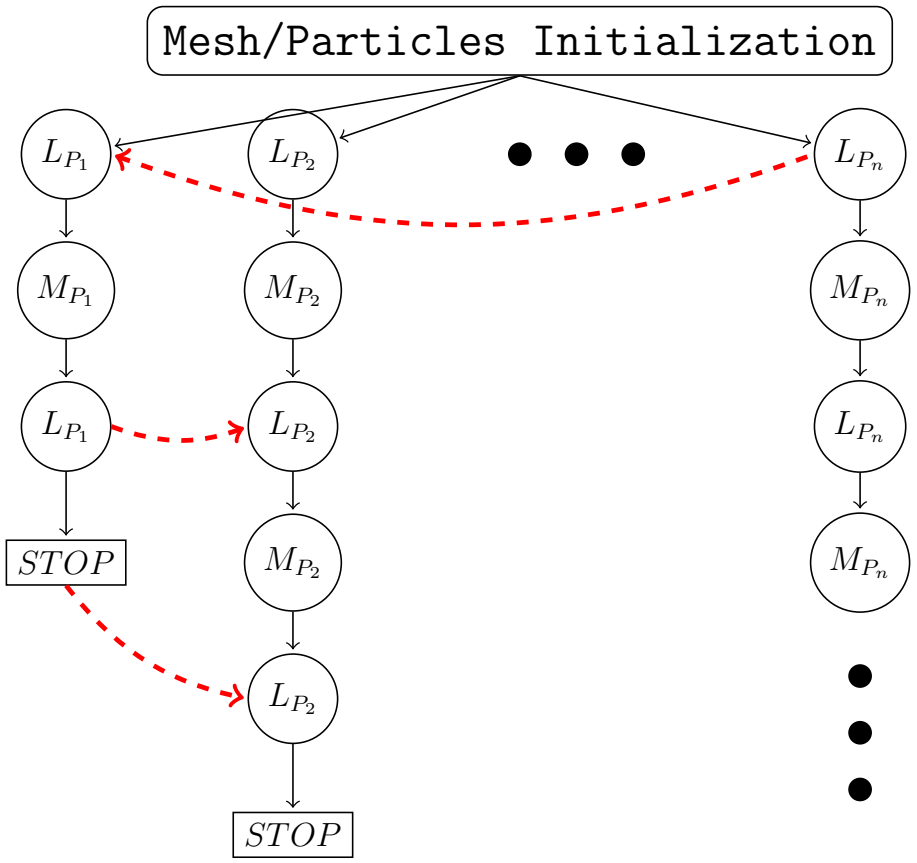


Figure 3.1: Particle tracking task graph. P_n stand for the n^{th} particle, a group of particles or a group of processes, L stands for Localization operation and M for Particle Movement.

Figure 3.1 is a sketch of operations used in particle tracking. We can see that two main operations are used to track particles in the Lagrangian way : the particle localization and the particle movement. This figure also shows two things: particle tracking is an embarrassingly parallel problem and as particles are independants, they call independent operations and datasets. We can sketch processors or computing units and memory spaces with the same graph as the different columns can stand for a group of particles, a group of processes or a group of memory spaces. Dashed arrows between processes represent the communications as a process may need a dataset, a part of the mesh to track its particles. In this example the first process P_1 imports the data owned by the n^{th} process,

P_n, to localize the particle during the first localization step and exports data to help the second process to localize its particle during the second step. This figure gives a good overview of the main difficulty to express high parallelism: a well balanced workload. As processes would work during different duration depending on particle's lives, a process that has finished all its work (first process in figure 3.1), will wait for other processes to send it some data. This is an opportunity to give it some particle to track. As the particle localization is the operation in which a process needs to import data, this step is implicitly a synchronization step.

According to these comments, we can extract a pattern, an architecture from the task graph.

3.3 Proposition of a software architecture.

The design of a software architecture is one of the most important step in a computer project. The explanation of this importance is that the entire code will be developed according to the decided design. This design has to reflect the main objectives of the application. In other words, it is not very efficient to design an architecture that renders the application difficult to use, exploit and maintain. An example of a bad architecture can be an interface that does not give any access to important and often used tools and services. We can imagine a mathematical library, a BLAS-like tool that only gives the user access to the Arnoldi's iteration computation. The Arnoldi's method needs matrix-vector multiplications and vector-vector operations. Even if these operators have been developed in the library, the user will need to find another one to compute other operations on matrices or implement them again instead of reusing the already developed ones in the Arnoldi library. This is the kind of frustration a well thought software design can avoid.
In the case of particle tracking, the most important quality the design must have is the ability to reuse important and often used functions.

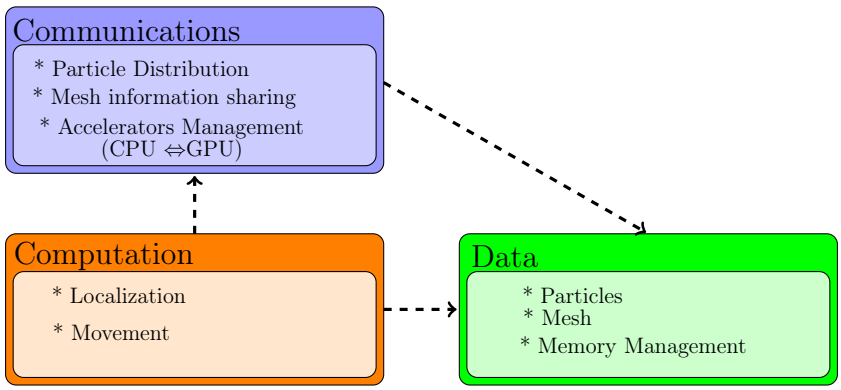


Figure 3.2: Component Graph of the library.

In this particular study applied to particle tracking, and according to the task graph particle localization and movement are the most important features that have to be accessible to the users. But the fact is that this study and the final library has to perform efficient particle tracking operations on massively parallel systems. So the final software must provide features to manage data and communications in a parallel context.
According to the graph of tasks seen earlier in this manuscript and taking into account the challenges

of high performance computing, the library is designed with a component based architecture. This architecture and the attached component graph is described in figure 3.2.
The architecture is based on three main components which are the Computation component, the Data component and the Communication component. On this component graph dependencies are also described. We tried to keep the components as independent as possible from each other but the dependency on data representation still remains. In fact this is a question of efficiency of a data representation that is not necessary a primitive one. In the following part, these components are quickly described and more technical and implementation details are available later in this manuscript.

Computation Component. The computation component is, as its name obviously says, the component used to do computation. In the study of particle tracking, most of the needed computations are geometrical based computations. In this component, the user finds any function that is needed to track particles for example, the user can use the function to compute the intersection point of a ray and a plane, the different functions and methods to localize a particle in a discretized environment or in a set of polyhedrons. Globaly, the functions implemented in this component are used to localize and move objects (like particles) and compute different interactions.
Most of these functions are instantiated for 32 and 64 bits to be more accurate in the case of heterogeneous architectures. The data representation to use this component is quite regular as it is based on primitive data representations. To be more explicit, this component uses integers, floats, bytes and their equivalent for 64 bits representations. In addition of these data representations, a special data structure is used to localize particles in a parallel context. This structure is based on bit manipulations and it is described later.

Data Component. The data component describes the special data structures and objects used to localize and move particles efficiently. The structures are used to describe an internal mesh partition, a set of cells or a communication protocol. Most of these structures are only for internal use, but a structure that is a light mesh representation. The most important structures that might be sent through the network of a parallel machine is implemented using bytes in order to render the communications protocol easier to use and to limit the number of communication calls. As the data access is more difficult using bytes, the data transformations are done internally.

Communication Component. The communication component is used to describe the communication protocols in a parallel system. This component manages data structures and services used to partition, distribute data and work over multiple processes and manage memory of accelerators. For example, this component contains MPI calls, CUDA stream creations and particle balance methods. This component also contains methods to read and write files especially log files that contain debug informations and also mesh readers and writers in standard formats like CGNS [76]and GMSH [77] files.

3.4 Functions and data strucutres adaptation for parallel context.

3.4.1 Integration to a calling solver.

The development of a new library that has the objective to reuse sequential and parallel code using external mesh and external simulation data leads to deal with external and specific data formats. The use of external data raises some questions:

1. can the imported structures be modified in order to coincide with the library,
2. does the imported data need to be cast to less complex data types,
3. can the memory allocated for imported data be modified,
4. can the data arrangement be modified and returned with a new data arrangement,
5. are the imported data types robust and suitable for particle tracking.

These questions can be reduced to a general one which is: the library has to deal with external data managed by an external component or with a copy of these data managed by an internal component.
A concrete example is simply the allocated data for the particle coordinates: during the particle tracking, some possibly leave the simulation. As these lost particles are not tracked anymore, the storage of their data (coordinates, velocity, diameter, weight, energy, localization, ...) is then useless. This useless memory space can be reallocated for another injected particle or a better parallel particle distribution.
Thus, the question is what does the particle tracking library do with the unused memory.
At this moment, the library is in charge of memory management. The current version of the library takes care of the allocation and deallocation of particle data. On the other hand, the computing mesh and the flow field are copied in the *Data* component in order to be less invasive as possible in the data storage of the whole simulation. Thus the data from the computing mesh is updated when the update service is called.
Future work will consist in adapting the particle data to this non invasive objective. To do so the variadic templates of *C++11* [78] are used and strides help the implementation of *AOS* data structures.

3.4.2 Parallel adaptation of a library.

Because the study is to perform particle tracking in High Performance Computing field, there are some rules to apply in order to be efficient at higher scales. In other words, some serial techniques and programming usuals are not efficient and adapted to HPC machines. This section is dedicated to some rules followed during the implementation of the library useful in this particular challenge of particle tracking and applicable to similar problems.

Keep data close to the local process and its neighbors.

This rule is very important and its efficiency and impact is one of the observations of this study. The principle is to keep as long as possible important data close to the core, the computing unit or the local shared memory space.
This is important especially in massively parallel machines because of the communication costs. In fact, communications efficiency in parallel computing is a very large field of study because of its importance in major parallel simulations and computations. The graph of tasks of particle tracking shows the high parallel capability of the operation due to the distribution of particles over the system. Because particles can be considered as independent objects they can be tracked independently of the parallel system : a computing system can track particles without communications as far as it owns the needed data. That is why the more data a core can keep, the better the performances will be. This sentence has to be understood properly. We do not say that all particles and all data have to be on the same memory space, but the computing units must have the maximum size of work and data with an acceptable load and work balance between the units. In other words, computing units have the same amount of work to do which corresponds to the maximum of work that can be given to a unit.

This rule is not only true for parallel computation, but also to any kind of application. Modern computers and machines have multiple cache and memory levels. They are in general named and classified by the proximity to the core and their latency. A universal characteristic is that the more the memory space is far from the core, the more space is available. It means that the more different data is needed, the more space and memory levels we need to compute an operation. To minimize memory need and transfer latencies, the idea is to do the maximum number of operations on the same data set.

Difficulties due to particle tracking implementation are mentioned later.

Minimize data needs.

Parallel computing often involves communication and data movement through a network. Communications are often hotspots in parallel softwares for multiple reasons : communications need synchronizations between two computing units that communicate, communication performances are limited by the network performance that depends on the machine and also on the distance between the two units and finally the size and number of communications highly impact the performances of communications.
The topology is also very important because the network topology can be designed to reduce distance between nodes. But the fact that the topology reduces maximum and average distance is not efficient if nodes communicate all the time with small sized messages, the communication phases are still hotspots to reduce.
One way to optimize communications and reduce their impact on overall performances is to try to reduce the number of messages and communication calls and optimize the size of these messages. In most cases, increasing the message size will yield better performance but this is limited by the

number of tasks (for MPI implementations). In addition this improvement is only valuable for a limited range of message size [79, 80]. This is explained when the message size approximates the network bandwidth. For sure, this bandwidth is saturated when the size of the message exceeds the network performances.
So the main idea is to reduce communication calls by regrouping data in a few number of messages, but sending packed data if the message sizes are too large. Another way to optimize the communications is to reduce the data needed for computations. Finding and using algorithms that need less external data is another way to reduce communications because any gap in a message can be replaced by needed data used later or for another operation.

The advice is thus to minimize the amount of data needed for computations and to gather a maximum amount of data in a few number of messages. On the other hand, this advice has a high impact on the software design and on the data representation.

Use universal and primitive datatypes.

A special attention must be paid to data representation and data types. The reason is that it impacts communication latency, data copies, computations and data access, vectorization capabilities and efficiency.

The data representation highly depends on the problem and on the targeted machine's architecture. To be more concrete, an attention must be paid on the size of a data structure and also on the regularity of accessing multiple data sets. In a SIMD model (**S**ingle **I**nstruction **M**ultiple **D**ata), the model executes a few number of instructions on multiple data sets. These data sets can be represented with different data types and structures. An instruction can be vectorized and easily optimized if these multiple data sets are encoded in the same way and with the same size. A well known application in computer sciences is to align data structures to the size of a page or a block. This way the instruction pointer does the same jumps in memory. In parallel computation, this principle is also good to applied for the same reasons and adds to the efficiency of communications and vectorization.
For particle tracking, this principle can be applied to communications to accelerate data transfers mostly on particle and mesh movements through the global network. The second optimization observed in a parallel context is the improvement of the vectorization. Some computations on particles can be vectorized depending on the size of a vector unit and the number of particles. In order to accelerate these computations, one can modify the size of a particle or the number of particles per instruction block.

Due to the usage of several protocols for communications, work distribution and due to heterogeneity of targeted parallel machines, primitive data type are also a good way to be efficient and adaptable to multiple data representations. For example in our case the data transfers are only done with bytes representation in order to send mesh data, particles data and internal data using the

same protocol. It means that communicated data has to be encoded with types of size of a byte. In current languages, this can be done using unsigned char data types.

Data representation and data types usage is important for the global performances of the application. Developers really have to take care of it in the implementation of a library or any software.

Ordering, arrange and precondition data to converge to an ideal case.

Many good practices are based on data, may it be on data representation, data flow and movement or on data computation. These good practices are used to be close to the ideal case an application is built for. The more a case is close to this ideal case, the more the application is efficient to compute the case. However preconditioning data sets is an operation that can be very expensive in terms of execution time. This preconditioning operation can include communications on a wired network, data copies and imports from a memory space to a lower cache level, data gathering for cache blocking and per block computations and data re-ordering.

Data re-ordering and arrangement is one of the rules that can be very efficient and that can have a more durable impact on performances. In fact, data arrangement impacts several steps and operations during the runtime, from data communication to complex computations. A concrete application of this principle applied in particle tracking is the reordering of particles. Particles are sorted according to their position. The more the particles are close to each other, the more the engaged data are close in memory. This improves the spatial an temporal localities. Spatial locality is improved because the close particles use close data sets in memory. Therefore, references, instruction pointers and addresses are close and do not need a lot of computation. In addition this can allow vectorization in accepting architectures.
Temporal locality is improved by data rearrangement because close data can be reused on different time steps or different operations that follow each other. For particle reordering, the temporal locality is improved when two particles are in the same location and use the same datasets. Tracking the first and the following particles then uses the same data sets and no data movement or reference computation is needed.

Generally, data arrangement is a principle that has a real impact on the entire runtime. On the other hand, arranging data can be a very complex operation. For example particle sort is completed with a complexity of $O((n_{Particles}) \times log(n_{Particles}))$ with $n_{Particles}$ corresponding to the number of particles with a quick sort and $O(n_{Particles} + n_{buckets})$ with a bucket sort of $n_{buckets}$ buckets. That is why a preconditonner is more efficient if it has an impact on the maximum amount of computations. If an expensive preconditioning method is only efficient for a small number of operations of the simulation or for a reduced number of iterations and time steps, then this preconditioning method can not be qualified as useful.

A special attention must be paid to the complexity of the method and its temporal efficiency.

3.5 Evaluated metrics

There are several metrics evaluated, the most are direct metrics like execution time and memory occupancy. Three calculated metrics are also used: speedup, parallel efficiency and distribution quality. The distribution quality has already been defined but its definition is recalled here:

$$Q = 1 - \frac{n_{external}}{n_{total}} \tag{3.1}$$

where Q the distribution quality, $n_{external}$ is the number of imported cells and n_{total} is the number of imported and local cells on the memory's process.

The speedup of an application is defined as the relation between an old quantity and the newest. For example, the speedup calculation applied to execution times is defined as this :

$$Speedup = \frac{T_{old}}{T_{new}} \tag{3.2}$$

where T_{old} is the old execution time to compare with the new one represented by T_{new}. In a parallel context, the speedup can represent the scale up obtained by the parallelization. This metric is obtained with the same formula but where T_{old} is the sequential performance and T_{new} is the parallel performance.

Applied to parallelism, the third metric is the parallel efficiency of a software seen as this :

$$Efficiency = \frac{T_{seq}}{T_{par} * nProcs} \tag{3.3}$$

where T_{seq} is the performance of the sequential run, T_{par} is the performance of the parallel run and $nProcs$ is the number of processes or the number of computing units used in the parallel run. This metric gives a percentage to represent the parallelism quality.

3.6 Technical details about structures and implementation

This section is a brief talk about data structures used during the implementation of this library. The previous section gives some advices in order to implement and adapt efficient code in a parallel context. Obviously, we have tried to follow these tips for the creation of the library.

During particle tracking, the input data are the mesh (cells, faces, vertices, flow field, boundaries, ...) and the particles coordinates and direction vectors. Entities that must be modelized are coordinates (vertices, points and vectors) and lists of elements with correlations (the connectivities between elements).

There are two well known ways to store coordinates and vectors. The first one consists in storing the coordinates of elements one after the other. In a single array, each element members are stored in a contiguous way. For example in a 3D coordinates element set, the first three elements of the coordinates array correspond to the coordinate of the first point, the three following correspond to the

three coordinates of the second point, and so on. This storage method is so called array of structures (AOS) [55]. The advantage of this storage method is the fact that a single object or block is stored in the same area in memory. This improve spatial locality. The drawback is that vectorization is difficult with this shape as members of objects are not contiguously stored.
The second method used to store data is named SOA for Structure Of Arrays. This method consists in storing object elements continuously in multiple arrays. This design is very useful to vectorized and SIMD operations but needs more cache loads for more general operations on data. The following code presents an example of AOS and SOA designs.

```
typedef struct {
  float x[N], y[N], z[N];
  int icell[N];
} SOA_3DCoordinates;

typedef struct {
  float x, y, z;
  int icell;
}AOS_3DCoordinates;
```

For the implementation of the library, the array of structure pattern has been chosen because of the simplicity to implement memory management. In fact, the library is designed to localize, move particles and also balance workload and communicate particles. Communicating arrays of structures is more efficient for message passing protocols and communications occupy a very important part of the computations. Another reason for this choice is the fact that particle localization needs a lot of data from the mesh, the vertices, the cells, faces planes and normals and object connectivities. This kind of operation is greedy in terms of memory accesses and using a lot of different data sets is not shaped for vectorization. On the other hand, an AOS design is more likely to be efficient because of the spatial proximity of elements in memory.

Structures from mesh data are also designed with AOS format as it is a often a format to describe finite elements. Other format exist [81] and are more adapted to more specific mesh forms like face set based meshes for example that are specific to triangle faces. But the main advantage of AOS structures is the compatibility with many meshes. Briefly, the structures can be described with a couple of arrays. One array describes the element and its members and the second array stores the addresses to access to an element and its member. This design is well known to store compressed sparse matrix (CSR and CSC) [82].
These two formats are used for all the library and input data is assumed to coincide with these formats.

Concerning the data types used to represent the mesh, there are two ways to store information : input mesh is assumed to be described with primitive data types (integers, floats, double, boolean, ...), and these data types are described using template structured and functions. Most of these oper-

ations require two types, unsigned integers and signed reals. In order to be compatible with multiple architectures, template functions and structures are instantiated for 32 and 64 bits. The library can eventually be improved by instantiating for extended and smaller precision.

As the mesh data is imported and will probably be used after going through the particle tracking library, the library must not modify any data from this mesh. For this reason, the required data from the mesh is copied in order to be managed, transformed and distributed. Thus, cloned mesh data can be stored with specific data types. In a parallel context, communications are frequently computed and in order to accelerate and facilitate communication implementations and flexibility, a unique primitive data type is used to store mesh data copies. In fact, important data from the mesh is copied and stored in the form of a single array of bytes. This way, memory management, data exchange and communications are very easy to implement and do not depend on the original data types and problem precision. The difficulty lies on the data access performed by interfaces. A mesh partition is internally modelized using the same data format which is similar to a compressed sparse matrix, but all the mesh partition data is stored in a single bytes array. A second array stores the addresses to access the different elements in the array of bytes.

The data of the mesh and geometric entities inside the library is represented with the same method of modern meshes. Figure 3.3 gives an example of the storage methods used in the library.

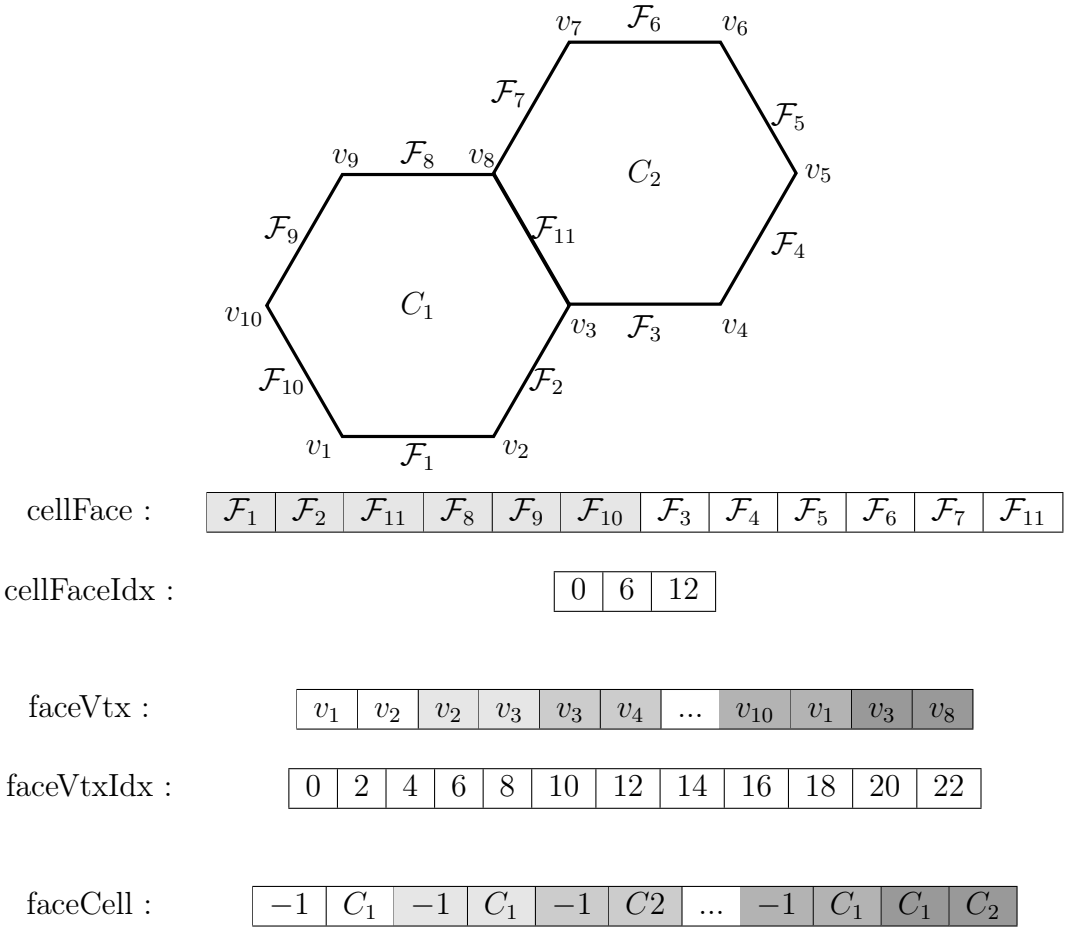


Figure 3.3: Representation and storage method of mesh data.

In this chart, 2 mesh cells are modeled with 11 faces and 10 vertices. In this example, the two cells

are connected together and also with the different geometric objects. C_1 and C_2 represent the cells, $\mathcal{F}_{1-11}$ represent the faces of the mesh and v_{1-10} are the vertices.
To connect these geometric entities and to represent their relations, connectivity arrays are used in the form of two distinct arrays: the first array ($cellFace$, $faceVtx$) stores the data. This array gives the other objects to which the object we are looking is connected. To give an example, $cellFace$ gives the faces connected to each cell. According the cellFace array of the chart, the cell C_1 is connected to the faces $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_{11}$, $\mathcal{F}_8$, $\mathcal{F}_9$ and $\mathcal{F}_{10}$. The second array is an index array. It represent the virtual adresses to access the data connected to an object. With the same example, the faces connected to the cell C_2 are given by *cellFaceIdx[*C_2*]*. This array *cellFaceIdx* is of size $nCell + 1$, with $nCell$ the number of cells in the current mesh partition. Thus, the faces connected to C_2 start from the index *cellFaceIdx[*C_2*]*$= 6$.
The advantage of this format is that the access to a set of data is direct and contiguous. In addition, the index array implicitly stores the number of entities connected to an object. Indeed, the number of faces connected to the cell C_2 is equal to *cellFaceIdx[*$C_2 + 1$*]* - *cellFaceIdx[*C_2*]*. This is the reason why the size of this array is n_{object}+**1**.
The same storage method is applied on multiple object in the mesh.

The third array represented in figure 3.3 is *faceCell* which represent a different kind of data. This array gives the different cells connected to each face. It is assumed that, a face is connected to a maximum of 2 cells (1 or 2 cells, a non connected face is not taken into account). Each cell correspond to a "*side*" of the face, and this "*side*" can correspond to the number of a cell, which is an integer, or can have a value out of the range of the cells numbers. Here this special value is set to -1. This special value defines the boundaries of the mesh partition. It can be a wall, an obstacle, or the end of the space discretization and the solution range.
This array is of size $nFace \times 2$, where $nFace$ corresponds to the number of faces in the mesh partition. This size is fixed for the single reason that a face is considered as possibly connected to 1 or 2 cells. For this reason there is no need to implement an index array as the "*sides*" of a face $\mathcal{F}_i$ are accessible with *faceCell*$[\mathcal{F}_i \times 2 + \{0|1\}]$.

3.7 Conclusion

The architecture of ParOPTIC's library is designed in order to take into account the high parallelism capability of particle tracking problem. This is finally a problem that is highly parallel if particles are well distributed, that has a high re-use potential in the way that all particles are tracked using the same operations, and that can be easily adapted to heterogeneous machines. The component-based library corresponds to these qualities as this approach allows high potential of re-usability, it allows high potential of parallelism due to the independency of the components, and the component-based design is easily adapted to heterogeneous machine due to the same quality.
In the next chapter, the algorithms that compose the components are described.

CHAPTER

4

Optimization of a set of basic algorithms.

Contents

4.1 Introduction

As shown in the graph of tasks, particle tracking in a meshed environment is basically a couple of operations which are particle localization and particle movement. These operations are very basic geometrical operations scientists have been working on for centuries. The computation of these operations has been studied for many cases from visualization and light modelization to the treatment of boundary conditions in CFD simulations. This chapter is dedicated to the implementation, optimization and analysis of these operations used to localize and move particles in very large meshes.

4.2 Particle Localization.

4.2.1 Particle Localization in a single Cell.

This operation can be done at different grains, at the scale of a single cell, a set of cells and set of mesh partitions but these algorithms depend on the algorithm used to determine if a particle is inside a cell or not. This is also known as a containment problem which is the subject of many studies [83, 84, 85, 86]. Our study is not about the different methods and problems linked to the containment problem but a focus on two methods that are highly used to determine the position of a point compared to convex polyhedrons and polygons [47, 87].

The first method to determine if a particle is inside a polygon (or a polyhedron) consists in computing the plane equations of the polygon's faces (or polyhedron's faces) and to determine, using the face's normals, the position of the particle compared to each face. In other words, this algorithm consists in determining on which side of the plane the particle is.

The advantage of this algorithm is that it is relatively efficient for regular cell shapes. In fact, this algorithm is quite efficient on convex regular cells such as tetrahedrons and cubic cells. On the other hand, it does not work at all on more complex cells such as concave cells.

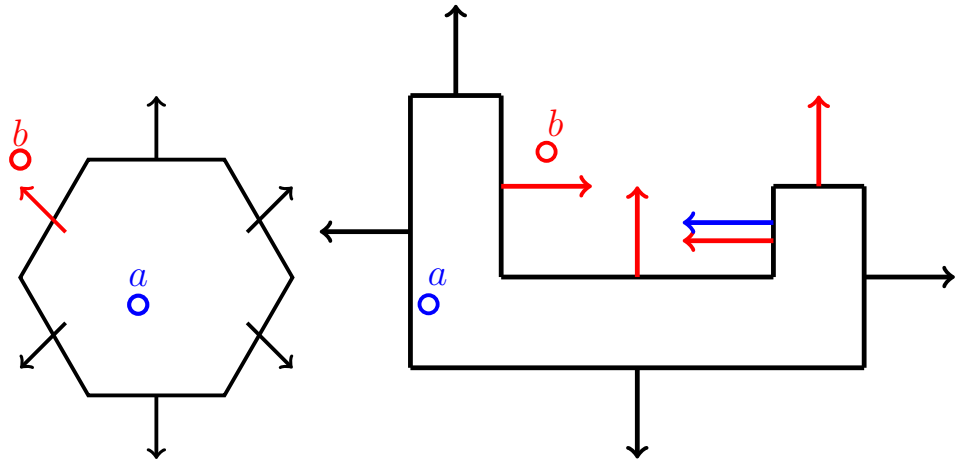


Figure 4.1: Example of the application of the first method to determine if a polygon contains a point using face positions. On the left, two particles a and b are localized using the outsiding normals of the convex polygon's faces. On the right the same particles are localized in a non-convex polygon. Colored normals indicate that the particle of the same color is outside of the polygon.

The figure 4.1 shows an application of this first method on two different shapes of polygons: the first polygon on the left is a convex polygon, the second one, on the right is concave. In both cases, two particles a and b are localized and a is localized inside the polygon whereas and b is outside. This first method consists in determining for each face of the polygon if the vector from the particle to its orthographic projection on the current face has the same direction than the outsiding face normal. In the figure 4.1, colored vectors correspond to vectors that are not in the same direction that the projection vector from the particle. In other words, colored vectors indicate that the particle (of the same color) are not inside the polygon.

In the first case, on the left, which corresponds to localizing particles in a convex polygon, a and b are both correctly localized, a single face normal is not in the same direction than the projection

vector from b, so b is calculated as being outside the polygon, which is currently right.
On the other hand, this method does not correctly calculate the position of the particle a in a concave polygon on the right as one of the face normal indicates that a is outside of the polygon.
So this first method is not adapted to irregular meshes and cells with non-convex shapes.

The second method to determine the position of a point in relation to a polygon or polyhedron consists in firing a ray from this point and count the number of faces of the polyhedron this ray intersects. This method has the advantage to be efficient with irregular geometric objects like concave polygons. On the other hand, this method requires the computation of ray-planes intersections which is very greedy in terms of computation time. Another disadvantage of this method is that all intersections with all the faces of the cell has to be computed, so no optimization can be done by computing a reduced number of intersections. The reason of this last disadvantage is that the result depends on the number of intersections, if the ray from the point intersects an odd number of faces, the point is then inside the object. Otherwise, the point is outside of the object.

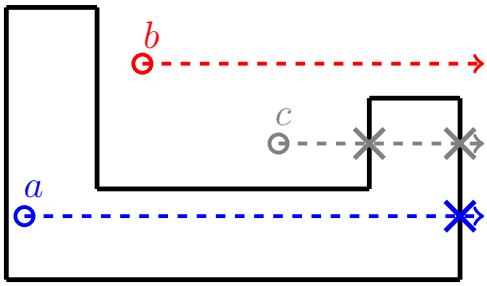


Figure 4.2: Example of the application of the second method to determine if a polygon contains a point using rays.

Figure 4.2 gives an example of this method applied to three points. In this example, we have to determine if these points are inside the drawn polygon. To do this, a ray is traced with a random direction (here the direction is set to a unit vector of coordinates (1, 0, 0, ...)). The three points a, b and c have to be localized using this method. By counting the number of faces each ray is intersecting, we can figure out that the ray from point a intersects only one face, the ray from b does not intersect any face and the ray from c intersects 2 faces. So the rays from b and c intersect a even number of faces whereas the one from a intersects an odd number of faces. With this parity, we can determine that a is inside the polygon, whereas b and c are outside.
The algorithm of this method is proposed in 1.

We encountered a particular case when this method does not work and that is discussed by Hormann et al. [87] and Kalay [47], the point localization can not be determined when the ray intersects the faces and the intersection point is a vertex of the polygon. Figure 4.3 presents this particular case.

In figure 4.3, we retrieve two points that have to be localized. Rays are launched from these two points and unfortunately these rays intersect some faces at a vertex of the polygon. The ray that comes from a intersects faces p and q at the common vertex of both faces. According to the algorithm 1, a is considered as being outside of the polygon which is completely wrong. The same

```
nb_intersections ← 0;
for iface in polygon do
    is_intersecting ← intersects(iface);
    if is_intersecting is true then
        nb_intersections ++;
    end
end
if nb_intersections is even then
    /** the point is outside of the polygon **/
else
    /** the point is inside of the polygon **/
end
```

Algorithm 1: Localization of particle in polyhedra using ray/plane intersections.

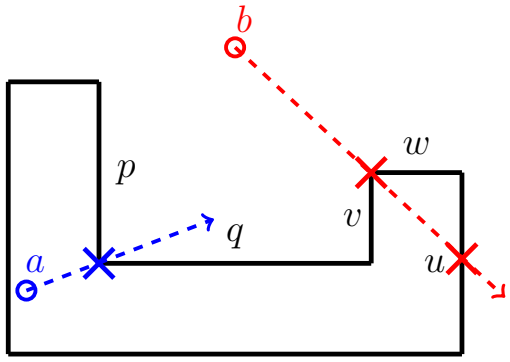


Figure 4.3: Example of the application of the second method to determine if a polygon contains a point using rays.

effect is noticed with point b, its ray intersects 3 faces u, v and w. b is then considered as being inside the polygon.

It can be noticed that if we consider a ray intersecting a vertex as a not intersected face, the problem still remains.

Two main solutions can be carried out, the first consists in launching several random rays from the point and hope the problem will not persist with the new ray. This first solution has the advantage of not being intrusive if the problem is not encountered but has the drawback to force the process to restart until the localization is rightly determined. The second solution consists in determining a ray that is surely not passing through a vertex of the polyhedron. This solution has the advantage to be easily implemented and to give an instant right result, but on the other hand, requires to compute a ray for each particle. This ray has to be computed dynamically.

The computed ray is determined as the one that goes to the barycenter of one face of the cell. The drawback of this solution is that if the barycenter of the face is outside the face, in the case of a concave shape of this face, launching a ray to this external barycenter will not work properly. That is why during the face loop, for each face of a polyhedron, the face is divided into triangular faces. The problem is that even with all these computation, when a ray is passing through a face barycenter and hits other faces at a vertex, the localization can not be determined. Another computation has to be made. The final test to add is to count the number of vertices that are hit. If this number is greater than 0, in other words, if the ray hits any vertex or edge of the polyhedron, the ray has to

be changed.

The next ray corresponds to the ray hitting the barycenter of another triangular face. The algorithm is given in algorithm 2

```
/* Faces are triangles */;
nb_intersections ← 0;
nb_rays ← nb_faces;
/* compute the different rays, an additional random ray is added at the end */
for iray in nb_rays do
    /* while a right ray is not found */
    ray_vector ← compute_ray_to_face_barycenter(iray);
    nb_intersections ← 0;
    for iface in nb_faces do
        intersectionPoint ← intersects(ray_vector, iface);
        if intersectionPoint exists then
            nb_intersections ++;
            for iborder in iface do
                if intersectionPoint ∈ iborder then
                    nb_intersections ← 0;
                    /** Stop and change the ray "iray" **/
                end
            end
        end
    end
    if nb_intersections > 0 then
        /* We found a good ray and the number of intersections */ iray ← nb_rays;
    end
end
if nb_intersections is pair then
    /** the point is outside of the polygon **/
else
    /** the point is inside of the polygon **/
end
```

Algorithm 2: Localization of a particle in a polyhedron using ray/plane intersections taking into account singularities.

The algorithm written in algorithm 2 can determine precisely when a point or a particle is inside a cell or not. In the simulation field, meshes are more often composed of several cells.
To localize a particle or a set of particles in a set of cells, the first method that can be implemented is to iterate over the cells and for each cell, determine if a particle is contained in this cell. This method is very expensive as the number of cells grows. In addition the complexity of algorithm 2 is very high and is about $O(n^2 \times m)$ for a single particle where n is the number of faces of the cell and m the number of vertices of a face which is 3 for triangles.

4.2.2 Particle Localization in a set of cells.

Localizing a point in a polyhedron is a very expensive operation as it requires several singularity checks and intersection computation with all faces of the polyhedron. This operation is then expensive in terms of computation time and also intensive in terms of number memory accesses. The main idea from multiple authors is then to reduce the number of cells to check to find the final one in which the point is localized. Several techniques have been experienced, some consists in choosing a subset of candidate cells by space discretization. The point is first localized in a coarse grain subset of subspaces that contain a limited number of candidate cells. Another technique consiste in localizing the point in approximated cells. The candidate cells are approximated with a bounding box or a regular shape to select a limited number of candidates.
We implemented a method that reduces drastically the number of candidate cells. The idea is to select a subset of cells due to their proximity to the particle. To build this subset, the vertex of the mesh that has the minimal distance with the particle is determined. This algorithm reduces the candidate cells to the cells that own the vertex closest to the desired point. It also has the advantage to be very easy to implement and is presented in algorithm 3.

ivtx_tmp $\leftarrow$ $ivtx_1$;
minimal_distance $\leftarrow$ distance(particle, $ivtx_1$)
for *ivtx in mesh* **do**
  if *minimal_ distance* $>$ *distance(particle,* $ivtx_i$*)* **then**
    ivtx_tmp $\leftarrow$ $ivtx_i$;
    minimal_distance $\leftarrow$ distance(particle, $ivtx_i$);
  end
end

Algorithm 3: Determination of the minimal distance

Obviously the minimal distance is computed not using a square root which is a very expensive operator. As the accurate distance is not required, only an approximation is made to compare with other vertices, this distance is calculated with

$$distance_{AB} = (x_B - x_A)^2 + (y_B - y_A)^2 + (z_B - z_A)^2 \tag{4.1}$$

With these two algorithm executing at different scales, a set of particles can be localized by first determining a subset of candidate cells and in a second time determining among this subset of cells the one that contains the particles. This method at this scale has the advantage to be more efficient than a simple iteration over all known cells, but has the same drawback as the first method: if the number of cells and the number of vertices grows, even the computation cost of the distance can become very greedy and inefficient. The reason is the same, increasing the number of candidate vertices takes more time to compute.
In order to reduce again the computation time of particle localization in large meshes, another localization method can be added at the scale of a mesh partition.

4.2.3 Particle Localization in a large local mesh.

Particles can be localized in mesh blocks in order to reduce the local size of the mesh. The method used consists in localizing particles in a secondary mesh in the form of multiple implementations with advantages and disadvantages. Most of the implementations are based on tree structures. The idea is to transform the involved graph by discretizing nodes or gathering some of them to manage the accuracy of the search operation. More specifically, some close cells are regrouped in the same structure object that is used to reduce the search area of a localization.
There are multiple implementations of tree constructions from kd-trees to octrees and binary trees. Cartesian grids can be considered as trees, indeed the tree is single rooted which is the unique node with multiple leaves that represent the cells of the cartesian grid.
All the tree shapes have advantages and disadvantages depending on the memory occupancy, the search complexity and the parallelization capacity.
In a parallel CFD application, the computing mesh that simulates the fluid phase, is parallelized using partitions. These partitions are distributed on the different processes of the target parallel machine. The adjacent graph modelizes the processes and the partitions, a node corresponding to a partition or to a process.

Figure 4.4 gives an example of a structured grid on the left and a tree (a quadtree) on the right to localize particles in a large mesh.

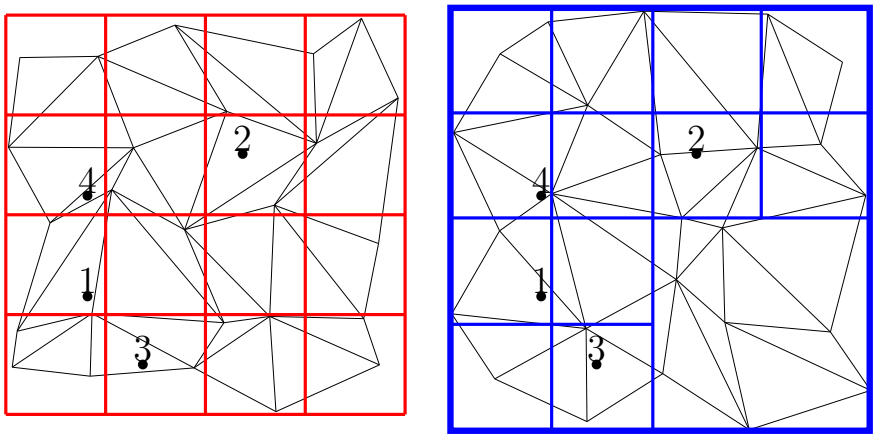


Figure 4.4: Cartesian grid overlapping a mesh (left) and quadtree(right).

To implement a tree to localize a set of particles has multiple advantages. First of all, a tree structure is very light to store. Another advantage of a tree structure is the complexity to access a leaf. This complexity is about $(log_2(n))$ with n the number of steps in the tree. In addition, according to figure 4.4, the quadtree requires less leaves to store because of the localization of the particles. Indeed, a region that does not contain particles or fluid is not important to store in the tree. In other words, an empty or less dense region can be less accurate for the search operation.

The structured grid has almost the same advantages such as the memory occupancy that is very light or the complexity to access a leaf or a cell of the structured grid. The time to access a cell with a structured grid depends on the number of directions of this grid but the time complexity is

constant. In fact, the access to a cell is done with a single division per direction:

$$i_c = \lfloor (\frac{(x_p - x_{min})}{(x_{max} - x_{min})} N_x) \rfloor + 1 \tag{4.2}$$

with x_p the particle's coordinates we want to localize, N_x the number of cells of the structured grid in a direction and x_{min} and x_{max} the minimum and maximum coordinates of the structured grid. This formula gives i_c which is the number of the structured cell in a direction where the particle is.

To compare the two localization methods, we can notice that the localization in a tree structure is more complex than localizing in a structured grid. On the other hand, the tree can perform localization more precisely than a static structured grid, as the constructions of leaves in a dynamic tree can be more accurate (due to regions with less density) than a static structure as structured and cartesian grids.
Figure 4.5 gives an example where the localization can be more precise with a dynamic tree structure. In this example, in terms of space occupancy, the structure is building more leaves than in

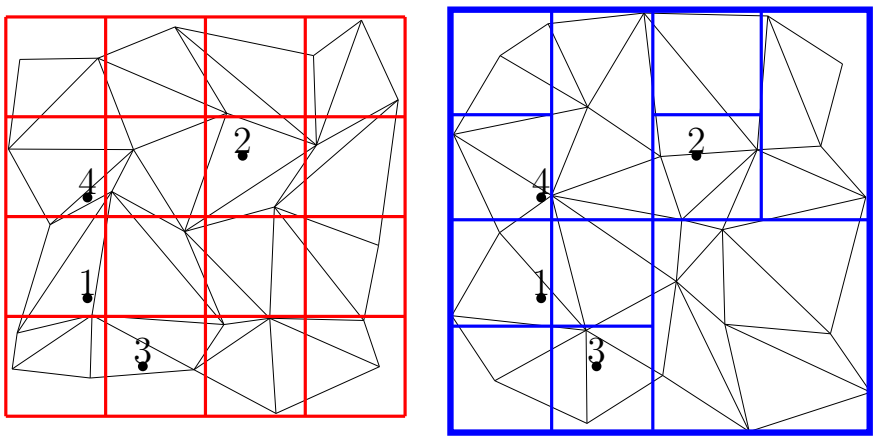


Figure 4.5: Cartesian grid on the left and Kd-tree on the right. Leaves of the Kd-tree are built according to the position of particles or the density of the fluid computation.

the equivalent tree. In addition, the tree is more accurate to localize these particles in the same area. This is due to the dynamic construction of a tree structure, in fact a tree node can be built instantly during the run of the localization step in order to be more precise by reducing the number of candidates cells in the computing mesh. In this case, the structured grid needs to build all cells (corresponding to the leaves in a tree structure) to localize particles anywhere in the computing mesh partition.
In our implementation, the structure we have chosen to localize particles at coarse grain, is the overlapping structured grid for the following reasons: the light space occupancy required, the complexity of the localization and the static construction of the grid. In fact this default of the structured grid is an advantage for particle tracking simulations because particles travel all over the computing mesh and they do not guarantee to be close during the simulation. In addition, linking leaves and structured cells to the computing mesh is quite expensive in terms of computation operations and number of memory accesses. This will be discussed in chapter 5 which is about particle tracking on

massively parallel systems.

4.2.4 Performances obtained.

This section presents some performances obtained for particle localization which is the most expensive operation for particle tracking. The results are obtained by localizing $640,000$ particles with random coordinates. Only the localization step is measured depending on the size of the overlapping structured grid.
The computing mesh is a structured mesh with $256,000$ cuboids and is not parallelized. The sequential performances are studied. Figure 4.6 gives the speed-up obtained by increasing the size number of the overlapping structured grid.

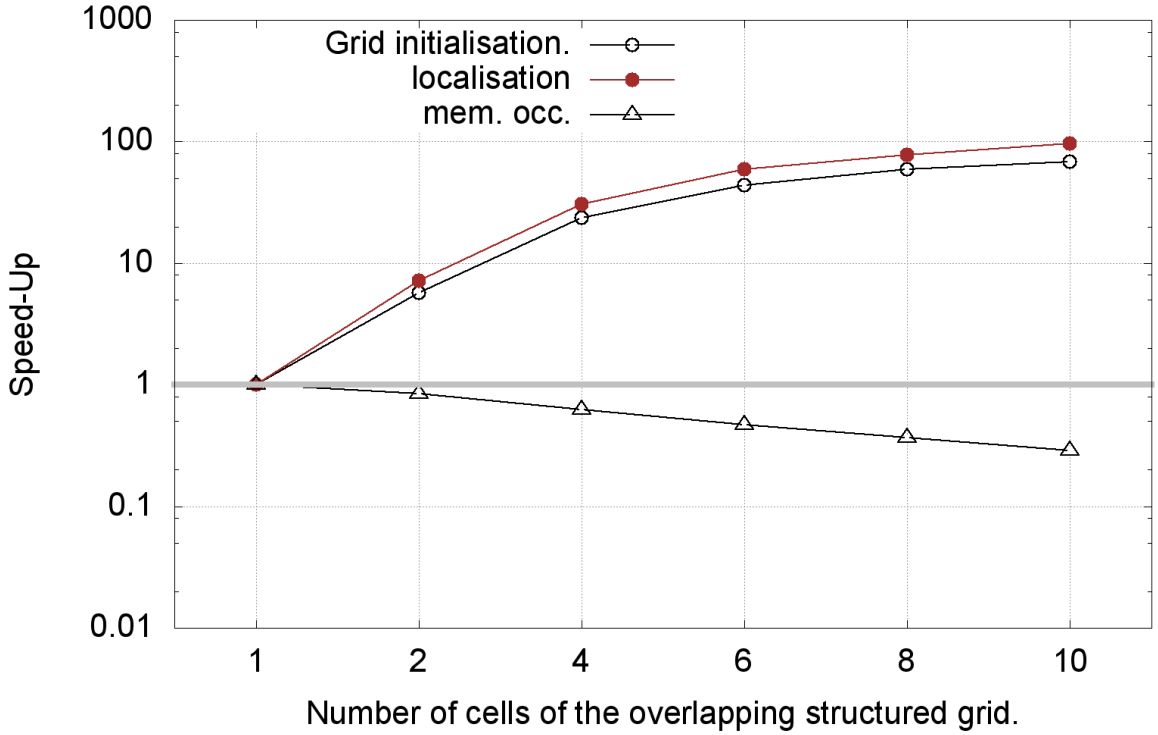


Figure 4.6: Speed-up obtained to localize particles

In the figure 4.6 the speed-up is calculated comparing a run without a structured grid to localize particles and different discretization of an overlapping structured grid.
The increase of the size of the structured implies that the number of cells of the original computing mesh per structured cell decreases. This directly impacts the grid initialization and the time to localize particles. In fact the more the grid is discretized, the less there are computing cells per box and the higher is the speed-up.
The maximum speed-up obtained for the execution of the localization and the initialization of the grid is obtained for the maximum size of the structured grid and is equal to 95 for the localization operation and 68 for the grid initialization. It means that for an overlapping grid of size $10 \times 10 \times 10$, localizing particles in a computing mesh of 256.000 cells is close to 100 times faster than localizing particles without a grid.

On the other hand the memory occupancy increases with the size of the structured grid. It is due to the creation of ghost cells explained in the next chapter that concerns the particle operations in a parallel context.

4.3 Particle Movement.

The movement of a particle is a simple operation to compute if all data are available. In fact, the movement of a particle is computed using equations of motion [88], although these equations depend on the underlying simulation, the computation can be summarized as the sum of all forces applied to the droplet.
This operation is then quite easy to implement and to compute, the only difficulties come from the methods to gather data from the mesh. As we are talking about these methods, it can be noticed that there are many methods depending on the computation the flow field. But these methods do not concern the particle localization and tracking.
The fact that gathering forces applied to particles does not concern particle tracking computation and the fact that this operation is done by the fluid computation does not mean that *ParOPTIC* does not have to take care about the fluid phase. This is discussed later in the flow field section.

As a particle is moving, it is possible that this particle leaves the current cell of the computing mesh where the particle was localized before its move. Contrary to the particle movement computation, determining the localization of the particle after its movement can be an additional difficulty during the runtime.
In order to perform the particle localization, the previous algorithms can be used, they are quite efficient and particularly accurate. On the other hand, these previous algorithms remain very expensive in terms of computation time and complexity. This is the reason why, these algorithms have to be called the least possible. Other algorithms to localize efficiently with the available data during the runtime are then used.

The first algorithm that can be used is to use previous algorithm with a neighborhood visit. Previous algorithms can be used to determine the cell of the computing mesh described previously named the *in-cell* test. As the use of this algorithm highly depends on the number of candidate cells, during the runtime, and the previous location is still known, the number of candidates can hardly be reduced. Candidates during the particle movement are determined by the proximity of these candidates with the current particle location. In other words, this first approach consists in visiting neighboring cells of the current particle's location, moving the particle and determining the next location by visiting the neighboring cells. Figure 4.7 shows an example of this approach. The algorithm is presented below (Algorithm 4).

In figure 4.7, two particles are tracked : P and P'. P is moving from cell C to a neighboring cell. For the localization of the next position $P+1$ of particle P, the idea is to compute the in-cell test

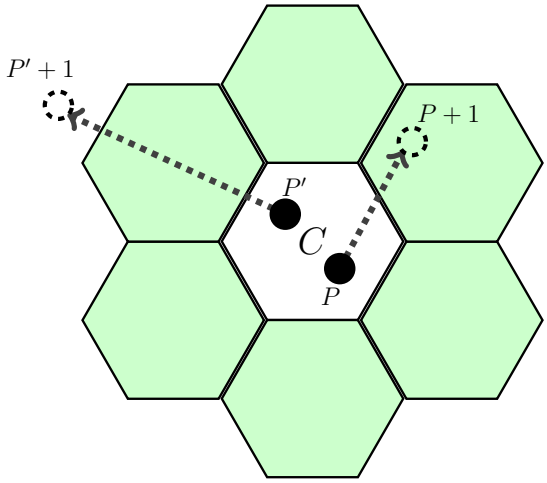


Figure 4.7: Particle localization using $in-cell$ test and neighboring cells visits.

```
/* The current localization of the particle is known*/
begin
    /*particle movement*/
    for ivtx to visit do
        for icell in visiting cell containing ivtx do
            if icell has not been visited then
                if localize_particle_in_cell() is true then
                    /* The cell is found -> Stop*/
                    stop
                else
                    for ivtx2 in icell do
                        ivtxToVisit ← ivtx2;
                    end
                end
            end
        end
    end
end
```

Algorithm 4: Particle localization using $in-cell$ test and neighboring cells visits

for all neighbours of C. P is then localized in a maximum of 6 iterations in this example.
The other particle P' goes faster or further than P. P' is so fast that it leaves C and its directly neighboring cells. As for a naive path finding algorithm, this particle localization algorithm visits and checks the direct neighbors of C, then the next neighbors of the direct neighbors and so on.
In the end, if the particle is not in any cell of the mesh partition, the particle is declared lost and has to be relocalized in the grids.

The example presented in figure 4.7 shows the advantages and drawbacks of this approach. The advantages are in fact the reusability of an already used algorithm and so the ease of implementation. Another advantage is the very low number of candidates to visit in order to determine the new location of the particle. On the other hand, a drawback is immediately linked to the low number of candidates. The fact that the number of candidate cells is low, is a consequence of a particle that moves slowly. In the case of a particle at very high speed or in the case of a very precise mesh

discretization, the particle has higher chances to be localized further than the closest neighboring cells. So, in this particular case, the number of candidate cells is considerably higher which has a real and high impact on the performances. It can be noticed that in this particular case, the precision of the simulation is directly impacted too as the particle that moves further than the closest cells is not integrated and does not take into account the field of the forgotten cells. In simulation field some time integration methods are used to associate time and space discretization, Runge-Kutta methods and Euler schemes [89] can be given as examples. These methods can also be used in particle tracking but still do not guarantee that no cell will be forgotten. In the example of figure 4.7, P' direction and speed does not integrate the velocity modelized by the direct neighboring cells of C as P' passes through these temporary cells.
In addition, regarding the algorithm 4 the algorithm complexity is around $\mathcal{O}(n \times m)$ if the most internal loop is considered as negligible compared to both other loops and where n is the number of vertices per cell and m is the number of cells containing a vertex.

Regarding these drawbacks, we looked for another approach to be more efficient in terms of computation time and complexity and in terms of precision. The authors Haselbacher et al. [44] developed an algorithm able to localize particles with more precision, with a single memory access and with less memory occupancy. The algorithm consists in memorizing the connectivity between faces and cells of the computing mesh and determining the face the particle trajectory is going to cross. A face can be connected with any number of cells greater than 0. In our implementation, the case where faces are connected to 1 and 2 cells is only supported.
So with the information that a face is connected to one or more cells, a connectivity association can be set in order to determine the number of cells connected to each face. This way, the intersection point between the particle path and the faces of the current cell is computed and determined. If the particle is not moving enough to cross the face, the particle stays in the current cell. Whereas, if the particle crosses a face and is going to leave the current cell, then the cell is updated and the particle's path is computed until the time step is finished. Figure 4.8 shows an example of this algorithm. The final algorithm is written below in algorithm 5

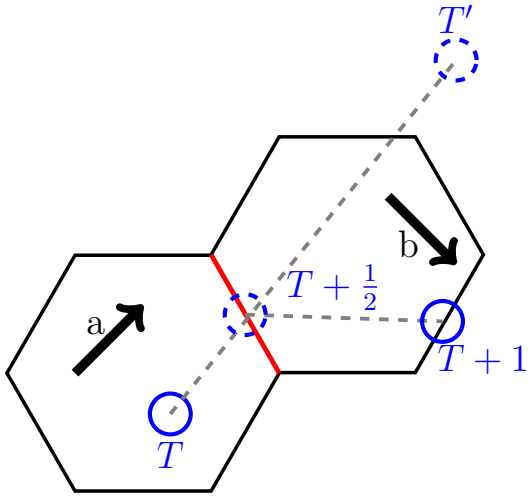


Figure 4.8: Particle movement and localization by computing particle-face intersections.

In figure 4.8, a particle is moving and is leaving its current cell. In this example the ray/face

```
/* The current localization of the particle is known*/
begin
    /*particle movement*/
    d_min ← MAX;
    pt ← 0;
    for iface ∈ faces_currentcell do
        ptIntersection ← intersects(ray, iface);
        if ptIntersection exists then
            if d_min > distance (particle, ptIntersection) then
                d_min ← distance (particle, ptIntersection);
                pt ← ptIntersection;
            end
        end
    end
    /* pt is the intersection point */
    /* d_min is the distance between the particle and pt */
    return {pt, d_min}
end
```

Algorithm 5: Particle movement and localization by computing particle-face intersections.

intersection is computed and as it is shown in the figure, allows the field of the crossed cells to be taken into account. As it is equivalent to discretizing the particle's movement and the time step, the ray-face intersection is iteratively computed until the particles movement is finished.
The advantage of this algorithm is that no computing cell can be skipped and this renders the particle tracking more accurate compared to the previous one. Another advantage of this algorithm to be noticed is that the same algorithm of face/ray intersection is used. In fact, the only difference comes from the input data. The drawback is obviously the usage and the need of this face/ray intersection call. We have seen that this intersection operation is quite expensive in terms of computation time, but compared to the neighboring cells algorithm, the complexity is constant and equal to the number of faces of the current cell as the next cell is obtained with a single memory access. So this last algorithm has been chosen and implemented.

An important problem still remains, with the particle localization using ray intersections, the use of ray-face intersection for particle movement encounters the same singularities : problems come when particles leave their current cell and intersect an edge or a vertex of the cell. The result of this problem is that the edge and the vertex are possibly connected to more than 2 cells. Indeed, in the case of a complete face, this face is connected to two cells : the current one and the neighboring cell. If a particle intersects an edge or a vertex, the edge or the vertex is possibly connected to more than 2 cells and the more the cells the more the solutions, the more the solution is hard to find. In fact, it is very hard in our implementation to determine and choose the right cell, as all candidate cells are right solutions (an edge and a vertex can belong to more than 2 cells). A solution is to move further the particle and not take care about the intersection point and determine the final cell to localize the lost particles. We believe that a better solution can be found and it will be the subject of our future work.

4.3.1 Flowfield

The flow field that is in general represented by a velocity field is computed with different methods than particles. We have seen above in this manuscript that Eulerian schemes are often used to compute the fluid phase compared to the Lagrangian approach that is used to track particles.
As the objective of this work is to study algorithms used for particle tracking and to deliver an efficient library that performs these operations, we chose not to implement methods that solve flow velocities because of the large number of existing methods.
On the other hand, the flow field has to be represented in a reduced way in order to compute particles positions. As the particle movement depends on the forces applied on it, which are computed by the fluid phase, a general and independent representation has to be done to represent these forces. The flow representation must be independent concerning the vector computation methods and the number of forces and physical variables because of the existence of multiple physical models.
The choice has been made to store the flow field in the form of a unique velocity field. Our implementation does not store the physical quantities and the different forces for the reason above, but the final velocity field is stored in the form of an array of vectors. The velocity field stored modelizes the velocity of the fluid in the cells. The array that stores the forces is then of the size of the number of cells of the computing mesh. Updating this field is very easy in a shared memory context, the interface can easily allow to update the velocity field.
On the other hand two particles in the same cell can go to different directions in simulations that take into account momentum or that do not depend on the fluid (for example the light generation). For these cases the velocity of a unique particle is important and is computed outside the library.
The flow field member is very useful to manage velocity field of a partitioned mesh with a different distribution than the particle phase which is completely within our case of study.

4.4 Conclusion

This chapter describes the different algorithms used to compute particle tracking. These are very basic algorithms and have already been dealt with in the literature. Algorithms have been selected in order to be efficient in a sequential run, cheap in terms of data access and memory occupancy and that can be intensively reused. Indeed localization algorithm and particle movement (which corresponds to the particle localization during its movement computation) both use efficient ray intersections.
All these algorithms have been adapted for AOS structures to optimize vector operations on independent particles.
Sequential tests also show the performances of the algorithms and especially the performances of an overlapping structured grid that decrease the number of candidate solutions for expensive algorithms.
The next chapter talks about the algorithms, adaptations and performances for a parallel system and for distributed memory architectures.

CHAPTER

5

A Design for localizing and tracking particles on a remote memory space.

Contents

5.1 Introduction

In the previous chapter, a set of structures and algorithms has been discussed in order to localize and move particles in a particle tracking simulation with large meshes. In a parallel context, this kind of simulation is run on massively parallel machines which have multiple memory spaces dispatched

on an irregular network. As each node of this network has possibly different components, different architecture and different computing performances, each node is considered as singular.
At this point, implemented algorithms are efficient on a local and limited environment. This efficiency is essentially due to the way to parallelize particles on a system that does not need to import remote data from another memory space.

In order to study the different algorithms, communications have been implemented with message passing tools using MPI [73]. The global approach is reproducible with other communication protocols but the implementation and the different optimization of the code or the structures allocations are specific to MPI usage.

5.2 Particle Localization in a different mesh partition.

A mesh partition is defined as a set of vertices, faces, cells and other numerical entities and a structured grid defined earlier for particle localization at high scale. This grid overlaps any mesh partition as it is based on the maximum and minimum vertices coordinates. The partitioned mesh is then scattered on multiple remote memory spaces in the form of mesh partitions that are considered as local meshes.
These local meshes are used to localize and track our particles. When the particle or a set of particles are localized in a different mesh partition there are different solutions in the literature.

The first solution consists in sending the set of particles to the remote memory space where the mesh partition is stored and where the particles will be localized. This first solution is quite efficient when particles are equally, homogeneously dispatched in the global computing mesh [2, 90, 91, 92]. On the other hand, the particle tracking can become sequential if all or a majority of the particles to be tracked are localized in the same area (in the case of an injector for example) all particles are sent to the same memory space that belongs to a single node. With this approach there are no communication needs except for particles migrations which is achieved with high efficiency. The drawback is the risk of the simulation to turn into a sequential simulation or in the best of cases the system has an unbalanced and irregular workload during the run time.

A second solution is to store a copy of mesh partitions on every memory spaces of the computing system. This solution has the advantage to perfectly balance the workload, which corresponds to the particle distribution in particle tracking simulations. Another advantage is that it has an important impact on the entire simulation, as no communications are needed during the entire run in the case where the all partitions are copied on every remote memory space. The particles are perfectly distributed and no data transport is needed to keep the balance quality. This last advantage impacts a lot the entire simulation as it does not really need any barrier or any form of synchronization between the processes except in simulations where particles interactions are observed.
On the other hand, several drawbacks come with this approach. The first one concerns the memory

occupancy of the simulation. Scientists are interested in HPC and parallelism for two main reasons: the possible acceleration of their application and the possibility to run simulations on larger problems due to the storage parallelization. So storing several mesh partitions on the same memory space drastically limits the maximum size of the studied problem.
The second drawback also concerns an objective of HPC usage : parallelizing a problem generally accelerates the simulation. For the particle tracking, the problem does not exists as particles are still well distributed so the work is well balanced. The problem is the balance of the flow field computation. In fact, for simulations that use multiple phases (as Euler/Lagrange methods), the fluid phase is solved in parallel. Having copies of the solution of the fluid phase on all the memory spaces does not impact the parallelization of the fluid phase but it adds an update step. In order to keep the whole mesh up to date for all copies, the whole solution of the mesh has to be sent and received. This additional operation requires lots of communications with very large messages (the size of a mesh partition) with synchronization problems and message concurrencies.

A third solution is a mix of both previous solutions that consists in balancing and distributing particles and mesh partitions and proceeding to the transfer of subsets of mesh partitions, from 1 to a limited number of computing cells [20]. The extreme cases are obviously sending 1 cell and sending all cells of the mesh partition. This third approach has the advantages of the second solution that consists in sending the whole mesh partition to other remote memory spaces but may occupy less memory space as it becomes possible and reasonable to receive and remove subsets of mesh partitions when they are not used. Another advantage also due to the second solution is that a particle can be localized and tracked for several time steps depending on the size of the partition subset received. The larger this subset is, the less communication calls it needs, the second solution being the particular case where no communication is needed as all memory spaces possess all mesh partitions. There are unfortunately some drawbacks with this approach, the first one is the fact that a method to determine a subset of cells to be communicated has to be implemented. The second drawback concerns the need of synchronizations and frequent communications. In fact, this drawback depends on the particles behaviour, as particles can stay in the same area during the entire simulation. But in general, any time a particle moves and visits another subset of computing cells, this subset has to be called and received. This is a good agreement between particles and mesh balance.

This last approach has been chosen to localize and track particles in partitioned and distributed mesh over multiple memory spaces for these multiple advantages and the good agreement between memory occupancy and workload.

As we said earlier in the drawbacks of this hybrid approach, a method is needed to determine the number of computing cells by subsets, by boxes to communicate.

5.3 Particle Localization using a grid.

The adopted solution is the one that consists in communicating subsets of computing cells. This solution is presented in figure 5.1. This figure shows an example where a set of 8 particles are tracked. These particles are distributed as 4 of them (n^o1, 2, 3, 4) are stored in memory space ⓐ and the rest (5, 6, 7, 8) is stored in memory space ⓑ.

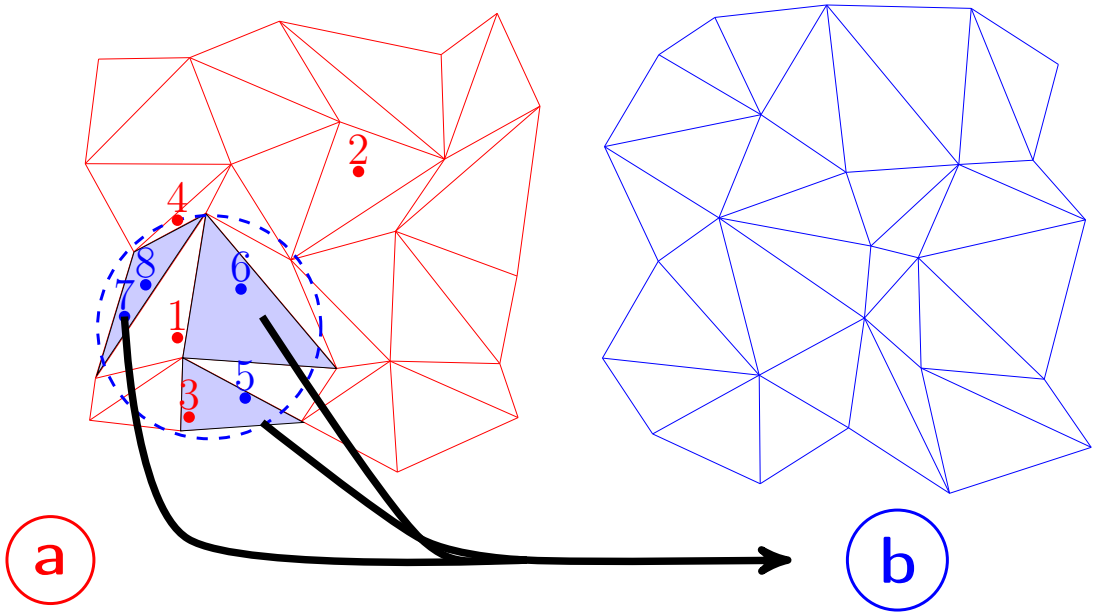


Figure 5.1: Example of cells imported and exported to localize and track removed particles. ⓐ and ⓑ are processes that own a different mesh partition and all particles are localized in ⓐ. Particles 1, 2, 3 and 4 are tracked by ⓐ whereas 5, 6, 7 and 8 are tracked by process ⓑ. A set of cells are possibly communicated, the three triangle cells, the entire mesh of ⓐ, or the cells contained inside the dashed circle.

The particles stored in ⓑ are localized in the mesh partition owned by ⓐ, so mesh data has to be imported by ⓑ in order to track particles 5, 6, 7 and 8. These particles are localized in 3 triangle cells stored in ⓐ memory space. The approach allows to import these 3 cells from ⓐ, but the particles have a high chance to leave these cells and go to the neighbouring cell. This forces ⓑ to import again mesh data from ⓐ, which is expensive in terms of communication time. We have seen that the entire mesh partition stored in ⓐ can be imported but this approach requires a lot of memory space and is not efficient in terms of spacial locality.
A good agreement in this example presented in figure 5.1 can be to import the list of cells inside the dashed circle from ⓐ. This will import the 3 cells where the particles are and the neighbouring cells, allowing particle tracking without communications for a limited time.

We previously introduced a data structure, a structured grid, in order to localize particles at the scale of a mesh partition. This structure is discretized to form cells and kind of boxes used to localize and track particles. If the boxes (created with the cells of the overlapping structured grid) are considered as independent mesh partitions, these boxes can be sent and received easily to and from other memory devices. In fact, we used this implementation to define subsets of cells to import.

The figure 5.2 presents the same example shown in figure 5.1 with an overlapping structured cells composed of independant boxes. In figure 5.2 the example is the same than the one presented in

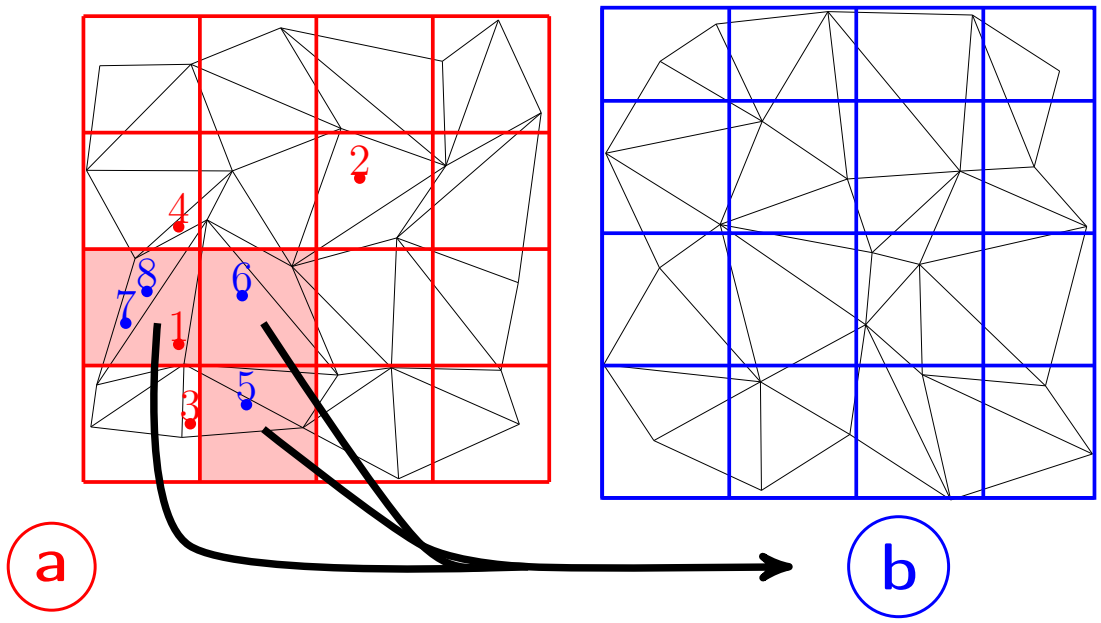


Figure 5.2: Example of structured cells import and export to localize and track removed particles. ⓐ and ⓑ are processes that own a different mesh partition and all particles are localized in ⓐ. Particles 1, 2, 3 and 4 are tracked by ⓐ whereas 5, 6, 7 and 8 are tracked by process ⓑ.

figure 5.1 as two memory spaces are represented (ⓐ and ⓑ), each memory space is used to store a mesh partition and the balanced number of particles. The problem is the same, ⓑ has the task to track particles $\{5-8\}$ and does not store the required environment that is stored on ⓐ. The proposed solution is to import the three boxes, where particles 5, 6, 7 and 8 are, from ⓐ and ⓑ is finally able to track particles in these boxes.

With this approach, particle distribution is not changed, as the workload and the distribution of the mesh and the number and the size of communications is reduced as also the memory occupancy of external boxes.
The different structures and algorithms to describe a box and to communicate are detailed in the next section.

5.4 Implementation details.

5.4.1 Identification of a subpartition.

In order to characterize the particle localization in the structured grid, a special structure is used. This structure is described in the code snippet on the right. This structure is a set of three unsigned integers. For the sake of performance, this structure is aligned to 32 bits. In the case of very large meshes, this structure can be scaled up in order to represent large number of boxes and large number of processes.

```
BoxID {
  unsigned int icontext : 8;
  unsigned int igrid : 8;
  unsigned int icell : 16;
};
```

The structure is composed of three integers that represent the following: *icontext* is the numeral representation of the computing mesh, in fact it is possible to launch several meshes and simulations at the same time. In chapter 7 this element will be used for another usage. The member *igrid* is the numeral representation of the mesh partition, this number represents the mesh partition and the associated structured grid. *Icell* represents the cell number of the structured grid. As we considered the cells of the structured grid as independent sub-partitions, this number represents the static location of the sub-partition in the mesh partition represented by the structured grid. This is the local identification of the sub-partition.
By default, the bit representation of the three different components is aligned to 32 bits with the following representation : *icontext* and *igrid* are encoded on 8 bits and *icell* is encoded on 16 bits. This representation allows to instantiate up to 511 simulations of 511 partitions each discretized with 131071 boxes each.
In the idea to assign a single partition to a single process, it can be noticed that this representation only allows 511 processes per context, and so per simulation launched at the same time.
This structure is very important as it allows to identify the box, where particles are probably localized. This structure is obviously accessible from outside the library.

5.4.2 Definition and communication of a subpartition.

So all local particles are localized using the structure *BoxID*. All processes have the knowledge of the size of all partitions, this way all processes can determine the *BoxID* of its particles. As this approach is a pre-localization step that determines the approximative area of the final computing cell where the particle is, a particle can be localized in multiple boxes. Figure 5.3 shows an example case where a particle can be localized in multiple boxes at the same time.

As boxes are localization approximations, particles can be localized in several grids. In figure 5.3, 4 grids are represented NW, NE, SW and SE with 3 particles A, B and C. As we can see, particle A is only localized in the grid SE, whereas particle B is localized in both NW and NE grids and C is localized in all grids. As the four grids are structured grids or four sets of boxes, it means that A

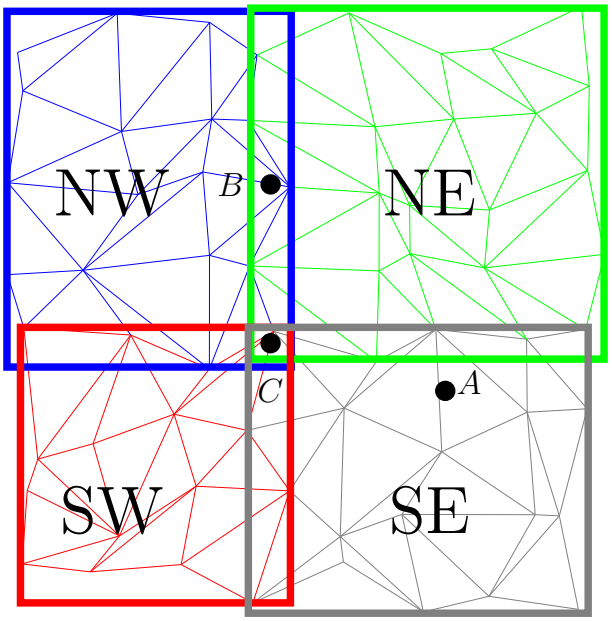


Figure 5.3: Example of a particle localized in multiple boxes.

is localized in a single box, B is possibly in 2 different boxes and C is localized in 4 different boxes. In addition, boxes also own ghost cells which are duplicated cells imported from other boxes and other mesh partitions. In other words a particle that is localized in a ghost cell is also localized in another box. On the other hand, ghost cells are added after the boxes creation. It means that ghost cells that are added are all situated outside the boxes. So the particle localization step firstly localizes the particle in a box. It means that a particle localized in a ghost cell can not be localized in multiple boxes of the same grid but can be in multiple boxes of different remote grids.
The local box number is the *.icell* member of the *BoxID* structure.

Once the group of boxes where a particle can be are singled out, they are all received from other remote memory spaces. As the box is identified with a unique *BoxID*, the process location is directly found. The target process then receives the set of needed boxes from all other known processes, stores them, localizes particles in all the boxes it owns (internal and external boxes) and determines the unique box and the unique cell for each particle.
In terms of communications, this operation requires 4 communication calls that are resumed in algorithm 6.

The 4 calls consist in communicating the number of required boxes, communicating the *BoxID*, receiving the sizes of each boxes in order to store them properly and finally receive the asked boxes. As we can see, this algorithm needs the communication of messages of variable size to each known processes due to the communication of the number and the sizes of the boxes. This is highly not recommended, as some processes may do more communications than others. This is due to the chosen overlapping structured grid where some cells of the grid may own more data than others especially where the fluid is mode dense. A dynamic box construction can solve this.

```
/* All processes run this algorithm */
/* Requested boxes are sorted in terms of the grid */
for grid_i ∈ {grids/processes} do
    send(n_Rbox, grid_i);
    // Send the number of boxes to recv from grid_i
    recv(n_Sbox, grid_i);
    // Recv the number of boxes to send to grid_i

    send(RboxId, grid_i);
    // Send the box ids to recv from grid_i
    recv(SboxId, grid_i);
    // Recv the box ids to send to grid_i

    send(SboxSize, grid_i);
    // Send the size of boxes to send to grid_i
    recv(RboxSize, grid_i);
    // Recv the size of boxes to recv from grid_i

    send(Sbox, grid_i);
    // Send the boxes to send to grid_i
    recv(Rbox, grid_i);
    // Recv the boxes to recv from grid_i
end
```

Algorithm 6: Communication and exchanges of sub-partitions using Boxes.

As a few number of long message is better recommended in order to reach the maximum rate of the network, the first communication is not efficient as it exchanges the number of boxes in the next messages which corresponds to a single integer per exchange.
Another approach is implemented to reduce the number of communication calls and to increase the size of the messages. The found solution consists in communicating to each process an array of booleans which indicates true if the box is needed and false otherwise. This way, the number of needed boxes and the box ids are sent in the same message. This is due to the fact that the size of the message can be precomputed and is static as all processes have the size of each grid of the parallel system.
With this algorithm a local process sends to each other process a vector and receives from each process another vector that contains the needed boxes from other processes. Globally a matrix is sent and another matrix is received.

The next instruction consists in sending and receiving the boxes and their data to and from other processes. (see algorithm 7).
This algorithm that consists in transmitting an array of booleans can be considered as an administrative form or a Multiple Choice Question a process has to complete and return to the process to which he is asking boxes.
A very useful information is the fact that processes store different mesh partitions. It means that a process can only send boxes it manages and can only receive boxes it does not own. This information is very helpful for the completion of the form and for communications destinations.

```
/* All processes run this algorithm */
for grid_i ∈ {grids/processes} do
    bool Rarray[nTotalBoxes_i] ← {false};
    bool Sarray[nTotalBoxes_me] ← {false};
    for Rbox_j do
        if Rbox_j ∈ grid_i then
            array[Rbox_j.icell] ← false;
        end
    end
    send(Rarray, grid_i);
    // Send the array of bool to recv from grid_i
    recv(Sarray, grid_i);
    // Recv the array of bool to send to grid_i

    for Sbox_j do
        if Sarray[j] = true then
            send(SboxSize_j, grid_i);
            // Send the size of boxes to send to grid_i
            send(Sbox_j, grid_i);
            // Send the boxes to send to grid_i
        end
    end
    for Rbox_j do
        if Rbox_j = true then
            recv(RboxSize, grid_i);
            // Recv the size of boxes to recv from grid_i

            recv(Rbox, grid_i);
            // Recv the boxes to recv from grid_i
        end
    end
end
```

Algorithm 7: Exchanges of boxes using forms filled by processes.

Now that all processes store all needed boxes to localize local particles, they must be stored in order to keep boxes that are only needed for the particle movement. In the set of received boxes, some of them are not used because of the fact that a particle is possibly localized in multiple boxes or some boxes do not contain any data of the mesh partition.

So a particle is localized in cells of each box with the *in-cell* test seen earlier in this manuscript. The returned cell is considered as a unique solution so as soon as the *in-cell* test is satisfied, the algorithm stops and goes to the next particle to localize.

When all particles are localized (or not, some particles may be outside the computing mesh), the boxes that are marked as unused are removed from local memory.

A box is identified with the *BoxID* structure defined in the following and as it is used to represent a sub-partition of the mesh, it needs data from the mesh to localize and track particles. The needed data is defined on the right hand side. A *Box* contains multiple object connectivities of cells, faces and vertices that are used to track particles in any memory space. This is a condensed mesh data only useful for particle tracking. No data concerning particles is stored in this structure. In addition to the mesh data, an array of floats named *flowfield* is stored. This array has the size of $dimension \times nCell$. It represents the vector field of a cell. The reason is that there are multiple ways to store physics data in the mesh and also multiple ways to compute the acceleration vector of a particle. In order not to have to store all of these data that drastically increases the size of the messages and the final size of stored boxes, the choice was made to let the flow phase compute alone the vector field and update the final vector field used to move particles in cells of the mesh. The drawback of this choice is that particles in the same cell compute the same direction vector from the field. Multiple particles localized in the same computing cell can have different directions from the field with this approach but it requires to update the vector field between the computation of two continuous particles. Direction vectors that do not depend on the flow phase are obviously computed independently.

```
Box {
  BoxID id;
  int nCell, nFace, nVtx;
  int cellFace [];
  int cellFaceIdx [];
  int faceVtx [];
  int faceVtxIdx [];
  {. . .}
  float vtxCoords [dim x nVtx];
  float borders[dim x 2]

  float flowfield [dim x nCell];
};
```

Because of the *MPI* library, the data members of this *Box* structure are condensed into two main arrays *data* which is an array of bytes that stores all data except *nCell, nFace, nVtx* and *BoxID* and an array *dataIdx* which is a fixed size array of integers to store the addresses of each structure member. This the same method to store data that is used that consists in two arrays, one for the data values and the other to store the data addresses. With this representation, the data access is not harder and it allows to send the whole structure in a few number of messages independently from the data types.

The update of the flow field can be a very expensive operation as all boxes in the parallel context and all mesh partition must be updated. This operation is described in the next section.

5.4.3 Optimizations on communications

Communications are very expensive in terms of execution time. As a particle localized in a remote box, need the data of this remote mesh partition, this box will travel through the network and will be imported on the local process memory. Such a communication is very expensive as a very important amount of data is received and no computation on the particle can be started before all data has arrived.

This part of the particle tracking is known as communication bound problem, it means that the

performances, the execution time depends on the time of communication. Generally in HPC field the solutions developed to reduce communication time are:

- reduce the number of communications,
- reduce the size of messages,
- do some work and computations while the data are received.

In our implementation the size and the number of messages can be tuned with the size of a box. Indeed the smaller are the boxes, the smaller are the messages and the more frequently they are exchanged. On the other hand, bigger boxes allow the track of particles for a longer time but require more storing space and bigger messages.
In order to optimize time spent in communications and not waste any time waiting data, some computation can be done during boxes exchanges. These exchanges are done during particle localization and for this reason the only computation that can overlap communications is the localization of other lost particles.
To do so, non-blocking communications are used to transfer boxes. Thus, processes do not need to wait for the communications to be complete to start working. Thanks to the *MPI* routine *MPI_ Test* the completion of a *MPI* communication can be tested. The processes that do not receive all data frequently test the communication in order to start particle localization. Some particles can be localized while the data arrive, for example particles that are localized on the local mesh which do not need to wait for any communication to be complete.
The algorithm is presented in algorithm 8.
During the particle localization, the particles are stored in an array. For each particle the communications are tested, if the particle is in a box that has been received or that is already stored in local memory, the particle is localized. If the box has not arrived, the particle is not localized and is moved to the end of the array. This way the particle will be localized later.

In this algorithm 8, the processes try to localize all particles. Some particles that are located in the local mesh partition or in an imported box that has already arrived can be localized.
On the other hand, some of the required boxes are not already received, and particles that can not be localized are moved to the end of the loop in order to try to localize them later.

The advantage is that no time is wasted to wait for boxes and communications to be complete. If a set of particles can not be localized because of missing data, they are moved in order to be localized later. It allows to compute particles that do not need to wait for additional data.
On the other hand, a singularity can happen, because of communication failures. If one or more communications fail for any reason, the algorithm will loop and run out of memory. In a future work, a watch-dog will be added, a set of tests in order to garanty the robustess of communications.

```
/* All processes run this algorithm */
/* Requested boxes are sorted in terms of the grid */
for process_i ∈ {grids/processes} do
    // Recv the boxes from process_i with a non-blocking message.
    irecv(n_box, process_i);
end

for iparticle ∈ list_of_particles do
    /* If the box has been received, the particle can be localized */
    if box_iparticle exists then
        localize(iparticle);
    else
        /* If the box has not been received, the particle is added at the end of the list. The
           particle will be localized later, at the end */
        list_of_particles ←^add iparticle;
    end
end
```

Algorithm 8: Receive of Boxes and communication overlap.

5.4.4 Flowfield update

As multiple mesh partitions are exported to other memory spaces, a modified mesh has to be updated on all these memory spaces. As structural changes in the mesh render the current boxes and grids out of date, the flow field update operation only considers the changes on the fluid.
This update operation is very simple as it consists in updating the values of local and exported velocity vectors.
So first, the local vector field is updated, for each local box, the flow field array is modified. In a second time, the list of outdated boxes have to be determined.

5.4.5 Definition of the grid, the set of local boxes.

We already have seen that the mesh partition is subdivided into a fixed number of boxes. These boxes are arranged as a structured grid in order to have all local boxes as unique. This way the localization of a point in a structured grid has a single solution. The grid is numerically represented on the right.

```
Grid {
  int igrid; // Same as the BoxID.igrid
  int nBox;

  float borders[dim x 2];

  Box boxes [nBox];

};
```

We already talked about the parallelization of this structure and the fact that in order to accelerate the particles localization with the *BoxID* precomputation, the data of all grids have to be stored on every memory space of the parallel system. To be more precise, and now that the *Grid* is defined, the amount of data that is sent to other

processes is equal to the sum of the size of *igrid*, *nBox* and *borders* members. It means that the total amount of the whole problem on each process is equal to :
$nGrid \times (igrid + nBox + borders) + boxes$
which corresponds to 2 integers plus $2 \times dim$ floats plus the size of local boxes. This is the starting and minimum memory occupancy of the structures per process. Obviously the memory occupancy increases as remote boxes are imported to track external particles.
The shared data (*igrid*, *nBox* and *borders* members) is represented in the same way as the Box and mesh data. An array of bytes stores the three members. The particularity is that because the size of these members are known and fixed, the address array is not required.
The grid unique number is the *.igrid* member of the *BoxID* structure.

We can notice that the range of identification numbers (icontext, igrid and icell) is limited to the local range of a block. In other words, these numbers represent the local number inside a same block (inside a single box, inside a single grid or inside a single context). To be more accurate, the box indentification number is valid in the range of the number of boxes in a single grid, the grid's identification number is in the range of the total number of grids inside a single context and finally the identification number of a context is valid in the range of the total number of existing contexts. The following definitions summarize the ranges:

$$\begin{cases} ibox \in [0, nBox_{igrid}] \\ igrid \in [0, nGrid_{icontext}] \\ icontext \in [0, nContext] \end{cases}$$

where $nBox_{igrid}$ corresponds to the number of boxes in the $igrid^{th}$ grid, $nGrid_{icontext}$ corresponds to the number of grids in the $icontext^{th}$ context and $nContext$ corresponds to the total number of existing contexts in the run.

5.5 Acceleration obtained with traveling boxes.

This approach that consists in creating sub-meshes is applied to a parallel particle tracking simulation. In this application, 12.8 millions particles are localized using the structures developed above and the next position of each particle is computed.
The graph in figure 5.4 gives the speed-up obtained by partitioning local meshes into boxes for the particle localization phase. In this graph 5.4, the execution time of particle localization is measured on different number of processes and different number of local boxes. The localization speed-up is obtained by comparing the execution time of particle localization to show the parallel scaling of traveling boxes. The speed-up is based on the average time as it has been run with parallelized processes, 2 different processes can execute particle tracking operations with different efficiencies.
The x axis refers to the number of boxes per local grid. The represented number corresponds to the number of boxes per dimension. For example the results associated to a structured size of 4

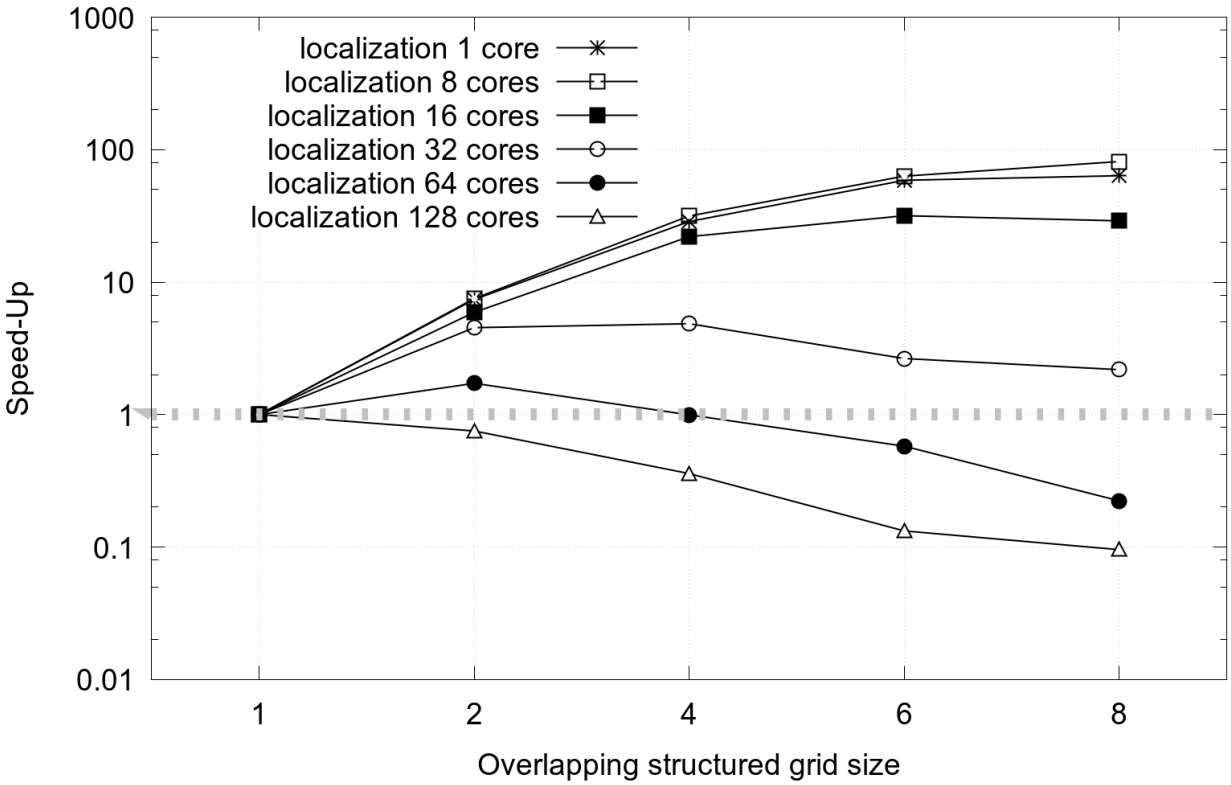


Figure 5.4: Speed-up obtained using boxes of different sizes on 128 processes.

corresponds to the speed-up obtained with a structured grid of 4 boxes (so 4 structured cells) per dimension, which corresponds to a total number of $4 \times 4 \times 4$, so 64 boxes per local grid.
The cross points represent the execution time (the associated speed-up) of particle localization using multiple sizes of boxes in a sequential run.
The speed-up is calculated by comparing the execution of a parallel run without the use of boxes (that is equivalent to a single box per local grid) and the same execution with the same number of processes but with several boxes and grid discretizations. For example the open circles represent the speed-up of particle localization in multiple grid sizes on 32 cores compared to the execution time with a single box per process on 32 cores. This is the reason why all curves start with a speed-up of 1(100%).

According to figure 5.4, the more the particle localization is parallelized, the less is the speed-up. This is due to the growing part occupied by communications. Indeed, the communications are more important as the number of boxes increases.
On the other hand the use of boxes is very efficient on low number of processes. The use of numerous boxes is efficient and brings high speed-up for simulations run on 16 processes or less.
The cost of communications become too heavy compared to the acceleration obtained with the use of boxes starting from 32 cores where the speed-up starts to decrease as the number of boxes per grid increases.

Speed-up values lower than 1 mean that the localization is more time consuming than the original one that do not requires boxes. These values are encountered for parallel runs on more than 64 processes. This is the sign that communications cost more thant the acceleration brought by boxes.

The interpretation we can do is that if the communications are not optimized, the use of travelling boxes is efficient for 32 processes or less. This is again a common problem in parallel applications, especially in parallel particle tracking that is bounded by communication in the case of distributed and dispersed particles.
Another remark we can make is the fact that the simulations are the same in terms of parameters and data states. In other words, the same mesh and exactly the same particles are used in this test. So, the run on different number of processes differs from the parallelization of the mesh (the number of computing cells per process), the particle distribution (the number of particles per process) and also the number of computing cells per box. Thus, in cases of high parallel runs with the maximum number of boxes per grid, the number of cells per box is the lowest. This has a high consequence in terms of performances and execution time because it increases the number of box swaps for moving particles. In fact, as particles move and leave boxes, they have to be localized again in the right box in order to continue the track computation. This movement requests new communications as the new boxes to which the particles go must be imported if they are not already stored in local and accessible memory.

As we are talking about memory, the next figure 5.5 shows the memory occupancy of the run used in figure 5.4. The memory usage of the application is analyzed compared to the number of boxes and the number of processes in the parallel run.

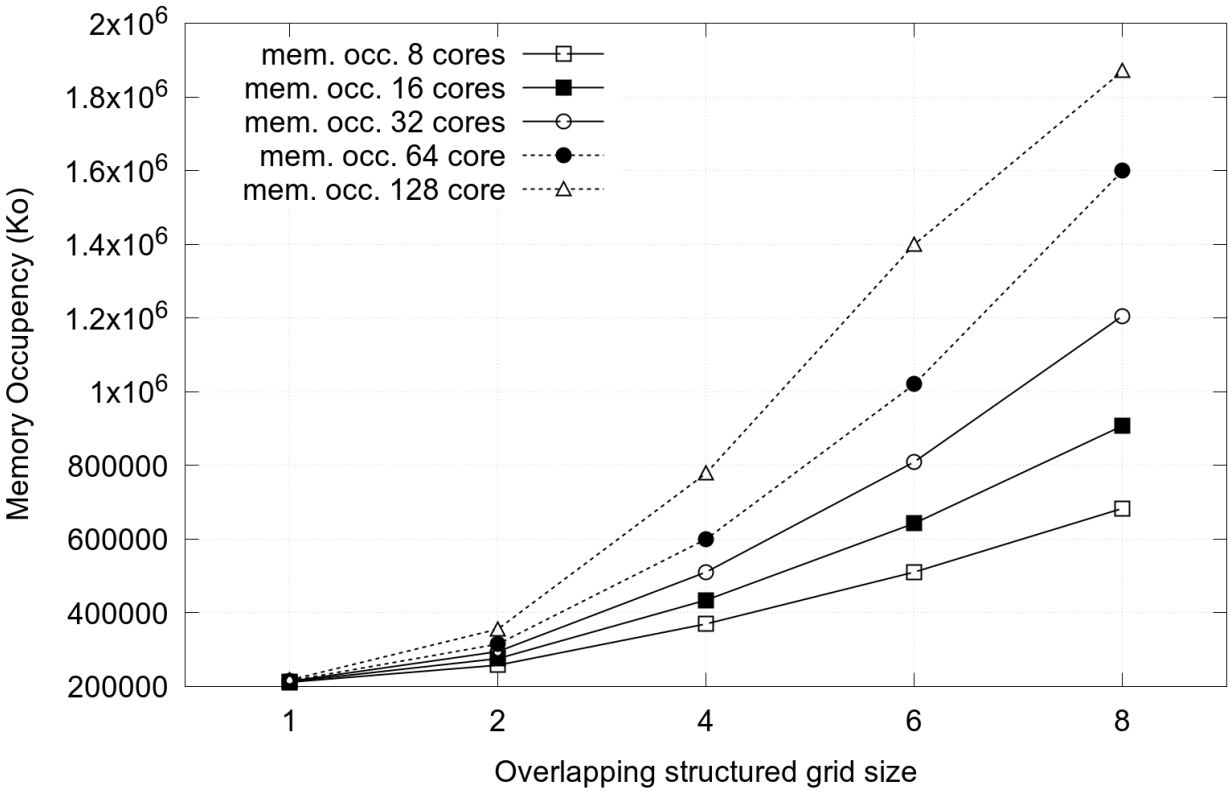


Figure 5.5: Memory used by the boxes (intern+imported boxes) per process.

The trend we can figure out is that the more the mesh is discretized, the more the memory per process is used. This is explained by two different effects: first, a box is a set of cells, faces and vertices extracted from the original computing mesh partitions. On each mesh partition, a set of

boxes is initialized in order to share mesh subpartitions with other memory spaces and to accelerate particle localization in local grids. As boxes contains cells, faces and vertices inside this box and also ghost cells, a box is slightly heavier than the sum of geometric objects that contain a box. Indeed, ghost cells are cells that are directly connected to the cells localized inside a box. Thus, ghosts cells regroup cells, faces, and vertices that are outside of a box that are connected to the geometric objects inside a box. These ghost cells are cell clones localized in neghbouring boxes. In other words, the size of ghost cells is added to the size of each box in memory. This is the first reason why the memory occupancy increases as the number of boxes increases.
On the other hand, the boxes have been implemented in order to send and receive blocks of mesh partitions instead of entire mesh partitions to localize and track particles. So, the second reason of this high memory occupancy on massively parallel runs lies in the particles spatial localization. In the test case, the particles coordinates are randomly initialized in the range of the global computing mesh. It means that all processes have to localize and track particles initialized in all the partitions of the mesh. So during the particle localization, all existing boxes of the simulation are received by all processes and the final memory occupancy corresponds to the maximum possible as all processes store all mesh partitions. In this condition of particle distribution, the grid structure and the associated boxes are completely useless in a massively parallel context.
This second remark also means a lot in terms of performances: first, contigous particles have very low chances to be localized in the same box, which means that the localization of a set of particles implies that data of boxes are not reused from one particle to another. There are then lots of cache misses and page replacements between two particles.

The other observation we can do is that the use of boxes does not improve the efficiency of next particle movement. This is explained by the fact that the particle movement is computed using local and targeted data. The required data objects for particle movement are already stored in local memory and known by the processes. The particle movement computation is then not improved by the implemented structures.

For technical reasons, a test case with a larger mesh could not be launched because of the too important memory occupancy per computing node. We believe that performances using traveling boxes is highly connected to the number of computing data (cells, faces, vertices, ...) in each box. In other words, depending on the size of the problem, the performance of the cartesian grid is tunable with the size of a box. In all cases, the tendencies of the performance is always similar, the speed-up increases until a maximum size of box that corresponds to the mesh partition discretization, that is to define, and from this adapted box size, the performance decreases as the size of boxes increase on a limited number of processes. On the other hand, starting from 32 processes, the communications are too heavy to show high performances in particle localization.

5.6 Conclusion

The structures developed in this section are used to perform particle tracking on remote memory spaces. The local mesh is partitioned into a set of boxes that are considered as independent meshes. These boxes are sent and received from and to other memory spaces in order to track particles stored on remote memory spaces.
Using boxes that represent partitions of the local mesh partition allows to efficiently distribute particles through the parallel system. Indeed, particles are distributed in order to balance the workload (the number of particles to track), regardless of flow field phase computation.

Using sub-partitions such as these boxes improves the particle localization step. Indeed, the localization time is divided up by almost 100 for 8 parallel processes.
On the other hand, the implementation has no impact on the computation of particle movement. As this computation is a local operation with data that is already in place, the impact of reducing partition size is null. In addition, the chosen implementation that uses ghost cells implies a growth of the memory occupancy. In fact, the more the boxes are small, the more there are ghost cells and the higher is the memory occupancy.
Another reason why the obtained speed-up does not scale with higher number of processes is the increase of communications. Indeed, the number of boxes that are exchanged with other processes increases. This additional fact decreases the parallel performances.

It has to be noticed that the tested case simulates the random creation of particles. It means that all particles of the simulation are initialized with random coordinates. In this simulation, all cores own particles that are randomly localized everywhere in the global computing mesh. Localizing and moving a single particle needs the import of an entire box. This implies that the current distribution of particles stores the entire computing mesh in the form of boxes on each memory space of the parallel system. This is shown by the distribution quality of particles 5.4 that is equal to 0.78125% for every size of boxes which corresponds to the rate of a single partition in the global computation mesh ($\frac{1}{128}$). This explains the growth of the memory occupancy.
The particle distribution algorithm used is detailed in the next chapter 6 as a proposed optimization to organize particles and to gather close data on the same memory space.

CHAPTER

6

Proposition of an Efficient Particle Distribution Method.

Contents

6.1 Introduction

The parallelization of particle tracking is similar to any HPC challenge as it depends on the workload of the simulation. As we have seen in the graph of tasks, particle tracking parallelization (the work balance) is defined by the amount of particles to track on each process. It means that the parallelization of particle tracking highly depends on the particle distribution efficiency. This section describes an algorithm to distribute tasks we have chosen and some optimizations in order to distribute particles and tasks in a parallel context.

6.2 Particle Distribution on a shared memory machine.

Particle tracking is very easy to parallelize in a shared memory context. The particles are all localized in the same memory device and no communication protocol is needed. In HPC and computer science the use of these architectures is very often especially today with the concept of accelerators and GPGPUs used to compute graphics and also more scientific simulations.
So particles can be easily parallelized and distributed over the parallel processes. At the very beginning of this thesis, the test has been made on a single processor of 32 cores, the parallel efficiency for this test was equal to 93%. This test on a single processor was performed by *OpenMP* [93, 94].
The conclusion we made at this time is that particle tracking is very efficient in a local parallel context but there is many cache misses during the run time as data of particles that are contiguous in memory and continuous in the runtime are far from a particle to another. The local cache is then constantly renewed and the local data is not reused as much as possible.
To solve this problem and try to bring closer particle data, the idea was to renumber particles according to their position and more precisely, according to the *Box* in which they are localized. To do so, a bucket sort [95] is used that has a complexity equal to the number of particles to sort. Here the buckets are the local boxes of the overlapping structured grid.
By applying a sort on particles, sorting them by their box localization, the speed-up obtained is about 20% on the overall simulation (particle localization + particle movement).
Renumbering particles in order to optimize data reuse, spatial and temporal locality improves the overall runtime and the sorting algorithm is particularly adapted to the use of boxes to localize particles. This is a way to keep data close to the core and to reduce temporal locality of data. Sorting and arranging data on local memory was the premise of our particle distribution algorithm.

6.3 Particle Distribution Algorithm on remote memory spaces.

There are many algorithms in the literature to distribute and balance tasks on a parallel system, some authors propose multiple algorithms to distribute particles and in general tasks [19, 66]. In order to distribute particles, we chose an algorithm among them all proposed by O'Brien, Brantley and Joy named Partner Processor Algorithm. This algorithm consists in iteratively assigning to each process a partner process. Each process chooses a list of other processes with which it communicates and balances particles.

The algorithm runs on a parallel system where processes have unbalanced work to do. This amount of work per process is determined iteratively, the number of iteration depending on the number of processes. In fact the number of iterations is determined with :

$$\lceil log_2(nProcs) \rceil \tag{6.1}$$

where $nProcs$ is the total number of computing processes.

The work balance is done through several steps, first each process chooses a partner process with which it will communicate. The selection of the partner is done with the following operation :

$$rank_{myself}^{(1<<iteration)} \tag{6.2}$$

where $rank_{myself}$ is the number of the current process, the local process, and $iteration$ is the number of iteration which is the number of previous partners.
So the idea is to choose iteratively multiple partners and both partners balance their work. The final result is impressive as the work balance is reached very fast and is very accurate. In fact the final work balance is perfect.

To better understand the effect of this algorithm an example is presented in figure 6.1. In this

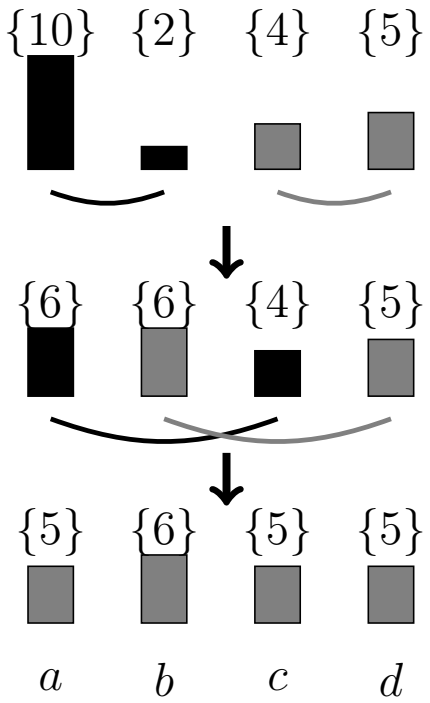


Figure 6.1: Distribution of tasks over 4 processes using the *Partner Process Algorithm*

figure 6.1, 4 processes (a, b, c, d) have a set of 21 particles to track. Because the workload is not well balanced, particles have to be distributed in order to optimize the parallelization rate of the application. Before the execution of the distribution algorithm, particles are distributed as this : 10 particles are tracked by process a, 2 particles belong to process b, 4 particles belong to process c and 5 particles belong to process d. We can figure out that in the end of this example, each process has to track at least 5 particles. To execute the distribution algorithm, the first step is to choose a subset of process partners with which a process is going to balance their particles. During the first iteration, the partners couples formed to balance the workload are $\{a, b\}$ and $\{c, d\}$. After two iterations (corresponding to two partners to each process), the workload of the entire application is perfectly balanced.

In this paragraph, we discuss the different advantages and disadvantages of this algorithm to

distribute tasks over multiple computing units. Let us begin with the pros. The first advantage of this algorithm is its simplicity to implement it. The complexity of the algorithm remains in the communications between two processes, this algorithm does not care about communication priorities and global communications. Indeed, there is no need to share any information on the global network as any information that has to be shared is communicated to the partner during each iteration. This gives another advantage of this algorithm that is a reduced and limited number of communications. The number of communications depends on the number of processes and the number of partners. In fact the number of communication calls is equal to the number of iterations of the algorithm. The final advantage of this algorithm is its efficiency. The load-balancing is obtained at up to $\lceil log_2(nProcs) \rceil$ iterations, where $nProcs$ is the number of processes in the system.

There are some disadvantages coming with this algorithm, the first one is the limitation in the adaptation with different number of processes. Indeed, the number of processes has to be a power of 2 in order to select a process partner for each process.

A very important remark to add is that the algorithm has been chosen for its ease of implementation. The algorithm performances are not studied in this manuscript and is not compared to other algorithms. The modifications and the different approaches used are applied on this particular algorithm but this can be applied on every other algorithm to distribute tasks.

6.4 Approach to optimize Particle Distribution Algorithm.

Particles can be easily and efficiently localized with the developed structures and especially with the *BoxID* structure. The fact is that a set of random particles has to be localized and moved efficiently in different times. The first time is to write efficient algorithms to localize and move a single particle. In a second time these algorithms are used for multiple particles. The fact is that when a particle has moved and the process goes to the next particle to track, some data can be reused if both continuous particles are close to each other. In other words, if two particles are close in the environment, local data have a higher chance to be reused and cache memory has a chance to be reused accelerating execution time. Thus the particle numerical localization is very important compared to the numerical localization of other particles in the same memory space.

So now that the main objective is defined as trying to group close particles in the same memory space, the idea is to modify the distribution algorithm in order to bring closer in memory the data that is numerically close. To do so, the algorithm is modified in order to take into account the particle position during the distribution.
The formula used to select a partner process is not modified, neither the number of iteration but some steps are added before any communication between the both partner processes.
Particles are locally sorted in 5 categories :

- A: particles that are located in the environment of the partner process,

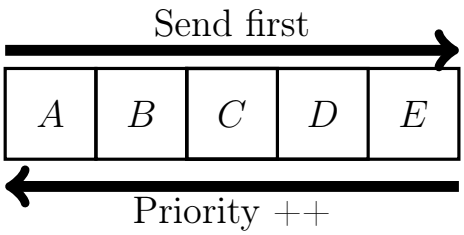


Figure 6.2: Array of particles to send to partner.

- B: particles that are located in one future partner of the current partner process,
- C: unlocalized particles ($\notin \{A \cup B \cup D \cup E\}$),
- D: particles that are in a future partners of the current process,
- E: particles that are in current process.

So the modification of the particle distribution algorithm is made in 5 important steps that have to be executed in the right order :

1. send particles that are located in the environment of the partner process,
2. send particles that are located in one future partner of the current partner process,
3. send unlocalized particles,
4. send particles that are in a future partners of the current process,
5. send local particles.

The general idea is that particles closest to the local process are sent the latest. The figure 6.2 represents the array of particles that have to be sent with the priorities. As the A category represent the category with the highest priority and E with the lowest one, particles that are categorized with the highest priority are sent first to the current partner.

Obviously, the sending priorities are locally computed as all process can localize particles in any grid of the simulation, compute the list of partners of all processes and so determine the category of each particle. The modified algorithm is detailed in algorithm 9.

In order to make it clearer, the example presented in the previous section concerning 4 unbalanced processes is modified and sketched in figure 6.3.

In this figure 6.3, 4 processes (a, b, c, d) have a set of 21 particles to track. Here, the patterns represent the particles localization, for example all particles managed by process d are localized in the mesh partition and the associated grid stored and managed by process c. Because the workload is not well balanced, particles have to be distributed in order to optimize the parallelization rate of the application.

The context and the problem are the same as the example presented in the previous section : 10 particles are tracked by process a, 2 particles belong to process b, 4 particles belong to process c and

```
begin
    list_partners[⌈log2(nProcs)⌉];
    for ipartner < ⌈log2(nProcs)⌉ do
        list_partners[ipartner] ← rank_myself^(1<<iteration);
    end
    for each ipartner ∈ list_partners[] do
        box_ids ← localise_particles(particles);
        n_ideal ← (n_local + n_partner)/2;
        exchange(n_local, n_partner, ipartner);

        int categories[5];
        for each iparticle ∈ list_particles[] do
            if (iparticle ∈ A) /* Send particles localized in A catgory */
            categories[0] ←add iparticle;

            else if (iparticle ∈ B) /* Send particles localized in B catgory */
            categories[1] ←add iparticle;

            else if (iparticle ∈ C) /* Send particles localized in C catgory */
            categories[2] ←add iparticle;

            else if (iparticle ∈ D) /* Send particles localized in D catgory */
            categories[3] ←add iparticle;

            else if (iparticle ∈ E) /* Send particles localized in E catgory */
            categories[4] ←add iparticle;

        end
        /* Send/receives particles to/from partner */
        send (categories, ipartner);
        recv (particles, ipartner)
    end
end
```

Algorithm 9: Modified Particle Distribution Algorithm.

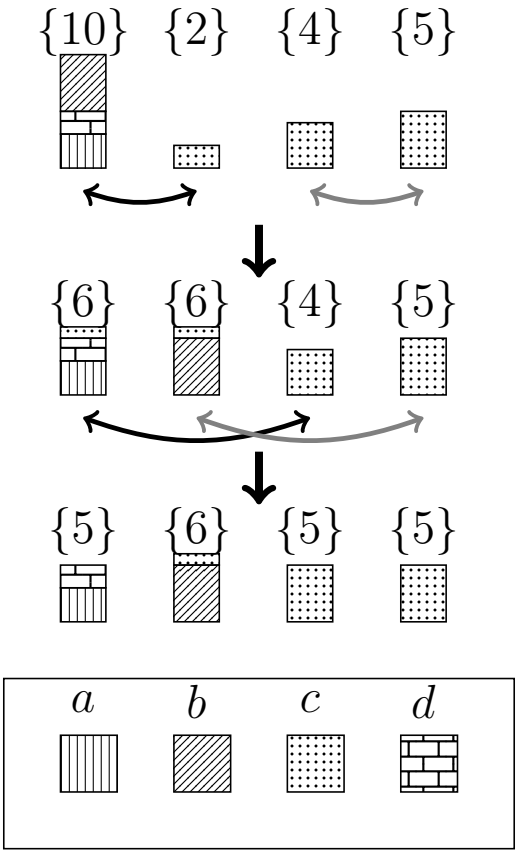


Figure 6.3: Distribution of tasks over 4 processes using the modified *Partner Process Algorithm* that takes into account the particles localization in grids.

5 particles belong to process d.
The difference lies in the objective to send particles with different priorities. At the start of the algorithm, the particles localization is distributed in the memory of the process in this way a : 3 particles are localized in the process a, 2 particles are localized in process d and 5 particles are localized in process b. All other particles stored in the memory of other processes are localized in the mesh partition stored in process c.
At the end of the algorithm's execution, the particles distribution is perfectly balanced again and the localization has been taken into account.
The figure also shows a limit to the current algorithm :process a owns all particles localized in process d whereas this process d does not store any particle that can be localized in its local mesh partition. This is explained by the fact that a and d did not communicate during the execution. On the other hand, the limit can be fixed in this example by applying the additional step of our future work that consists in sending as a second priority particles that are localized in the future partners of the current partner.
In this example, during the first iteration, a sends in priority the particles localized in b and, as process d is one of the future partner of b, a sends in a second time particles localized in the grid of d instead of receiving particles from b that are localized in c.

In this paragraph, we discuss the different advantages and disadvantages of this algorithm to distribute tasks over multiple computing units. Let us begin with the pros. The first advantage

of this algorithm is its simplicity to implement it. The complexity of the algorithm remains in the communications between two processes, this algorithm does not care about communication priorities and global communications. Indeed, there is no need to share any information on the global network as any information that has to be shared is communicated to the partner during each iteration. This gives another advantage of this algorithm that is a reduced and limited number of communications. The number of communications depends on the number of processes and the number of partners. In fact the number of communication calls is equal to the number of iterations of the algorithm. The final advantage of this algorithm is its efficiency. The load-balancing is obtained at up to $\lceil log_2(nProcs) \rceil$ iterations, where $nProcs$ is the number of processes in the system.

There are some disadvantages coming with this algorithm, the first one is the limitation in the adaptation with different number of processes. Indeed, the number of processes has to be a power of 2 in order to select a process partner for each process.
The second disadvantage is the fact that all processes do not communicate. It means that some particles are not sent to the right process, in the process they are localized. This is the case of processes a and d.

6.5 Acceleration obtained with our distribution algorithm.

The different modifications we implemented in order to distribute particles compared to their localization give in theory a better distribution quality and a better data proximity. This improves the execution time of multiple operations of particle tracking.

Figure 6.4 shows the speed-up obtained by tracking 64 millions particles in a mesh with a total of 1.728 million cells with a $4 \times 4 \times 4$ overlapping cartesian grid on each mesh partition. This test case has been experimented on several number of processes and the results are presented in our previous proceeding paper of the High Performance Computing Symposium [96]. The chart shows that the particle localization and the computation of the particle's next position are from 50% to almost 150% faster than the original particle distribution. We remember that the difference with our algorithm is that our algorithm takes into account the particle's localization and generates a priority rank compared to the other particles in order to decide to send it or not.
It is important to notice that the obtained speed-up is high because of the memory occupancy of the application. Ideed, during this run, the original distribution algorithm that distributes particles regardles of their localization, randomly dispatched the particles and brought the situation where all processes need to track particles located in all the mesh. This situation implies that all processes need data of all the partitioned mesh and after the particle localization phase, all processes store a copy of the entire mesh and of all mesh partitions.
With the number of data used for this run, we noticed that some of the nodes use the swap memory that has slower data access. So the speed-up obtained in this test is mainly due to the current machine memory that can not store all the data on all its node.

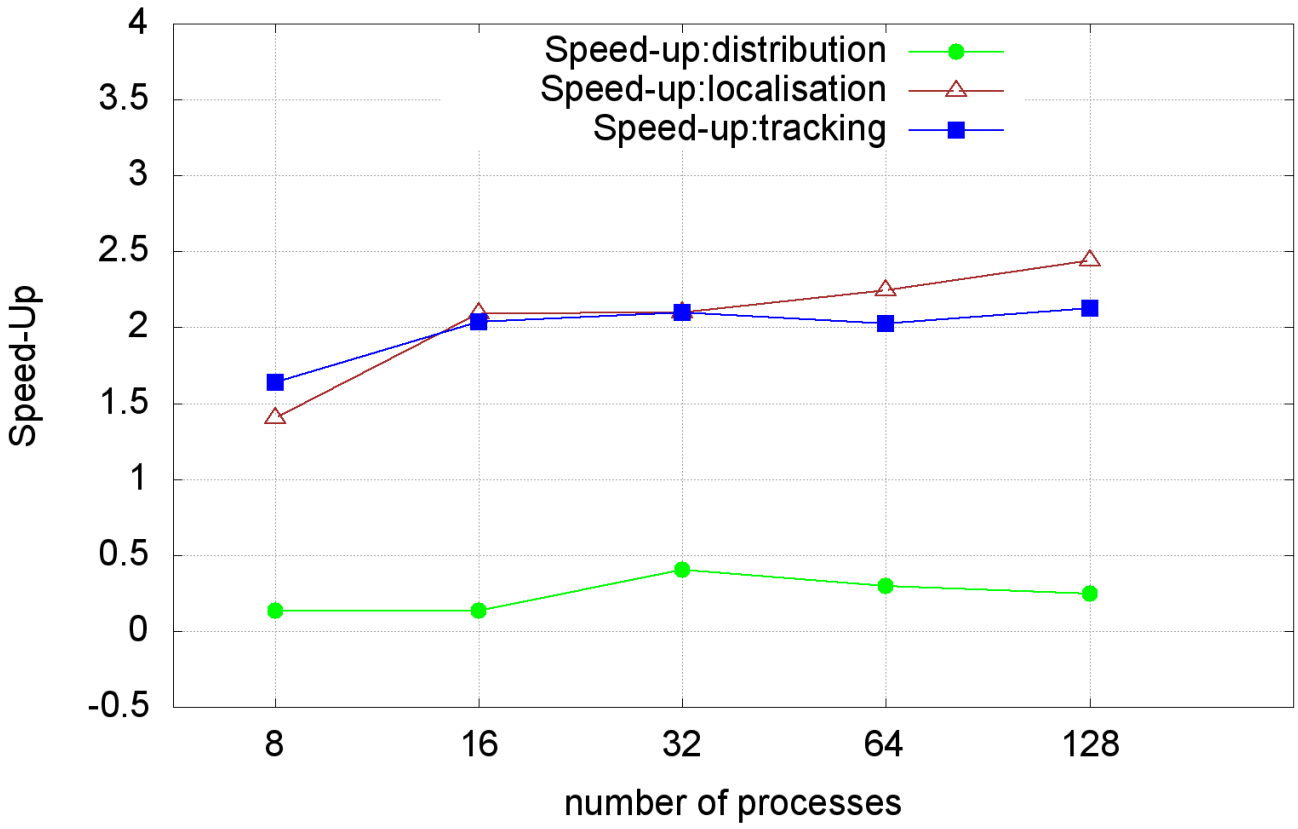
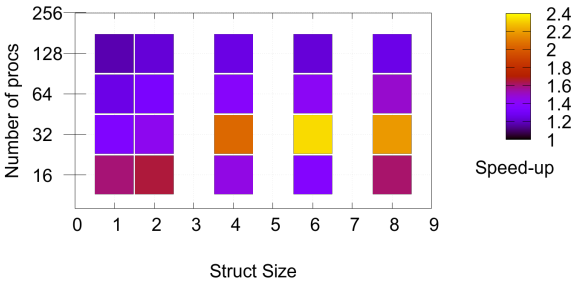


Figure 6.4: Speedup obtained on particle localization, distribution and particle movement.

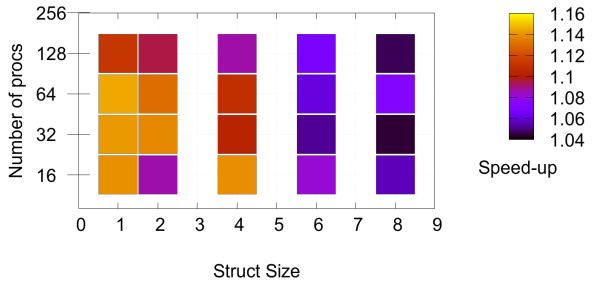
So the first conclusion we can figure out is that our algorithm saves a lot of memory as particles localized in the local boxes are sent with the very last priority. Processes keep particles that are close to the local mesh and do not require to receive additional boxes from remote processes.
In order to observe the speed-up obtained without the influence of memory capacity of weaker nodes, the problem size has been reduced.

The following charts show the speed-up for distribution quality, the obtained speed-up to localise particles, the speed-up obtained for movement computation and on the memory occupancy. The initial particles are stored in the memory of the first process of the system. Again, the particles are initialized with random coordinates in the range of the global mesh with random initial direction and velocity vectors.
The computing mesh is made of 1 million cells and 12.8 millions of particles are followed for a single time step. It means that a single call to particle distribution is done and that particles are localized and move only once. The speed-ups of the 2 operations (particle localization and particle movement) as the gains obtained on distribution quality and memory occupancy on different number of processors are shown in figure 6.6.

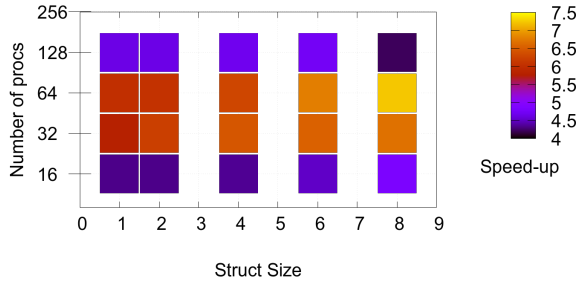
According to these charts of figure 6.6, the particle distribution algorithm helps to improve performances of particle localization, particle movement computation and the memory occupancy as the distribution quality. Indeed, our particle distribution algorithm that takes into account the particle



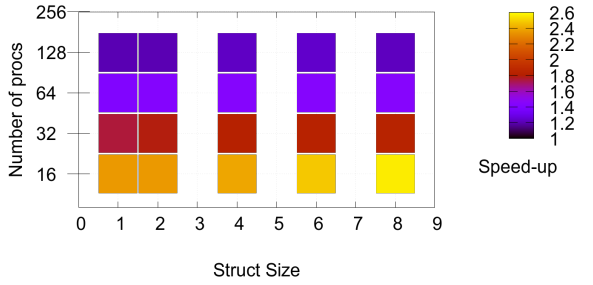
a) Localization



b) Movement



c) Distribution Quality



d) Memory occupancy

Figure 6.5: Speed-up obtained by distributing particles with the modified algorithm

localization before sending a particle to another process, decreases the execution time to localize particles.

The obtained speed-up are lower than the speed-up obtained with larger mesh and more particles because of the memory occupancy of the original algorithm. In fact, our modifications saved more than 130% of memory occupancy (6.6.d). On the other hand, the more there are processes in the run, the less is the speed-up obtained concerning memory occupancy. To summarize the speed-up reported on figure 6.6, the localization speed-up is obtained with 32 processes and a grid size of 6, the particle movement computation get 15% of acceleration with 32 and 64 processes and no cartesian grid, the best memory save is obtained with the most discretized grid and 16 processes, and finally, the distribution quality is the best for 32 and 64 processes.
It can be noticed that the speed-up obtained on 128 processes are not the worst but the speed-up seem to decrease as the number of processes increases. Globally, the distribution algorithm is more efficient on an average of 32 processes.
The particle movement phase is accelerated because the particle distribution algorithm implicitly does a sort on particles according to their position. Indeed, particles in the same grid will be gathered and sent together. This sort is not doing enough to sort particles according to their box, so contiguous particles are not necessary localized in the same box. This is why the obtained speed-up are low but observable.

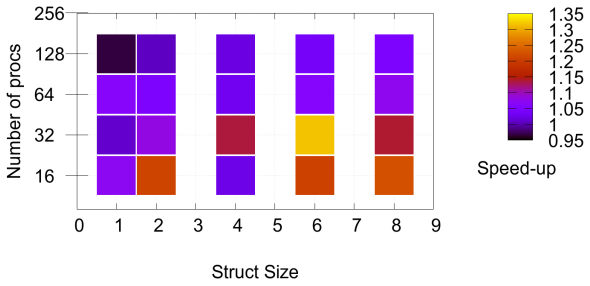
The reason why the best perfomances are obtained for a limited number of processes is because of the original algorithm. As the chosen algorithm iteratively deals with pairs of processes and because a process does not encounter all other processes in this algorithm, some particles are never placed in the right memory device.
A workaround can be to implement another distribution algorithm or increase the number of partners in order to make each process be the partner of every other partner. A round robin shuffle can determine such a list of partners.

As a reminder, the particles in this test are all initialized in the memory of the first awakened process and coordinates are randomly initialized using a pseudo random generator with a discrete distribution. In other words, a particle have a constant chance to be localized in any box. So the particles are equally distributed in space. The consequence of this fact is that the first rank well distributes the particles to the partners it encounters.
In addition, the other ranks that do not have any particle to share at the begining, skip the first partners because of this lack of particles to send and receive. For these two reasons, this test can be considered as false or guided.
In order to observe the particle distribution in a different situation, during the next test case the particles are distributed using the original algorithm is order to randomly distribute data on processes. Then our algorithm is called.

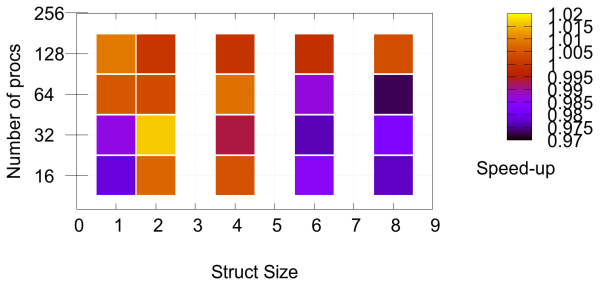
Figure 6.6 shows the same graphics as the previous case, that represent the speed-up obtained on particle localization, particle movement the distribution quality and the average memory occupancy.

According to these new tests, the best accelerations are obtained with low number of processes and high number of boxes. Our particle distribution brings 30% of speed-up on distribution quality that reverberates on particle localization and memory occupancy.
On the other hand, the particle movement computation is no longer impacted according to the values of figure 6.6.

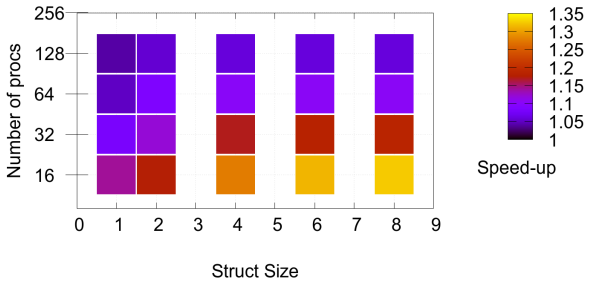
The obtained speed-ups are very low compared to the previous tests because of the starting particle distribution. As particles are equally distributed on processes and randomly initialized, each process has the same amount of particles located in all the mesh. Calling our distribution algorithm is equivalent to improving a little the particle distribution law.
Again, the distribution improvement is limited by the distribution method that consists in selecting a subset of partners with which the particles are shared.
According to the particle distribution algorithm, some particles are not sent to the right processes and are used to complete the work balance corresponding to the number of particles. For this reason, the distribution quality is lower and directly impacts the other operations especially on a high number of processes as the number of process partners is not linear compared to the total number of processes.



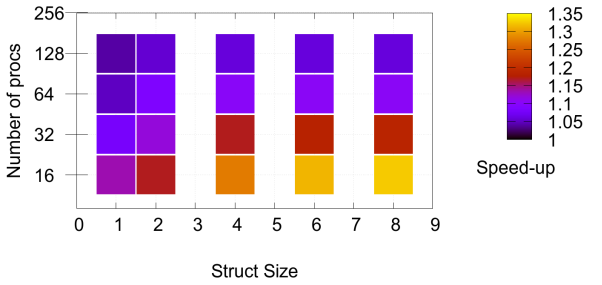
a) Localization



b) Movement



c) Distribution Quality



d) Memory Occupency

Figure 6.6: Speed-up obtained by distributing particles with the modified algorithm

Our conclusion with this final test is that our algorithm still improves particle distribution but is limited to the number of processes in the entire system because of the partners selection method. One workaround that consists in changing the partner selection method of the distribution algorithm can improve the quality distribution.

6.6 Conclusion

This chapter reveals an approach and its good performances on the execution time and on memory occupancy of the whole particle tracking solver. The modifications we brought to the original algorithm show the importance of keeping close data near to the core where they are. Taking into account some data proximity has multiple effects and positive impacts at different steps of the application: it reduces the number of communications which highly impacts the localization execution time, it increases the proximity of particles reducing the temporal and spatial locality of data during the movement of particles, and it reduces the memory occupancy on each process as the number of imported boxes.
On the other hand, the time required to distribute particles obviously increases as the modifications added instructions to the original algorithm.
The other conclusion we can make is the accuracy of the distribution quality metric. This metric represent the percentage of imported mesh parts on a process. As the quality increases, the performances also increases with the same tendencies. It means that this quality is quite useful and gives an idea of performances.

As shown in the results, as the number of processes increases, the distribution quality decreases and is close to the original algorithm for a run on 128 processes.
It means that the algorithm is not adapted for a high number of processes if we continue the particle localization and data affinity. This is an often encountered problem, when a parallel algorithm is efficient on a limited and reduced number of processes. Our simple approach to improve the particle distribution, detailed in the next chapter solves this problem by adapting a reduced environment in order to execute efficient algorithms in a close to ideal environment and case. In other words, the algorithms are executed in an encapsulated range of data close to an ideal case.

CHAPTER

7

Algorithm Adaptation to Large Parallel Systems.

Contents

7.1 Introduction

Many parallel algorithms have difficulties to scale to a large number of processes. This is generally due to the growing cost of communications. Many papers and studies have the objective to try to scale up an application for a maximum number of processes or a modern architecture and particle tracking is by far not an exception.
In fact, our implementation also shows communication difficulties in the previous chapter
The problem with particle tracking (and with several other parallelized problems especially data parallelism) is that the more the simulation is parallelized, the more the network is occupied by additional processes. The risk is then that communications occupy more space in the whole execution time. There are many techniques to accelerate parallelized applications and to maximize the involved speed-up. The main idea is in general to reduce the size of communications so as not to invade the network or to pack data in order to reduce the number of communications.

The fact is that in our library, *ParOPTIC*, the particle localization is the most expensive operation and the more the processes , the less the localization phase scales. The difficulty to scale is also found in the distribution method of the particles. Indeed, the developed algorithm quickly looses interest as it gives an interesting distribution only up to 16 processes.
We have two important algorithms that give very good results for a limited number of processes. In order to scale up to higher number of processes, we developed an approach that consists in reducing the number of processes and isolate them instead of finding and optimizing other algorithms that have better performances on higher number of processes. This way, we can approach the ideal case for an algorithm.

The next sections introduce this concept, giving some implementations for the particle localization and distribution and give general results obtained with this approach. As particle movement and face/ray impacts are only computed with local data, there is no need to adapt this part of particle tracking work flow.

7.2 Approach to adapt an efficient algorithm to large HPC systems.

The concept is very simple and has already been used by some researchers, the main idea is to regroup processes into smaller and more adapted groups than the global one formed with the processes of the entire system.
The concept can be compared to modern architectures of workstations as a group of processes corresponds to the processes located in a single NUMA node and the parallelization is adapted to each of these nodes. Each group, or node communicates with the others in range. The size of these groups are determined in order to complete a specific task. For example and in our study of particle tracking, the implemented particle distribution algorithm works for a number of processes equal to a power of 2. It means that the number of processes within a single group has to be a power of 2 to distribute particles in the same group. The figure 7.1 shows the concept in a sketch.

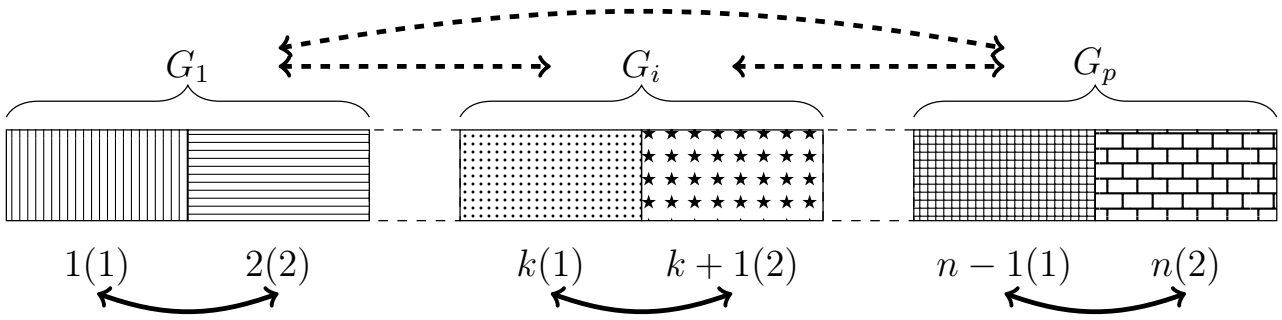


Figure 7.1: G_i is the i^{th} group composed with 2 processes. The first number is the global rank or id of the process from 1 to N, the second number between parentheses is the process local rank in the group.

In figure 7.1, decorated boxes represent a process in the global system. These processes are as-

signed to a list of operations. A total of n processes has to be distributed into p groups. In this example all groups own 2 processes. The dashed arrows represent the communications, the possible data flows between groups and the dense arrows represent the priority communications inside a single group including collective communications.

All processes of the same group execute the same instructions and multiple groups can execute the same instructions set. That is why two groups can communicate with each other. The way in which groups communicate is based on point to point communications. In other words, a set of groups can not communicate with collective communications. But an algorithm has been developed in order to allow collective communications between two groups. This is done thanks to group unions. A temporary group is created when two groups are initiating communications. This new group is the union of both communicating original groups. That way, the processes of both communicating groups can communicate with the same protocol and are able to use collective communications. When the communication operation is done between two groups, the temporary group is deleted.

The figure 7.2 gives an example of this temporary group. G_i and G_j are two groups that are composed

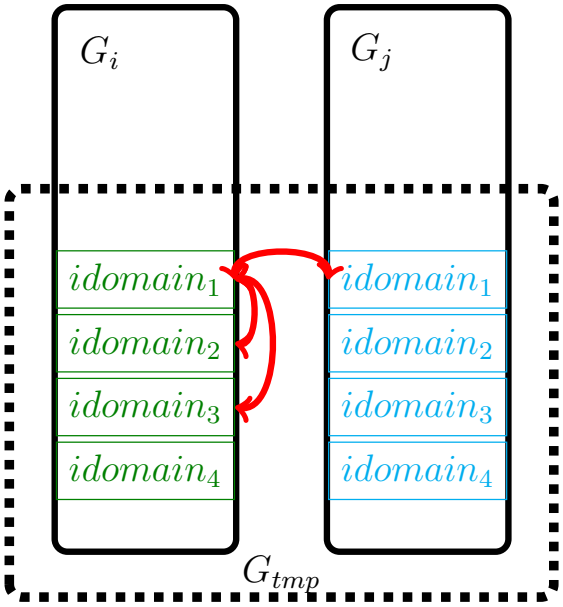


Figure 7.2: G_{tmp} is the temporary group composed with 2 groups G_i and G_j. G_i and G_j are formed with 4 processes each, so G_{tmp} allows these processes to communicate.

with 4 processes each. Each group, G_i and G_j, can distribute particles between the process inside the group. For example, the processes in group G_i, can distribute particles with the other processes inside G_i. This permit the processes inside a single group to maximize the distribution quality. But as all the groups do not track the same number of particles, groups also have to communicate and distribute particles with each other. To do so, G_{tmp} is temporary group formed with all processes of two groups, then all processes can distribute particles with all others within the global parallel context.

In the figure 7.2, the communications and the partner processes of the first process of G_i are shown. According to the implemented distribution algorithm, the total number of processes becomes 8 in G_{tmp} group and the distribution is executed.

The groups can theoretically be formed according to several qualities like the hardware proximity of processes, CPUs and accelerators, like the numerical proximity of stored data, or like any reordering function. This is theoretical, no group arrangement has been tested yet, but it can be done in an additional study. We believe that cache usage and memory proximity can be optimized using groups management.
In the same way all tested groups execute the same instructions sets. The groups are theoretically shaped to execute different instructions but no tests have been done at this moment.

For the next sections, groups are used to optimize localization phase and particle distribution.

7.3 Adaptation of Particle Localization

In this section, communication groups are used in order to optimize the particle localization operation. As a reminder, localizing a particle is the most expensive operation of the particle tracking and has been optimized to accelerate the execution. First we have chosen an algorithm efficient for any shape of the cell. In a second time, the number of candidate cells is reduced by selecting cells that own the nearest vertex of the particle. Then this nearest vertex is selected from a reduced number of candidate vertices by localizing the particle in an overlapping structured grid. Given that a structured grid describes a local mesh partition, a particle can be localized by finding the possible structured grids. Indeed, a particle can only be localized in a single cell of the computing mesh but this cell can be owned by multiple structured grids as these structured grids include ghost cells. This way, a particle can be localized in multiple structured grids.
To be brief, this first check that consists in testing if a particle is localized inside a structured grid, has an average complexity of $O(n)$ where n is the total number of structured grids. This is due to the fact that all structured grids have to be checked in order to test all mesh partition candidates where the particle can be. This complexity is linear but can be reduced by grouping processes and their mesh partitions. Indeed this complexity can become close to $O(log(n))$.
Applied to parallel particle tracking with our implementation this is completely different as particles can be localized in multiple mesh partitions. For this reason, all groups and all mesh partitions have to be checked. It means that for the particular case of particle tracking, the particle localization is always in the worst case. So the complexity is always of $O(n)$ for both approaches.

Modelisation of the group concept was very easy to implement for particle localization. We earlier introduced an internal data structure in order to identify a particle localization with a 32 bits sized object. The group id is added to this structure and the size of other members is managed. This new structure is described in figure 7.3.

In this structure presented in figure 7.3, the *igroup* member is the identity of the group, *igrid* represents the identity of the grid or the process within the $igroup^{th}$ group and *icell* is the number of the cell in $igrid^{th}$ structured grid. Just for the record, the maximum number of groups with this

```
BoxID {
        unsigned int igroup : 8;
        unsigned int igrid : 8;
        unsigned int ibox : 16;
}
```

Figure 7.3: Members of the data structure BoxID that implements the identification of a cell in an overlapping structured grid.

implementation on 32 bits is 511, as the maximum number of grid/processes per group. The maximum number of cells per grid is then 131071 cells, this can be reached with a $3D$ cubic grid of 50 cells per dimension. This is the default configuration, and can be adapted to a particular simulation.

Algorithm 10 presents the adapted algorithm to localize particles in a set of mesh partitions gathered in multiple groups.

```
for i from 0 n_groups do
    if particle ∈ group_i then
        for j from 0 n_grid[group_i] do
            if particle ∈ grid_j then
                particle is localized in a structured cell of grid_j in group_i;
            end
        end
    end
end
```

Algorithm 10: Particle Localization using process groups.

Compared to the original algorithm which consists in determining the grid where the particle is localized and then the box inside this grid, this adaptation has an additional external loop and an additional condition statement in order to localize particles in the borders of groups (sets of grids). On the other hand, the loop ranges are reduced. But as we noticed earlier, the grids all have to be checked to localize particles. This renders the complexity of both approaches equal to the worst case. In terms of execution time, the cost of the additional condition statement is added to the original algorithm.

Figure 7.4 presents the comparison of execution time with a single group of 128 grids, and multiple groups of 2, 4, 8, 16, 32, 64 and 128 grids each. The test is configured for 1 grid per process and localizes a set of 12 millions particles. The case of 128 processes in the same group corresponds to the case where no groups are used. Indeed if all processes and the associated grids are in the same group, no communication and operations between groups are called.

Figure 7.4 shows that the number of groups and the size of each group do not impact the execution time of particle localization. This is explained by the fact that the complexity does not change, all grids from all groups have to be checked as a particle can be localized in several grids as in several groups.

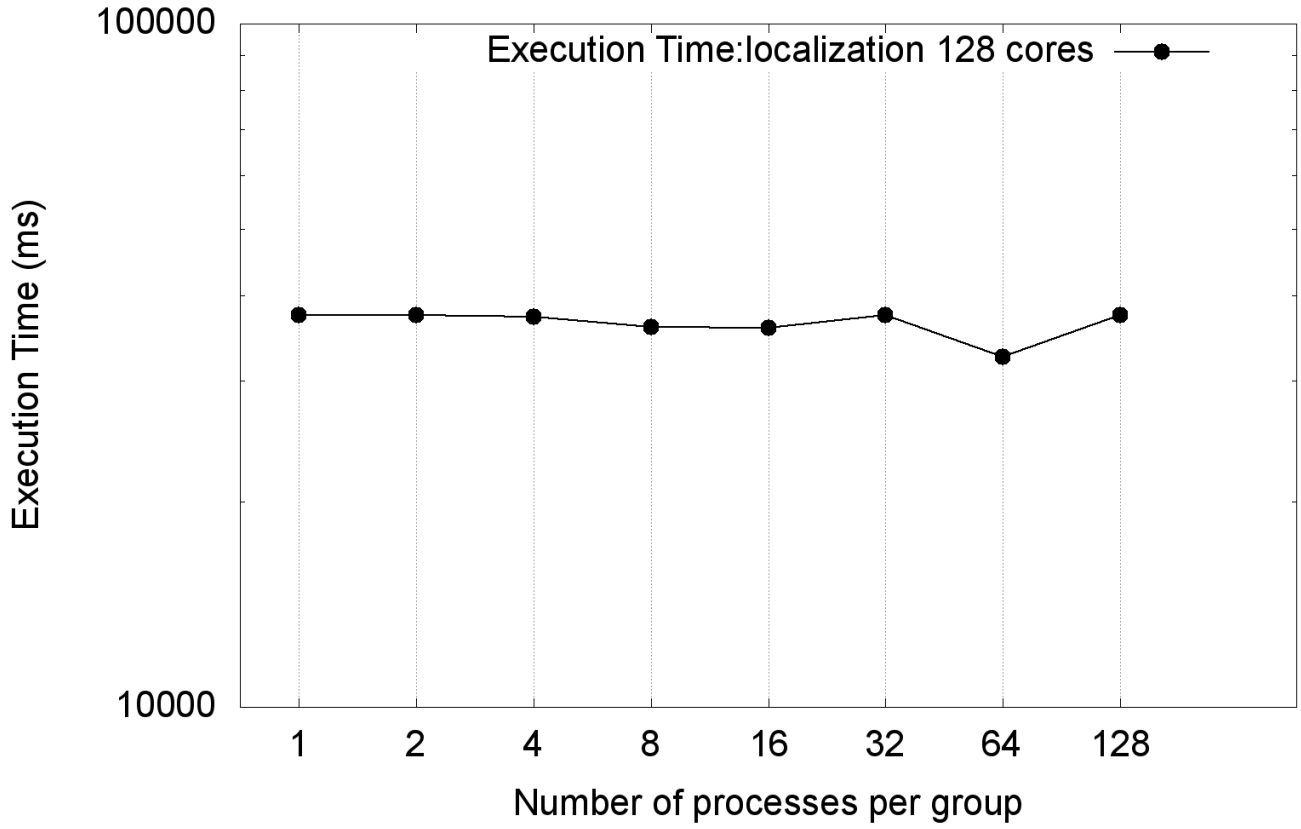


Figure 7.4: Execution Time to localize 12 millions particles. The localization operation is run in parallel on 128 processes. Some processes are gathered into groups depending on the size of a group.

7.4 Adaptation of Particle Distribution

The distribution algorithm we implemented and customized gives a high speed-up on localization operations. It means that the particle localization has an impact as it determines the number of communications needed to localize particles and the spacial and temporal locality of mesh data.
As we have noticed earlier, the quality of particle distribution becomes worse as the number of processes grows. In fact, it was revealed that the number of processes corresponding to the peak of distribution quality (and the highest speed-up obtained for particle localization and movement phases) is from 2 processes to 16 processes. For a number of processes higher than 16, the distribution quality and corresponding speed-up decrease drastically until being as efficient as a standard distribution run.
Grouping processes and executing efficient algorithm on reduced numbers of processes can make the algorithm efficient and scalable on massively parallel systems with large number of processes. As the original algorithm requires a specific number of processes that has to be a power of 2, the groups are created in order to satisfy this constraint.

The adapted and modified algorithm is explained in 11.

This adaptation 11 is very simple to implement and shows that any algorithm can be easily adapted this way. The same distribution algorithm is executed with processes owned by the both groups G_{local} and G_i at each iteration. This means that the sum of processes in G_{local} and G_i must

for $G_i \neq G_{local}$ **do**
  $G_{tmp} \leftarrow G_{local} \cup G_i$;
  $distribute\_particles(G_{tmp})$;
end

Algorithm 11: Algorithm to distribute particles on multiple groups. A pair of groups is formed and their context is unified, the distribution is executed on the process members of the couple.

be equal to a power of 2 for this particular case.

In order to create pairs of group, the algorithm of particle distribution has been chosen: a group selects a partner group iteratively in order to unify the processes involved and distribute particles with the implemented and modified algorithm. There are two main advantages: the pairs are formed in parallel, there is no concurrency between groups when partners are selected and the second advantage is the efficiency of the convergence, indeed because not all groups are visited, the number of iteration and the number of partners are reduced.
On the other hand, the disadvantages of the distribution algorithm appears because the number of groups must be a power of 2 and as a reduced number of groups are visited, it impacts the accuracy of the particle distribution quality.
Another method of partner selection can be implemented and perform better efficiency, but the general idea of process grouping in order to reduce large problems to the resolution of multiple smaller cases still remains and is appropriate.

Figure 7.5 gives the efficiency, obtained with the adaptation of the distribution algorithm according to the number of processes per group with a total number of 128 processes. The execution times are compared to the same execution with all processes gathered in a single group.

Results presented in figure 7.5 show multiple remarks : first, it confirms the previous conclusions that the distribution algorithm is very efficient in a limited range of number of processes. The performances (execution time and memory occupancy) decrease as the number of processes per group increases. The exception is the execution time of distribution operation, indeed, the more processes are in a group, the less is the execution time to distribute particles. This is due to the number of iteration of the distribution algorithm which is logarithmic. Iterating the algorithm on twice the number of processes converges faster than the sum of two groups iterations.
Another thing that can be noticed is that the peak is reached for 8 processes (16 groups of 8 processes each). This confirms previous observations.
If we compare these results with those presented previously, we can also notice that the performances are not as good as expected. In fact, the study of the distribution quality explained this lack of efficiency (figure 7.6).

Figure 7.6 presents the average quality of the particle distribution of the same run with a total of 128 processes dispatched into 1 to 128 groups. The quality gives back the trend shown by the study of efficiencies in figure 7.5. The distribution quality is quite related to the computing performances of particle tracking. In this case, the average quality is shown but in reality, the quality distribution

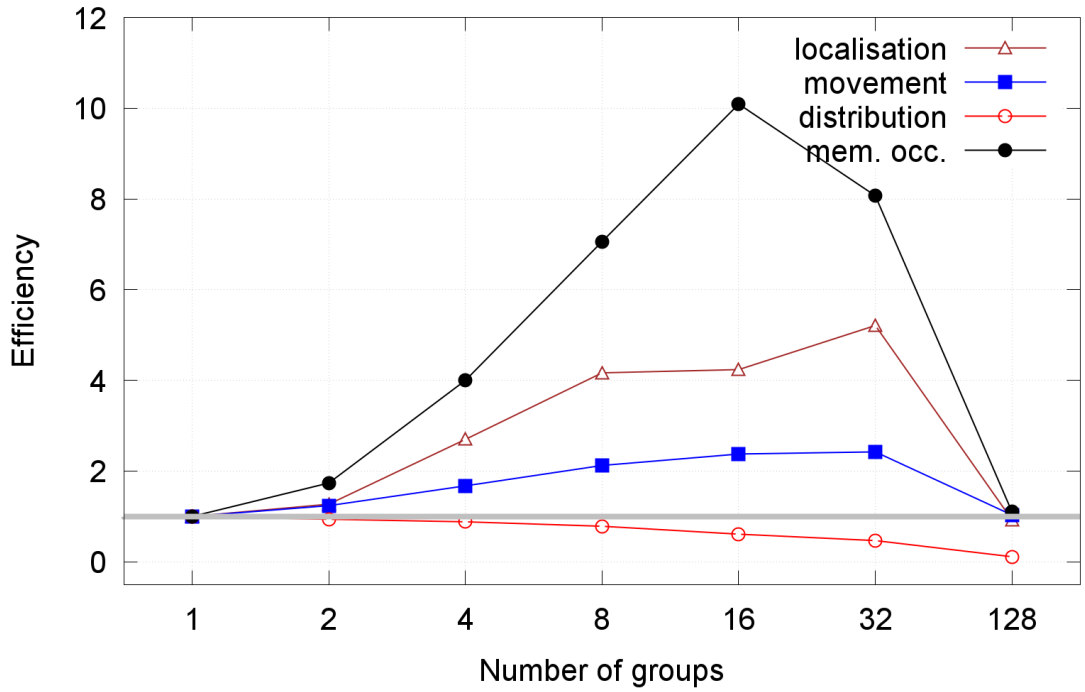


Figure 7.5: Speed-up of operations and quantities with groups of multiple sizes. The speedup are compared to the performances of the group created with 128 processes.

of each process is not well distributed. One can have an excellent particle distribution with a quality distribution of 100% (so all mesh data are local data, no communication is needed), and another can have a very bad particle distribution and receive dispatched particles as its distribution quality criterion can be less than 5%. This is again related to the chosen algorithm that makes the processes visit a limited number of partners.

The algorithm 11 shows that the approach encapsulates both groups that balance their particles. As both groups have particles not localized in this encapsulated temporary group, particles that do not remain in one process in the group are randomly dispatched instead of being organized to be sent to future partner groups.

The structural reason is that the work flow is based on the local shape of a group and does not take into account other groups. A workaround can be to pre-compute future partner groups and add these future groups and the underlying processes to the list of future partners, so that particles localized in these future partners will not be sent in priority. This workaround has not been tested during the writing of this manuscript but can be a future work in order to improve the library.

7.5 Conclusion

The approach that consists in creating groups of processes has been implemented and shows in the case of the distribution algorithm fast improvements. The implementation was quite simple and allows to easily adapt an algorithm not shaped for large number of processes.

Many tests remain to be done, for example the optimization of the distribution algorithm that is still a particular case applied to this approach. Other tests to be done that could show more general interests are the different strategies to manage groups like renumbering processes in groups, grouping

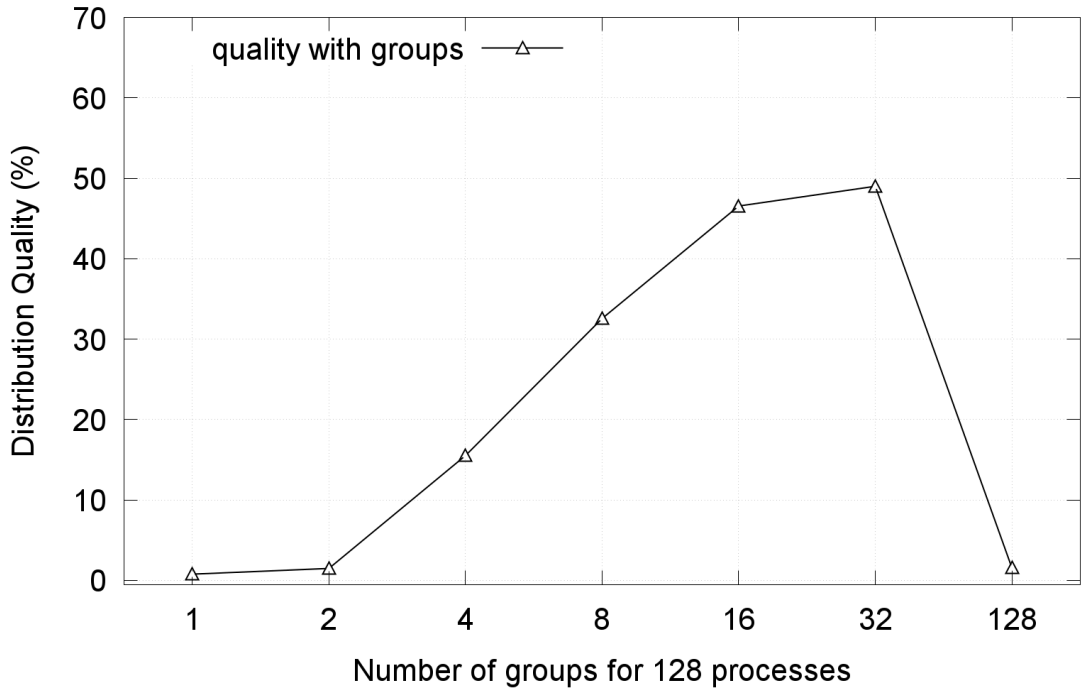


Figure 7.6: Distribution quality with groups of multiple sizes.

criteria groups balance and heterogeneity.

These tests are not done at this moment but a priority strategy to form groups, for the particular case of particle tracking, can be grouping processes that are numerically close. In other words, processes that own close particles and close mesh data are gathered in the same group in priority. Another strategy that can be adapted for particle tracking is the grouping of processes close in the hardware. This way processes of the same socket or with the same memory controller are regrouped together.

CHAPTER

8

Evaluation of the particle tracking library.

Contents

8.1 Test implementation and machines specifications.

Performances of the distribution algorithm are measured with the execution time of a parallel particle tracking simulation. Particle coordinates and their direction vectors are randomly initialized on process of rank 0, which means that all other processes have no particles to track at the beginning of the simulation. The mesh represents a three-dimensional cube discretized with cubic cells. Two situations are evaluated, the first test case measures the performances on re-injected particles. During this case, particles that come out of the global mesh are removed from the simulation and the same number of particles are randomly created in the same simulation, in the memory of the same process. The second test case measures the performances when leaving particles are not replaced. Particles leaving the simulation's mesh are simply removed and not tracked anymore. So the number of particles decreases according to the time steps.
The reason why particles and their velocity are randomly initialized is because we thought that having lot of particles in the same area as this is a particular case which is very efficient with any

approach. We want to simulate general cases where particles localization is undetermined and where the cache efficiency is undetermined and can be optimized.

The cube's dimension is $100 \times 100 \times 100$ cubic cells for a total of 1.000.000 cells. The dimensions of the overlapping structured grid used to manage traveling boxes are fixed to a cube of 4 boxes each side. It means that the grid is formed with 64 boxes per process.
In order to perform particle distribution, all particles are initialized in a single process, the first one. All other processes do not own any particle to track. This garantees the correct balance of particle distribution in terms of work balancing.

To move the particles, we choose to set a number of time steps represented by the particle/face intersection computation. Each time step, the particles move to the intersection point on the face of the cell the particle is crossing. Performances are measured on the same problem size which corresponds to the track of $12,8$ millions particles and on different number of processes. The more processes there are, the more particles and mesh are distributed on cores.
Table 8.1 summarize characteristics of the used nodes.

Node	Number of cores	CPU Model	Clock Speed (GHz)	Local Memory (Go)
1	12	Xeon X5670	2.93	48
2	12	Xeon X5670	2.93	36
3	12	Xeon X5670	2.93	36
4	12	Xeon X5670	2.93	36
5	12	Xeon X5670	2.93	36
6	12	Xeon X5670	2.93	36
7	24	AMD Opteron 6168	1.9	32
8	24	AMD Opteron 6168	1.9	32
9	32	AMD Opteron 6274	2.2	32

Table 8.1: Characteristics of computing nodes used during tests.

The distribution step is done a single time before any particle movement. This allows us to comment the distribution quality at each time steps and perhaps determine when a particle distribution can be called again to balance the workload.

To perform communications we used Message Passing Interface(MPI) on all cores initializing one MPI thread per physical thread. The machine is a set of heterogeneous nodes with different number of cores, different type of processors and different memory sizes. These nodes are connected to a single switch. This topology corresponds to a star network. The network uses gigabit ethernet technologies. The tests are run with a total of 128 MPI processes, one process per core.
The main goal of these tests is to evaluate the performances of the particle tracking library and particle distribution by studying the evolution of distribution quality for moving particles.

8.2 Parallel Reusability of the components.

A large part of this study is to use efficient software engineering techniques to develop a maintainable, easy to use and reusable in parallel and sequential contexts.
We can summarize the different blocks and structures that define the particle tracking implementation to prove and evaluate the code reusability and the library architecture efficiency.
The first chapter that treats algorithms and implementation details is chapter 4. This chapter details sequential algorithms used to solve general Lagrangian particle tracking. These algorithms are summarized in two operations: particle localization and particle next position computation. For both operations, the same algorithm is used that consists in using face-ray intersections, first to determine the particle localization compared to a polyhedron and then to compute the next interaction between the particle and the mesh.
In addition, a data structure which is an overlapping cartesian grid, is introduced in order to accelerate particle localization.
The chapter 5 talks about the techniques used to export mesh data to remote memory spaces and then parallelize particle tracking resolution. To do so, the previous cartesian grid is used and each cell of this grid is expressed as independant mesh partitions. Thus, the same data structures are used to localize particles in sequential computations and also used to communicate and export data to remote memory spaces. So the same data structures and the same operations are used in sequential and parallel contexts to localise and move particles.
This reusability is mainly due to the Lagrangian point of view of particle tracking.
The last approach is developped in order to adapt non scaling methods and algorithm that encounter difficulties to be parallelized. This approach consists in grouping processes in multiple encapsulated groups able to execute a set of instructions. This approach highly depends on the reusability of operations as the same instructions can be executed in a large group of processes as in a group containing a single process.

A lot of efforts have been made to reuse the same instructions in sequential as in parallel runs and these efforts participate in the maintainability and the upgrade of the library. In addition of the high reusability of methods, the component-based structure of the software allows to separate these methods and improve the reusability of these methods independently to the problem's nature.

8.3 Particle tracking with reinjected particles.

This test case treats moving particles during 50 time steps with re-injected particles. It means that particles that come out of the computing mesh are re-injected in the same memory space of the lost particle but with a random velocity and random coordinates. The new injected particles have to be localized.
Figure 8.1 illustrates the evolution of particle distribution quality at each time steps.

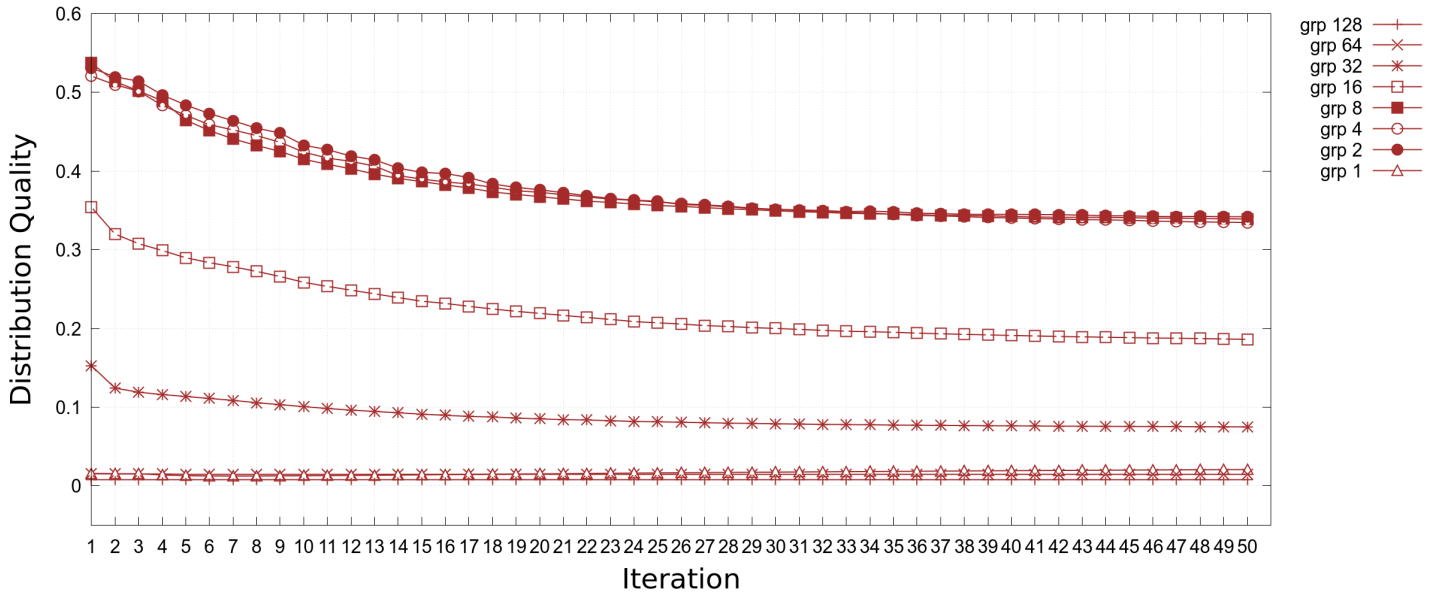


Figure 8.1: Distribution Quality for moving and re-injected particles.

The presented results in figure 8.1 include the distribution quality for a fixed grid size. The structured grid size is fixed to 4 boxes per dimension.

As shown in figure 8.1, the distribution quality decreases rapidly with the first iterations but begins to decrease slowly and continuously around the 20^{th} iteration for groups of size 2, 4 and 8 processes (sizes that correspond to the ideal number of processes where the distribution algorithm is the most efficient).

As a reminder groups distribute their particles by iterations and each iteration partner groups (groups that are going to balance particles), a new group is formed as a result of the union of the two partner groups. This transitory group then contains the sum of processes of the two original groups. Thus, the particle distribution of two groups of size 2 corresponds to the distribution of particles of 4 processes, as the particle distribution of groups of size 8 corresponds to the particle distribution of 16 processes.

This is due to the random initialization of new particles and the movement of particles already tracked. Indeed, new particles have equal chance to be localized in the local mesh than in another remote mesh partition. In addition, some external particles move from a remote mesh partition to the local one, which balances the quality loss generated by lost particles and by particles moving to a remote mesh partition.

More generally, the distribution quality decreases as iterations increase and converge to the minimal distribution quality that corresponds to the distribution quality generated by the non modified algorithm (in the case of 128 mesh partitions, the minimal quality is equal to $\frac{1}{128}$).

This effect is illustrated by group sizes that are unadapted to the distribution algorithm: the distribution quality stays low or slowly converges to this minimal limit.

8.4 Particle tracking with deleted particles.

This test treats moving particles during 50 time steps with removed particles. Particles that come out of the computing mesh are removed from the simulation and not replaced.
The main goal is to see the evolution of distribution quality when particles are removed from the simulation.
The figure 8.2 illustrates the particle distribution quality at each time steps.

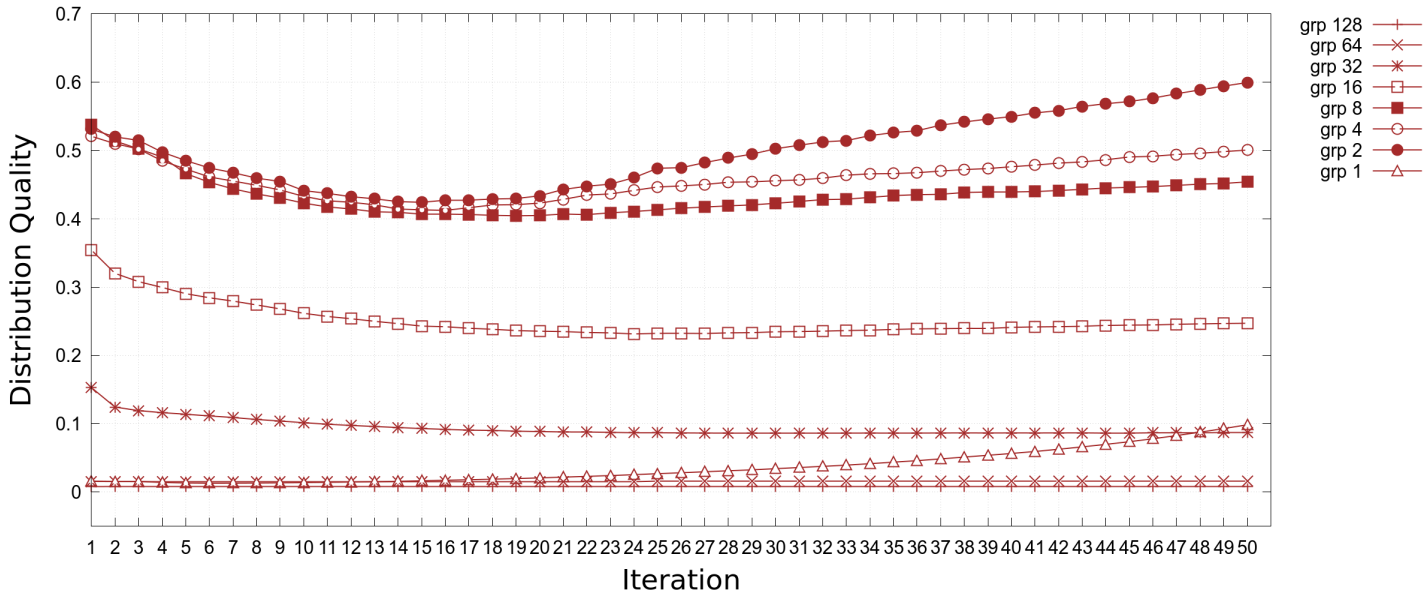


Figure 8.2: Distribution Quality for moving and deleted particles.

Statistically, the distribution quality does not strongly change as processes have the same chances to have a set of particles that leaves the simulation.
If lost particles are removed from the simulation, the distribution quality evolves interestingly: the quality decreases as observed in the case of re-injected particles, but instead of continuously decreasing in order to converge to the minimal quality, it increases starting from the 20^{th} iteration again.
This observation can be made for all sizes of groups, but the increase is more noticeable for groups of ideal size which correspond to groups of 2, 4 and 8 processes and for the smallest one, a group formed with a single process.
Our assumption is that removed particles participate in deleting imported boxes which increases the distribution quality.
Indeed, as the number of particles slowly decreases, around the 20^{th} iteration the remaining particles do not require as much boxes as at the begining of the run. The less there is remaining particles, the less the processes require external boxes.

8.5 Conclusion

The two test cases illustrate two cases often encountered in particle tracking simulations : some particles may leave the simulation by being deposed on a wall for example and some other may appear during the runtime in the case of fuel droplets continuously injected for example.
The two examples try to simulate general cases where sets of particles are injected and removed during the simulation.

These examples demonstrate that the distribution quality decreases as time goes by when the number of particles does not change drastically and is very impacted in cases where the number of droplets changes.
The idea is then to adapt the frequency of distribution calls in order to balance the particle distribution in the case of unbalanced processes when particles are removed from the simulation and in order to optimize the particle localization in memory.
More various tests have to be run because the two current examples illustrate the evolution of distribution quality in terms of data proximity. Indeed, this metric does not show the particle balance, the evolution of number of particles to track per process.
This metric, the distribution quality must also be combined with the number of particles per process. in order to represent more accurately the quality of work distribution of a simulation.

Other tests can be run, which for example show the evolution of the particle number distribution in the case of particles that are injected in the memory of a single process. In this case, the work slowly decreases as the injected particles are initialized in a single process. On the other hand, the current distribution quality that only shows the particle proximity to the mesh data, will increase for the same reasons encountered in our second test case: the particles are too few to request high number of external boxes.

CHAPTER

9

Conclusion, Discussion and Perspectives

This study made a general state of the art of operations used in particle tracking simulations with a Lagrangian point of view. There are multiple methods to localize particles and compute interactions with the environment in order to track particles in large meshes. Some of these methods have been studied, implemented and optimized for HPC applications.

A final library, *ParOPTIC*, based on component-based architecture is developped which regroups the studied operations summarized in three components: the Communication component dedicated to communication and network operations and management, the Computation component that regroups the operations on particles and meshes, especially geometric operations and the Data component that stores and manages internal data structures and brings some functions to manage external data.

The current library architecture is designed to be easily modified and updated by external actors and developers. In addition, this architecture allows to easily apply optmizations as the chosen methods are implemented in order to maximize the reusability of operations. Indeed, *ParOPTIC* reuses the same code in sequential-parallel context and also in distinct operations, for example the particle localization uses the same method as the particle boundary collision computation which consists in compute ray-plane intersections. This ray-plane intersection computation is used in sequential context as well in a massively parallel context.

This implementation is facilitated by the high parallel capability of particle tracking in a Lagrangian point of view.

Operations are optimized in order to run in a sequential context as particles are considered as independent objects in the Lagrangian point of view. It means that the performances of a run correspond to the sum of performances of the track of each particle.

Much remains to be done such as optimizations in every component but we think we can consider the structures as efficient message-based protocols as the data are stored in the form of arrays of bytes. This is why there is no need to cast any complex structures in order to be compatible with the communication protocol.

Anyway, a great deal of optimization is still to do especially in the communication component as it is the hotspot of the application. Even with the communications overlapped with some particle localizations, the application is still memory bound as shown in chapter 5. In addition, the particle proximity improvements we have done are not sufficient to improve particle localizations. On the other hand, the method has shown good speed-up for very few of messages and iterations. We believe that another method of distribution, a more accurate one, can perform better performances. The particularity of the distribution algorithm we have chosen and developped by O'Brien is that a limited number of partners are visited. It means that the number of particles that are not localized in the set of partners can be high. This number can be reduced by using another algorithm that increases the number of visited processes. Increasing the number of partners increases the quality of the distribution. For example, the list of partners can be calculated using Round-Robin scheduling. This algorithm garantees that every process visits and communicates with all other process.
The global idea does not change as the presented results proved that a distribution algorithm have to take into account the numerical data proximity with the data already stored on the different remote memory spaces.

The graph of tasks reveals the implicit synchronizations between localization and particle movement steps and, as the movement speed of an army is dictated by the slowest cart, the synchronization and the time a process is waiting, is equal to the required time to wait for the slowest one.
In the configuration of large test cases, the particle tracking is run on an heterogeneous parallel machine, so that the execution time measured is in fact the execution time of the slowest process. This is a lack of work-balance if we take into account the performances of each node. Indeed, slowest nodes must have less work to achieve than fastest ones in order to balance computation times.
There is a related problem that also concerns communications and node concurrency: in order to perform communications between nodes, some barriers are essential especially during particle distribution, when processes exchange their particles 2 by 2, and during particle localization when processes look for mesh parts and ask other processes. More researches need to be done in order to perform non rendez-vous communications with data stealing and general NUMA strategies.
Indeed, processes communicate with each other because of the exchange of data and explicit rendez-vous. In the case of data stealing or perhaps open memory spaces where processes can perform deposits and withdrawals, the processes could communicate without barrier and became entirely independent.

At this moment, boxes are sent independently. They are sent and received through multiple messages composed with a single box per message. Another approach can be implemented in order

to reduce the number of messages that consists in packing box data in a single message per asking process. If a process requests a set of boxes, these boxes are packed in a single array of byte, the sizes of the boxes are sent and then the resulting array is finally sent. This can improve communications between processes as particle exchanges have been improved by sending groups of particles and not particles one by one for the distribution operation.

The developpment of such a library is very long and difficult, but the chosen architecture allows to easily improve existing functions and add new ones. We intend to maintain and upgrade this library as particle tracking also involves more specific operations to be computed in scientific simulations. An update is in progress that consists in treating the particles data as a set of fields. The global structure that represent particles is a set of unknown fields related to particles. For example the particle coordinates are stored in a field of this structure as also the diameters of the particles.
These fields are designed to store unknown types in order to store single and double precision types, integers as reals, thanks to variadic templates of *C++11*.
The structure is based on a simple interface that consists in adding and deleting fields of different size in order to manage particles data. This structure allows to manage data easily as the storage is based on *C++* standard vectors, so the memory management is left to the language standard library. Another advantage of this approach is that the storage strategy is not important. In fact, this new structure can manage data Structure of Arrays as Array of Structures because to use SOA format, multiple fields of multiple data types can be added in the structure, whereas a single field of the same data type can be added to perform AOS data format.
Because the structure is based on a set of vectors to represent several fields, the access to these fields is performed by using field index that are integers given to the user to access data. An example of use is given in algorithm 12.

```
particle_data<real, real, integer, char, LeafID>();

/* add particles coordinates in AOS format [x1, y1, x2, y2, x3, y3, ...] */
idx_coordsAOS = particle_data.add(2*nParticles, pcoords);

/* add particles coordinates in SOA format in distinct fields [x1, x2, x3, ...][y1, y2, y3, ...] */
idx_coordsSOAx = particle_data.add(nParticles, pxcoords);
idx_coordsSOAy = particle_data.add(nParticles, pycoords);

/* add particles densitiy */
idx_density = particle_data.add(nParticles, pDensity);

/* add particles box localization */
idx_box = particle_data.add(nParticles, pBoxes);

/* get and print box localization of the third particle */
print (particle_data.get(idx_box, 3));
```

Algorithm 12: Example of use of our new data structure based on unkown fields.

This new data representation has the advantage to let the user manage the memory used by the

particles on the side of the user domain. On the other hand, the library manage a copy of particles and is allowed to distribute data for better performances. The disadvantage of this implementation concerns the memory occupancy as it requires to copy particle data as mesh data inside the Data Component.
Lots of improvements are possible because of the chosen software architecture and many can be implement in a little time.

The main ideas developped in this study are efficient and adaptable to massively parallel systems. They can be summarized by the following good practices. Traveling data must travel or be sent close to other data numerically close. For particle tracking, moving particles must be sent with neighbouring mesh data and must be sent to processes that manage mesh data numerically close to particles. In other words, the data proximity is very important and has a high impact on execution time.
The second good practice concerns the adaptation of algorithms to large HPC machines. Grouping processes can be a fast solution to adapt a code to parallel machines by enclosing processes in adapted environments to solve a part of the computation.
The third good practice concerns the software architecture. The main idea to retain is that independant components make the maintainability, the use and the development of the application.

Bibliography

[1] F. P. Kärrholm, *Numerical modelling of diesel spray injection, turbulence interaction and combustion*. Chalmers University of Technology Gothenburg, 2008.

[2] D. Darmana, N. G. Deen, and J. Kuipers, "Parallelization of an euler–lagrange model using mixed domain decomposition and a mirror domain technique: Application to dispersed gas–liquid two-phase flow," *Journal of Computational Physics*, vol. 220, no. 1, pp. 216–248, 2006.

[3] U. Schäferlein, M. Keßler, and E. Krämer, "Aeroelastic simulation of the tail shake phenomenon," 2017.

[4] L. Kang and L. Guo, "Eulerian–lagrangian simulation of aeolian sand transport," *Powder technology*, vol. 162, no. 2, pp. 111–120, 2006.

[5] R. Soulsby, C. Mead, B. Wild, and M. Wood, "Lagrangian model for simulating the dispersal of sand-sized particles in coastal waters," *Journal of Waterway, Port, Coastal, and Ocean Engineering*, vol. 137, no. 3, pp. 123–131, 2010.

[6] D. A. Steinman, "Simulated pathline visualization of computed periodic blood flow patterns," *Journal of biomechanics*, vol. 33, no. 5, pp. 623–628, 2000.

[7] "The green500." https://www.top500.org/green500/. Accessed: 2018-08-10.

[8] "Energy Aware High Perfomance Computing workshop." http://www.ena-hpc.org/. Accessed: 2018-08-10.

[9] D. Etiemble, "45-year cpu evolution: one law and two equations," *arXiv preprint arXiv:1803.00254*, 2018.

[10] J. Soman, K. Kothapalli, and P. Narayanan, "Some gpu algorithms for graph connected components and spanning tree," *Parallel Processing Letters*, vol. 20, no. 04, pp. 325–339, 2010.

[11] A. Schäfer and D. Fey, "High performance stencil code algorithms for gpgpus.," in *ICCS*, pp. 2027–2036, 2011.

[12] D. G. Merrill and A. S. Grimshaw, "Revisiting sorting for gpgpu stream architectures," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 545–546, ACM, 2010.

[13] S. J. K. K. N. P. J, "Some gpu algorithms for graph connected components and spanning tree," *Parallel Processing Letters*, vol. 20, 12 2010.

[14] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[15] M. Arora, "The architecture and evolution of cpu-gpu systems for general purpose computing," *By University of California, San Diago*, vol. 27, 2012.

[16] K. Schloegel, G. Karypis, and V. Kumar, *Graph partitioning for high performance scientific simulations*. Army High Performance Computing Research Center, 2000.

[17] J. Castro, M. Georgiopoulos, R. Demara, and A. Gonzalez, "Data-partitioning using the hilbert space filling curves: Effect on the speed of convergence of fuzzy artmap for large database problems," *Neural Networks*, vol. 18, no. 7, pp. 967–984, 2005.

[18] R. Glowinski, Q. Dinh, and J. Periaux, "Domain decomposition methods for nonlinear problems in fluid dynamics," *Computer methods in applied mechanics and engineering*, vol. 40, no. 1, pp. 27–109, 1983.

[19] C. Fonlupt, P. Marquet, and J.-L. Dekeyser, "Data-parallel load balancing strategies," *Parallel Computing*, vol. 24, no. 11, pp. 1665–1684, 1998.

[20] S. J. Plimpton, D. B. Seidel, M. F. Pasik, R. S. Coats, and G. R. Montry, "A load-balancing algorithm for a parallel electromagnetic particle-in-cell code," *Computer Physics Communications*, vol. 152, no. 3, pp. 227 – 241, 2003.

[21] M. Garcia, *Development and validation of the Euler-Lagrange formulation on a parallel and unstructured solver for large-eddy simulation*. Theses, Institut National Polytechnique de Toulouse - INPT, Jan. 2009.

[22] L. Liu and M. Barigou, "Lagrangian particle tracking in mechanically agitated polydisperse suspensions: Multi-component hydrodynamics and spatial distribution," *International Journal of Multiphase Flow*, vol. 73, pp. 80–89, 2015.

[23] M. Mahdavimanesh, A. Noghrehabadi, M. Behbahaninejad, G. Ahmadi, and M. Dehghanian, "Lagrangian particle tracking: model development," *Life Science Journal*, vol. 10, no. 8s, pp. 34–41, 2013.

[24] P. Khare, S. Wang, and V. Yang, "Modeling of finite-size droplets and particles in multiphase flows," *Chinese Journal of Aeronautics*, vol. 28, no. 4, pp. 974–982, 2015.

[25] A. J. Chorin, "Numerical solution of the navier-stokes equations," *Mathematics of computation*, vol. 22, no. 104, pp. 745–762, 1968.

[26] R. Temam, *Navier-Stokes equations: theory and numerical analysis*, vol. 343. American Mathematical Soc., 2001.

[27] V. Girault and P.-A. Raviart, *Finite element methods for Navier-Stokes equations: theory and algorithms*, vol. 5. Springer Science & Business Media, 2012.

[28] U. Ghia, K. N. Ghia, and C. Shin, "High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method," *Journal of computational physics*, vol. 48, no. 3, pp. 387–411, 1982.

[29] J. Kim and P. Moin, "Application of a fractional-step method to incompressible navier-stokes equations," *Journal of computational physics*, vol. 59, no. 2, pp. 308–323, 1985.

[30] M. W. BEALL and M. S. SHEPHARD, "A general topology-based mesh data structure," *International Journal for Numerical Methods in Engineering*, vol. 40, no. 9, pp. 1573–1596, 1998.

[31] R. B. R. C. Strawn, "Tetrahedral and hexahedral mesh adaptation for cfd problems," *Applied Numerical Mathematics*, vol. 26, 1998.

[32] K. Yamamoto, K. Tanaka, and M. Murayama, "Comparison study of drag prediction for the 4th cfd drag prediction workshop using structured and unstructured mesh methods," in *28th AIAA Applied Aerodynamics Conference*, p. 4222, 2010.

[33] S. V. P. W. Longest, "Evaluation of hexahedral, prismatic and hybrid mesh styles for simulating respiratory aerosol dynamics," *Computers & Fluids*, vol. 37, 2008.

[34] R. Biswas and R. C. Strawn, "Tetrahedral and hexahedral mesh adaptation for cfd problems," *Applied Numerical Mathematics*, vol. 26, no. 1, pp. 135 – 151, 1998.

[35] T. Ahmed, M. T. Amin, S. R. Islam, and S. Ahmed, "Computational study of flow around a naca 0012 wing flapped at different flap angles with varying mach numbers," *Global Journal of Research In Engineering*, 2014.

[36] A. Vié, H. Pouransari, R. Zamansky, and A. Mani, "Comparison between lagrangian and eulerian methods for the simulation of particle-laden flows subject to radiative heating," *Annual Research Brief*, pp. 15–27, 2014.

[37] E. Riber, V. Moureau, M. García, T. Poinsot, and O. Simonin, "Evaluation of numerical strategies for large eddy simulation of particulate two-phase recirculating flows," *Journal of Computational Physics*, vol. 228, no. 2, pp. 539–564, 2009.

[38] D. Paulhiac, *MODELING OF SPRAY COMBUSTION IN AN AERONAUTICAL BURNER*. Theses, INP Toulouse, Apr. 2015.

[39] Z. Zhang and Q. Chen, "Comparison of the eulerian and lagrangian methods for predicting particle transport in enclosed spaces," *Atmospheric Environment*, vol. 41, no. 25, pp. 5236 – 5248, 2007. Indoor Air 2005 - 10th International Conference on Indoor Air Quality and Climate (Part II).

[40] S. Subramaniam, "LagrangianâĂŞeulerian methods for multiphase flows," *Progress in Energy and Combustion Science*, vol. 39, no. 2, pp. 215 – 245, 2013.

[41] J. Capecelatro and O. Desjardins, "An eulerâĂŞlagrange strategy for simulating particle-laden flows," *Journal of Computational Physics*, vol. 238, pp. 1 – 31, 2013.

[42] M. Chiesa, V. Mathiesen, J. A. Melheim, and B. Halvorsen, "Numerical simulation of particulate flow by the eulerian-lagrangian and the eulerianâĂŞeulerian approach with application to a fluidized bed," *Computers & Chemical Engineering*, vol. 29, no. 2, pp. 291 – 304, 2005.

[43] L. Tessé and J. Lamet, "Radiative transfer modeling developed at onera for numerical simulations of reactive flows.," *AerospaceLab*, no. 2, pp. p–1, 2011.

[44] A. Haselbacher, F. Najjar, and J. Ferry, "An efficient and robust particle-localization algorithm for unstructured grids," *Journal of Computational Physics*, vol. 225, no. 2, pp. 2198 – 2213, 2007.

[45] P. Chopra and J. Meyer, "Tetfusion: an algorithm for rapid tetrahedral mesh simplification," in *Proceedings of the conference on Visualization'02*, pp. 133–140, IEEE Computer Society, 2002.

[46] R. Klein, G. Liebich, and W. Straßer, "Mesh reduction with error control," in *Visualization'96. Proceedings.*, pp. 311–318, IEEE, 1996.

[47] Y. E. Kalay, "Determining the spatial containment of a point in general polyhedra," *Computer Graphics and Image Processing*, vol. 19, no. 4, pp. 303 – 334, 1982.

[48] P. Schneider and D. H. Eberly, *Geometric tools for computer graphics*. Elsevier, 2002.

[49] J. Li and W. Wang, "Fast and robust gpu-based point-in-polyhedron determination," *Computer-Aided Design*, vol. 87, pp. 20–28, 2017.

[50] W. C. Thibault and B. F. Naylor, "Set operations on polyhedra using binary space partitioning trees," *SIGGRAPH Comput. Graph.*, vol. 21, pp. 153–162, Aug. 1987.

[51] D. Badouel, "An efficient ray-polygon intersection," in *Graphics gems*, pp. 390–393, Academic Press Professional, Inc., 1990.

[52] J. M. Snyder and A. H. Barr, *Ray tracing complex models containing surface tessellations*, vol. 21. ACM, 1987.

[53] T. Möller and B. Trumbore, "Fast, minimum storage ray/triangle intersection," in *ACM SIGGRAPH 2005 Courses*, p. 7, ACM, 2005.

[54] M. Shevtsov, A. Soupikov, A. Kapustin, and N. Novorod, "Ray-triangle intersection algorithm for modern cpu architectures," in *Proceedings of GraphiCon*, vol. 2007, pp. 33–39, 2007.

[55] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. April 2018.

[56] C. Lameter, "Numa (non-uniform memory access): An overview," *Queue*, vol. 11, no. 7, p. 40, 2013.

[57] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, "Mpi on a million processors," in *European Parallel Virtual Machine/Message Passing Interface UsersâĂŹ Group Meeting*, pp. 20–30, Springer, 2009.

[58] E. Jeannot and G. Mercier, "Near-optimal placement of mpi processes on hierarchical numa architectures," in *European Conference on Parallel Processing*, pp. 199–210, Springer, 2010.

[59] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in hpc applications," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pp. 180–186, IEEE, 2010.

[60] G. Karypis and V. Kumar, "Metis: Unstructured graph partitioning and sparse matrix ordering system, version 4.0 (2009)," *URL http://glaros. dtc. umn. edu/gkhome/views/metis. Accessed July*, 2013.

[61] G. Karypis, "Metis and parmetis," in *Encyclopedia of parallel computing*, pp. 1117–1124, Springer, 2011.

[62] R. Leland, "The chaco user's guide version 2.0," tech. rep., Technical Report SAND95-2344, Sandia National Laboratories, Albaquerque, NM 87185-1110, 1995.

[63] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *International Conference on High-Performance Computing and Networking*, pp. 493–498, Springer, 1996.

[64] L. Meng, C. Huang, C. Zhao, and Z. Lin, "An improved hilbert curve for parallel spatial data partitioning," *Geo-spatial Information Science*, vol. 10, no. 4, pp. 282–286, 2007.

[65] H. Nishimura, "Gpu computing for particle tracking," *Lawrence Berkeley National Laboratory*, 2011.

[66] M. J. O'Brien, P. S. Brantley, and K. I. Joy, "Scalable load balancing for massively parallel distributed monte carlo particle transport," tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2012.

[67] K. C. Kang, S. Cohen, R. Holibaugh, J. Perry, and A. S. Peterson, "A reuse-based software development methodology," tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1992.

[68] M. de Jonge *et al.*, *To reuse or to be reused: Techniques for component composition and construction*. PhD thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, 2003.

[69] E. Noulard and N. Emad, "A key for reusable parallel linear algebra software," *Parallel Computing*, vol. 27, no. 10, pp. 1299–1319, 2001.

[70] N. Emad and A. Sedrakian, "Toward the reusability for iterative linear algebra software in distributed environment," *Parallel Computing*, vol. 32, no. 3, pp. 251–266, 2006.

[71] N. Emad, S. Petiton, and G. Edjlali, "Multiple explicitly restarted arnoldi method for solving large eigenproblems," *SIAM Journal on scientific computing*, vol. 27, no. 1, pp. 253–277, 2005.

[72] N. Emad, O. Delannoy, and M. Dandouna, "Numerical library reuse in parallel and distributed platforms," in *International Conference on High Performance Computing for Computational Science*, pp. 271–278, Springer, 2010.

[73] M. Snir, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker, *MPI–the Complete Reference: The MPI core*, vol. 1. MIT press, 1998.

[74] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[75] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[76] D. Poirier, S. Allmaras, D. McCarthy, M. Smith, and F. Enomoto, "The cgns system," in *29th AIAA, Fluid Dynamics Conference*, p. 3007, 1998.

[77] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities," *International journal for numerical methods in engineering*, vol. 79, no. 11, pp. 1309–1331, 2009.

[78] N. M. Josuttis, *The C++ standard library: a tutorial and reference*. Addison-Wesley Professional, 2012.

[79] D. Vicente, "Tuning your mpi implementation." Barcelona Supercomputing Center, PATC, 2014.

[80] "Mpi performance topics." Livermore Computing Center, https://computing.llnl.gov/tutorials/mpi_performance/. Accessed: 2018-08-10.

[81] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt, "Openmesh - a generic and efficient polygon mesh data structure," 2002.

[82] Y. Saad, "Sparskit: A basic tool kit for sparse matrix computations," 1990.

[83] B. Chazelle, *The polygon containment problem.* Carnegie-Mellon University, Department of Computer Science, 1981.

[84] R. M. Freund and J. B. Orlin, "On the complexity of four polyhedral set containment problems," *Mathematical programming*, vol. 33, no. 2, pp. 139–145, 1985.

[85] P. K. Ghosh, "A solution of polygon containment, spatial planning, and other related problems using minkowski operations," *Computer Vision, Graphics, and Image Processing*, vol. 49, no. 1, pp. 1–35, 1990.

[86] S. J. Fortune, "A fast algorithm for polygon containment by translation," in *Automata, Languages and Programming* (W. Brauer, ed.), (Berlin, Heidelberg), pp. 189–198, Springer Berlin Heidelberg, 1985.

[87] K. Hormann and A. Agathos, "The point in polygon problem for arbitrary polygons," *Computational Geometry*, vol. 20, no. 3, pp. 131 – 144, 2001.

[88] M. R. Maxey and J. J. Riley, "Equation of motion for a small rigid sphere in a nonuniform flow," *The Physics of Fluids*, vol. 26, no. 4, pp. 883–889, 1983.

[89] C. Hirsch, *Numerical computation of internal and external flows: The fundamentals of computational fluid dynamics.* Elsevier, 2007.

[90] B. Rosa and L.-P. Wang, "Parallel implementation of particle tracking and collision in a turbulent flow," in *International Conference on Parallel Processing and Applied Mathematics*, pp. 388–397, Springer, 2009.

[91] D. Paulhiac, *Modélisation de la combustion dâĂŹun spray dans un bruleur aéronautique.* PhD thesis, 2015.

[92] J. Capecelatro and O. Desjardins, "An euler–lagrange strategy for simulating particle-laden flows," *Journal of Computational Physics*, vol. 238, pp. 1–31, 2013.

[93] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[94] O. A. R. Board, *OpenMP Application Program Interface.* OpenMP Architecture Review Board.

[95] E. Corwin and A. Logar, "Sorting in linear time-variations on the bucket sort," *Journal of computing sciences in colleges*, vol. 20, no. 1, pp. 197–202, 2004.

[96] F. Bonnier, N. Emad, and X. Juvigny, "Software architecture for parallel particle tracking with the distribution of large amount of data," in *Proceedings of the High Performance Computing Symposium*, p. 12, Society for Computer Simulation International, 2018.