

Thèse présentée pour obtenir le grade de

Docteur de l'université de Bordeaux

École Doctorale de Mathématiques et Informatique de Bordeaux

Spécialité Informatique

Emmanuel CIEREN

Molecular Dynamics for Exascale Supercomputers

Sous la direction de Raymond NAMYST & Laurent COLOMBET

Thèse soutenue le 9 octobre 2015
à la Maison de la Simulation

Jury			
M. CALVIN, Christophe	Chargé de recherche	CEA Saclay	Examineur
M. COLOMBET, Laurent	Chargé de recherche	CEA DAM-Île de France	Directeur de thèse
M. COULAUD, Olivier	Directeur de recherche	INRIA Bordeaux Sud-Ouest	Examineur
M. DESPREZ, Frédéric	Directeur de recherche	INRIA Grenoble Rhône-Alpes	Rapporteur
M. GERMANN, Timothy C.	Research Scientist	Los Alamos National Laboratory	Examineur
M. NAMYST, Raymond	Professeur	Université de Bordeaux	Directeur de thèse
M. PÉREZ, Christian	Directeur de recherche	INRIA Grenoble Rhône-Alpes	Rapporteur

Commissariat à l'énergie atomique et aux énergies alternatives (CEA)

Centre DAM-Île de France – Bruyères-le-Châtel, 91297 Arpajon Cedex

Département de Physique Théorique et Appliquée

☎ +33 (0)1 69 26 40 00

Molecular Dynamics for Exascale Supercomputers

Emmanuel CIEREN

2012 – 2015

*À Marion et ma famille, qui ont eu la patience de
me supporter dans cette épreuve ...*

Abstract

Molecular Dynamics for Exascale Supercomputers

In the exascale race, supercomputer architectures are evolving towards massively multicore nodes with hierarchical memory structures and equipped with larger vectorization registers. These trends tend to make MPI-only applications less effective, and now require programmers to explicitly manage low-level elements to get decent performance.

In the context of Molecular Dynamics (MD) applied to condensed matter physics, the need for a better understanding of materials behaviour under extreme conditions involves simulations of ever larger systems, on tens of thousands of cores. This will put molecular dynamics codes among software that are very likely to meet serious difficulties when it comes to fully exploit the performance of next generation processors.

This thesis proposes the design and implementation of a high-performance, flexible and scalable framework dedicated to the simulation of large scale MD systems on future supercomputers. We managed to separate numerical modules from different expressions of parallelism, allowing developers not to care about optimizations and still obtain high levels of performance. Our architecture is organized in three levels of parallelism: domain decomposition using MPI, thread parallelization within each domain, and explicit vectorization. We also included a dynamic load balancing capability in order to equally share the workload among domains.

Results on simple tests show excellent sequential performance and a quasi linear speedup on several thousands of cores on various architectures. When applied to production simulations, we report an acceleration up to a factor 30 compared to the code previously used by CEA's researchers.

Keywords: molecular dynamics, HPC, manycore, MPI, threads, TBB, vectorization, load-balancing, Intel® Xeon Phi™, C++, OpenCL.

Résumé

La dynamique moléculaire pour les machines exascale

Dans la course vers l'*exascale*, les architectures des supercalculateurs évoluent vers des nœuds massivement multicœurs, sur lesquels les accès mémoire sont non-uniformes et les registres de vectorisation toujours plus grands. Ces évolutions entraînent une baisse de l'efficacité des applications *homogènes* (MPI simple), et imposent aux développeurs l'utilisation de fonctionnalités de bas-niveau afin d'obtenir de bonnes performances.

Dans le contexte de la dynamique moléculaire (DM) appliquée à la physique de la matière condensée, les études du comportement des matériaux dans des conditions extrêmes requièrent la simulation de systèmes toujours plus grands avec une physique de plus en plus complexe. L'adaptation des codes de DM aux architectures exaflopiques est donc un enjeu essentiel.

Cette thèse propose la conception et l'implémentation d'une plateforme dédiée à la simulation de très grands systèmes de DM sur les futurs supercalculateurs. Notre architecture s'organise autour de trois niveaux de parallélisme: décomposition de domaine avec MPI, du *multithreading* massif sur chaque domaine et un système de vectorisation explicite. Nous avons également inclus une capacité d'équilibrage dynamique de charge de calcul. La conception orienté objet a été particulièrement étudiée afin de préserver un niveau de programmation utilisable par des physiciens sans altérer les performances.

Les premiers résultats montrent d'excellentes performances séquentielles, ainsi qu'une accélération quasi-linéaire sur plusieurs dizaines de milliers de cœurs. En production, nous constatons une accélération jusqu'à un facteur 30 par rapport au code utilisé actuellement par les chercheurs du CEA.

Mots-clés: dynamique moléculaire, HPC, architectures multi-cœurs, MPI, threads, TBB, vectorisation, équilibrage de charge, C++, Intel® Xeon Phi™, OpenCL.

Remerciements

Je tiens tout d'abord à remercier Raymond, mon directeur de thèse, de m'avoir accordé ta confiance pour mener à bien ces travaux de thèse, et d'avoir été présent pendant ces trois ans malgré la distance entre Bordeaux et Paris. Je n'ai pas encore eu le temps de regarder ta liste de nanards, mais j'aurai une pensée pour toi pour les aventures de *Doc Savage*. Merci à Laurent pour ton encadrement quotidien. J'ai beaucoup appris grâce à toi et je suis fier de compter parmi tes thésards. Je me souviendrai longtemps de tes gâteaux pour oublier les heures passées à déboguer. Un grand merci également à Laurent, pour avoir ouvert ton équipe de physiciens à un extra-terrestre mathématico-informaticien, et d'avoir patiemment répondu à toutes mes questions sur la dynamique moléculaire.

Je voudrais aussi remercier l'ensemble du jury, et en particulier Messieurs Desprez et Pérez, pour avoir pris le temps de relire mon manuscrit. Je ne manquerai pas de prendre en comptes vos remarques et conseils. Merci à Tim, à qui je dois mes séjours américains, d'avoir fait le déplacement depuis le Nouveau-Mexique, d'où je ne garde que des bons souvenirs.

J'ai une pensée spéciale pour tous ceux que j'ai pu côtoyer au CEA pendant ces trois ans. Que ce soit pour de l'administratif, des questions de C++ ou de L^AT_EX, des sorties course à pied, ou des discussions qui auront contribué (ou pas . . .) à me faire avancer, je remercie Adrien, Ahmed-Amine, Alexandre, Antoine, Aurélien, Bastian, Boris, Cédric, David, Émilien, Esther, François, Gêrôme, Gilles, Grêgoire, Guillaume, Jean, Jean-Baptiste, Jean-Bernard, Jean-Charles, Jessica, Joëlle, Jordan, Julien, Lucas, Marc, Mickaël, Nicolas, Noël, Olivier, Olivier (bis), Paul, Ronan, Sandra, Valentin, et tous les autres.

Enfin, un grand merci à toute ma famille, avec une pensée particulière à Inès, qui a relu attentivement le manuscrit et m'a ainsi permis de corriger une certaine quantité de coquilles. Merci à mes colocs, Arthur et Julie, et au demi-coloc Éric, pour les moments de décompression à coup de saucisson, pinard et de pizzas cantal-bleu-noix. Je garde le meilleur pour la fin, merci à Marion d'avoir toujours été là, de m'avoir soutenu et d'avoir pris soin de moi pendant la rédaction. Je suis conscient que ce ne fut pas une partie de plaisir, et j'espère sincèrement pouvoir te rendre un jour un peu de ce que tu m'as donné.

Summary

Abstract	v
Résumé	vii
Remerciements	ix
Summary	xi
1 Introduction	1
I Molecular Dynamics	5
2 MD Background	7
3 MD Applications: Then and Now	21
4 MD Tools	27
II ExaSTAMP Project	31
5 Motivations	33
6 Framework Architecture	41
7 Implementation and Preliminary Results	59
8 Dynamic Load Balancing	85
III Validation	103
9 Overall Performance	105
10 Ejecta Simulations of Metal Under Shock	113

11 Conclusion	119
Appendix	123
A Processors Used in our Tests	125
B Atoms and Potentials Parameters	127
Résumé Détaillé	129
Bibliography	135
List of Tables	143
List of Figures	145
List of Algorithms	147
Listings	149
Contents	151

Experiments, Modeling and Simulation

The natural purpose of Science is to describe processes occurring in nature. In most cases, direct observation limits are reached as these processes are complex and in constant interaction with others. The scientist has to recreate the process (often in a simpler manner) in a controlled environment: he performs an experiment. This approach lets the scientist investigate the behavior of the system regarding chosen conditions, the ultimate goal being the mathematical description of the observed phenomenon. The set of resulting equations is referred to as a *model*.

The processes can be described according to these principles extend over a large range of orders of magnitude in terms of length, time and mass scales. Distances can vary from 10^{-15} m in quantum mechanics to 10^{23} m in astrophysics; similarly, time and mass scale vary from 10^{-15} s and 10^{-27} kg to 10^{17} s and 10^{40} kg respectively. These particularly wide ranges point that experiment can sometimes be hopeless to conduct, which means certain phenomena will be hard if not impossible to describe. This typically happens at large scales, e.g. in astrophysics, where there are only few possibilities to verify models by observations and experiments and to thereby confirm or reject them. Many models, for example in materials science or astrophysics, consist in a large number of interacting bodies, e.g. galaxies cluster or polymers. As the number of particles frequently reaches the million, there is no hope to determine a solution of the underlying equations with paper and pencil (which happens for $n \geq 3$). Furthermore, accurate models can be really complex to solve and hardly have an analytic solution. Therefore, simplified models that are easier to solve are developed, but they come with restricted validity: for instance, the simple pendulum becomes an harmonic oscillator using the “small-angle” approximation.

These are the main reasons that accounted for the rise of computer simulation in science, alongside theoretical and experimental approaches. It has even become an essential tool in fields where experiment are too complex to develop. By computer simulation, we actually mean the mathematical prediction of a physical process issued from a model using a computer system. Let us consider a set of equations that are valid on continuous sets of time and space (e.g. Navier-Stokes equation in fluid mechanics); the sets where these equations live are being *discretized*, i.e. we are going to solve these equations on a discrete sample of these sets. Operators acting in our set of equations are being approximated by discrete operators (e.g. partial derivative can be evaluated with difference operators); we eventually obtain a discretized equation, that will be solved on the

discrete sets built right before. In some cases, it is even possible to demonstrate that the discrete solution can be as close as requested to the exact solution from the continuous equation, as long as we select enough points in our samples: this property is called *consistency* (see [60] for more details). Once the results from a simulation are interpreted, they are compared against those obtained from an experiment and/or theoretical predictions. Models can be then accepted, refined or refuted. Figure 1.1 outlines well the interactions between theory, experiment and computer simulation.

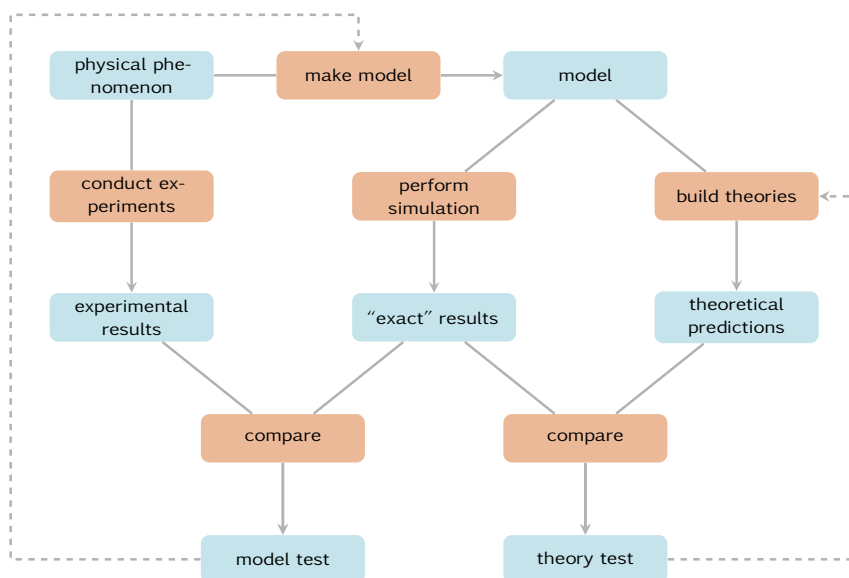


Figure 1.1 | Connection between experiment, theory and computer simulation. Freely inspired from [3].

The fact that solutions of a mathematical model are obtained from a computer, combined with the huge improvement in computer capabilities has allowed scientists to investigate more complex and realistic models than those that were reachable with analytic methods. Furthermore, computer simulation allows more flexible and less expensive experimental setups. A computer does not require safety conditions to work at a thousand atmospheres or at Kelvins, or does not mind if the phenomenon under investigation takes place for a few femtoseconds or several billion years. Computer simulation is also very useful to perform parametric study with relatively little effort. Besides, it does not mean that computer simulation can cut loose from experiment, which is the only proper way to approve or refute a model. Computer simulation eventually acts as a link or a shortcut between theory and experiments.

Particle Models

A significant portion of numerical simulation is held by *particle models*. In these representations, the goal is to track trajectories of entities, in interaction with an external force field or with each other. These entities are called *particles*, but they can represent atoms and molecules as well as stars and galaxies. Most of particle simulations follow classical laws of physics: equations of motion derivated from Newton's second law bring a system of ordinary differential equations. For a given initial state (positions and velocities) for the particles, the whole system's state and its evolution in time are supposed to be completely established.

The following list, which is obviously non-exhaustive, presents some meaningful examples of physical systems that can be represented by particle systems.

Astrophysics: In this area, simulations hardly serve other goals aside from testing theoretical models. Particles in astrophysics can be clusters of stars or entire galaxies; they interact through gravitation laws.

Biochemistry: The dynamics of macro-molecules as proteins or viruses at the atomic level is one of the most productive applications of particle methods. Examples for phenomena studied in biochemistry are molecular fluids, liquid crystals, nuclear acids . . .

Material Physics: The simulation of materials at an atomic scale is primarily used in the analysis of known materials and in the development of new ones. Particles methods are used in solid state physics to simulate metals under shock, formation of cracks initiated by pressure, propagation of sound waves, or the impact of defects in the structure of materials. This is the area of application of this thesis.

Evolution of Modern Computers

Computer simulation is not a magic tool: in most cases, complex models require a huge amount of computing time and memory and tradeoffs have to be accepted. The rise of computer simulation contributed to the expansion of several research areas: new, faster, and less memory-consuming methods and algorithms had to be worked out (e.g. fast Fourier transform by Cooley and Tukey in 1965 [25]).

During the same time, better compilers, systems and hardware kept being developed. If the term “Super Computing” was first used in 1929, in reference to large tabulators made by IBM for Columbia University, we can consider that modern *supercomputing* was born in the mid-1970s with the Cray-1. This computer popularized *vectorization*, which means that computing units are able to process similar arithmetical operations on data stored in a vector for roughly the cost of the same scalar operation. The Cray-1 successors ensured the leadership of this family until the end of the 1980s.

A parallel computer is the assembly of several processors into one computing system, these processors “talking” to each other with different means to solve a large problem together. First parallel computers consisted in two or more processors connected to the same memory unit. Yet, hardware limitation prevented such *shared memory* systems to get a high number of computing units: for instance, the Cray-2 had “only” eight processors. The 1990s match the rise of massive parallelism in supercomputers: the advent of *distributed-memory* systems, i.e. several computers connected to each other through a network, allowed to overcome several orders of magnitude in terms of number of processors [28]. It was not unusual then for a supercomputer to have several thousands of them. This breakthrough was well illustrated by the ASCI family, whose *Red* model was the first computer that reached the teraflops per second (i.e. it was able to process 10^{12} floating point operations per second) on the LINPACK benchmark [37]. This computer system, which consisted of more than 9200 processors, held the first spots of the Top500 [93].

The Top500 project has been listing and ranking the 500 most powerful computer systems in the world every six months since 1993. Computer performance is measured in flop/s, using the LINPACK benchmark. Development over time of computing capabilities from the Top500 are

showed in Figure 1.2. Nowadays, supercomputers must expand the number of cores and the use of accelerators such as GPUs to gain performance. Increasing clock speed is not an option anymore, as the ratio of flop/s over power consumption is not profitable.

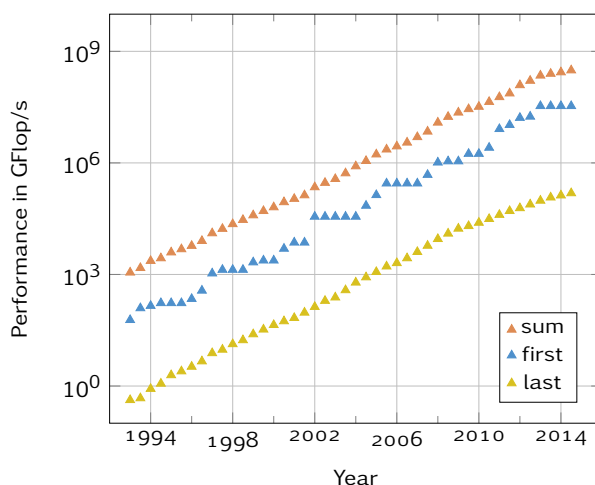


Figure 1.2 | Performance development of Top500 Supercomputers over time. All data have been extracted from [93]. The average performance has had an almost linear growth since it started. If it keeps following this progression, the Exaflop barrier (10^9 GFlop/s or 10^{18} Flops/s) should be broken by a single supercomputer around 2019. As a comparison tool, we can note that the processor in a typical modern smartphone performs roughly at a few GFlop/s.

Thesis Goals and Outline

Computer simulation is a relatively recent tool that has proven quite useful in building complex and accurate models, alongside theory and experiment. Particle methods are some approaches that successfully took advantage of its development. Yet, if performance race has allowed a significant boost in computer capabilities, the counterpart is that programming models to run efficient simulations on supercomputers has become extremely complex. This will be even more true when it comes to overcome the exaflop barrier.

For this thesis, we designed and developed a simulation framework suited for both current and future supercomputer architectures, as no available application was able to fit all our needs. The chosen area of application is a particle method, *Molecular Dynamics*. This framework, that will be used for production, will have to be general enough to cover a large range of situations. Furthermore, all programming complexity induced by performance goals should be separated and hidden from physics modules, so that non-expert developers do not need to worry about performance concerns.

After this introduction, the first part focuses on molecular dynamics (MD): Chapter 2 introduces some MD formalities, Chapter 3 gives several examples of MD applications, before going through some reference tools to perform MD simulation in Chapter 4. The next four chapters are dedicated to the project's inner core. After detailing our motivations in Chapter 5, we cover the complete design (Chapter 6) and implementation (Chapter 7). Chapter 8 describes the introduction of dynamic load-balancing in our application. The last part consists in the validation of our conception and implementation: Chapter 9 presents our framework performance in various test cases, followed by a “real-life” case-study in Chapter 10. We finally conclude and draw some perspectives in Chapter 11.

PART I

Molecular Dynamics

MOLECULAR dynamics (MD) is a method used to compute the dynamical properties of a particle system, widely spread in fields such as Materials Science, Chemistry and Biology. The main principle of MD consists in numerically integrating Newton's equation of motion for a set of particles, where the force on one particle depends on the interactions with all others. In this chapter, we will briefly introduce basic notions of Molecular Dynamics: equations, integrators, and potentials. We will refer to [3] for a more complete description.

2.1 Time Integration

To describe a molecular system, we theoretically need to solve the time-dependent version of Schrödinger's equation, for all electrons and nuclei. Despite recent progress of computing capabilities in the last fifty years, these calculations are nowadays limited to systems up to thousands of atoms¹ and a few picoseconds of simulation time [33].

In order to study phenomena as shocks or protein folding, a gain of several orders of magnitude is required. Therefore we choose to solve classical equations of motion for nuclei, which can be justified by the Born-Oppenheimer approximation. Furthermore, it is even possible to pass over all quantum effects, as in most cases, the thermal de Broglie wavelength [32] is far below the average distance between particles (see Figure 2.1). In this case, we will talk about *Classical* Molecular Dynamics; otherwise, when quantum effects must be taken into account, we will talk about *ab-initio* Molecular Dynamics.

2.1.1 Classical Molecular Dynamics Equations

Let us consider a system of N particles described by their coordinates ($\mathbf{q}_i$) and momenta ($\mathbf{p}_i$). In most cases, particles (e.g. atoms) are treated as points, and coordinates $\mathbf{q}$ consist simply in positions; however, particles can also stand for rigid molecules, thus the combination of their

¹For *ab-initio* system, the computation time depends on the number of electrons; therefore the number of atoms that can be reached for a simulation depends on the type of atom considered.

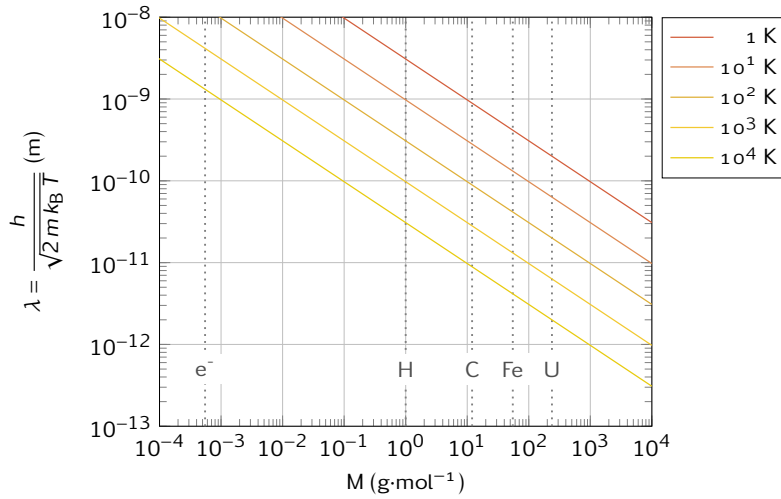


Figure 2.1 | Thermal de Broglie wavelength for different masses and temperatures. In order to omit quantum effects, the thermal de Broglie wavelength must be about an order of magnitude below a typical length of the system (e.g. the inter-particle distance). For metals around 1000 K, this wavelength is about the atoms size: all quantum effects are bounded to this extension and we can safely use classical laws of physics.

center of mass and orientation will be used. The time evolution of our system is given by [55, 56]

$$\begin{cases} \dot{\mathbf{p}} &= -\frac{\partial \mathcal{H}}{\partial \mathbf{q}} \\ \dot{\mathbf{q}} &= +\frac{\partial \mathcal{H}}{\partial \mathbf{p}} \end{cases} . \quad (2.1)$$

When we restrict ourselves to closed systems, the Hamiltonian is the sum of kinetic and potential energies functions, which respectively depend only on particles' coordinates and momenta

$$\mathcal{H} = \mathcal{K}(\mathbf{p}) + \mathcal{P}(\mathbf{q}) , \quad (2.2)$$

Besides, the kinetic energy function expresses itself with

$$\mathcal{K}(\mathbf{p}) = \frac{\mathbf{p}^2}{2m} ; \quad (2.3)$$

substituting (2.2) and (2.3) in (2.1), we obtain

$$\begin{cases} \dot{\mathbf{p}} &= -\frac{\partial \mathcal{V}}{\partial \mathbf{q}} \\ \dot{\mathbf{q}} &= \frac{\mathbf{p}}{m} \end{cases} , \quad (2.4)$$

which gives us

$$m\ddot{\mathbf{r}} = -\frac{\partial \mathcal{V}}{\partial \mathbf{r}} . \quad (2.5)$$

We recognize Newton's second equation of motion, in which the force expression is given by the negative gradient of a potential energy function. This *potential* is quite important, as it contains every single properties of the studied material; beyond numerical considerations, the precision of a simulation is directly related to the potential quality. More details about potentials will be

discussed in Section 2.3.

Equation (2.5) is numerically integrated to retrieve particles' trajectories. Given the chaotic nature of the N-body problem [4], it is delusive to access exact trajectories, especially for long time periods. Anyway, what really matters is the average behavior of trajectories, so that thermodynamical properties of the system may be computed with enough precision. This is only possible if the scheme used is a *symplectic* integrator [72], i.e. an integrator that preserves phase space metric.

2.1.2 Verlet Schemes

Beside symplecticity, other important properties for integrators are stability and precision [60]. The Verlet integrator provides all of these properties, as well as time-reversibility, for a low computational cost [73]. For the record, this method was first brought in astronomy by Delambre at the end of the 18th century, then used by Cowell and Crommelin in 1909, and also by Størmer; before being popularized in MD by Loup Verlet in 1968 [119].

Verlet Integration

In order to solve our problem, we discretize the time with a fixed time step² $\Delta t > 0$, which means we will approximate the exact solution at steps given by $t^n = n\Delta t$. The Verlet integration consists in using the central difference approximation to the second derivative term in (2.5):

$$\frac{\Delta^2 \mathbf{r}^n}{\Delta t^2} = \frac{\mathbf{r}^{n+1} - 2\mathbf{r}^n + \mathbf{r}^{n-1}}{\Delta t^2} \approx \frac{\partial^2 \mathbf{r}}{\partial t^2} = -\frac{1}{m} \frac{\partial \mathcal{V}}{\partial \mathbf{r}};$$

denoting $\mathbf{a}^n$ the acceleration computed at t^n with $-\frac{1}{m} \frac{\partial \mathcal{V}}{\partial \mathbf{r}}$, we eventually have

$$\mathbf{r}^{n+1} = 2\mathbf{r}^n - \mathbf{r}^{n-1} + \Delta t^2 \mathbf{a}^n. \quad (2.6)$$

By writing the Taylor expansions for positions at $t + \Delta t$ and $t - \Delta t$, it is easy to verify that the error for this integrator is in Δt^4 .

Yet, the major issue with this version is that velocities are not directly generated. Although they are not required for the time evolution, their knowledge is necessary to compute kinetic energy, which is needed to ensure the total energy conservation, or to exploit simulation results. Velocities can be computed from positions with a central difference approximation:

$$\mathbf{v}^n = \frac{\mathbf{r}^{n+1} - \mathbf{r}^{n-1}}{2\Delta t}. \quad (2.7)$$

This estimation is of order 2, canceling the order 4 obtained with (2.6). To overcome this, some variants have been developed. They produce exactly the same positions, but differ in the velocity evaluation: at $\mathbf{v}^{n+1/2}$ for Verlet Velocity (2.8), and at $\mathbf{v}^n$ for the Leapfrog method (2.9).

²Other methods exist with variable time-steps [102], but will not be discussed here.

Verlet Velocity:

$$\begin{cases} \mathbf{r}^{n+1} &= \mathbf{r}^n &+ \Delta t \mathbf{v}^n &+ \frac{\Delta t^2}{2} \mathbf{a}^n \\ \mathbf{v}^{n+1/2} &= \mathbf{v}^n &+ \frac{\Delta t}{2} \mathbf{a}^n \\ \mathbf{v}^{n+1} &= \mathbf{v}^{n+1/2} &+ \frac{\Delta t}{2} \mathbf{a}^{n+1} \end{cases} \quad (2.8)$$

Verlet “Leapfrog”:

$$\begin{cases} \mathbf{r}^{n+1/2} &= \mathbf{r}^n &+ \frac{\Delta t}{2} \mathbf{v}^n \\ \mathbf{v}^{n+1} &= \mathbf{v}^n &+ \Delta t \mathbf{a}^{n+1/2} \\ \mathbf{r}^{n+1} &= \mathbf{r}^{n+1/2} &+ \frac{\Delta t}{2} \mathbf{v}^{n+1} \end{cases} \quad (2.9)$$

2.2 Thermodynamical Ensembles

In statistical physics and thermodynamics, a *statistical ensemble* is an abstraction where we consider at once all the possible states a given system can reach [53]. It can be viewed as the probability distribution for the state of this system, considering external constraints. A *thermodynamic ensemble* is a specific case of statistical ensemble which has properties that are needed to derive quantities of thermodynamic systems from the laws of classical or quantum mechanics.

In other words, a thermodynamical ensemble provides a formal framework to model the action of external constraints on the system considered.

2.2.1 NVE

By construction, the integration of MD equations provides solutions in the Microcanonical ensemble, which means that the number of particles (N), the global volume of the simulation (V) and the total energy (E) remain constant over time. If we want to apply MD to other configurations where quantities like pressure or temperature are parameters instead of properties, e.g. bio-molecular system, we have to modify Newton’s equations.

2.2.2 NVT

The canonical ensemble represents a closed system in contact with a heat-bath at a defined temperature. The system can exchange thermal energy with the heat-bath, therefore its number of particles, its volume and its temperature remain constant. Various method are available to estimate MD simulation in the canonical ensemble, among them are the Andersen thermostat, the Nosé-Hoover thermostat and the Langevin thermostat.

Concerning the Andersen thermostat [6], the coupling between the system and the heat-bath is represented by stochastic forces applied on random particles. For the Nosé-Hoover thermostat [83, 84], a friction term is introduced in Newton’s equation. Its value is set according to the difference between the computed kinetic energy and the targeted kinetic energy.

The Langevin dynamics [102] consists in adding a constant friction term (dissipation) and a random force (fluctuation) in Newton’s equation. The average magnitude of the random forces and

the friction are related in a particular way, which guarantees NVT statistics. One of the advantages of the Langevin thermostat is that it operates on particles individually, and does not need to compute the total kinetic energy of the system. This can be important when performing large-scale simulation on parallel computers.

In practice, we only need to put the following fluctuation-dissipation step at the end of the desired integrator,

$$\mathbf{v}^{n+1} = \underbrace{\mathbf{v}^{n+1} - \gamma \Delta t \mathbf{v}^{n+1}}_{\text{dissipation}} + \underbrace{\sqrt{\frac{2\gamma \Delta t}{\beta m}} \mathcal{N}(0, 1)}_{\text{fluctuation}}, \quad (2.10)$$

where γ is a friction parameter, $\beta = 1/k_B T$ with T the targeted temperature and k_B the Boltzman constant, and $\mathcal{N}(0, 1)$ the reduced centered normal distribution.

2.2.3 NPT

The isothermal and isobaric case can be treated in the same way: we add another degree of freedom representing the simulation box volume. This volume is now a dynamical quantity evolving with the difference between the computed pressure and the targeted pressure, according to the Nosé-Hoover method. If we need to enforce a global constraint instead of a pressure, a similar process can be employed [88].

2.3 Potentials

In most MD simulations, particles are treated as points and the interacting force between particles is approximated as a gradient of a potential that depends on the particles' relative positions; force on particle i can be expressed as

$$\mathbf{f}_i = -\nabla_{\mathbf{r}_i} V \left(\{\mathbf{r}_j - \mathbf{r}_i\}_{j \neq i} \right). \quad (2.11)$$

The force computation is obviously the most challenging part: it contains all the physics of the simulation and can take up to 99% of its total time. Potentials from classical MD are empirical or semi-empirical; they are computed from an analytic formula, or they can be interpolated from tabulated values if the computation becomes too expensive [123].

2.3.1 Short and Long Range Potentials

In theory, MD potentials have an infinite range. To avoid excessive computation times, interactions are neglected below a given distance r_{cut} called the cutoff distance; this approximation is completely justified for solid materials, since distant atoms are “screened” by nearer atoms. Equation (2.11) becomes ($\mathcal{N}(i)$ is particle i 's neighborhood)

$$\mathbf{f}_i = -\nabla_{\mathbf{r}_i} V \left(\{\mathbf{r}_j - \mathbf{r}_i\}_{j \in \mathcal{N}(i)} \right), \text{ with } \mathcal{N}_i = \{j \neq i \mid \|\mathbf{r}_j - \mathbf{r}_i\| < r_{\text{cut}}\}. \quad (2.12)$$

The r_{cut} distance is not a global parameter for the simulation; it is specific to the potential.

In case of systems with electrostatic effects, long-range interactions cannot be omitted and special algorithms have been designed; most of them are based on Ewald's summation [44] or

on fast multipole method [54]. Unless told otherwise, all potentials raised in this thesis will be considered as short-range potentials.

2.3.2 Pair Potentials

Pair potentials are a particular but extremely spread case for potentials, where interaction between particles are reduced to pairwise interactions. Therefore, the potential between two particles will only depend on the distance between those particles; potential in (2.12) can be rewritten as

$$V\left(\{\mathbf{r}_j - \mathbf{r}_i\}_{j \in \mathcal{N}(i)}\right) = \sum_{j \in \mathcal{N}(i)} V'(r_{ij}), \text{ with } r_{ij} = \|\mathbf{r}_j - \mathbf{r}_i\|, \quad (2.13)$$

when the potential energy of the system will be

$$E_i = \frac{1}{2} \sum_{i, j \in \mathcal{N}(i)} V'(r_{ij}). \quad (2.14)$$

The purpose of the $1/2$ factor is to avoid counting energy twice, since the potential evaluates the interaction and not a single particle. In the future, we will only use V instead of V' . Pair potentials are quite effective to describe low-density materials such as liquids and gases. There are a lot of pair potentials, and we will only describe the most common ones.

Lennard-Jones Potential

Although it was first designed to study gases, the Lennard-Jones potential (LJ) [68] has been used in a large part of material science, and has become a standard benchmark for MD codes. Its expression is given by

$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^6 \right], \quad (2.15)$$

where ε and σ are parameters which denotes respectively the well depth and the bond length. The LJ potential is the sum of a repulsive ($1/r^{12}$) and an attractive term ($1/r^6$). If the attractive part represents the van der Waals force and its power of sixth can be justified [76], the power of twelfth, which is standing for the Pauli repulsion, has been chosen for its convenience; it gives good results nonetheless. Expression in equation (2.15) is plotted in Figure 2.2, and some values for parameters σ and ε are given in Table 2.1.

Morse Potential

The Morse potential [81] is a pair potential designed to study diatomic molecules, and has the advantage of offering an analytic solution to Schrödinger's equation for this case. As it contains effects such as bond breaking and unbound states, it proved itself a better approximation than the quantum harmonic oscillator to compute vibrational structure of molecules. The Morse potential only uses three parameters: an equilibrium distance r_e , a well depth D_e and a parameter a that controls the well's width; the energy function of Morse potential is

$$V(r) = D_e \left(1 - e^{-a(r-r_e)}\right)^2. \quad (2.16)$$

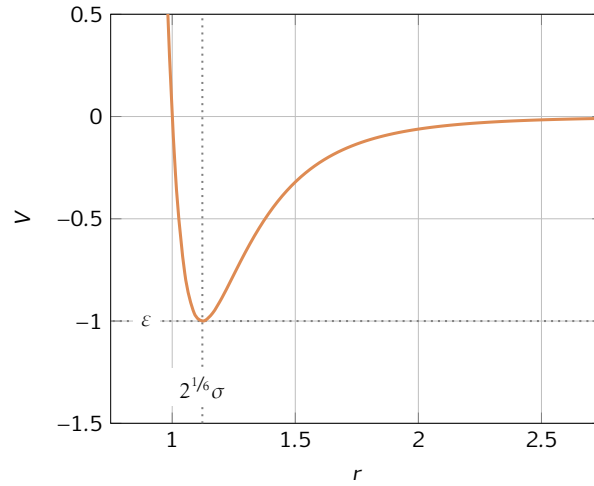


Figure 2.2 | Lennard-Jones potential. Here we represented the interaction energy given by the Lennard-Jones potential V between two particles distant by r , with $\sigma = \varepsilon = 1$.

Atom	Z	M	σ	ε
H	1	1.008	0.281	0.741
He	2	4.003	0.228	0.879
C	4	12.011	0.335	4.412
N	7	14.007	0.331	3.214
O	8	15.999	0.295	5.308
F	9	18.998	0.283	4.550
Ne	10	20.180	0.272	4.050
S	16	32.060	0.352	15.770
Cl	17	35.450	0.335	14.951
Ar	18	39.948	0.341	10.324

Table 2.1 | Parameters for Lennard-Jones potential. Z is the atomic number, M the atom mass in u, σ in nm and ε in meV; σ and ε values come from [3]. For the r_{cut} parameter, a common rule is to choose 2.5σ .

Buckingham and Exp-6 Potentials

The Buckingham potential (2.17) [21] and the Exponential-six potential (2.18) [79] are variants of the LJ potential, with an exponential instead of an inverse power of twelfth for the repulsive part:

$$V(r) = A \exp(-Br) - \frac{C}{r^6}, \quad (2.17)$$

$$V(r) = \frac{\varepsilon}{1 - 6/\alpha} \left\{ \frac{6}{\alpha} \exp \left[\alpha \left(1 - \frac{r}{r_{\min}} \right) \right] - \left(\frac{r_{\min}}{r} \right)^6 \right\}. \quad (2.18)$$

Here the parameters A , B and C (or ε , α and $r_{\min}$) are constants fitted with experimental values. Although this exponential term is more realistic to model the electronic repulsion, the Buckingham potential is a bit trickier to use compared to the LJ one; it can be explained by the computational cost, since an exponential is much more complicated to compute than a few multiplications, and by the fact that it remains finite at small distances, allowing particles to overcome the Pauli repulsion.

2.3.3 The Embedded Atom Model

The accuracy of pair potentials remains limited when it comes to bonded interactions: for the study of metals and their alloys, effects from the electron charge density have a significant impact: the Embedded Atom Method (EAM) provides a general and accurate model and an acceptable computational cost [30, 48, 31]. The energy function for a particle issued from a pair potential is based on this particle environment, which is a ball which radius is the potential cutoff distance. The main difference introduced by an EAM potential is that the energy function also includes a term based on the particle's neighbor environment: the EAM is a multi-body potential. A general form for the energy function for one particle using an EAM potential is

$$\left\{ \begin{array}{l} V(\{\mathbf{r}_j - \mathbf{r}_i\}_{j \neq i}) = \frac{1}{2} \sum_{j \in \mathcal{N}(i)} \phi(r_{ij}) + F(\bar{\rho}_i) \\ \bar{\rho}_i = \sum_{j \in \mathcal{N}(i)} \rho_j \end{array} \right. , \quad (2.19)$$

where ϕ is a pair potential, ρ_j the contribution of the electron density near atom j , and F an embedding function representing the amount of energy required to place atom i in the electron cloud. Both ϕ and ρ are canceled beyond the cutoff distance. Common EAM potentials are for instance the Johnson potential [67] or the Sutton-Chen potential (see below).

Sutton-Chen Potential

The Sutton-Chen potential (SC) is an EAM potential which provides a reasonable description of various bulk properties, with an approximate many-body representation of the delocalized metallic bonding [113]. Its expression is given by substituting the following equations in (2.19):

$$\left\{ \begin{array}{l} \phi(r) = \varepsilon \left(\frac{a_0}{r} \right)^n \\ F(\bar{\rho}) = -c \varepsilon \sqrt{\bar{\rho}} \\ \rho_j = \sum_{k \in \mathcal{N}(j)} \rho(r_{jk}) , \text{ with } \rho(r) = \left(\frac{a_0}{r} \right)^m \end{array} \right. . \quad (2.20)$$

We can identify a repulsive part given by the pairwise part (ϕ), and an attractive one, $F(\bar{\rho})$. SC potential has three dimensionless parameters that describe its “shape”: c , m and n ; m and n are integers most of the time, and we must have $n > m$. The last two parameters are ε and a_0 , which respectively describe the energy and the length scale. It can be useful when using tabulated potential: if two metals have the same parameters values for c , m and n , we only need to store one table, and then rescale the potential with a_0 and ε . Examples of parameters for Sutton-Chen potential are given in Table 2.2.

2.3.4 Other Potentials

Besides pair and multi-body potential, there are a large number of possibilities to describe the evolution of a system of particles. Three-body potentials allow the introduction of angles, which

Atom	Z	M	m	n	ϵ	c	a_0
Al	13	26.981	6	7	33.147	16.399	4.05
Ni	28	58.693	6	9	15.707	39.432	3.52
Cu	29	63.546	6	9	12.382	39.432	3.61
Rh	45	102.906	6	12	4.9371	144.41	3.80
Pd	46	106.42	7	12	4.1790	108.27	3.89
Ag	49	107.868	6	12	2.5415	144.41	4.09
Ir	77	192.217	8	10	19.833	34.408	3.92
Pt	78	195.084	6	14	2.4489	334.94	3.84
Au	79	196.967	8	10	12.793	34.408	4.08
Pb	82	207.2	7	10	5.5765	45.778	4.95

Table 2.2 | Parameters for Sutton-Chen potential. Z is the atomic number, M the atom mass in atomic mass units, ϵ in meV and a_0 in Å. All values come from [113]. For each elements, these parameters have been fitted to give good results for a given domain (pressure, temperature, state), and may differ from others in literature.

has been used by Stillinger and Weber to develop a potential for Silicon crystals [110].

Another example is the MEAM potential. MEAM was designed to take into account the geometry in the electron density term of the embedded atom model: indeed, the EAM potential term $\bar{\rho}$ is a superposition of spherically averaged electron densities, and does not depend on the local symmetry [11].

In order to have a better understanding of chemical bonds, Tersoff [116, 115] made a model to describe single, double and triple bond energies in carbon structures. This *Reactive Empirical Bond Order* potential (or REBO) was later extended to radical and conjugated hydrocarbon bonds by Brenner [20]. The REBO potential writes itself as the sum of two pair potentials, the second being weighted by a multi-body term depending on the local environment. More information about the REBO potential is available in [101].

2.4 MD Specifics

2.4.1 Truncated/Shifted Potentials

The introduction of a cutoff distance in potential induces a discontinuity of the energy function, which can be detrimental for MD simulations. As potentials are defined within a constant, they are usually shifted such that their value at the cutoff distance is zero: if we are given a potential denoted $V(r)$, we will use an effective potential $V^*(r)$ such that $V^*(r) = V(r) - V(r_{\text{cut}})$. An example for a truncated Lennard-Jones potential is presented in Figure 2.3.

It is important to notice that this shifting process does not guarantee the force continuity at the cutoff distance. If a very high accuracy in computation is required, specific adjustments will have to be performed, such as the introduction of spline functions in order to smooth the force.

2.4.2 Boundary Conditions

Periodic Boundary Conditions

The behavior of finite systems is very different from very large or infinite ones. Unless we want to describe well-defined clusters of atoms or molecules, the number of particles used to simulate

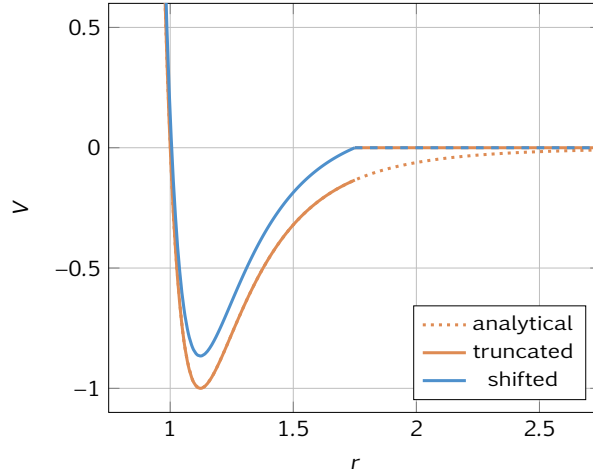


Figure 2.3 | Truncated and shifted Lennard-Jones potentials. Here we used the same parameters as Figure 2.2, except for the cutoff distance that we set to 1.75σ in order to have a better understanding of the situation.

macroscopic systems is of primary importance for the computation statistic accuracy. Despite recent and future progress in computer capabilities, this number of particles in simulation will remain negligible for a long period of time when compared to the number of particles contained in a macroscopic system ($\approx 10^{23}$ particles).

The most convenient solution to overcome this problem is to use periodic boundary conditions: particles are enclosed in a simulation box, which is replicated in all three space directions, as showed in Figure 2.4. All the “image” particles move in coordination with their original one from the simulation box, which will be the only box to be effectively simulated. If only short-range interactions are in effect, we only need to replicate a part of the box which thickness $\geq r_{\text{cut}}$. When a particle enters/leaves the simulation box, an image particle respectively leaves/enters this region, such that the number of particles remains constant. Note that the simulation box must be large enough so that size effects can be legitimately neglected.

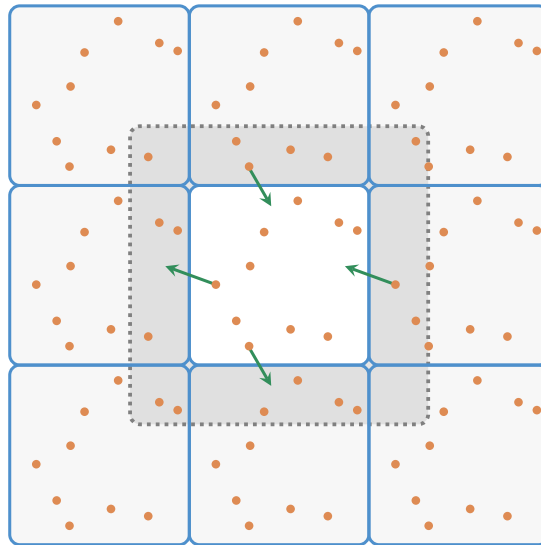


Figure 2.4 | Periodic boundary conditions. In this 2D-example, the main simulation box (white) is replicated in eight (27 for the 3D case) boxes (light gray). In case of short range interactions, only the zone delimited by the dotted line needs to be replicated. If a particle is moving out of the main simulation box (green arrow), a particle image will enter.

Fixed Boundary Conditions

In other cases, e.g. a closed system with walls, we need another type of boundary conditions, as fixed boundary conditions. Unlike periodic ones, there are two major ways to introduce fixed boundary conditions: continuous barrier potentials and rigid/semi-rigid particle walls.

Potentials used for a wall can be just a repulsive part from a well-known potential (e.g. LJ, see below) or are specific potentials designed to describe a particular surface, as the 10-4-3 potential in [109]. Particle walls consist in an arrangement of particles that remains artificially static during the simulation (as if they were of infinite mass): the electron repulsion in the chosen potential will prevent moving particles from crossing this barrier.

Let us come back to wall potentials, and take the repulsive part from a Lennard-Jones potential. If we want a smooth transition to zero for both interaction energy and force as explained in Section 2.4.1, we get something similar to

$$V_{\text{wall}}(\mathbf{r}) = \begin{cases} \left(1 - \frac{w_{\text{cut}}}{|r_{\text{dim}} - r_{\text{wall}}|}\right)^{12} & \text{for } |r_{\text{dim}} - r_{\text{wall}}| < w_{\text{cut}} \\ 0 & \text{for } |r_{\text{dim}} - r_{\text{wall}}| \geq w_{\text{cut}} \end{cases}, \quad (2.21)$$

where

- $\mathbf{r} = (r_x, r_y, r_z)$ is the particle position;
- dim is the direction (x , y or z) of a normal to the wall;
- r_{wall} is the wall position on axis O_{dim} (usually one of the simulation box boundaries), such that $|r_{\text{dim}} - r_{\text{wall}}|$ is the distance of the particle to the wall;
- w_{cut} is the wall cutoff distance.

It is easy to verify that this “potential” and its first derivative, are continuous at w_{cut} .

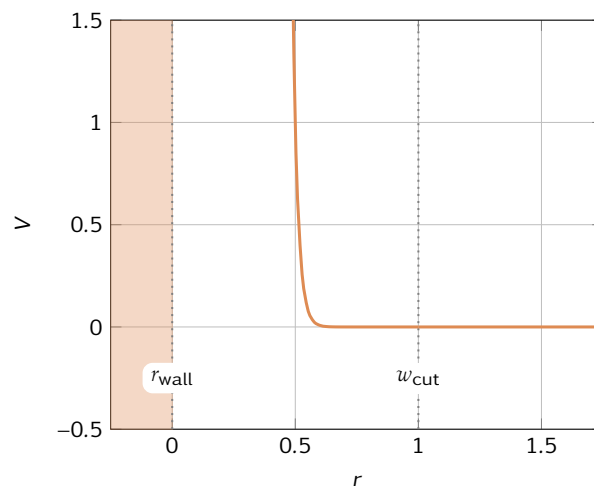


Figure 2.5 | Potential for a “wall” boundary condition. To build this “wall potential”, we took the repulsive part from a LJ potential, truncated it at a distance w_{cut} and performed adjustments to get both continuity and derivability at this distance. Parameters values are 0 for r_{wall} and 1 for w_{cut} .

2.4.3 Neighbor Search

When it comes to the force computation, each particle should get a list of its neighbors within the cutoff distance. The naive solution, which consists in running through all pairs of particles to test whether their distance is lower than the cutoff distance, is a $O(N^2)$ algorithm for the number of particles; it is clearly not suitable for large scale simulations. In order to overcome this, several methods have been developed. Here we present two possibilities widely spread in MD software to speed-up the neighbor search process: the Verlet lists and the cell-lists methods (see Figure 2.6).

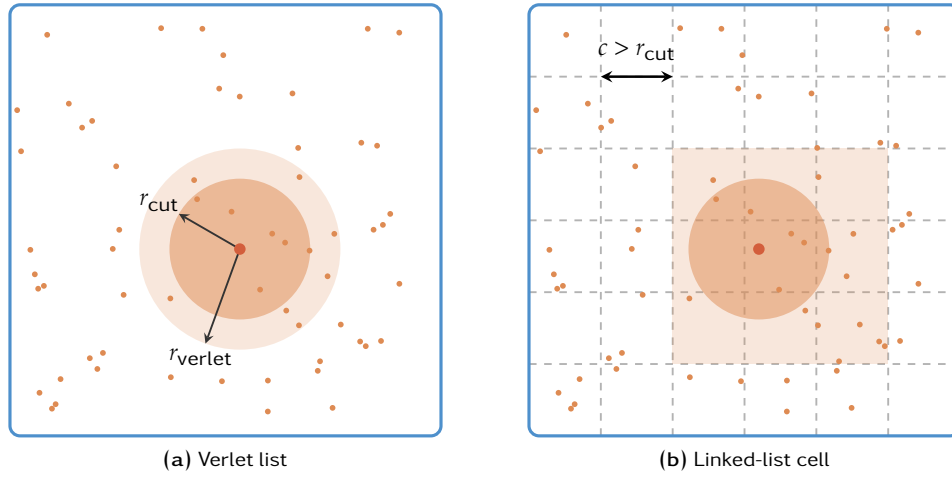


Figure 2.6 | Verlet neighbor list and linked-list cell methods.

Verlet Lists

In practice, particles move quite slowly regarding the integration time step. Therefore, between two iterations, the neighbor list of a particle is hardly modified: having to recalculate the same neighbors would certainly be a waste. The idea behind Verlet neighbor lists is to compute a list of neighbors within a distance r_{verlet} ($r_{\text{verlet}} > r_{\text{cut}}$), and to refresh this list every n ($n > 1$) iterations. The value of parameter n is chosen such that $r_{\text{verlet}} - r_{\text{cut}} > n\Delta t\bar{v}$, with Δt the simulation time step and $\bar{v}$ a typical particle velocity. A much better criterion is to keep track of the maximum displacement within the neighbor list and to relaunch the neighbor computation when the accumulation of this displacement is greater than $(r_{\text{verlet}} - r_{\text{cut}})/2$. In that case, the neighbor list update follows the system's dynamic, which ensures a better precision without wasting time in useless computations.

It is important to notice that the neighbor list search is still a $O(N^2)$ algorithm. We just minimized the call frequency of the process.

Linked-Cell Lists

Let us partition the domain with a virtual cubic mesh of size c , such that it is slightly greater than the cutoff radius. Given a particle, we only have to look for its neighbors in the cell where it lives and its neighboring cells, which makes a total of 27 cells to explore (in a three-dimensional space). This is how the linked-list cell method [3] reduces the naive pair search into a $O(N)$ algorithm.

It is also possible to blend both methods for a maximum efficiency: we keep the virtual mesh with a cell size $c > r_{\text{verlet}}$ and perform a “Verlet” style neighbor search only for particles in the neighboring cells.

2.5 MD on Parallel Computers

There are three basic ways to parallelize work in MD simulations: parallelization over particles, parallelization over forces and domain-decomposition. As explained in [92], the first two proved inefficient for large simulations, as they require too many communications over the interconnection network, leaving the third one as the only possibility despite potential load-balancing issues.

Domain Decomposition

The latter method is typically used in MPI implementations: the global domain is split and each process is assigned to a sub-domain. To compute interactions on the edges, each sub-domain will be enclosed in a ghost layer, which consists in a copy of the boundaries with its neighboring sub-domains (see Figure 2.7).

For pair potentials, the length of the ghost layer needs to be at least r_{cut} ; if we use a cell mesh for neighbor search, the ghost layer is measured in number of cells. Cells need to be up-to-date so that the force computation is correct: at each step, right before the force computation, two sessions of communications are performed:

1. particles moving from a domain to another are being exchanged;
2. once domains’ “real” part has been updated, they send their edges to their neighbors, which keeps their “ghost” part up-to-date.

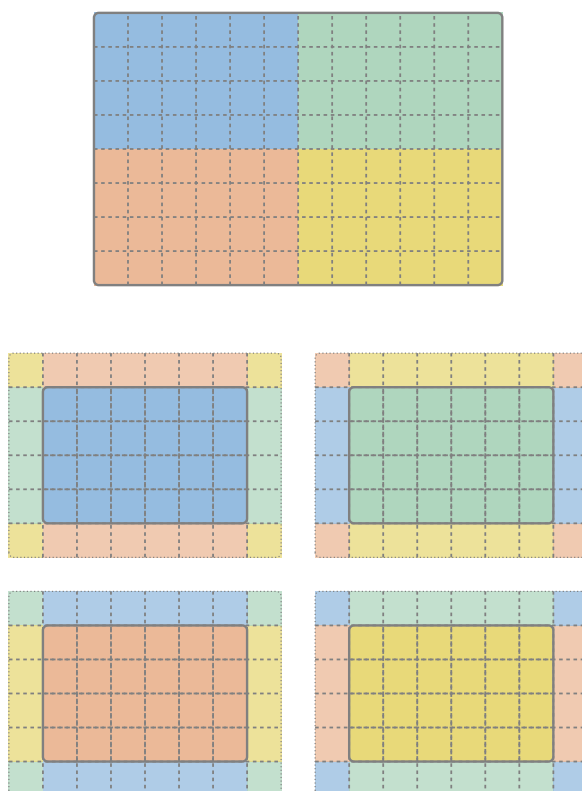


Figure 2.7 | Domain decomposition. 2D-example with periodic boundary conditions.

3

MD Applications: Then and Now

THE method of molecular dynamics is widely spread in fields such as Materials Science, Chemistry and Biology. The amount of publications regarding MD has kept increasing since its first steps in the early 1950s. Using data provided by ISI Web of Knowledge [96], Figure 3.1 shows that the ratio of the number of publications about molecular dynamics to the total number of publications (all fields combined) has more than doubled since the end of the 1980s.

In this chapter, we will begin by detailing some examples of MD application through an overview of historical simulations. We will continue with a focus on shock physics, which is an important point as it is the area of study of the work presented in this thesis, before picturing some examples of the work performed at CEA.

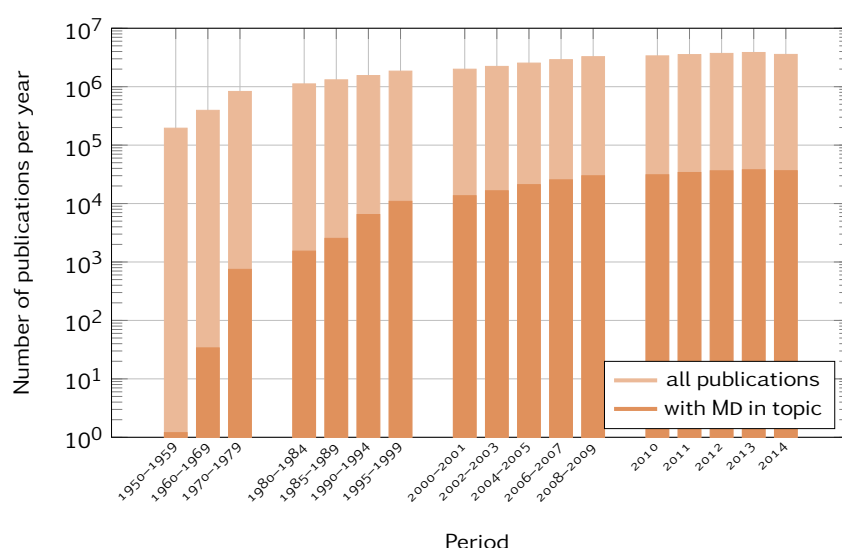


Figure 3.1 | Amount of publications for the period 1950–2014. Data extracted from ISI Web of Knowledge. Note the logarithmic scale for the y axis. The number of MD-related publications has grown faster than the total number of publications: its ratio went from less than 0.1% to 1% in 2014. We can also observe that the number of MD-related publication in the period 1950–1999 is roughly equal to the one for the period 2010–2014.

3.1 General Applications

During the early 1950s, MD was a brand new method restrained to theoretical physics due to the lack of computer capabilities (or existence ...). In 1957, Alder and Wainwright [2] were the first to perform a MD simulation using the hard-sphere model. Seven years later, Rahman employed a continuous potential for a study on liquid argon [95]. Then, thanks to its scalable structure, MD took a substantial step in the 1970s with the ever increasing computer capabilities: one of the first simulation about protein folding was achieved by Levitt in 1975 [74], who later won the 2013 Nobel prize in Chemistry with Karplus and Warshel. MD simulations then successfully coped afterwards with a million particles systems in the 1990s [62], before reaching one billion particles in 2005 [51]. In the last decade, efforts were rather concentrated on increasing models complexity and simulations time scales than on the system size. This can be partially explained by the relative difficulty in extracting relevant results from massive amounts of data. Specific methods are examined to overcome these difficulties, on a much wider area than MD itself [9, 38].

Two very productive MD applications areas are material sciences (see next section) and biology, where it has been involved in numerous scientific developments, among which we can mention the following examples (see also Figure 3.2).

- Tobacco mosaic virus (TMV) is a virus known to infect a wide range of plants, including tomato, pepper and cucumber. The infection causes specific patterns on the leaves, hence the name. As TMV is easy to produce in large quantities, and cannot be transmitted to animals, it has been used extensively in structural biology. As an example, Freddolino et al. [49] performed an all-atom molecular dynamics simulation (one million atoms for fifty nanoseconds) of the satellite tobacco mosaic virus as a whole. This simulation enabled the analysis of essential physical properties of the virus particle, which is a necessary step in the understanding of the virus infection mechanism.
- Gaucher's disease is a hereditary genetic disease caused by a recessive mutation that affects the glucocerebrosidase. This enzyme takes part in the process of assimilation of waste produced by cells. Its deficiency provokes the accumulation of fatty substances in several organs, causing enlarged spleen and liver, skeletal disorders and neurological complications among various other symptoms. Offman et al. [85] applied MD simulations to predict the molecular basis of the most common protein mutation causing Gaucher's disease. Results obtained provided a new framework for determination of the structure of other Gaucher's disease mutants and enabled suggestions about new approaches for rational drug design.

3.2 Shock Physics

Shock physics focuses on modeling matter behavior under extreme conditions (stress, pressure and/or temperature). Understanding condensed matter response under these conditions has been central to advances in fundamental science and modern technology, as they exist in many natural and man-made environments. For instance, naturally occurring high pressure and high temperature phenomena take place in meteorite impacts and in earth internal structure. For industrial applications, such conditions are used in semiconductor research [57] or new materials design and synthesis [42].

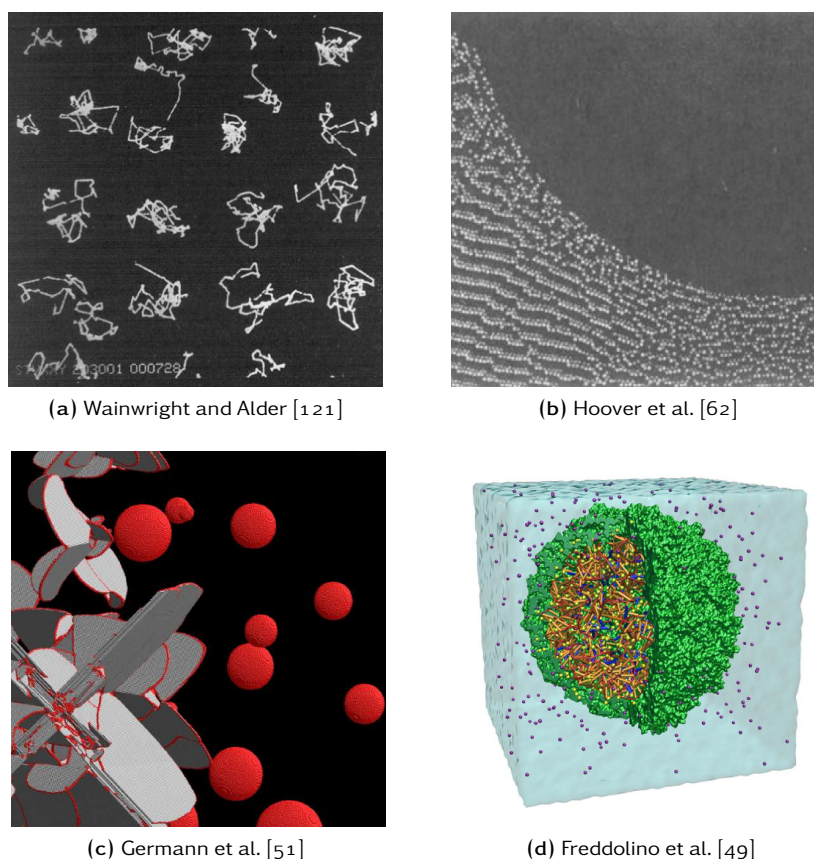


Figure 3.2 | Examples of MD simulations in material science and biology. (a) “The traces made up by a succession of positions of a 32 particle hard sphere system at a density of 1.525 in a cubic box projected onto the xy plane.” (b) “Portion of a typical configuration of the 1,036,800-atom Lennard-Jones-spline simulation showing damage below the indenter (...).” (c) “A third view of the same 2.1 billion atom simulation, at 18 ps.” (d) “Schematic Representation of the STMV Particle (...), solvated in a water box of dimensions $220 \times 220 \times 220 \text{ \AA}$.”

Example: Microscopic View of Structural Phase Transitions Induced by Shock Waves

Structural transformations in steel have been used since ancient time to forge various tools, especially in the process of hardening by rapid cooling. The underlying physics was first seriously explored at the end of 19th century by the German metallurgist Adolf Martens, who gave his name to both a steel structure (martensite), and a type of diffusion-less phase transition (martensitic transformation).

In their study, Kadau et al. [69] performed molecular-dynamics simulations in order to investigate the shock-induced phase transformation of solid iron. They highlighted a nucleation process occurring above a critical shock strength, preceded by an elastic precursor regarding the front shock direction. The dynamics and orientations of this process depend on the shock's strength and direction, and orientational relations between the unshocked and shocked regions are similar to those found for the temperature-driven martensitic transformation in iron and its alloys.

3.3 MD at CEA

At CEA, classical molecular dynamics has become a standard tool to capture a portion of the microscopic physics underlying models used in production codes working across the continuum mechanics. In this section, we present two representative examples of studies performed by CEA scientists. Another important example is the case of micro-jetting simulations. As they are part of the validation process of this thesis, they will be the object of a specific development further away; please refer to Chapter 10 for more information.

3.3.1 Molecular Dynamics Simulations of Shock Compressed Heterogeneous Materials

The transformation of graphite into diamond is observed in a variety of natural environments: diamond forms under static pressure below 150 km depth in Earth's inner layers, and it can be formed as well during impacts between planetary bodies. On Earth, diamonds have been reported in several impact craters in western Europe and Russia. Simplified physical models are usually used to explain the formation of diamond and its destabilization along the shock history. The mechanism of formation of aggregates of nanometer-sized diamond phases in some of these terrestrial and extraterrestrial samples has long been controversial and two conflicting mechanisms were put forward [47, 71]. Both scenarios are difficult to develop experimentally because real samples are rare, with a complex structure, and the time and length scales involved are quite short. On the opposite, they are likely to be addressed by high performance atomistic simulation methods which are capable of giving detailed information on the relevant time and length scales, provided that an accurate and efficient potential is available for the target materials and thermodynamic regime.

The aim of this study performed by Pineau et al. [91] is to run a set of molecular dynamics simulations designed to lighten the G/D transition in a shock compressed heterogeneous system made of a graphite inclusion embedded in a copper matrix (see details on Figure 3.3).

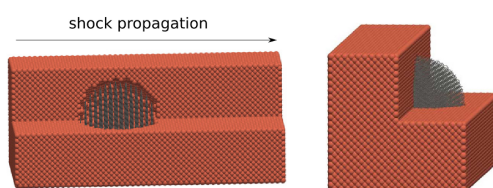


Figure 3.3 | Sample model of graphite inclusion in copper matrix. From [91]. The FCC lattice is oriented along the axis of shock propagation (x). Interactions for Cu/Cu, C/C and C/Cu were respectively modeled with SC, LCBOPII [77, 78] and LJ potentials. The integrator was a Verlet-Velocity, with a time step of 2 fs. Simulations were performed with different shapes and sizes for the carbon inclusion, using the STAMP code (see Chapter 4).

Results obtained allowed the determination of a threshold velocity between 1.2 and 1.5 kilometers per second for the G/D transition. Besides, this transition is shape and size independent within the explored range. Yet, the structuration in nanocrystallites strongly depends on the orientation of the graphite stacking with respect to the shock propagation, a co-linear stacking leading to larger crystallites, as pointed in Figure 3.4. These qualitative results agree well with the experimental observation of nanodiamonds embedded in graphitic matrices in several post mortem Martian meteorite samples. In particular, they are compatible with the antigenic mecha-

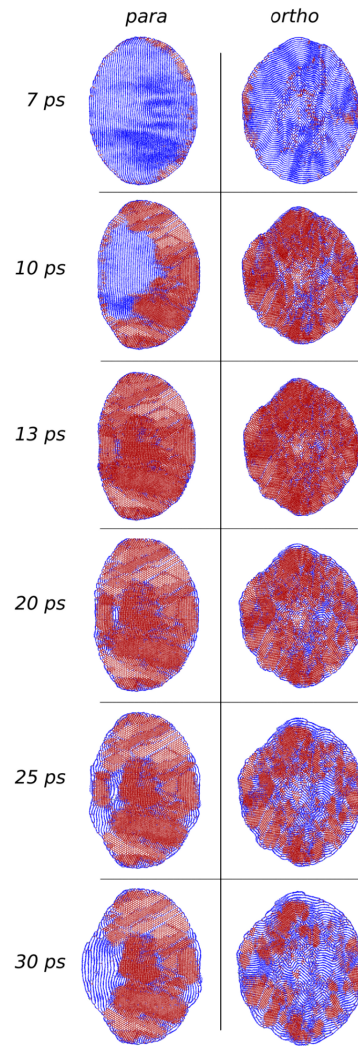


Figure 3.4 | Structural evolution of the carbon inclusion structure. From [91]. Different orientations are proposed. In the *para* case (left), graphite planes are parallel to the shock front, whereas they are perpendicular to it in the *ortho* case (right). The color code indicates the local coordination: graphite-like structures are in blue, diamond-like structures in red.

nism where pressure plays a crucial role and temperature can lead to counterproductive effects. However, the alternative allogenic mechanism cannot be assessed with this simulation design, as it involves much more complex mechanisms with time and length scales that cannot be accessed yet with MD simulations.

3.3.2 Experimental and Numerical Study of the Tantalum Spallation

Spall is a process where snippets of a material are torn off a large solid body. It can be triggered by several mechanisms, among them are projectile impact, corrosion, weathering, or cavitation. Spalling and spallation both describe the process of surface failure in which spall is ejected. In their paper, Soulard et al. investigated elementary processes of spallation in single crystal of tantalum [107]. In this case, the spallation is induced by a laser pulse which provokes an intense but unsustained shock inside the crystal. The study focuses on the number and shape of pores

resulting from the tensile inside the material when the incident shock reflects on the opposite face, by comparing results from both experiments and MD simulations.

Given the size of the samples used ($10\text{mm} \times 10\text{mm} \times 100\mu\text{m}$), a pragmatic approach to analyze experimental results is the post-mortem analysis, but there is a risk to observe effects of other phenomena which occur after the actual damaging process. Besides, post-mortem analysis does not allow a kinetic study of the spallation process. Those difficulties can be overcome by MD simulations, which are carried out at reduced scale considering the experimental sample size: the numerical sample contains about 60×10^6 atoms. After having selected and evaluated a Johnson potential [67], simulations are performed (see picture at Figure 3.5).

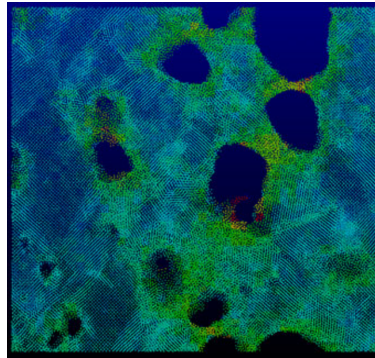


Figure 3.5 | Temperature map from MD simulation. From [107]. This sample, taken at $t = 64$ ps, shows that atomic structure is clearly disordered (amorphous material or liquid) around pores.

The analysis of number, size and shape distributions of the resulting pores reveals strong similarities with the experiment. They also exhibit power laws, which are common features of a scale invariant process. Most pores have an ellipsoid shape like a rugby ball. Yet the average orientation of pores shows some differences: it is along the shock axis propagation according to the experiment when none particular direction is found with simulations.

THERE are a lot of tools to perform MD simulation, especially in the area of biochemical systems, among them we can cite AMBER [22], GROMACS (see further), NAMD [82] and NWChem [118]. GROMACS excepted, we will focus here on MD software that primarily target material sciences, especially condensed matter and shock physics. Indeed, as biomolecular and condensed matter systems are very different, they do not share the same constraints in the software design process. A biochemical system is considered “large” when it has around a million particles, whereas a condensed matter one has to be tens or hundreds time larger. However, potentials involved in biomolecular systems are often much more complicated than an EAM. Besides, electrostatic effects cannot be neglected in proteins or polymers, and having to take into account long-range interactions highly affects data structure: for instance, both methods cited in Section 2.3.1 use a tree structure to maximize their efficiency.

Among MD software that target material science, only a few are able to run on a (very) large number of cores. In this narrow list, DL_POLY and MOLOCH can be considered as “closed”, since their source code is either under a specific license or simply unavailable. LAMMPS is eventually the simulation tool that gets the closest to our needs, this is why we will spend more time on this one.

The Mantevo project [59] gathers several open-source software packages called *mini-applications* or *miniapps*, in order to study and improve general high-performance computing applications. It covers a wide field of applications, including implicit unstructured PDE, explicit dynamics, circuit simulation and molecular dynamics. CoMD and MiniMD are two MD packages from the Mantevo project. With a limited set of features, they cannot be considered properly as MD software; we will however use them to compare performance, since they are easy to handle.

4.1 DL_POLY

DL_POLY [104] is a general purpose simulation software for classical molecular dynamics developed at Daresbury Laboratory (UK). It covers a large range of applications, with various molecular systems, force fields, boundary conditions or integration schemes available [64]. The latest version (4) provides scalable performance from workstations to supercomputers, with a parallelization based on the static/equi-spatial domain decomposition model. DL_POLY is written

in Fortran 90, and is supplied in source form under a specific license¹; however, older versions are available under BSD license.

The package neither provides thread parallelization to fully tap multicore processors, nor a dynamic load-balancing capability to compensate a poor workload distribution in the domain-decomposition process. These two points are nonetheless part of DL_POLY's future developments.

4.2 GROMACS

GROMACS [13, 94] is a multipurpose package designed to perform molecular dynamics simulations on biochemical systems from hundreds to millions of particles, which is distributed under a GNU Lesser General Public License. If it was first designed to run on powerful desktop machines and medium-sized clusters [1], it now claims itself as *“provid[ing] extremely high performance compared to all other programs”*, from laptops to supercomputers. Indeed, several algorithmic optimizations have been introduced in the code (e.g. a specific set of routines to calculate the inverse square root).

In parallel, GROMACS uses either the standard MPI communication protocol, or a custom “Thread MPI library” for single-node workstations. OpenMP is now present in most modules to run in conjunction with MPI on multicore processors, and GPU acceleration on recent NVIDIA hardware is supported as well. In recent versions, innermost loops are written in C using functions that the compiler easily transforms to SIMD² instructions. If these writings provide significant acceleration, they request an advanced knowledge in low-level coding, which means that developments must be reserved to expert users.

Considering its area of applications, GROMACS is not designed to perform large-scale shock physic simulations. We will nevertheless draw on some of its angles in our own work.

4.3 LAMMPS

LAMMPS [92, 98] is a classical molecular dynamics simulator designed for parallel computers released under a GNU Public License. Its large number of features include the simulation of atomic, polymeric, biological, metallic, or mesoscale systems, using a variety of force fields and boundary conditions. LAMMPS benchmarks include simulations up to several billions atoms on tens of thousands of cores.

Domain-decomposition and load-balancing. LAMMPS runs as well on a single desktop processor as on large cluster nodes using domain-decomposition and MPI to exchange messages. A dynamic load-balancing capability has been recently integrated to equally divide the workload between domains. Yet, methods used in this capability restrict domains' topology to rectangular shapes, when other methods which can be more efficient do not [34].

Thread-based parallelization. A thread-base parallelization with OpenMP is also present to run on multicore processors. This functionality is however only available as an additional package, meaning that it cannot be used on all LAMMPS's capabilities. Moreover, the documentation does not give a precise statement about the parallel efficiency of this package, except that it depends on many components of the input.

¹See http://www.scd.stfc.ac.uk/SCD/research/app/ccg/software/DL_POLY/40526.aspx.

²See Section 5.1.3.

In LAMMPS, all EAM potentials are tabulated, meaning that instead of computing a value from an analytic formula, the value is interpolated from a data file. Besides precision and portability concerns, this could be a major issue when using thread parallelization on a large number of cores, since in this case the force computation would be memory bound instead of being compute bound.

Architecture Limits. LAMMPS is written in C++, which is supposed to make it highly portable and is easy to modify or extend. In practice, we can see that several important structures produce *public* attributes instead of access methods: this means that the use of one of these structures depends on its implementation. A small modification in a low-level class can have a strong impact on a large part of the code, making it harder to maintain.

LAMMPS' sources are also subject to code duplication, especially in force computation routines. For instance, the formula for the Lennard-Jones potential is hard-coded at least a dozen times. If we need to introduce an optimization (i.e. change the `std::pow` call to a faster integer power) or correct a bug, the process has to be repeated several times. Besides being a problem for code maintenance, code duplication is quite confusing for new developers who try to understand an architecture. It seems that LAMMPS may be too general to cover efficiently our needs for shock physics on exascale machines.

4.4 MD in Mantevo Project

4.4.1 CoMD

SPaSM (Scalable PARallel Short-range Molecular dynamics) is a MD code which was developed in the early 1990s at Los Alamos National Laboratory by Lomdahl et al. [75, 12] in order to simulate the dynamical behavior of materials under extreme conditions. More recently, SPaSM has been re-designed to perform very large simulations on the Roadrunner Heterogeneous Supercomputer [52].

CoMD [45] is a miniapp which implementation is based on SPaSM, hence it focuses only on atomic simulation (single material), with LJ and tabulated EAM potentials. CoMD comes in three different versions: serial, MPI-flat and hybrid MPI-OpenMP. MPI-based parallelization relies on static domain decomposition; there is no possibility to balance workload between domains.

A characteristic of CoMD is that the neighbor search process is performed on the fly during the force computation. If it saves a lot of time and memory in the LJ case (or any other pair potential), the gain is not obvious in the EAM case. Besides, this strategy prevents any attempt in vectorizing the force computation.

4.4.2 MiniMD

MiniMD [59] is a simplified version of LAMMPS: it contains about 3000 lines of code when LAMMPS has more than 100,000 (this number can double if additional packages are included). This lighter version is much more convenient to test new ideas and algorithms on a large number of compute platforms; the most promising ones are meant to be integrated into LAMMPS for the benefit of the broader user community.

MiniMD's set of features is quite similar to CoMD's, with only all atom simulations (single material) available, a LJ potential and a (tabulated) EAM. As usual, the first level of parallelism is based on domain-decomposition; unlike LAMMPS, there is load-balancing capability. In the same way as CoMD, MiniMD sources are constituted of several variants:

- a reference hybrid (MPI and OpenMP) version;
- an OpenCL version (single-precision only), with MPI to run over multiple devices;
- two different version using Kokkos arrays [43];
- a version optimized by Intel®, with specific intrinsic version of LJ force function and neighbor search to run on Intel® Xeon Phi™ (see Section 5.1).

According to the documentation, MiniMD should be very close from LAMMPS in terms of performance.

4.5 MOLOCH

MOLOCH [100] is a molecular dynamics software which has been developed since 2000 at the Russian Federal Nuclear Center – Zababakhin Institute of Applied Physics (RFNC–VNIITF) in Russia. This code is dedicated to condensed-matter simulation, with a large range of potentials, thermodynamical ensembles and complicated geometry constraints implemented. Written in C++ to easily integrate new functionalities, MOLOCH can perform simulations up to several billion atoms. Code performance are boosted by specific algorithms for neighbor search and domain-decomposition; a dynamic load-balancing capability is available as well.

Yet, apart from the previous reference, there are very little information about this code, which makes deeper analysis and comparison with well-known software arduous.

4.6 STAMP

STAMP is a classical MD code designed for production which has been developed at CEA for about fifteen years [106]. Designed especially for shock physics, it covers a large range of materials using various potentials, including the simulation of shock waves in coarse-grained systems. STAMP has been used for simulations up to a billion atoms on about ten thousand cores.

In parallel, STAMP uses a classic domain-decomposition with MPI. There is neither thread-based parallelization for multicore architecture, nor data-structures to enable vectorization. Besides, the absence of dynamic load-balancing proved itself critical in some unfriendly cases (see micro-jetting simulations in Chapter 10).

PART II

ExaSTAMP Project

5

Motivations

IN the past few years, computer design has encountered significant changes, among them non-uniform memory accesses, multicore processors or the generalization of accelerators. These changes have a large impact on programming models, and thus cannot be ignored by developers who want to meet performance.

With a constant need from physicists to perform simulations of bigger and more complex systems, and considering limitations of current MD software to scale on future supercomputers, we decided to design a new MD framework to take over from STAMP.

In this chapter, we will go over this evolution in details, before itemizing different goals of STAMP's successor.

5.1 Computer Evolution

5.1.1 From SMP to Hierarchical Memory Systems

Symmetric MultiProcessing (SMP) is the most basic way to consider for a parallel architecture: two or more identical processors are connected to a shared central memory system (see figure Figure 5.1) through a system bus or a crossbar switch. These processors also usually have some fast and private memory (cache). Such systems allow any processor to work on any task wherever the data for this task are located in memory, as long as each task in the system is not executed by more than one processor at the same time. The bottleneck in the scalability of SMPs is the memory bandwidth or the cost of an efficient crossbar switch, so that such systems with more than eight processors are hardly encountered [28].

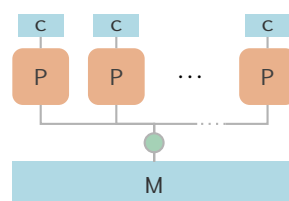


Figure 5.1 | SMP architecture. Several (identical) processors (P) with fast and private memory (c) are connected to a shared central memory system (M).

On a *Non-Uniform Memory Access* (NUMA) architecture, processors are arranged in different SMP-like nodes, as shown in Figure 5.2. All processors still share a common memory, but the latency time to access it strongly depends on the relative position between the processor accessing the data and where, i.e. in which node, the data are stored: compared to a local memory access, a distant one can be from 10 to 100% more expensive (depending on the machine hardware and topology) [8]. The more powerful computing units are, the more those effects get critical. NUMA architectures are not limited to a single level: they can be arranged in tree structures where the NUMA factor depends on both level and location of concerned nodes. As a consequence, a programmer who intends to execute efficiently some code on a NUMA architecture will have to focus on where the data will be allocated and executed [15].

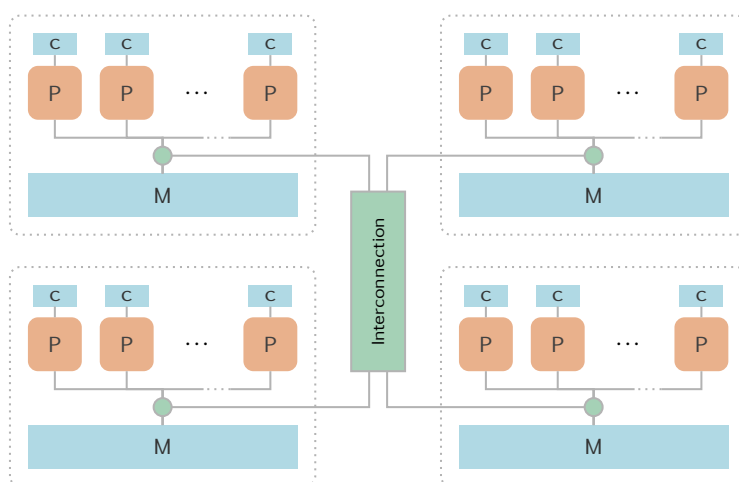


Figure 5.2 | NUMA architecture. Example with four NUMA nodes. The interconnection network used in NUMA architecture is specific to the vendor: those from Intel® and AMD are respectively called *Quick Path Interconnect (QPI)* and *HyperTransport*.

5.1.2 Multicore/Manycore Processors

Before early 2000, a common method to increase performance of computing units was to increase their clock rate. When the processor components size kept decreasing, it raised several problems in terms of power consumption and heat dissipation: indeed, the power consumption is proportional to square of the clock rate, considering only Joule heating. This small size permitted nonetheless to carve more than one computing unit on a chip: multicore processors were born.

A multicore processor is therefore a processor with two or more independent computing units on the same chip (see Figure 5.3). These units, called *cores*, can run instructions simultaneously and independently. In fact, two cores on one processor behave roughly as one SMP with two mono-core processors: the difference is that cores may share some level of cache memory, which makes communication between them much faster (at least in theory) than inter-processor communication (still within one SMP node).

Figure 5.4 shows the time-evolution in the Top500 ranking of the number of cores per socket ($\approx$ NUMA node). If the very first multicore chips were designed in the 1980s [97], it shows that multicore processors started to appear during the early 2000s, and experienced a substantial expansion in 2007. Nowadays, almost every random laptop sold has a *dual*- or a *quad*-core processor. The term *manycore* is reserved to architectures with tens or hundred of cores. Sixteen-

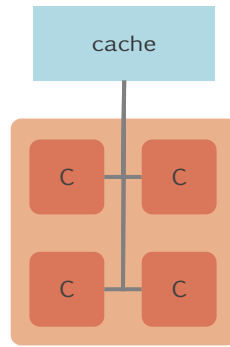


Figure 5.3 | A multicore processor. Example with a four-core (*quadcore*) processor. In practice, cores share high levels of cache (L2 or L3) and conserve a small, very fast one (L1).

cores processors (which makes NUMA nodes up to 64 or 128 cores!) are becoming standard in HPC environment.

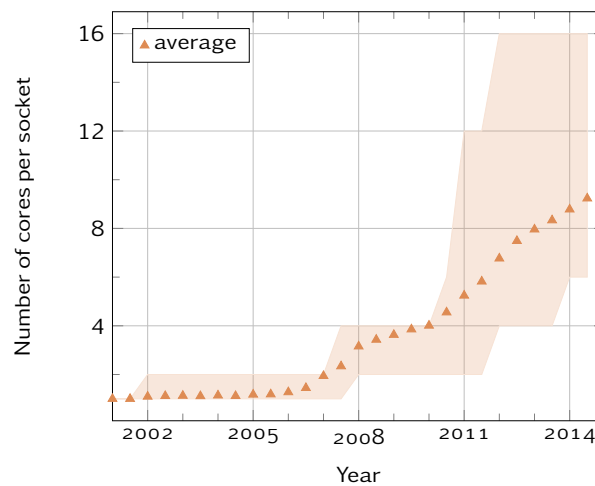


Figure 5.4 | Average number of cores per socket for Top500 supercomputers. The light orange zone delimits for each date the minimum (resp. maximum) value of the 5th (resp. 95th) percentile. This allows to get a precise trend of the moment.

Multithreading

*Multithreading*¹ is a term to describe the ability of a single computing unit to run multiple threads. For instance, when executing an application, a thread can find itself waiting for data for a certain amount of clock cycles. Another thread can use those “wasted” cycles to execute some work on its own.

This execution mode is very efficient for “memory bound” applications, e.g. applications which generate a lot of cache misses. On the other hand, multiple threads can interfere with each other when sharing hardware resources such as caches, which is especially the case for highly-optimized HPC applications [65]. This is why multithreading is often turned off on standard clusters.

¹The term *HyperThreading* is an Intel® implementation of multithreading.

5.1.3 Vectorization Returns

Vectorization, also called *Single Instruction Multiple Data* (SIMD) parallelism, is a form of parallelization in which a computing unit is able to perform a single operation on several data elements at the same time. To this end, data elements must be packed in arrays or vectors, hence the name.

Vector processing is a major feature of both conventional and modern supercomputers. It was an essential component to get performance in the early hours of supercomputing, as on systems such as Crays. Then even if it was still used in some specific applications (e.g. graphic applications or accelerated computing), vectorization has been left aside in front of more significant advances.

In 2008, Intel® and AMD proposed the AVX extension set for *x86* architectures (which is the architecture for most processors), quickly followed by AVX2 which expanded vectorization register from 128 to 256 bits [27]. In other words, it is possible to perform vector operations on eight single-precision floating-point numbers (float), or four double precision ones (double) at the same time: with such a theoretical speedup for qualified parts, vectorization must be taken into account seriously. Moreover, the Intel® Xeon Phi™ architecture (see next section) has been anticipated with 512 bits vectorization registers [66], making it critical to get decent performance.

5.1.4 Accelerators

Application-specific accelerating hardware with specific capabilities have been used for a long period of time in the context of scientific computing, with *Application Specific Integrated Circuits* (ASIC) in 1980, *Field-Programmable Gate Arrays* (FPGA) in 1985 and ClearSpeed boards, which were the first computing accelerators to enter the Top500 in 2006. In spite of significant advantages in terms of computation performance and power consumption, these accelerators are produced in very small amounts and cannot compete with graphic cards which are driven by video game industry.

GPU– GPGPU

Graphics Processing Units (GPUs) are a typical example of accelerators first designed to provide a response for a specific compute-intensive task, and used later for other purposes. At first, programmers had to go through graphic APIs to write code on GPUs, which restricted the use of such hardware from non-experts. Later, high-level environments were developed, which enabled the implementation of general purpose algorithms on top of GPUs and marked the advent of *General Purpose computing on GPUs* (GPGPU) [87].

Nowadays, the development of APIs such as CUDA and OpenCL in connection to the implementation of standard math library on GPUs offered mainstream developers the possibility to exploit their computing capabilities. Yet, obtaining significant acceleration is either reserved for a specific range of applications or for the result of a long and difficult optimization process. Figure 5.5, which represents computer systems using accelerators within Top500, well illustrates this trend: in 2014, about 10% systems were equipped with graphic-based accelerators, but these systems contributed to almost 20% of the total performance (in Flop/s).

Hybrid Manycore Processors

Besides external accelerating devices, several processors also feature heterogeneous processing cores that permit to offload compute intensive or critical operations on specific pieces of hardware.

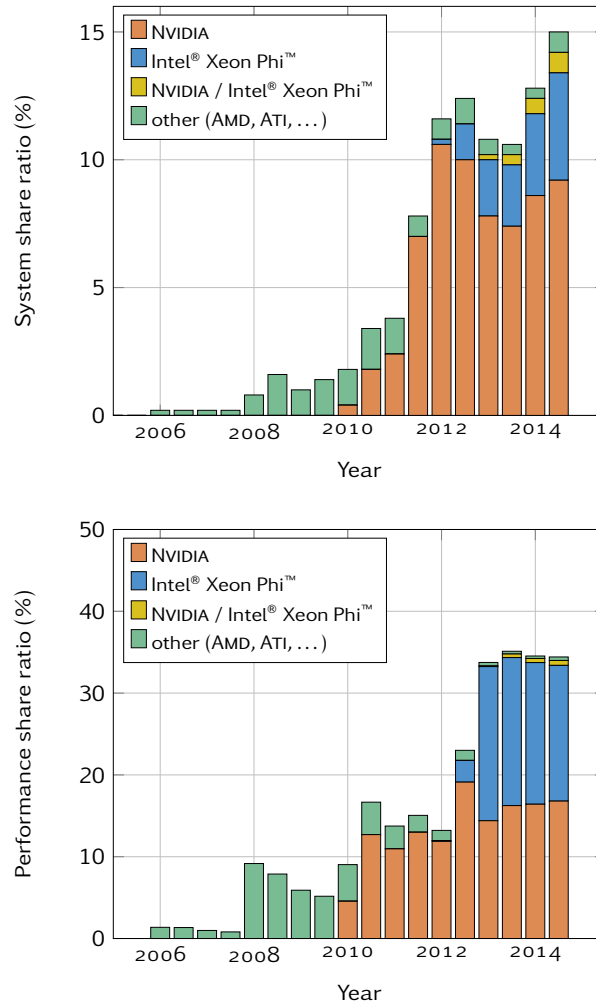


Figure 5.5 | Ratio of Top500 supercomputers with an accelerator. Top is in number of systems (out of 500), bottom is the total performance (Flop/s) ratio. The difference in terms of ratio can easily be explained: the Top500 project use the LINPACK benchmark to rank computer systems, which is basically dense linear algebra operations, which are known to perform very well on GPUs. If real-life applications were used, the performance ratio would drop substantially.

For instance, this was the case for the STI's (Sony, Toshiba and IBM) *Cell* processor, which is first known to have equipped Sony's PlayStation 3, but has also been used on HPC platforms. We can cite as well the *Fusion* series from AMD as well, which had the idea of integrating an accelerator within the processor to reduce data transfers. However, none of these architectures managed to compete with GPUs, being either too complex to program or to upgrade.

In order to bring a response to GPGPU, Intel® chose to integrate GPUs' characteristics (large number of cores, high-degree of multithreading and advanced vector capabilities) into a *x86*-compatible architecture, called Intel® *Many Integrated Core* (MIC). This compatibility is a huge effort towards programming flexibility, since a classic C/C++ or Fortran code with thread parallelization should compile and run without any modification. If raw performance is not expected to meet those of most powerful GPUs, paying the price for this effort, its peak should be much more easier to reach. In practice, a high-level of performance on such architecture will not be achieved without deep modifications. These modifications are supposed to be beneficial on "standard" processors, meaning that only a single version of a given code has to be maintained. This is obviously not the case on GPUs, as a code for these devices will either not run (CUDA) or need

to be carefully tuned (OpenCL) to run on a classic CPU. Indeed, to reach better performance, some of OpenCL specific concepts, such as *work-items* or *work-group* have to match hardware quantities, which are very different between CPUs and GPUs. Besides, MIC processors can be gathered together in a cluster using MPI, whereas a good speedup on several GPUs is also very hard to achieve, considering for instance the cost of data transfers.

All processors based on the MIC architecture will be part of Intel® Xeon Phi™ family. It contains in particular the *Knights Corner* (KNC), which is a major component of *Stampede* supercomputer [93], and the *Knights Landing* (KNL), scheduled to be released in late 2015. The KNL should feature 72 Atom™ cores with four threads per core (i.e. 288 total threads!) and 512 bits registers for vectorization.

5.2 Need for Bigger and More Complex Systems

The constant quest for bigger systems is neither a useless race to boost egos, nor a waste of computational resources. If some simulations exhibit scale invariant processes (see Section 3.3.2) and can settle for a few hundred million atoms, other situations involving for instance grain-boundaries [105] or pores implosion [108] would certainly benefit from winning one or two orders of magnitude in system size. If the mole ($\approx 6 \times 10^{23}$) is obviously not reachable, most material-related should be covered by trillion particles systems (10^{12} or 1000 billion). But who knows what will be found before we get to these scales ...

Beyond size considerations, another barrier in MD simulation is the time: some phenomena require simulations up to a microsecond, which represents about a billion iterations. If specific methods for advanced integration schemes are studied to accelerate the time-integration process (e.g. multiple time step algorithms [102]), another path is to speed-up the serial execution of a MD software, instead of relying only on parallel capabilities.

5.3 Objectives

Considering the constant-increasing need in computer capabilities to perform simulation of very large systems, the recent but major evolutions of supercomputer architectures and the current design of MD software raised in previous chapter, we decided to revise STAMP's conception from scratch into a new framework fitted for exascale supercomputing: ExaSTAMP.

The goal of this thesis is obviously not to rewrite a hundred thousand lines of code. Following the concept of mini-applications raised in Chapter 4, we will design and develop the base components of a Molecular Dynamics Simulation framework, targeting specifically manycore architectures. Once this part is advanced enough, beta-developers will implement their own modules.

Yet, an important difference with the mini-application concept is that ExaSTAMP is needed as a production tool by CEA physicists, to carry on specific and expensive simulations. If STAMP will keep being used considering its polyvalence, the new framework should enter production as soon as possible, at least for some specific situations.

A High-Performance, Scalable Framework

ExaSTAMP must be able to perform its simulations on a large number of cores (possibly more than 10^5), and also on a various range on architectures. It should be able to take advantage of shared-memory systems or the presence of accelerators. Indeed, half of TERA 1000, CEA's future supercomputer will be composed of KNL from Intel® Xeon Phi™ family.

To this end, we must operate on three different levels of parallelism: one for distributed memory to perform on large clusters, one for shared memory to take advantage of multicore/manycore cluster nodes or powerful desktop machines, and one for vectorization, to fully exploit performance of recent processors.

A MD Tool for Shock Physics in Dense Materials

With ExaSTAMP, we intend to simulate very large systems (up to several billion particles as a standard simulation) in the domain of shock physics and condensed matter. As dense material are very different from biochemical ones, we expect that the approach in the design of ExaSTAMP will vary from several MD codes.

A Friendly Code for Non-Expert Developers

The people that developed STAMP were not computer scientists, but physicists who were familiar with the art of coding. With some exceptions for specific and important modules, we expect future developers from ExaSTAMP to come from the same areas. The code architecture should be simple enough to be apprehended and modified by such non-expert developers. This will imply a high level of abstraction and a clear separation between physics module and parallelism tools, all without compromising performance.

6

Framework Architecture

THIS chapter details the design of ExaSTAMP and its essential components. Each object we have created has a mathematical, physical or computing meaning and a limited scope related to this meaning. This angle may confuse people used to conservative code architectures inherited from C and/or Fortran, where almost everything was gathered in a few structures, but it eventually allows much more flexibility and minimizes code duplication, which is exactly what we intend to do. Moreover, abstraction cost (e.g. calls a virtual method) in terms of performance has been seriously reduced thanks to progress made by compilers. When handled carefully, there are almost no differences between a C and a C++ code which both do the same thing.

An overview of concepts detailed in this chapter is represented on Figure 6.1; relations between objects will be outlined in *pseudo-UML*, when all implementation issues will be discussed in the next chapter. The first three sections will respectively picture particle objects and numerical schemes to integrate particles' trajectories, before focussing on some angles of the force computation process. In parallel, the force computation requires communications in order to have particles updated. In our approach, the global domain is overdecomposed with respect to the underlying cluster nodes. Section 6.4 is therefore dedicated to the *Node* concept and how it integrates a *Domain*. Then, the following sections respectively depict *Domain*, *Grid* and *Cell* objects. We finish by a quick description of our I/O system and the integration of specific optimized kernels for accelerated computations.

6.1 Particles

6.1.1 Particles

Particles are the building blocks of MD; they can be atoms, molecules, or even macro particles. In most cases, MD simulations involve only one *family* of particle at the same time, even if we can imagine mixing several of them (e.g. a protein in a solvent, which would be a huge flexible molecule with rigid ones).

Particle. A point particle can be represented in space phase with only its position ($\mathbf{r}$) and impulsion ($\mathbf{p} = m\mathbf{v}$, where $\mathbf{v}$ is the velocity). We also need to keep its force ($\mathbf{f}$) and potential energy (e_p) to compute thermodynamical quantities (kinetic energy can be computed from impulsion or velocity).

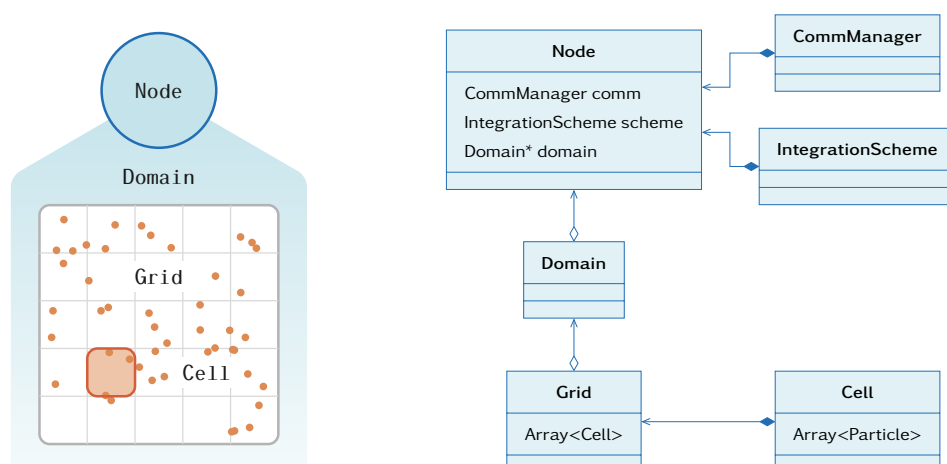


Figure 6.1 | ExaSTAMP architecture overview. A `Node` is the top structure of the code. It contains the integration scheme, a list of domains, and a communication manager. The node is also responsible for I/Os. The `CommManager` structure is an object-oriented object for communications. It allows a developer to create his own custom types and provides wrappers to use these types in communication. The `Domain` concept gathers an interface and its possible implementations. It proceeds to a reorganization and code factorization of these requirements for their implementation in lower-level classes. Every domain is itself decomposed in `Cells` in anticipation of the neighbor search. A `Grid` is basically an array of cells which contains all thread parallelization. `Particles` are eventually placed in cells.

In order to distinguish two particles, a global ID will be used. Any type of particle will inherit from class `Particle`, as shown on Figure 6.2.

Atom. Given our previous definition of particles, we do not need to add any attribute to describe an atom. Yet, we keep both classes in case someone would need to append a relevant functionality to it.

Molecule. Rigid molecules are a model with some internal degrees of freedom frozen. It is widely spread in fields like *Dissipative Particle Dynamics* (DPD) [61]. In this case, we will use the position inherited from a `Particle` to store the position of the molecule's center of mass, and we will add a quaternion to describe the molecule orientation against a reference configuration.

Other models, as flexible molecules, are actually all-atoms simulations with distinct potentials for inter- or intra- molecules interactions. We will not focus on this type of representation in our architecture; we rather left it intentionally as an example for a future developer who would like to add his own models.

6.1.2 Particle Types

Within a family of particles, we have different *types*: atoms can be carbon (C), iron (Fe) or bromine (Br); molecules can be water (H_2O), nitromethane (CH_3NO_2) or methanol (CH_3OH). Two carbon atoms have exactly the same mass¹; this means that the mass of a particle is an attribute of the particle type, not the particle itself.

Thus, we need to separate concepts of particle and particle type. With a single instance of each particle type, we will be prevented from storing needlessly a mass for each particle. All traits in a `TypeParticle` will be accessible directly from the `Particle` itself, leaving this process transparent for the developer.

¹Considering a single isotope.

TypeParticle. Two obvious common traits for all particles types are their name and their mass. To avoid costly divisions during computation phases, we will also enclose the inverse mass. This will define the base `TypeParticle`. If needed, all particle types can be extended to fulfill specific requirements.

TypeAtom. As atoms and ions only differ by their charge², we will use only one `ParticleType` to describe them. The `TypeAtom` class will inherit from the class `TypeParticle`, with the addition of a charge and an atomic number. When we need to considerate isotopes, several `TypeAtom` objects with different masses may be involved (e.g. ^{12}C , ^{13}C and ^{14}C).

TypeRigidMolecule. The `TypeRigidMolecule` class will inherit from class `TypeParticle`, and will store a complete description of it: number of atoms and their configuration in space, type of atoms, ...

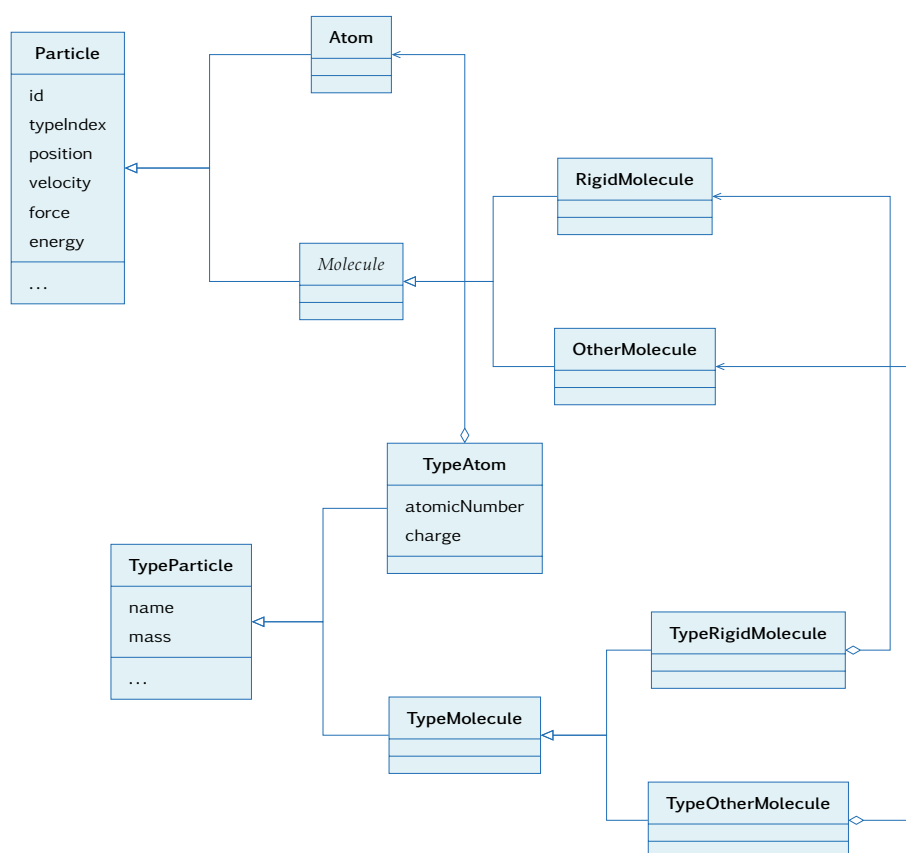


Figure 6.2 | UML representation of particle-related classes. Both trees have a similar structure; and we use an aggregation relation between a `Particle` and its `Type`.

6.2 Time Integration

MD allows the time integration of Newton's equations to retrieve a set of particle's trajectories. Section 2.1 presented some classic schemes to perform this integration. In the same way, the concept of `IntegrationScheme` provides us with a general framework to execute this integration.

²This means we neglected electron's mass, which is orders of magnitude smaller ($\approx 9.1 \times 10^{-31}$ kg) than proton's or neutron's one ($\approx 1.67 \times 10^{-27}$ kg).

6.2.1 Decomposition of Verlet Schemes

When analyzing Verlet schemes presented in equations (2.8) and (2.9), we can pull out three basic steps besides the force computation which will give us the acceleration:

- push positions (first order);
- push positions (second order);
- push velocities (first order).

For the Langevin scheme (2.10), we just have to add the fluctuation-dissipation step; therefore all three schemes can be described with a sequence of these functions, with half or a full time-step (see Figure 6.3). This decomposition will prevent us from a lot of code duplication, as several blocks are used in a lot of different schemes.

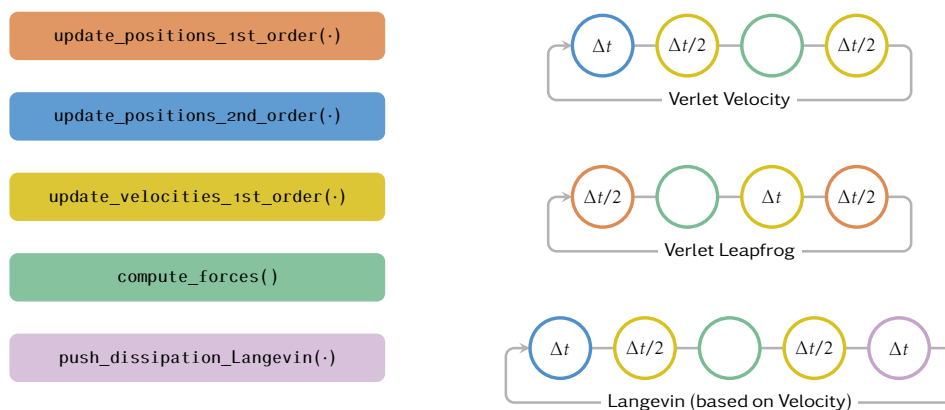


Figure 6.3 | Decomposition of Verlet Integration Schemes. All elementary blocks needed for the three schemes raised in Section 2.1.2 are printed on the left; these schemes are then detailed on the right, revealing the use of various similar blocks.

6.2.2 Classes for Integration Schemes

The `IntegrationScheme` class is an interface defined with a single (pure virtual) function called `oneStep()`, which will be implemented by its children, classes `VerletVelocity`, `VerletLeapFrog`, `Langevin` and whatever schemes that will be used in the future (see Figure 6.4). In our design, the `oneStep()` function does not “touch” particles directly, as it would need to know all possible implementations. Instead, it acts through an interface of the objects where particles actually live, using the elementary functions described in Figure 6.3.

6.3 Force Computation

In this section, we will focus on the force computation, which is by far the most expensive part of a simulation. We saw that the force expression can be derived from an energy function called *Potential*, and that various potentials have the same formalism and can be ordered in different classes. After having ranked potentials in a hierarchical set of classes, we will point out algorithms to compute the force using two families of potential, before centering on some technical angles of the force computation.

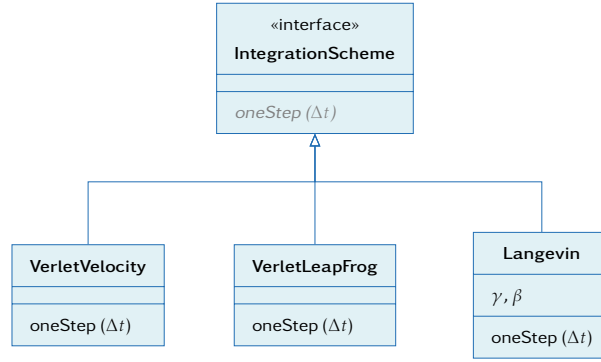


Figure 6.4 | Classes for time-integration schemes.

6.3.1 Potentials

During the design of ExaSTAMP, we focused on two classes of potential, pair and EAM potentials, as they already cover a large range of situations. We nonetheless kept in mind that other potentials should be easy to add into our architecture.

In our approach to draw the class diagram for potentials (on Figure 6.5), we reproduced the classification sketched in Chapter 2: from a common interface, we draw the `ShortRangePotential` family, which common attribute is to have a cutoff radius. Even if they do not take part in this work, there is still the possibility to draw a `LongRangePotential` object from the main interface.

From short-range potential, we have pair and EAM potentials. Both gather a large set of different potentials under a unified formalism. Pair potentials require only one function (the potential energy function), whereas EAM potentials have three; these functions are obviously pure virtual as it makes no sense to have an instance of these potentials. For each function we added the corresponding cutoff value in order to smooth energy variations (see Section 2.4.1).

On Figure 6.5, only one example for pair and EAM have been represented, but we also have an Exp-6 potential and another EAM potential (see Chapter 10). The ideal gas potential is a special case, as it pictures the absence of interaction. We connected it directly to the interface, since there was no better choice.

6.3.2 Force Computation Patterns for Potentials

Pair Potential

Let us consider a particle denoted by the index i , and let us suppose that we already have a list of particles in i 's neighborhood, $\mathcal{N}(i)$. Using notations from Chapter 2, the potential energy of particle i is given by

$$E_p(i) = \frac{1}{2} \sum_{j \in \mathcal{N}(i)} V(r_{ij}), \text{ with } r_{ij} = \|\mathbf{r}_j - \mathbf{r}_i\|, \quad (6.1)$$

and the force by

$$\mathbf{f}(i) = \sum_{j \in \mathcal{N}(i)} -\nabla_{\mathbf{r}_i} V(r_{ij}) = \sum_{j \in \mathcal{N}(i)} -\left(\frac{\partial V}{\partial r}\right)_{(r=r_{ij})} \frac{\mathbf{r}_i}{r_{ij}}. \quad (6.2)$$

The important thing to notice here, is that we only need to go through all pairs of particles *once* to compute all forces, as shown in Algorithm 6.1.

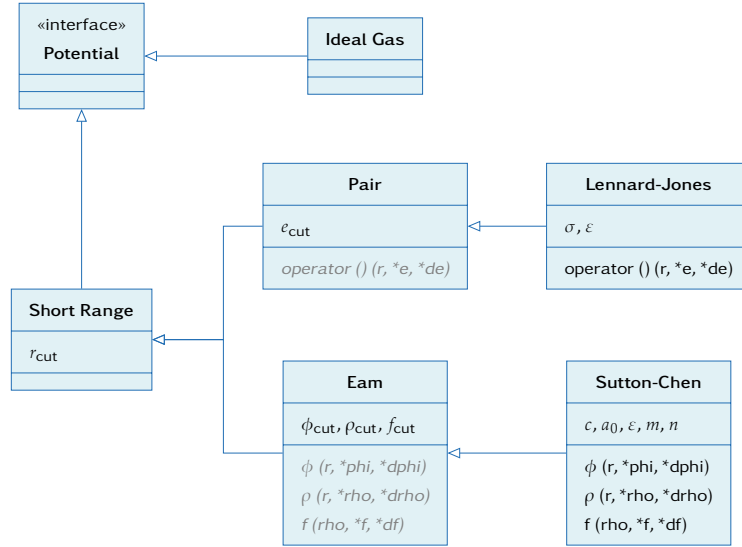


Figure 6.5 | Potential classes. When more than one Type of particle is present, there are more than one potential, or at least the same potential with different parameters. Hence, we may have to deal with several instances of the same potential.

Algorithm 6.1 Force computation for a pair potential

Require: $\forall i, E_p[i] \leftarrow 0$ and $\mathbf{f}[i] \leftarrow 0$

for $i = 1..N$ do

for all $j \in \mathcal{N}(i)$ do

$$E_p[i] \leftarrow E_p[i] + \frac{1}{2} V(r_{ij})$$

$$\mathbf{f}[i] \leftarrow \mathbf{f}[i] - \left(\frac{\partial V}{\partial r} \right)_{(r=r_{ij})} \frac{\mathbf{r}_i}{r_{ij}}$$

end for

end for

EAM Potential

The EAM case is more complicated, as neighbors of particle i 's neighbors are involved. With the same notations as last paragraph, we have:

$$E_p(i) = \frac{1}{2} \left(\sum_{j \in \mathcal{N}(i)} \phi(r_{ij}) \right) + F \left[\sum_{j \in \mathcal{N}(i)} \left(\sum_{k \in \mathcal{N}(j)} \rho(r_{jk}) \right) \right], \quad (6.3)$$

and

$$\mathbf{f}(i) = \sum_{j \in \mathcal{N}(i)} - \left\{ \left(\frac{\partial \phi}{\partial r} \right)_{(r=r_{ij})} + \left[F \left(\sum_{j \in \mathcal{N}(i)} \rho(r_{ij}) \right) + F \left(\sum_{k \in \mathcal{N}(j)} \rho(r_{jk}) \right) \right] \left(\frac{\partial \rho}{\partial r} \right)_{(r=r_{ij})} \right\} \frac{\mathbf{r}_i}{r_{ij}}. \quad (6.4)$$

Once again, the important point is that we need to go *twice* through all pairs of particles to get the forces for the whole system. Algorithm 6.2 highlights these loops well.

6.3.3 Neighbor Search

For the neighbor search, we preferred a Linked-Cell like method over a Verlet one. If the second appears more efficient at first glance, it tends to lose this efficiency on large dense systems [10], which is exactly what ExaSTAMP is aiming for. We therefore have a virtual grid/mesh on domains,

Algorithm 6.2 Force computation for an EAM potential

Require: $\forall i, E_p[i] \leftarrow 0, \mathbf{f}[i] \leftarrow 0, \bar{\rho}[i] \leftarrow 0$ and $\bar{F}[i] \leftarrow 0$

```

for  $i = 1..N$  do
  for all  $j \in \mathcal{N}(i)$  do
     $\bar{\rho}[i] \leftarrow \bar{\rho}[i] + \rho(r_{ij})$ 
  end for
end for

for  $i = 1..N$  do
   $\bar{F}[i] \leftarrow F(\bar{\rho}[i])$ 
   $E_p[i] \leftarrow E_p[i] + \bar{F}[i]$ 
end for

for  $i = 1..N$  do
  for all  $j \in \mathcal{N}(i)$  do
     $E_p[i] \leftarrow E_p[i] - \frac{1}{2} \phi(r_{ij})$ 
     $\mathbf{f}[i] \leftarrow \mathbf{f}[i] - \left\{ \left( \frac{\partial \phi}{\partial r} \right)_{(r=r_{ij})} + [\bar{F}[i] + \bar{F}[j]] \left( \frac{\partial \rho}{\partial r} \right)_{(r=r_{ij})} \right\} \frac{\mathbf{r}_i}{r_{ij}}$ 
  end for
end for

```

with cells of size $c \geq r_{\text{cut}}$. The algorithm to build a neighbor list for each particle is very straightforward, as pictured in Algorithm 6.3. Some implementation tricks to speed up the process will be presented in the next chapter (see Section 7.2.1).

Algorithm 6.3 Neighbor lists construction

Require: $\forall i, \mathcal{N}(i) = \{\}$

```

for all cells  $c$  in domain do
  for all particles  $i$  in  $c$  do
    for all cells  $n$  neighbors of  $c$  do
      for all particles  $j$  in  $n$  do
        if  $r_{ij} < r_{\text{cut}}$  then
           $\mathcal{N}(i) \leftarrow \{\mathcal{N}(i), j\}$ 
        end if
      end for
    end for
  end for
end for

```

For pair potentials, it is even possible not to build the neighbor list and get a particle's neighbors on the fly during the force computation: in Algorithm 6.3, line " $\mathcal{N}(i) \leftarrow \{\mathcal{N}(i), j\}$ " is replaced by lines " $E_p[i] \leftarrow \dots$ " and " $\mathbf{f}[i] \leftarrow \dots$ " from Algorithm 6.1. This is also applicable for EAM potentials, but with a loss in performance as the test $r_{ij} < r_{\text{cut}}$ would be repeated twice. In ExaSTAMP, we decided to keep the neighbor list in order to have the same neighbor search process for all potentials.

6.3.4 Keeping Edges Up to Date

To compute the force acting on a particle, we need the positions of all neighbors of this particle within the cutoff radius for a pair potential, and the neighbors and the neighbors' neighbor for an EAM potential. All particles' positions must be up-to-date before starting the force computation.

On a parallel simulation with domain decomposition, particles moving between domains must be relocated. Besides, the case of particles on a domain's edge must be considered carefully to include neighbors from other domains as on Figure 6.6: once moving particles have been successfully relocated, each domain send their edges to their neighbors to update a “buffer zone”, used to correctly access a particle's neighbors. Often called *ghost* layer, its depth depends on the potential(s) used in the simulation: one cutoff radius for pair potentials, and two for EAM potentials, as we need the positions of neighbors' neighbors. In our case, the cell structure used to build neighbor lists is helpful as its size is close to and slightly larger than the cutoff radius; “ghost” particles are therefore particles in the “edge cells” of the domain.

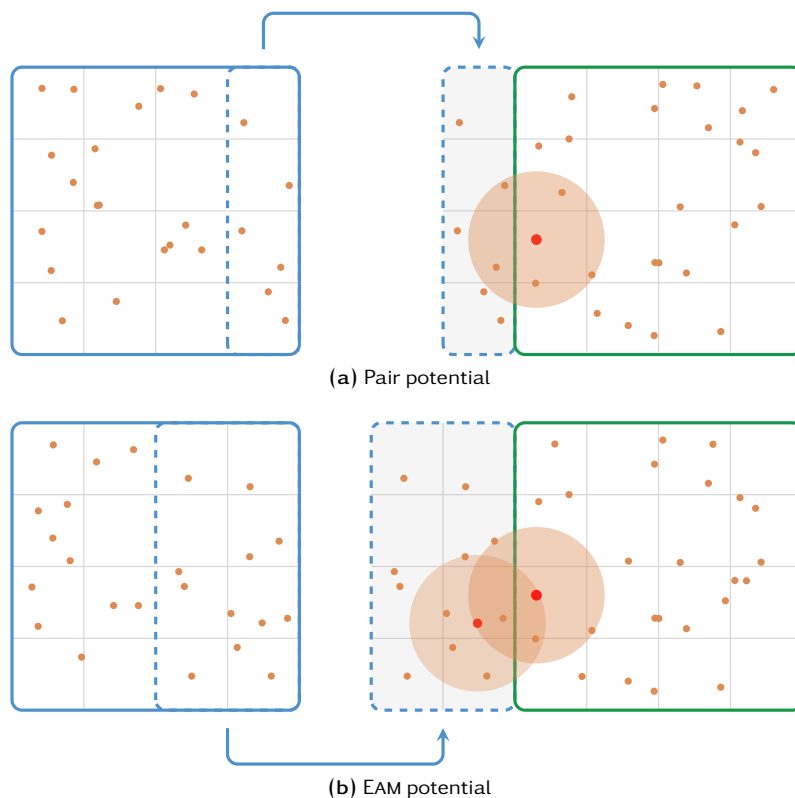


Figure 6.6 | Ghost layers for pair and EAM potentials. In order to properly compute interaction on a domain's edges, a part of the neighboring domain must be copied. The copied part is called the ghost layer; its depth depends on the potential: for pair potentials, all particles which distance to a domain's edge is lower than the cutoff radius may be needed by a particle on the neighbor domain. Concerning EAM potentials, we need two cutoff radii, since neighbors' neighbor are involved.

6.3.5 Overlapping Communications

Until the 2000s and the massive use of Infiniband® networks, communications were in most cases the main bottleneck for performance. Overlapping communication with computation is a classic method to overcome this bottleneck: instead of waiting for some communications to finish, processors can leverage this time to perform some computations.

In MD simulations, this process can take place during the ghost layer update. Indeed, ghost layers are only needed to compute the force acting on edge particles, meaning that inside particles are already ready for the force computation. The classic process

1. exchange particles;

2. wait for (1);
3. update ghost layer;
4. wait for (3);
5. compute force on *all* particles;

eventually becomes

1. exchange particles;
2. wait for (1);
3. update ghost layer;
4. compute force on *inside* particles;
5. wait for (3);
6. compute force on *edge* particles.

An important consideration is that the size of the edge layer is determined by the potential, in the same way as the ghost layer (respectively one and two cutoff radius for pair and EAM potentials). On recent networks, this gain of performance is not as important as what it used to be, especially when considering the cost of modifications. As we started ExaSTAMP from scratch, we decided to include both possibilities, using a *strategy* pattern [50].

6.3.6 Symmetrization

Another scope to reduce simulation times is to take advantage from Newton's third law: if $\mathbf{f}_{ij}$ denotes the force on a particle i , resulting from the interaction of particles i and j , the force acting on particle j resulting from this interaction is $-\mathbf{f}_{ij}$. In other words, we can divide the number of calls to potential energy functions by a factor two if we update the potential energy and the force of both particles i and j . To this end, we will *symmetrize* neighbor lists: for two neighbor particles i and j , we had $i \in \mathcal{N}(j)$ and $j \in \mathcal{N}(i)$; we now have $i \in \mathcal{N}(j)$ if $i < j$ so that we do not write twice in the same particle's attributes.

With EAM potentials, the symmetrization process allows the use of a single-depth ghost layer at the price of extra communication during the force computation. Besides the gain in computation time, a thinner ghost layer consumes far less memory. Performance of computations with and without symmetrization enabled are shown in the next chapter (see Figure 7.4).

6.4 Parallel

The first level of parallelism consists in splitting the work on a distributed memory system. As explained previously in Section 2.5, it will involve domain decomposition, which is the only rational choice³; domains will interact through the network using the MPI library [80].

³We recall that we only consider short-range potentials here; for long-range potentials (e.g. for electrostatic interactions), other parallelization types may be considered.

6.4.1 Node and Domain

Most MD software have been created before the multicore era and do not take advantage from capabilities offered by shared memory. This means that each processing unit (CPU or core) is usually affected to a domain, with important resources waste like extra communications and memory usage. Indeed, having more domains means more exchange between them, and more space to store ghost data needed for computation on edges. Furthermore, with those considerations, the notion of domain is inherently bonded to the concept of an MPI process.

In ExaSTAMP, we propose to explicitly separate these notions. On one side, we have the **Node** concept: during the execution, there will be one **Node** for every MPI process, which has been designed to sit at the top of a shared memory structure, i.e. a node in a cluster environment, hence its name. On the other side, we have a **Domain**, which actually corresponds to a fraction of the global simulation box. In a **Domain**, the workload is shared among threads (see Section 6.6 for more details). We can have one or several **Domains** attached to a **Node**. In this last case, inter-Node communications are replaced by memory copies to release the communication library. Note that we make no assumptions about a domain's topology: it neither needs to have a cuboid shape nor has to be connected. This consideration will be critical in Chapter 8.

This decomposition allows us to perfectly match hierarchical memory systems: let us consider for instance, a cluster where each node is made of eight-cores processors arranged in two NUMA nodes each. If we execute our program with one **Node** per cluster node and two **Domains** per **Node** (i.e. one **Domain** per NUMA node), we (1) relieve the network with less messages, (2) benefit from the parallelization over cores to speed-up work within a **Domain**, all (3) without wasting time in distant accesses, as **Domains** on a **Node** work (almost) independently.

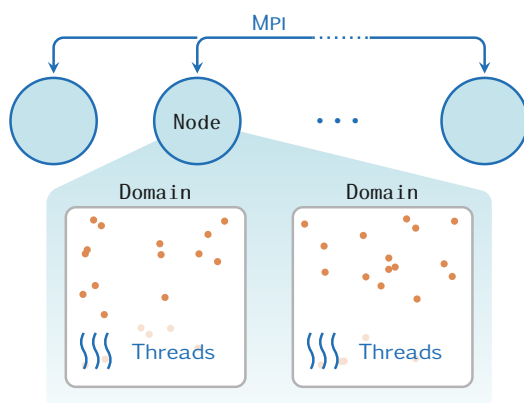


Figure 6.7 | **Node and Domain.** Example with two **Domains** on one **Node**

6.4.2 Communication Manager

Within a **Node**, the communication manager is the object in charge of communications. Communications can be separated in two types: point-to-point (or domain-to-domain) communications and collective communications. Relations between **Node**, **CommManager** and other objects for communications are outlined in Figure 6.8.

In both cases, our mainstream developer manages **Domains** as he would manage MPI processes in “classic” codes. The fact that a **Node** may contain several **Domains** is fully taken over (if necessary) by the **CommManager**.

Collective Communications

Collective communications have been wrapped in the communication manager, which provides a simpler interface than MPI: with a *template* signature and a specific class for arrays, the developer does not have to care about the size and the type of messages. Methods for collective communications are accessible from a `Domain`, and the fact that there can be several `Domains` per `Node` remains transparent from the developer's point of view.

Point-to-point Communications

In MD, all point-to-point communications are part of a more global exchange process (e.g. domains exchange moving particles or the ghost cell update mechanism, both outlined in Section 6.3.4), where domains send and receive messages only with their neighbors. To simplify these processes for the mainstream developer and to avoid unnecessary code duplication (as they do not depend on the type of particle used, unlike MPI calls), all point-to-point communications are embedded in specific structures.

From a `Domain`, the developer has access to several `Message Centers`: each `MessageCenter` manages all point-to-point communications corresponding to a specific process (e.g. particle exchange or ghost update). From a `MessageCenter`, he pulls a `MessageSend`, which is the structure that he fills with data to send, using the `push()` method. There is no need to use different messages for a `Domain` at this point, even if there are various recipients: all sorting and dispatching operations will be performed in the background.

Once the message is ready to send, our developer just has to use the `MessageCenter`'s `send()` method. This will trigger, through the `Session` object, the sorting and processing of messages from all domains, before sending them (or copying them if there are intra-Node communications) to the right recipient.

To ensure the end of communications and process the received messages, the developer will use the `collect()` method from the `MessageCenter`. The call of this method guarantees that the `MessageRecv` object has been properly built. He can now extract data from the message and process it as he needs. The last thing to do once data have been extracted is to call the `clean()` method, in order to free memory used for messages and data used in the background.

6.5 Domain

The diagram on Figure 6.8 has been a bit simplified for the `Domain` object. The `Node` does not actually handle the `Domain` directly, but through an interface. This interface gathers all services required by `Node` and `IntegrationScheme` objects. This interface has only one implementation for now. We chose this design to allow the connection of other possible implementations as the project grows.

An overview of the interface and implementation for domain objects is available on Figure 6.9. The main role for our `Domain` objects is to specify complex services into more elementary functions: we have already seen in Section 6.3.5 that the force computation routine could have two versions, depending on whether we want to enable communication overlap. The `updateCells()` function, which is used to synchronize particles, is composed by three distinct actions:

- `exchangeParticles();`

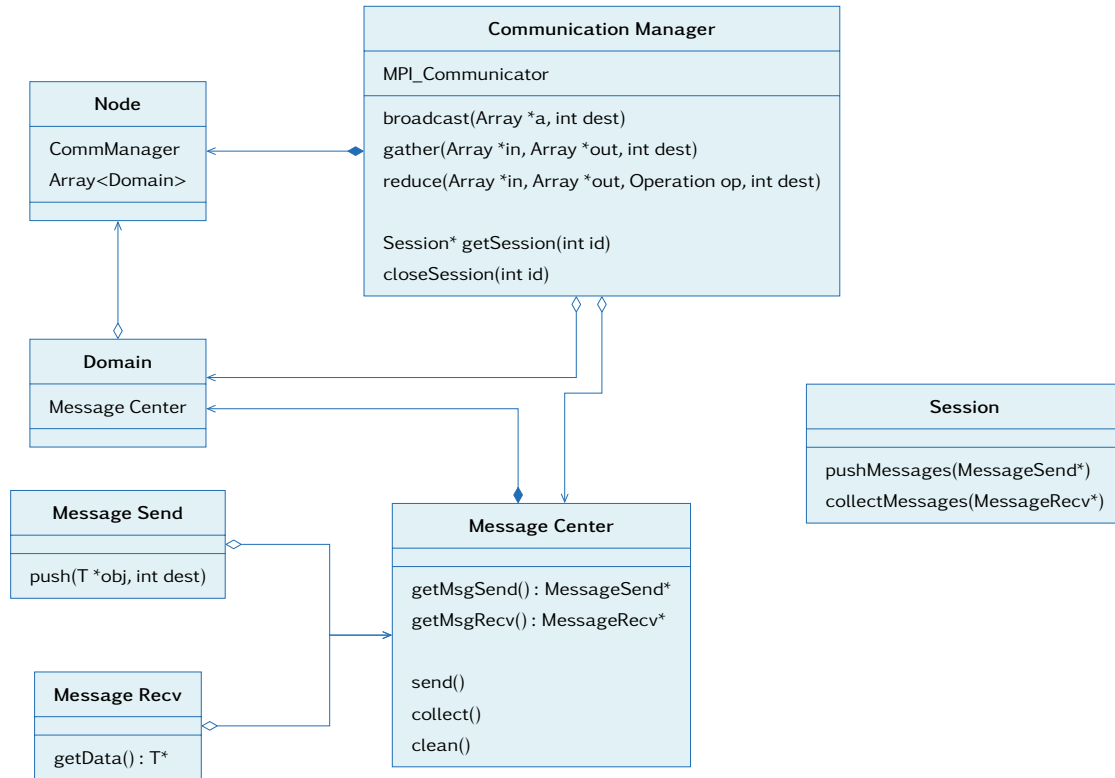


Figure 6.8 | Communication Manager.

- `updateGhost()`;
- `internalReorganization()`.

If the first two have been explained before, the third is due to the cell structure that we adopted to compute neighbor lists: within a domain, particles can move between cells and must be reassigned to the right ones in order to get the right neighbors. The `collectParticles()` and `collectGhost()` functions are here to ensure the end of communications, respectively for the particle exchange and the ghost update process.

6.6 Grid

If the `Grid` reflects the cell structure in which we placed particles for our neighbor search, its role is more global than providing an environment to build neighbor lists. The `Grid` transposes a “global” instruction to the cells (or particles) where it will be executed. For instance, if the `pushPositions` methods are applied to all particles, the function `exchangeParticles` is only about particles on the domain’s edges. Besides, the `Grid` is the root for thread parallelization: all loops on cells or particles are supposed to be executed concurrently. The `Grid`’s ability to build neighbor lists for particles is embedded in the decomposition of force functions: the global compute force process from `Domain` class can be arranged in a specific order :

- reset force and energy;
- build neighbor list;

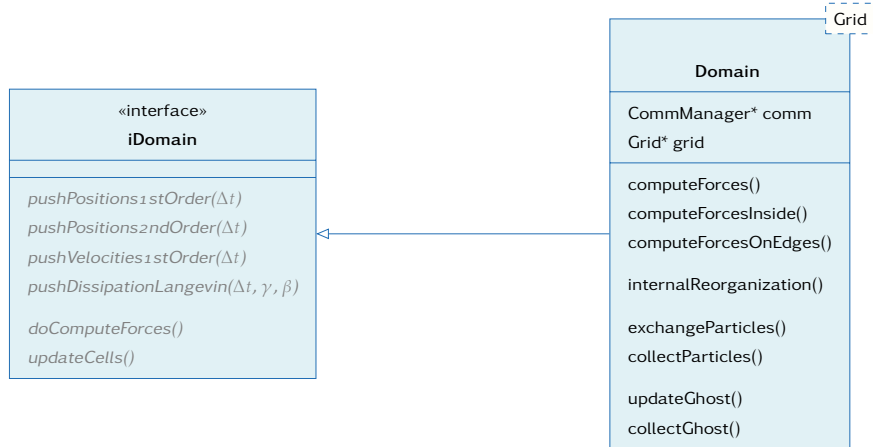


Figure 6.9 | Interface and implementation for a domain object. Some basic access functions like the total number of particles or kinetic/potential energy reduction have been omitted for readability reasons. In the same way, `Domain` object also implements all pure virtual functions from the interface.

- compute force;
- delete neighbor lists;

Thus, our `Grid` object ends up with three more methods that can apply on all, inside or edge particles.

In practice, grid cells size is determined by

$$c = \frac{L}{\lfloor L/r_{\text{cut}} \rfloor} \quad (6.5)$$

in each dimension, L being the domain's extension and r_{cut} the cutoff radius of the potential used for the simulation. If more than one material is involved, the maximum cutoff radius over all potentials is chosen.

The `SingleSpecGrid` class is a specialization of the `Grid` for simulations involving a single species of particles, e.g. only atoms or only rigid molecules simulations. The case of multi-material simulation, e.g. simulation with several atoms types are obviously considered by `SingleSpecGrid`.

6.6.1 Cell Traversal

Many `Grid` methods are acting on the same collection of cells. In order to avoid a systematic re-computation of these collections, we defined the `Traversal` object. A `Traversal` may be simply considered as set of cells that we run through to perform any given operation.

The `Traversal` improves code readability and provides a more efficient execution. It is a valuable tool for the developer as it silently takes into account parameters such as the grid's topology or the ghost thickness. For instance, there are no differences between going through the edges of a cuboid domain and an potato-shaped one. Besides, if new methods requiring original traversal were to be implemented, `ExaSTAMP` provides several tools to create them.

Table 6.1 shows `Grid`'s main methods, alongside the `Traversal` on which it gets executed, and a brief description. An uni-dimensional graphic representation of traversals is also available at Table 6.2.

Method	Affected Cells	Description
pushPositions1stOrder	real	to perform time-integration-related operations on particles
pushPositions2ndOrder		
pushVelocities1stOrder		
pushDissipationLangevin		
resetForceAndEnergy	all	to set potential energy and force to zero before the force computation
makeNeighborLists	all*	to build neighbor lists in concerned cells
makeNeighborListsInside	δ -inside	
makeNeighborListsOnEdges	δ -edge*	
clearNeighborList	all*	to delete neighbor lists
computeForce	real	to compute force in concerned cells
computeForceInside	δ -inside	
computeForceOnEdges	δ -edge	
exchangeParticles	1-edge	to locate particles that are leaving the domain
collectParticles	1-edge	to place arriving particles in the right cells
internalReorganization	1-inside	to move particles between cells
updateGhost	δ -edge	to send edges cells to neighboring domains
collectGhost	ghost	to get neighbors' edge in ghost cells

Table 6.1 | Grid Methods. δ stands for the ghost layer thickness; as said before, it is equal to one when pair potential are in effect, and two for EAM potentials. For each method we have a quick description of it, and the cells through which we go to perform these methods. Table 6.2 gives a visual representation of what these keywords mean.

Traversal	$\delta = 1$	$\delta = 2$
all		
real		
ghost		
all*		
δ -inside		
δ -edge		
δ -edge*		
1-inside		
1-edge		

Table 6.2 | 1D representation of Grid Traversals. In the case $\delta = 1$, some traversals are identical and should not be mixed.

6.6.2 Particle Storage

Grid integration in our framework was highly affected by the design of an efficient particle storage solution. We first used a design with a particle list which has later been given up because of performance issues at the thread level. A successful idea was to consider cells as independent *objects*. Both solutions are presented in Figure 6.10, and implementation details will be given in the next chapter.

Particle List

The Grid concept has no physical meaning: it is just an implementation trick used in order to make the neighbor search easier. This is why we first thought of separating cells from particles. The Domain had two main attributes, a ParticleList and a Grid, which covered the following needs:

- given a cell, provide (from the list) particles that belong to that cell;
- given a cell, provide the neighboring cells for neighbor list making;
- given an edge cell, provide the neighboring domains for communications;
- quickly distinguish real cells from ghost ones.

The first idea behind the ParticleList was to store particles in a single array (within a Domain), which allowed an efficient parallelization of compute-intensive functions and minimized the number of allocation/deallocation operations. Yet, hybrid performance happened to be somewhat disappointing, with speedups not as good as expected above a hundred cores. We found that it was due to a sequential region which became too important following Amdahl's law [5]. A more detailed explanation with a performance graph is available in the next chapter (Section 7.1.3).

Cell Array

Our solution to this issue was to fully take advantage of the cell structure. Instead of considering it as virtual, we explicitly included it in our design: the ParticleList concept is removed and replaced by an array of Cell objects. This collection of cells is integrated to our architecture as a Grid attribute, meaning that a Domain cannot access particles without going through the Grid. Each Cell is able to manage its own particles independently, which is really comfortable for thread parallelization.

6.7 Cells

A Cell is the smallest space subdivision in our architecture. It is the structure that effectively holds particles and performs operations on them. The Cell can be viewed as a list, with `push()` and `pop()` methods to manage incoming or leaving particles; however we will see in Chapter 7 that the implementation is quite different from a list, as we need a structure suitable for vectorization. The Cell also holds the neighbor lists for each of its particles.

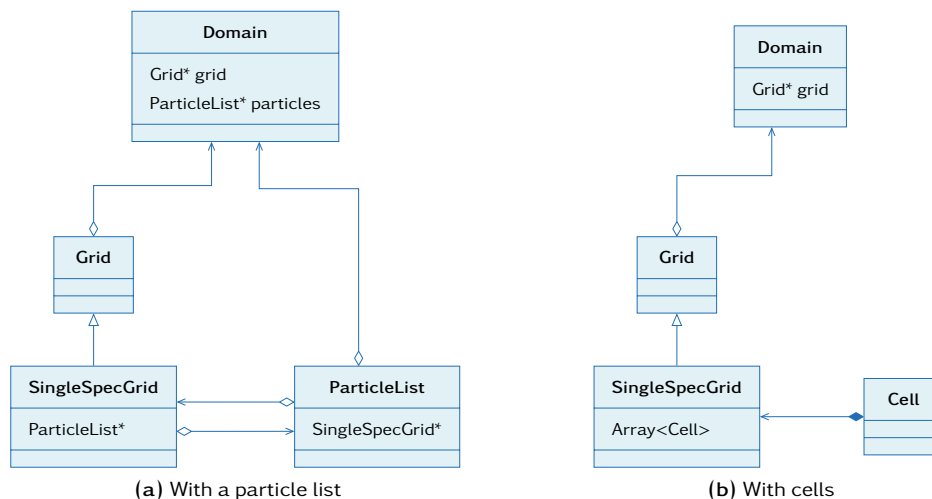


Figure 6.10 | Evolution of Grid design.

6.8 I/Os

If the constant growth of computing capabilities allows the simulation of ever larger systems, data produced by those simulations follow the same path. During production simulations, storage and file systems on clusters are constantly stressed out, and too few people are willing to face those issues. In ExaSTAMP, we have two separate ways to produce outputs: a “legacy” checkpoint-restart manager which uses the same format as STAMP, and a complete (checkpoint-restart and post-processing files), high-performance system manager provided by an external library.

6.8.1 Quick Diagnostics for Debug

First outputs have been used as a debugging tool in the early hours of ExaSTAMP’s development. In order to check the code, we need to perform visualization and statistical analysis on particles, as histograms of positions and velocities. Generated outputs are meant to support parallel execution (MPI and threads) and be human-readable (i.e. it will produce text files instead of binaries).

To visualize particles, we selected Paraview [58] with VTK file format, when we decided to have our own custom text formatting for our statistical analysis. The strategy to write our debugging outputs in parallel is to gather all data on one Node and let this Node write the whole file. As both formats use the same data, we have a common environment to gather it and specialized classes to write in the desired format.

Though they proved themselves very useful, our debugging tools have important limitations: as we gather everything on a node, we cannot use them on large systems without exceeding memory; moreover, the fact that only one MPI process is able to write induces sequential regions which are terrible for scaling; at last, the human-readable requirement prevents from using binary format which is much faster to write and read. As a result, these tools are reserved to developers and may not be available in production releases.

6.8.2 A First Checkpoint-Restart System

Application check-pointing is a production tool which consists in periodically saving on disk the state of a running program, in order to later resume (restart) the simulation. Checkpoint-restart (CR) is critical in HPC environments since jobs can run for times up to several weeks; if a job stops due to an internal error, node crashing or power failure, it will resume from the last successful checkpoint instead of starting from the beginning.

Many reasons led us to develop a “legacy” CR system (i.e. a CR that uses the same format as in STAMP) in addition to a standard one. It gives us an alternative to check the validity of our framework and the possibility to continue with ExaSTAMP simulations initiated by STAMP. Moreover, compatible CR files enable us to take advantage of all post-processing features that have been integrated in STAMP.

The legacy CR files are written with MPI-I/O [80]: each MPI process writes its own part of a common file, using basically the same method than our diagnostic tools (up to the `gather()` step). In this context, our Node structure brings multiple advantages: write operations are parallel, and we use far less cores than full-MPI applications, which relieves both network and file system.

6.8.3 Advanced CR & Outputs

Though our legacy CR system provides an efficient short-term solution to analyze and process simulation data, we do not intend to rely on STAMP to complete these tasks in the future, especially when billion-particle simulations will be involved. At CEA, dedicated tools have been internally designed to provide generic solutions to manage and analyze huge amounts of data on our supercomputers.

HERCULE and LOVE

The HERCULE project [120] aims to improve handling of voluminous data produced by numerical simulation codes on massively parallel computers. It addresses three different kinds of usage, which are grouped within a single library: inter-codes exchanges, post-processing actions, and checkpoint/restart.

LOVE (Large Object Visualization Environment) is a visualization system which has been designed to operate on large datasets issued from simulation on massively parallel computers. LOVE is built on the top of Paraview and VTK, and is fully compatible with HERCULE.

I/O Manager

In the same way that the communication manager was in charge of communications, we now have an object to perform all input/output operations, the `IoManager`. The `IoManager` (see Figure 6.11) has its own buffers to gather data before a write operation; it delegates to `Domains` the write operation of raw data into these buffers so that we have the same methods for both outputs and checkpoint files. Once buffers are filled, dedicated methods are enabled to write data in the right format.

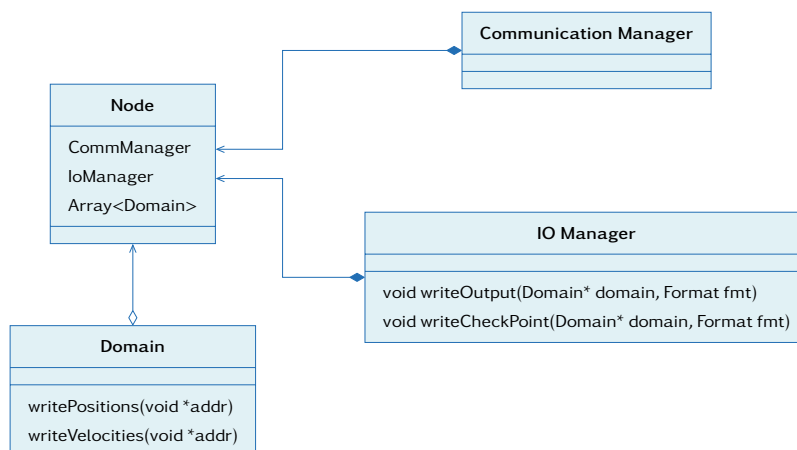


Figure 6.11 | Input-Output Manager.

6.9 Specific Kernels for GPU Accelerated Simulations

In the previous chapter, we saw that accelerating computing with GPUs/GPGPUs could result in a significant increase of performance, though at important development costs. If a physicist developer is hardly likely to handle highly-optimized CUDA or OpenCL code, there is no reason not to include such opportunity of performance.

The ultimate goal for the future is to benefit from ultra-optimized kernels for our MD potentials in the same way as BLAS [16, 35, 36] or LAPACK [7] libraries provide linear algebra functions. Although these are long-term considerations, we cannot afford to ignore them in an architecture that intend to reach the Exascale barrier; we however do not expect this exploratory work to enter production at the same time as our “standard” branch.

Therefore we selected a few simulation cases to be adapted on GPUs. Developments have then been carried out by GPU specialists through a collaboration at INRIA Bordeaux. Along with performance, efforts were focused on minimizing data duplication between host and device. In ExaSTAMP, we derivated a new `Grid` object specific to simulations on a device (see Figure 10.2). We chose this level in the architecture as we want a different storage solution than a standard `Grid`, since the computations will be performed on the device. Moreover, it allows us to benefit from some high-level capabilities of the framework, especially MPI communications.

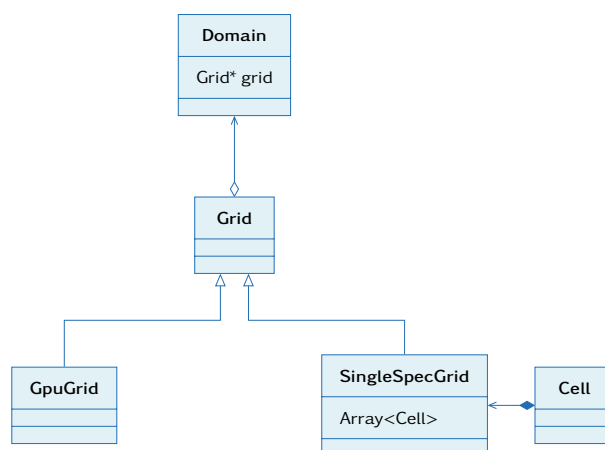


Figure 6.12 | Specific Grid for GPU accelerated simulations.

7

Implementation and Preliminary Results

IN this chapter, we discuss the implementation of ExaSTAMP. We chose the C++ language [112, 111] to achieve the implementation, as it provides both high and low level features, and is also widely distributed. Furthermore, we decided to use the most recent C++ capabilities, with the inclusion of several features from the 2011 standard.

This chapter is organized as follows. We start by following the architecture from high to low levels, with the achievement of `Domain`, `Grid` and `Cell` objects. We spend there some time on thread parallelization and the search of an efficient solution to store particles. Details about force computation are separated in their own section, before focusing on a tool to enable explicit vectorization. We end this chapter with a last section about the implementation of communication patterns.

7.1 Domain, Grid and Particles

7.1.1 Domain

As specified in the previous chapter, the `Domain` concept is an interface from where we can draw different implementations. For the time being, we only have one, but we can expect others in order to explore options we have not provided, such as Coulomb interactions. We note that having an interface at this level in the architecture does not raise any performance issues.

Our implementation of the `Domain` interface is a template class with the `Grid` class as an argument. Besides holding the strategy to compute force with or without communication overlap, and divide the `updateCells()` method in three separate services, the `Domain` class mostly delivers its methods to the `Grid`.

7.1.2 Grid

Avoiding Overhead Induced by Multiple Virtual Calls: the Curiously Recurring Template Pattern

In theory, the `Grid` is a base class containing several pure virtual functions, meaning that it cannot be instantiated itself. The `SingleSpecGrid` inherits from the `Grid` for simulations involving only

```

template <class OP> class CRTP {
public:
    uint64_t value() { return m_value; }
    void method(const uint64_t& n) { return op().method(n); }
5
protected:
    uint64_t m_value;

private:
10    OP& op() { return reinterpret_cast<OP*>(*this); }
};

class cAdd : public CRTP<cAdd> {
15 public:
    void method(const uint64_t& n) { m_value +=n; }
};

class cMul : public CRTP<cMul> {
20 public:
    void method(const uint64_t& n) { m_value *= n; }
};

25 template <class OP> void run_crtp(CRTP<OP>* obj) {
    for (uint i=0; i<N; ++i) {
        for (uint j=0; j<i; ++j)
            obj->method(j+1);
    }
30 }

```

Listing 7.1 | An example of the Curiously Recurring Template Pattern

a single species of particles. At this level, the virtualization cost is important and cannot be neglected, considering the number of times that some methods are called.

To avoid this issue, we examined the Curiously Recurring Template Pattern (CRTP) [26], which is a form of static polymorphism in which a class derives from a class template using itself as an argument. For instance, let us consider a base class with an integer as an attribute; this base class has two derived classes that can perform operations (add or multiply) on the base's attribute. This simple example is implemented in Listing 7.1 and the “classic” version of it in Listing 7.2.

Both listings also have a `run_XXX()` method that we use to compare CRTP's performance with dynamic polymorphism's. We ran these pieces of code on three different machines¹, using different compiler options and problem sizes (N). In all cases, the CRTP is much faster. Results in Table 7.1 show that the compiler does not have a significant impact on performance of both versions. Regarding Figure 7.1, the speedup provided by the CRTP increases with the problem size to reach a limit, which probably means that runs with low values of N are perturbed by side effects. The limit value for CRTP acceleration is around 3.5 for westmere, and 7.0 for both sandybridge and haswell. This factor of two can be explained by vectorization, since westmere supports only SSE instructions when sandybridge and haswell have AVX. The use of gcc's vectorization report indicates that loop at line 27 of Listing 7.1 is the only one which takes advantage of auto-vectorization, which confirms our words.

In comparison to dynamic polymorphism, the CRTP provides much more performance and enables vectorization (with a speedup between 3.5 and 7), without being harder to implement. If we remove the acceleration due to vectorization (2 for SSE and 4 for AVX since our tests used 64

¹Names and specifications of processors used are in Appendix A.

```

class Interface {
public:
    virtual ~Interface() {}
    uint64_t value() { return m_value; }
    virtual void method(const uint64_t& n) = 0;

protected:
    uint64_t m_value;
};

class iAdd : public Interface {
public:
    virtual void method(const uint64_t& n) { m_value +=n; }
};

class iMul : public Interface {
public:
    virtual void method(const uint64_t& n) { m_value *=n; }
};

void run_interface(Interface* obj) {
    for (uint i=0; i<N; ++i) {
        for (uint j=0; j<i; ++j)
            obj->method(j+1);
    }
}

```

Listing 7.2 | Inheritance version of Listing 7.1

bit data), the CRTP remains $1.75 \times$ faster. We therefore decided to take advantage of this idiom to replace dynamic polymorphism in our low-level classes.

Compiler	Version	Dyn. Polym.	CRTP
GNU (g++)	4.4	0.944	0.315
	4.8	1.100	0.315
	5.1	0.943	0.315
Intel®	15	1.100	0.315

Table 7.1 | CRTP vs. Dynamic Polymorphism (I). Average runtime (on 100 samples) in seconds for different compilers, with O2 optimizations enabled. The use of O3 optimizations did not improve the results. N value was set to 10,000. Runs were performed on a westmere machine.

Thread Parallelization

We chose Intel®’s *Thread Building Blocks* (TBB) library [63, 24] for our thread parallelization. It provides both algorithms and concurrent data structures which allow a developer not to manually manage threads. In TBB, parallel operations are treated as “tasks”, which are executed according to a dependency graph. TBB uses work stealing to balance the workload among available cores. TBB library is implemented in modern C++, using various features from the 2011 standard. For instance, templates are used extensively in both parallel algorithms (e.g. `parallel_for`) and data structures (`concurrent_vector`, ...).

Compared to TBB, the OpenMP API [29, 86] provides much more low-level control. It is also easier to parallelize an existing code with it. However, considering that we are writing our code from scratch by insisting on the object-oriented part, we preferred TBB over OpenMP. Besides,

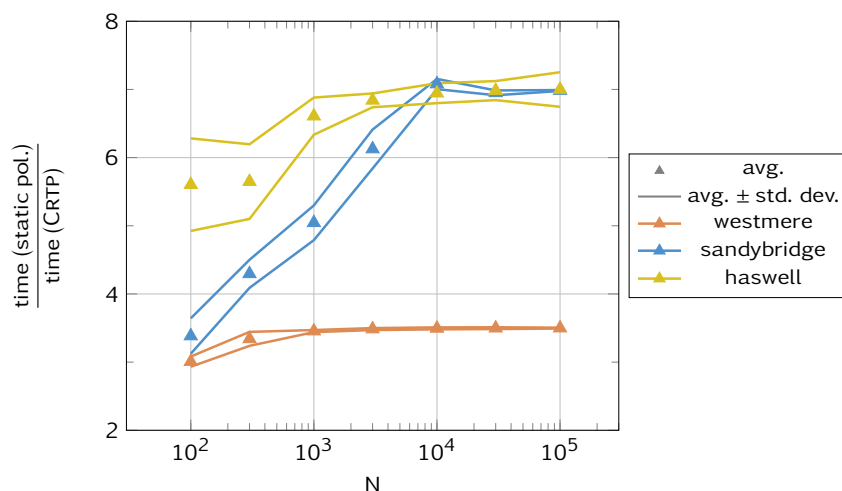


Figure 7.1 | CRTP vs. Dynamic Polymorphism (II). Acceleration provided by the CRTP for different values of N on different processors (desktop- westmere and sandybridge, and cirrus-haswell). We represented the average speedup over 100 runs, with plus/minus standard deviation. Runs were performed using version 4.8 of GNU compiler and O2 optimizations.

version 4 of OpenMP specification was not fully implemented when we started the integration of thread parallelization into the code, which definitely sold the case. In the end, TBB perfectly fits into our framework and keeps parallelism out of sight from the mainstream programmer. We nonetheless recently initiated the development (still in progress) of a branch using OpenMP instead of TBB to compare both systems on manycore architectures.

MD algorithms do not particularly exhibit task parallelism, hence we stuck to a classic “for loop” parallelism. To this end, we extensively used the `tbb::parallel_for` template function with C++ lambda functions, which we wrapped into a “parallel region”. This wrapping masks all TBB structure from the developer and would allow us to easily switch to another thread parallelization system, should the need arise. For the development of a new functionality, the developer only has to write a lambda function (which contains a serial algorithm) and select a traversal to dispose of a parallel algorithm.

As we explained in the previous chapter, thread parallelization strongly relies on the data structures used (see more in Section 7.1.3). If we had at first on each domain a large array of particles where we shared work, the `Cell` array now provides us with a much better grain size and particle locality for a better cache management.

In practice, all operations on the `Grid` are acting on a specific collection of particles that can be located according to their positions in cells, this statement allowing us to define a `Traversal`. As the example in Listing 7.3, all parallel operations are effectively divided using `Traversal`’s elements.

7.1.3 Particle List and Cell Array

Particle List

The idea behind the particle list was to separate particles from the cell structure, which was itself a mean to accelerate the neighbor search process. Operations which concerned all particles (ghost excepted), as time-integration or force computation (after neighbor lists have been computed) do not need to know in which cells these particles lie. These are treated directly in the `ParticleList`

```
// >>> file thread.hpp -----
template <class I, class J, typename Lambda>
inline void parallel_region(const I& begin, const J& end, Lambda lambda) {
    tbb::parallel_for(tbb::blocked_range<J>(begin, end),
        [&](const tbb::blocked_range<J>& r) {
            lambda(r.begin(), r.end());
        });
}

// >>> file singleSpecGrid.hpp -----
inline void SingleSpecGrid::pushPositions2ndOrder(const double& time) {
    const auto& real_cells = this->getTraversal(TraversalManager::REAL);

    parallel_region(0, real_cells.size(),
        [&](const uint begin, const uint end) {
            for(uint i=begin; i<end; ++i)
                m_allCells[ real_cells[i] ].pushPositions2ndOrder(time);
        });
}
```

Listing 7.3 | Parallel region

object. However other operations like neighbor search and pre-communication work precisely rely on the cell structure, to locate either potential neighbors or particles on the edges of a domain. Such operations are implemented in the Grid class. Both Grid and ParticleList object can access each others' data. Besides, threads are present in both classes, with a parallelization over cells in Grid, and particles in ParticleList.

The ParticleList object got its name as it behaves as a list from a certain angle, with particles entering and leaving it in the course of the simulation. Yet, lists are neither adapted for direct access of data, nor provide contiguous storage for vectorization. What we need is an extensible array such as the `std::vector` container provided by the Standard Template Library. This leads to the development of our own structure, which is detailed in Section 7.1.5.

If we obtained good performance on a single cluster node using thread parallelization (no communications), hybrid performance proved (relatively) disappointing above a hundred cores, as shown on Figure 7.2. A profiling study showed that all communication-related functions were inside serial regions at the thread level; according to Amdahl's law, the ratio of time spent in this serial region over the total time was becoming more and more important, reducing dramatically our acceleration. If serial regions cannot be completely eliminated, since effective MPI calls are not thread-safe, they must be significantly reduced. For instance, we may consider filling buffers and perform add/remove particles operations in parallel. Unfortunately, this is not possible with our particle storage solution: it is not possible to add/remove particles (at any position) in a single array concurrently.

Neither the addition of locks, nor the use of concurrent containers (`tbb::concurrent_vector`) allowed an improvement in performance. In order to properly address the issue, we tracked it back to its origins, and decided to change the Grid design, which lead us to remove the ParticleList in favor of a new structure: the Cell object.

Cell Array

As explained in the previous chapter, the new `Grid` explicits the cell structure which is used to search for neighbors. A `Cell` object is a simplified, smaller version of a `ParticleList`, as it contains particles of a `Cell`. The `Grid` is therefore composed of a set of `Cells` (which is implemented using an array), a structure to manage neighbor `Cells` and another to handle `Traversals`.

Having `Cells` in an array allows them to be treated independently. Even if memory allocation/deallocation form a lock, it does not have any impact since the number of cells is much bigger than the number of threads. To prevent `Cells` from being modified by two threads at once, we use mutexes provided by TBB.

This change in design eventually allows us to create parallel regions where only a single was in action when we used the `ParticleList`. We are now able to add and remove particles from the `Grid` in parallel, and also fill or process communication buffers concurrently. Figure 7.2 illustrates these improvements well: by executing the same simulation on the same processors, we were able to compare speedups provided by both versions (particle list and cell array). Beyond 40 cores (2 ivybridge nodes from Airain supercomputer), the cell array version is systematically better. On 200 cores, parallel efficiency was improved from 76% to 86% with 40 MPI $\times$ 5 TBB, and from 83% to 91% with 20 MPI $\times$ 10 TBB. Although we did not compare both versions on the Intel® Xeon Phi™, since the one with the particle list did not compile on this architecture (Intel® compiler was not fully supporting the C++ 2011 standard), we can nevertheless claim that the observed difference could only be more important, of not critical, with more that 60 cores. This is why we dropped the particle list for good in order to adopt a cell array design.

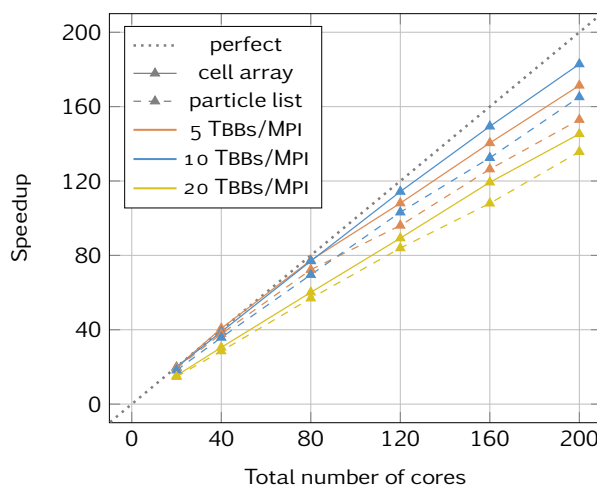


Figure 7.2 | Particle List vs Cell Array. Comparison of both versions with different combinations of MPI $\times$ TBB, using a strong scaling test. Simulations involved 8 million Copper atoms with a LJ potential for 512 iterations, and were performed on Airain's ivybridges. In all configurations, the version with the cell array scales better than the one using the particle list, since it contains less sequential regions. The fact that the curves with 20 threads per MPI process are quite lower than the others can be explained by NUMA effects, since Airain's ivybridges have two 10-cores sockets.

7.1.4 Particle Storage

Particle List: AOS vs. SOA

Within a `ParticleList`, we hesitated between two ways to store particles: either as an *array of structures* (AOS) or as a *structure of arrays* (SOA). The first one is generally preferred when data

are accessed as a whole (e.g. we need r_x , r_y and r_z at the same time to compute a distance), when the other is more efficient to access the same attribute of several particles at once (which happens in time integration and force computation). The AOS arrangement is also known for better cache handling, whereas a SOA is the only one that may benefit from vectorization.

We chose to implement both possibilities since there was no obvious choice. By default, the compiler was not able to vectorize any compute-intensive loop (i.e. in neighbor search or compute force functions), and results on sequential tests showed that SOA was slightly (around 5%) better than AOS. In profiling results on Figure 7.3 (columns 1 and 2), we observe that this advantage comes only from the force computation, since the neighbor search is faster in the AOS case.

When we eventually managed to enable explicit vectorization (see Section 7.3), the SOA arrangement proved definitely faster.

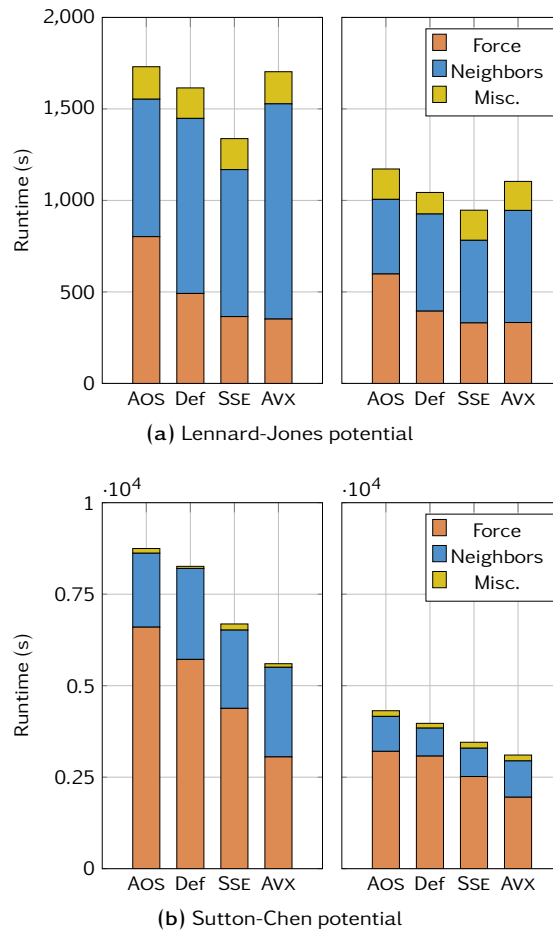


Figure 7.3 | AOS vs SOA. Runtime (in seconds) with profiling details of simulations with LJ (a) and SC (b) potentials. Left and right part are respectively without and with symmetrization enabled. Simulations involved 10^6 Copper atoms for 128 iterations, and were performed on an Cirrus' ivybridges. In each plot, the first column (AOS) corresponds to the AOS arrangement, when the others concern SOA. Def means default, i.e. we rely on compiler's (Intel®) auto vectorization, when SSE and AVX modes use explicit intrinsics. The neighbor search performance problem using AVX has been corrected in newer versions.

Cell Array: AOSOA

When we decided to express the cell structure, we could have kept the SOA arrangement to store particles: by selecting a constant N_{MAX} that represents the maximum number of particles per cell,

we can then locate particles from cell c starting index $c \cdot \text{NMAX}$. This option, which is in fact utilized in CoMD, works well on homogeneous systems but is certainly not adapted to heterogeneous ones. The NMAX parameter must be precisely estimated since a particle cannot enter a full cell, but if we choose a value too large, the simulation will not fill into the memory, as there is no difference between an empty and a full cell.

This is why we went for a different solution, where each cell is treated as an independent array. On the grid, we handle an array of `Cells`, and on a `Cell`, we have particles into a SOA, which makes the global storage solution an array of SOA, or AOSOA. Note that to ensure performance, and especially vectorization, cells must contain a minimal number of particles. Considering typical densities in materials used for shock physics simulations, we do not need to worry about that.

Using the AOSOA in our latest version, we restarted experiments from Figure 7.3. Results in Figure 7.4 show an average acceleration of 1.2 regarding the SOA arrangement. With a significant serial acceleration and a better scaling, we eventually adopted the `Cell` structure with AOSOA storage, and dropped the duo AOS/SOA.

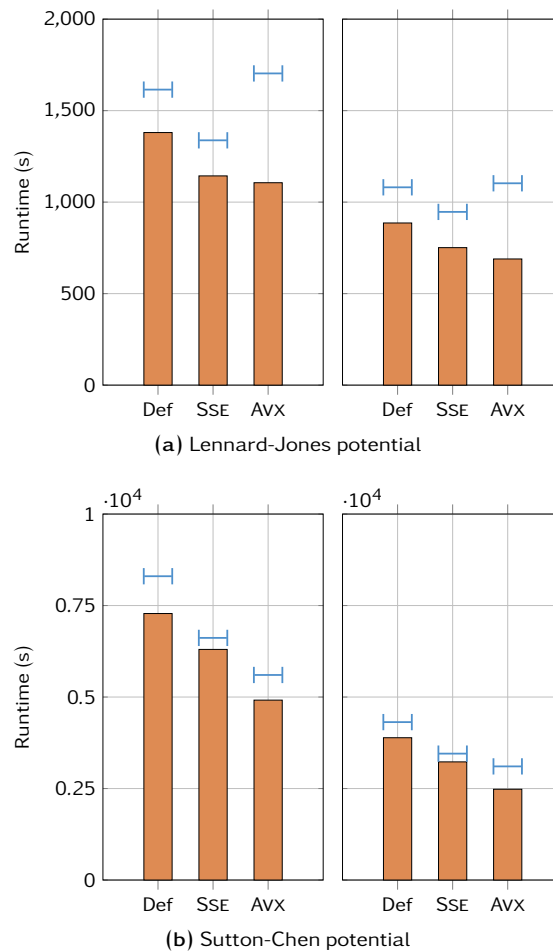


Figure 7.4 | AOSOA. Resuming of simulations from Figure 7.3 (same system, same processor, but without profiling details) with our latest version using an AOSOA arrangement to store particles. Blue bars are a reminder of Figure 7.3's results. We observe that this new version is faster in all cases, approving this storage solution for our architecture.

```

template <class T, uint32_t chunk=16, size_t align=0> class ExtArray {
public:
    ExtArray();
5   ExtArray(const uint32_t n);
    ExtArray(const uint32_t n, const T& value);
    ExtArray(const ExtArray<T>& array);

    ExtArray<T>& operator = (const ExtArray<T>& array);
10
    T& operator [] (const uint32_t index);
    T& back();
    T* data();

15   virtual uint32_t size();
    void reserve(const uint32_t n);
    uint32_t capacity();

    void clear();
20
    void push_back(const T& object);
    void pop_back();
    void pop(uint32_t index);

25   void resize(const uint32_t n);
    void resize(const uint32_t n, const T& val);

    void assign(const uint32_t n, const T& value);

30   void shrink_to_fit();
}

```

Listing 7.4 | The ExtArray container: public elements

7.1.5 Container Used in Cells

The `Cell` is the low-level structure which contains all compute-intensive operations. Its implementation is therefore critical for code performance. As particles may enter and leave the cell at every iteration, it has the behavior of a list.

We have just seen that particles within a `Cell` are stored as a structure of arrays so that we can benefit from vectorization. As particles may enter and leave the cell at every iteration, these arrays must be able to change size like the `std::vector` from the STL. Several points prevented us from using this container in `Cells`: the `std::vector` container is a *standard*, and we have no guarantees about its implementation and performance, which may change from a compiler to another; besides, we were not satisfied with the observed strategy for reallocation, which consists in doubling the size when the `std::vector` becomes full. Indeed, the fact of multiplying the capacity is a more general method to minimize the number of long-term allocations, the number of particles in a cell quickly stabilizes and oscillates around a mean value. A reallocation by blocks would be a better strategy for us. We therefore developed our own container to store particles. In other, less critical parts of the code, we kept using `std::vector`.

The `ExtArray` (which stands for extensible array) is a template container with three template arguments (see Listing 7.4), the first one (`T`) being obviously the type of element stored. The `chunk` parameter guarantees that the capacity of the container underlying array is *always* a multiple of its value. It has an impact when using the `push_back` method, where the reallocation is made by blocks of `chunk` elements, instead of multiplying the capacity. Moreover, since the call of every other method which modifies the capacity (`reserve`, `shrink_to_fit`, ...) will also result in an

array of size which is a multiple of `chunk`, this parameter prevents buffer overflow in vectorized loops. For instance, if someone uses SIMD instructions with chunks of size 2 on an array which has an odd number of elements, they have to perform a test in order not to overtake his array and cause a segmentation fault. When the parameter `chunk` is properly set² the developer does not have to worry about such considerations. The `align` parameter eventually guarantees that the underlying array is aligned on a `align`-byte boundary to quickly load elements into registers.

`ExtArray`'s set of public methods is similar to `std::vector`'s one. We nonetheless provided a `pop` method to remove an element anywhere in the array in constant time. This method, which copies the last element on the index to remove before deleting the last element, modifies the order in the array. This is not a problem for the `Cell` object, but it must be handled with caution elsewhere. We did not implement any exception mechanism in class `ExtArray`, and a bad utilization will cause undefined behavior. However, we dispose of a flag which enables the check of all sizes and accesses in debug mode.

7.2 Force Computation

In this section, we will discuss the implementation of the force computation process, which involves methods on both `Grid` and `Cell` objects. We will start with the neighbor search, then follow with the “real” force computation before explaining how we introduced symmetrization and communication overlap (see previous chapter) without duplicating too much code.

7.2.1 Neighbor Search

The neighbor search process has already been outlined in Algorithm 6.3: we take advantage of the cell structure to avoid a $O(n^2)$ algorithm. Neighbor lists are stored in `Cell` structures, which makes the neighbor search completely parallel within a `Domain` since thread parallelization is based on `Cells`.

We report here the evolution of the neighbor search implementation, with all important optimizations introduced. To account improvements, we used the same Lennard-Jones benchmark (131k atoms, 512 iterations) on a single core of a haswell processor. Atom and potential parameters are available in Appendix B. Results are summarized in Figure 7.5.

Simple Implementation

Neighbor search starts on the `Grid` level, with a `Traversal` as argument since the search can focus either on all cells or only on a part depending on whether communication overlap is enabled. Using its notations, we follow Algorithm 6.3, but only on cells given by the `Traversal` in entry instead of all cells.

When it comes to evaluate the distance r_{ij} , in most cases a norm computation is enough, except when the neighbor cell is a ghost cell across a boundary condition: the difference between coordinates $\mathbf{r}_i$ and $\mathbf{r}_j$ must be corrected. In general, whenever periodic boundary conditions are involved in a dimension d , all differences of coordinates $\Delta = r_i^{(d)} - r_j^{(d)}$ in this dimension are

²i.e. a power of two, or at least a multiple of 8 is a good start.

```

1  for (c : Traversal) {
2      for (n : neighbor[c]) {
3
4          for (i : cell[c]) {
5              for (j : cell[n]) {
6                  if (rij < rcut && i < j) add_neighbor(i, j)
7              }
8          }
9      }
10 }

```

Listing 7.5 | Neighbor search with symmetrization (I)

corrected according to:

$$\Delta' = \begin{cases} \left(r_i^{(d)} - r_j^{(d)}\right) + L^{(d)} & \text{if } \left(r_i^{(d)} - r_j^{(d)}\right) < -\frac{L}{2} \\ \left(r_i^{(d)} - r_j^{(d)}\right) & \text{if } -\frac{L}{2} \leq \left(r_i^{(d)} - r_j^{(d)}\right) \leq +\frac{L}{2} \\ \left(r_i^{(d)} - r_j^{(d)}\right) - L^{(d)} & \text{if } +\frac{L}{2} < \left(r_i^{(d)} - r_j^{(d)}\right) \end{cases} . \quad (7.1)$$

Having such a test for all distance evaluations is very expensive, as we can see on *simple* entry in Figure 7.5. When symmetrization is not enabled, the neighbor list computation takes 80% of the computation time.

Neighbor Cell Differentiation

The necessary test introduced in (7.1) only makes sense when we are in a cell located on the edges of the global domain, looking for neighbors in a ghost cell. Thus we can split these cases in a standard one without the correction test, and in another one which is only enabled for (global) edge cells. This split brings a speedup up to 1.3 on the neighbor search algorithm, making the overall simulation 27% faster (see *split* entry in Figure 7.5).

Test Refactoring in Symmetrization Case

When using symmetrization, the test $r_{ij} < r_{\text{cut}}$ to evaluate if particles i and j are neighbors becomes $r_{ij} < r_{\text{cut}}$ AND $i < j$ to count the pair only once. Once again, it introduces a test in the middle of a compute-intensive loop, which almost guarantees poor performance (see line 6 of Listing 7.5).

Our idea was to move this extra test higher in the algorithm, i.e. at the cell level, as you can see on Listing 7.6, line 4. As expected, this change has no effect when we are not using symmetrization. When enabled, this operation accelerates the neighbor search by 28%, and the global simulation by 20%, when comparing to the *split* case.

Other Optimizations

Two more optimizations have been implemented. In the first one, we managed to vectorize a part of the neighbor search, but we keep implementation details for Section 7.3.

The second one is a projection calculation of a particle position on a cell's edges: if the distance between a particle i and a neighbor cell n is already larger than the cutoff radius, there is no need

```

1  for (c : Traversal) {
2      for (n : neighbor[c]) {
3
4          if (c < n) continue
5
6          for (i : cell[c]) {
7              for (j : cell[n]) {
8                  if (rij < rcut) add_neighbor(i, j)
9              }
10         }
11     }
12 }

```

Listing 7.6 | Neighbor search with symmetrization (II)

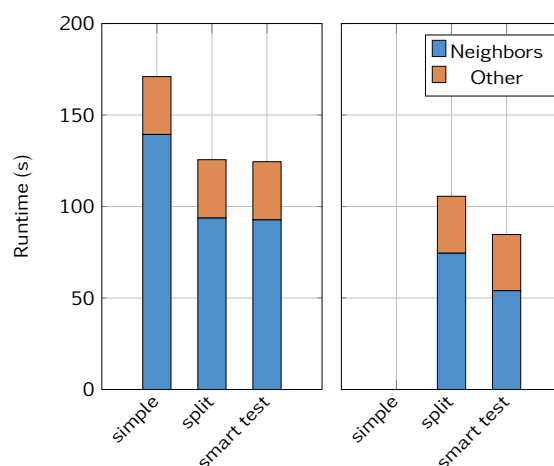


Figure 7.5 | **Optimizations of neighbor search.** Runtimes (in seconds) for successive optimizations implemented in the neighbor search process. Left and right are respectively non-symmetric and symmetric searches. Simulation with a Lennard-Jones potential (131k atoms, 512 iterations) on a single core of a haswell processor.

to test particles of n against i . This optimization is especially profitable when the cutoff radius of a potential is much smaller than the cell's size. This is more likely to happen when performing simulation involving more than one atom type. In our LJ benchmark, the projection optimization does not bring a noticeable acceleration.

7.2.2 Force Computation

Once neighbor lists have been created, we can start the force computation process. On a `Domain`, we loop through each potential object (one potential for each interaction), and call the `Grid` method corresponding to the pattern of the given potential. Indeed, we saw in algorithms 6.1 and 6.2 in the last chapter that we had recurrent force computation patterns for all pair and EAM potentials, the potential specificity being the expression of its inner functions (V for a pair potential, ρ , F and ϕ for an EAM). To avoid code duplication, we only implemented those patterns on the `Grid`. They run through cells in parallel and call the following `Cell` methods, which are easily identified to blocks observed in algorithms 6.1 and 6.2:

- `computeForcePair();`
- `computeForceEamRho();`

- `computeForceEamF()`;
- `computeForceEamFinal()`.

Among these methods, `computeForceEamF`, is a bit different since it does not involve neighbors. It just computes the embedded term $F(\bar{\rho})$ of an EAM potential for every particle in the cell, the $\bar{\rho}$ term being computed right before using `computeForceEamRho` method. Function $F(\cdot)$ expression is carried by the `EamPotential` object.

Methods `computeForcePair`, `computeForceEamRho` and `computeForceEamFinal`, are the core of the force computation and the most time-consuming functions. They happen to all have a similar structure: we go through every particle i in the `Cell`, then through each neighbor j of i , and compute quantities in function of r_{ij} , which is the distance between particles i and j ; these quantities are then written into particle i 's attributes. The main issue with this structure is that the computation of r_{ij} is full of indirections: under any circumstances, all neighbors of each particles can be stored contiguously in memory, preventing from the beginning all chances of vectorizing these parts of the code.

The `VectorizationBuffer` is an object composed of several small arrays which has been specifically designed to bypass the indirection problem in `Cell` force computation methods. After the neighbor search, all its internal arrays are allocated with the size of the largest neighbor list. For one particle i , we fill three of `VectorizationBuffer`'s arrays with the components of vector $\mathbf{r}_i - \mathbf{r}_j$. All computations can now be carried without indirections before writing results in `VectorizationBuffer`'s dedicated arrays. As the `VectorizationBuffer` object is declared out of a parallel region, it may be used by different threads at the same time if we do not pay attention. To prevent this, we used the `tbb::enumerable_thread_specific` template structure, which ensures that each thread will have its own private copy of the `VectorizationBuffer` to perform computation safely.

In functions `computeForcePair`, `computeForceEamRho` and `computeForceEamFinal`, we have mutualized writing and reading operations on the `VectorizationBuffer` which allows us to avoid code duplication once more.

7.2.3 Communications Overlap

Communication overlap is a technique where calculations are performed during communications to improve speedup. Algorithms to compute force with and without overlapping communication have been presented in the previous chapter (see Section 6.3.5). Implementation on `Domain` simply consists in a rearrangement of lower-level calls. To choose whether or not to activate communication overlap, we used a strategy pattern. On `Grid` level, the force computation function must exist in three different versions, to carry the computation on respectively *inside*, *edge*, and *all* cells. Using our `Traversal` structure, this split is easily made: neighbor search and actual force computation methods take a `Traversal` as argument, which allows us to keep one function and also prevents code duplication.

In order to compare performance, we run ExaSTAMP with and without communication overlap enabled. Simulations involved 8 million atoms for 2048 iterations on 200 cores of Airain's ivybridges. Once again, we tried different combinations of the number of MPI processes and the number of threads. On Table 7.2, we observe that enabling communication overlap does not increase performance, simulation with overlap being even slightly slower (about 3%). It can be

partially explained by the network (Airain uses an Infiniband QDR), which considerably reduces communication cost. Moreover, splitting neighbor search and force function in two parts may induce more overhead for thread parallelization.

Num. MPI	Num. TBB	Comm. Overlap	
		on	off
40	5	4.354	4.223
20	10	4.072	3.951
10	20	5.169	4.958

Table 7.2 | Impact of communication overlap. Performance is measured in *efficient time*, which is the average compute time (in microseconds) per particle per iteration per thread (lower = better). This measurement, which is also used by CoMD, allows us to easily compare simulations with different number of particles, iterations or cores. In all cases, enabling communication overlap does not allow a gain in performance.

In our case, modern clusters with fast network seem to make communication overlap technique non-effective. As its implementation does not increase the code size (except a few lines for the strategy pattern), we decided to keep it in case we meet some hardware where it may be worth it.

7.2.4 Enabling Symmetrization

Newton's third law ensures that, if $\mathbf{f}_{ij}$ is the force acting on particle i resulting from the interaction of particles i and j , then the force acting on particle j resulting from this interaction will be $\mathbf{f}_{ji} = -\mathbf{f}_{ij}$. There are therefore two ways of computing forces:

- if i is a neighbor of j , then j is also a neighbor of i and both particles compute interaction i/j ;
- if i is a neighbor of j , then j is not a neighbor of i ; when particle i computes interaction i/j , it also has to write the resulting force in particle j 's attributes.

The implementation of symmetrization in the neighbor search process has already been discussed. In the force computation, we have to modify the part where results are extracted from the `VectorizationBuffer` to be written in particle's attributes. As the force computation methods in `Cells` object are executed in parallel, all writing operations must be protected.

The TBB library offers several types of locks. Among them, the `tbb::spin_mutex` got our attention, since we are exactly in the situation described in the documentation:

`Tbb::spin_mutex` is non-scalable, unfair, non-recursive, and spins in user space. It would seem to be the worst of all possible worlds, except that it is very fast in lightly contended situations. If you can design your program so that contention is somehow spread out among many `tbb::spin_mutex` objects, you can improve performance over using other kinds of mutexes.

On a `Cell`, we have as much `tbb::spin_mutex` objects as particles, and therefore lock only a single particle for a writing operation. The memory cost of having one lock per particle is limited since `tbb::spin_mutexes` use one byte each. Profiling analyses with Intel® VTune® Amplifier showed that the time spent by threads waiting in `tbb::spin_mutex` during the force computation is close to zero, and can fairly be omitted in front of the time spent in synchronization barriers at the end of parallel regions.

As for performance, results in Figure 7.4, which we used to compare the AOSOA arrangements against AOS and SOA, also let us estimate the gain offered by symmetrization: simulations with

symmetrization enabled are on average 40% faster with a LJ potential, and 95% faster with the SC potential. The difference of acceleration can be explained by the fact that the LJ potential is much cheaper in computation time than SC: the ratio of the total time spent in the force computation is around 80%, whereas it gets near 100% with a SC potential.

7.3 Explicit Vectorization

We saw in Chapter 5 that vectorization was important to achieve some performance on current processors, and that it will become critical on architectures like the Intel® Xeon Phi™. Despite the progress made by compilers regarding auto-vectorization over the last years, writing code to maximize the number of vectorization opportunities detected by the compiler remains a delicate process. Moreover, they cannot outcome explicit writing of vectorized code, which is a arduous technique that we cannot expect to be mastered by standard developers.

In this section, we present a template tool that solves this issue by generating vectorized instructions from a sequential-looking code. We will start by presenting some test-kernels that we will use to show auto-vectorization limits. We will then describe our tool, before applying it to the same kernels to prove its efficiency.

7.3.1 Test Kernels

Test kernels are presented in Listing 7.7. They all consist in pieces of code that we can find in ExaSTAMP. The template parameter T is the data type on which the kernel operates. If ExaSTAMP uses only double precision, we kept the ability of using single-precision to be the most flexible possible.

Push. Kernel push is used in time-integration to update particles' position with velocities and forces using an Euler schemes of order 2.

Neighbors. This function tests whether a given particle which position is $\mathbf{r}_j$ is in the neighborhood of a set of particles which positions are stored in arrays rx , ry and rz . The neighborhood is defined by the (squared) cutoff distances in array rc in order to simulate a multi-material simulation. The loop computes a square norm for each particle, and compares it with the cutoff radius and writes the result in array out .

Lennard-Jones. This function computes the potential energy and forces between a particle and its neighbors using the Lennard-Jones potential. Array rx , ry and rz store the different component of $\mathbf{r} - \mathbf{r}_i$, $\mathbf{r}$ being the particle's position and $\mathbf{r}_i$ its neighbor positions. Potential energy is written in ep array and force in fx , fy and fz .

7.3.2 Auto-Vectorization Limits

Auto-vectorization indicates the ability of a compiler to transform scalar portion of a code into a vector one, i.e. one able to perform a single operation on several elements of data at once. In order to estimate this ability, we compiled all test kernels using different compilers, with vectorization and vectorization report enabled. With GNU 4.6.3, none of the test kernels are vectorized, the reason being that *“loop contains function calls or data references that cannot be analyzed”*. More

```

template <class T>
void push (const uint N, T dt, T *x, T *v, T *f) {

    T halfdt2=0.5*dt*dt;

5   for (uint i=0; i<N; ++i) {
        X[i] += dt*v[i] + halfdt2*f[i];
    }

10  }

template <class T>
void neighbors (const uint N, T rjx, T rjy, T rjz, T *rc, T *rx, T *ry, T *rz, T *out) {

15   T tmpo, tmp1, tmp2;

    for (uint i=0; i<N; ++i) {
        tmpo = rjx - rx[i];
        tmp1 = rjy - ry[i];
        tmp2 = rjz - rz[i];
        out[i] = (tmpo*tmpo + tmp1*tmp1 + tmp2*tmp2) < rc[i];
    }

25  }

template <class T>
void lennard_jones (const uint N, T *rx, T *ry, T *rz, T *fx, T *fy, T *fz, T *ep) {

30   T sigma=1.0, epsilon=1.0, ecut=1.0;

    T _sigma2 = sigma*sigma;
    T _2epsilon = 2*epsilon;
    T _24epsilon = 24*epsilon;
    T _minus_half_ecut = -0.5*ecut;

    T tmpo, tmp1, tmp2;

40   for (uint i=0; i<N; ++i) {
        tmpo = 1.0/( rx[i]*rx[i] + ry[i]*ry[i] + rz[i]*rz[i] );
        tmp1 = _sigma2 * tmpo; // tmp1 = (sigma/r)^2 = rt^2
        tmp2 = tmp1 * tmp1 * tmp1; // tmp2 = rt^6
        tmp1 = tmp2 * tmp2; // tmp1 = rt^12
        tmp2 = tmp1 - tmp2;
        tmp1 = tmp2 + tmp1;
        tmp1 = _24epsilon * tmp1 * tmpo; // tmp1 = 24*eps*(2*rt^12-rt^6)/r2
        ep[i] = _2epsilon * tmp2 + _minus_half_ecut; // ep = 2*eps*(rt^12-rt^6)-ecut/2
        fx[i] = rx[i] * tmp1;
        fy[i] = ry[i] * tmp1;
        fz[i] = rz[i] * tmp1;
    }

50  }
}

```

Listing 7.7 | Test Kernels for explicit vectorization

recent versions (4.8.3 and 4.9.2) are able to vectorize both push and neighbor kernels, but not the lennard-jones one (“*failure to determine dependencies*” and “*complicated access pattern*”). As for a recent Intel® compiler, it reports the vectorization of push and lennard-jones kernels, but fails on neighbors (“*vector dependence prevents vectorization*”).

In order to test real performance of auto vectorization, we ran these test kernels on a haswell (on Cirrus), and compared the average runtimes against those obtained with GNU 4.6.3, which has not vectorized anything. As the haswell supports AVX instructions, theoretical speedups are 8 on float and 4 on doubles. Results are reported in Table 7.3, where a bold green cell means that the kernel has been reported vectorized, whereas a red one indicates that it has not. We observe that Intel® compiler provides slightly better performance than GNU for the push kernel (which is the only beeing vectorized by both). On average, auto vectorization produces speedups respectively around 4–5 and 2–3 for single and double precision numbers, which are quite distant from what we expected.

Those results show that there is no compiler able to vectorize all three kernels, and that acceleration provided by auto-vectorized versions is disappointing. If we want to meet performance, especially on the Intel® Xeon Phi™, we must change the writing of our kernels.

Compiler	Push		Neighbor		Lennard-Jones	
	float	double	float	double	float	double
GNU 4.6.3	1.00	1.00	1.00	1.00	1.00	1.00
GNU 4.8.3	4.25	2.19	5.26	2.72	1.13	1.04
GNU 4.9.2	4.74	2.55	4.43	2.25	1.13	1.05
Intel® 15.0.2	5.32	2.79	4.53	0.89	3.92	1.11

Table 7.3 | Auto-vectorization limits. Acceleration regarding GNU 4.6.3 version (on haswell). A green cell means that the kernel has been reported vectorized, unlike the red ones. Average over 4096 runs, with arrays of size 256 (floats) and 128 (doubles). If compiler auto-vectorization provides a certain acceleration, it remains quite far from theoretical speedups (×8 on floats and ×4 on doubles for AVX instructions). We also observe that Intel® compiler accelerates the neighbor kernel (in single-precision) when it was not supposed to, and that it does not seem to vectorize the LJ in double precision, which was the opposite of what was announced.

7.3.3 A Template Intrinsic Generator

Intrinsics

An intrinsic or *builtin function* is a function that is directly implemented by the compiler, so that it can better integrate and optimize it for the situation. For instance, the push kernel version with intrinsic functions for double precision and SSE instruction is available in Listing 7.8: although it is explicitly vectorized, this code is hard to read and a pain to maintain. Indeed, we need another version for single precision, another one to work with AVX instructions . . . Overall, if we want both single and double precision, in sequential (for debug), SSE, AVX, and for the Intel® Xeon Phi™ (IMCI on KNC, AVX-512 for the KNL), we need eight versions of each kernel!

Intrinsics are very specialized, which is a necessary price to get performance. It raises the issue of code duplication and will be hardly handled by our developers. Without a workaround, they are very unlikely to be included in our implementation.

```

void push (const uint N, double dt, double *x, double *v, double *f) {

    __m128d dt      = _mm_set_pd1(dt);
    __m128d halfdt2 = _mm_set_pd1(0.5*dt*dt);

5   __m128d tmpo, tmp1;

    for (uint i=0; i<N; i+=2) {
        tmpo = _mm_load_pd(x+i);           // tmpo = x[i:i+1]
        tmp1 = _mm_load_pd(v+i);           // tmp1 = v[i:i+1]
10     tmp1 = _mm_mul_pd(dt, tmp1);         // tmp1 = dt*v[]
        tmpo = _mm_add_pd(tmpo, tmp1);      // tmpo = x[] + dt*v[]
        tmp1 = _mm_load_pd(f+i);           // tmp1 = f[]
        tmp1 = _mm_mul_pd(halfdt2, tmp1);  // tmp1 = (dt^2/2)*f[]
15     tmpo = _mm_add_pd(tmpo, tmp1);      // tmpo = x[] + dt*v[] + (dt^2/2)*f[]

        __mm_store_pd(tmpo, x+i);          // write tmpo back into array
    }
20 }

```

Listing 7.8 | Intrinsic implementation of push kernel for double-precision and SSE instructions

Intrinsic Wrapping

In order to keep both a high level of performance and our developers happy, we decided to hide intrinsics behind a custom template object: the `simd::vector`. Our vector object has two templates arguments: the underlying type of data that it represents (`float` or `double`), and the type of SIMD instruction that will be generated (see Table 7.4); it contains basic operators, comparison operators, and also some math functions (fast inverse, square root, exponential, logarithm, trigonometric functions, ...). When the code in Listing 7.9 is compiled with `T=double` and `instr=simd::sse`, the generated code is exactly the same as Listing 7.8.

instr_t	T	gen. type	chunk
simd::serial	float	float	1
	double	double	1
simd::sse	float	__mm128	4
	double	__mm128d	2
simd::avx	float	__mm256	8
	double	__mm256d	4
simd::imci	float	__mm512	16
	double	__mm512d	8

Table 7.4 | Overview of our intrinsic generator. The first two columns are the `simd::vector` templates arguments. The third one corresponds to the type which is effectively handled behind the vector, and the fourth, the “size” of the vector (which is used to increment the loop index).

We managed to get a code as easy to write as to understand, the only difference with the sequential implementation being the store and load methods. Unlike explicit intrinsics, the `simd::vector` is likely to be handled by developers. We also removed the code duplication issue, all different versions being implemented from the same template.

```

template <class T, simd::instr_t instr>
void push (const uint N, const T& t_, T *x_, const T *v_, const T *f_) {

    typedef simd::vector<T, instr> vector_t

5   vector_t x, v, f, result;
    vector_t dt(t_);
    vector_t half_dt_square(0.5*t_*t_);

10  for (uint i=0; i<N; i+=vector_t::chunk_size) {

        v.load(v_+i);
        x.load(x_+i);
        f.load(f_+i);

15         result = x + dt*v + half_dt_square*f;
        result.store(x_+i);

    }

20 }

```

Listing 7.9 | Implementation of push kernel with our intrinsic generator

Performance of the Intrinsic Generator

To evaluate the performance of our wrapper, we ran the exact same tests as in Table 7.3 using GNU 4.9.2 and Intel® 15.0.2 (AVX intrinsics are only included in recent versions), and compared the acceleration against the version with GNU 4.6.3 (see Table 7.5). This allowed us to estimate the level of vectorization we can reach. We then compared in Table 7.6 the performance of our wrapper against auto-vectorization on different processors.

Compiler	Push		Neighbor		Lennard-Jones	
	float	double	float	double	float	double
GNU 4.9.2	6.91	3.82	7.99	3.89	7.36	3.38
Intel® 15.0.2	6.95	3.83	7.41	3.89	7.39	3.30

Table 7.5 | Performance of our intrinsic generator (I). Acceleration provided by our intrinsic generator against a version without vectorization (using GNU 4.6.3). Average over 4096 runs, with arrays of size 256 (floats) and 128 (doubles). There are no significant differences between GNU and Intel® compilers.

Processor	Compiler	Push		Neighbor		Lennard-Jones	
		float	double	float	double	float	double
standard	GNU 4.9.2	0.99	0.55	1.00	1.01	3.12	1.89
	Intel® 15.0.2	0.77	1.02	1.17	3.31	1.03	1.87
haswell	GNU 4.9.2	1.46	1.50	1.80	1.73	6.51	3.22
	Intel® 15.0.2	1.31	1.37	1.64	4.37	1.89	2.98
KNC	Intel® 15.0.2	1.65	1.29	1.06	5.96	1.35	7.46

Table 7.6 | Performance of our intrinsic generator (II). Acceleration provided by our intrinsic generator against versions using compiler auto-vectorization. Average over 4096 runs, with arrays of size 256 (floats) and 128 (doubles).

Our intrinsic generator provides an acceleration between 6.9 and 8 with single precision, and between 3.3 and 3.9 with double precision when compared to the non-vectorized version (Table 7.5). This represents at least 86% and 82% of the theoretical speedup, which is the level of

performance that we can expect from intrinsics. When comparing the intrinsic generator to the same code that benefits from auto-vectorization (Table 7.6), it is (almost) always faster. In simple cases (push for floats and doubles, neighbor for floats only) and on processors using only SSE instructions, auto-vectorization has similar or slightly better performance. As soon as we start using advanced instructions, our intrinsic generator provides an appreciable acceleration. Its use becomes essential on “complicated” kernels (i.e. neighbors and LJ) with double precision, especially on the Intel® Xeon Phi™, where the excellent accelerations (respectively $\times 5.96$ and $\times 7.46$ for neighbor and lj kernels) is due to poor auto vectorization.

In this section, we proved that vectorization of compute-intensive kernels was mandatory to reach performance, and that compilers are not (yet) able to provide efficient vectorized code. Intrinsics functions are a way to reach performance, but their complexity makes them hard to operate on, especially on a framework which must run on different types of processors. With our intrinsic generator, we provide a tool that wraps intrinsics into template objects. This allows non-expert developers to implement sequential-looking functions, that are transformed into vectorized instructions at compiler time. Performance of generated functions are much better than those which benefited from auto-vectorization, which therefore validates the integration of our intrinsic generator in ExaSTAMP.

7.4 Communications

ExaSTAMP architecture has several Domains per Node in order to anticipate hierarchical memory accesses. As a Domain communicates with all its neighbors, we want to control the communication flow, i.e. we want to be able to replace communications between two Domains on a Node by explicit memory copies, or to group/ungroup messages depending on the machine used.

Once again, all these changes must not be noticeable for the developer, who will trigger communications with specific interfaces. To this end, we wrapped collective MPI function calls in template methods of the CommManager, and proposed a set of objects which makes point-to-point communications easier. If we did not implement the multi-domain case yet, the following communication algorithms already support this capability, which should facilitate its future integration.

7.4.1 Towards More Intuitive MPI Prototypes

For historical reasons, prototypes of MPI communication functions use generic addresses for buffers (`void*`), which means that the type and size of data has to be specified otherwise, as well as the communicator. In numerical simulation codes, communications are almost always operated on arrays with the default communicator; moreover, specific or custom classes that contains type and size information are available when using C++ language.

We present here some simplifications for the developer in the use of collective communications (no more direct handling of buffer sizes and data types). If we rely on a particular example, all collective communication in ExaSTAMP effectively benefit from these simplifications. For instance, the `MPI_Gather` function prototype in Listing 7.10 may be simplified into the one in Listing 7.11, provided certain conditions:

- the MPI communicator is reachable from the communication manager;

```
int MPI_Gather (void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtpe,
               int root, MPI_Comm comm);
```

Listing 7.10 | Standard prototype for MPI_Gather function

```
template <class T>
int CommManager::gather (Array<T>& in, Array<T>& out, int root);
```

Listing 7.11 | Custom prototype for our MPI_Gather equivalent

- Array container uses contiguous storage for its elements, with access to the first element's address (e.g. `data()` method from `std::vector`);
- Array container has a method to access its number of elements (e.g. `size()` method from `std::vector`);
- there is a way to get a `MPI_Datatype` from the template argument `<T>`.

Considering our design of the `CommManager` object, the first condition is fully satisfied, and our class `Array` (similar than a `std::vector` but lighter) has been defined with methods that meet second and third points. Besides, the simplified prototype can be adapted to any other container following these rules. Regarding the last requirement, the solution we provide is developed below.

A Tool to Handle MPI Data Types

The `std::type_index` class is a wrapper class around a `std::type_info` object that was introduced in the 2011 standard. It can be used as index in associative and unordered associative containers. Thus we can define a `std::map<std::type_index, MPI_Datatype>` container that stores `MPI_Datatypes`, with corresponding `std::type_index`s as keys.

This map is initialized with all MPI predefined types and ExaSTAMP custom types (e.g. `Particle`); the initialization takes place right after the `MPI_Init` call. During execution, a `MPI_Datatype` can be retrieved by the communication manager with a call to the template function `MPI__type_get()`. Right before the `MPI_Finalize` call, elements that are not predefined `MPI_Datatype` are freed.

Listing 7.12 gives an insight of what we have just described. The function `MPI__Init_types()` has been shortened to fit in the page: if we let the `MPI__Create_type_Particle()` function as an example, more custom types are effectively created.

Adding New Types

The creation of derived types uses directly functions from the MPI standard. So far, we used exclusively `MPI_Type_contiguous` and `MPI_Type_create_struct` to create our derived types. Once a new type has been committed, it must be registered to our map with `MPI__Type_add` to be used in the rest of the code. In the first developments by “external” programmers, this capability has been quite appreciated. Its handling did not bring any issue.

```

// >> file mpiTypes.hpp -----
typedef std::map<std::type_index, MPI_Datatype> MapTI;

// acces a MPI_Datatype
5 template <class T> MPI_Datatype MPI__Type_get () {
    extern MapTI MPI__TYPES_MAP;
    return MPI__TYPES_MAP [ typeid(T) ];
}

10 // register a MPI_Datatype
template <class T> void MPI__Type_add (MPI_Datatype type) {
    extern MapTI MPI__TYPES_MAP;
    MPI__TYPES_MAP [ typeid(T) ] = type;
15 }

// initialization of the MPI_Datatype database
void MPI__Init_types();

20 // free all MPI_Datatypes
void MPI__Free_types();

// >> file mpiTypes.cpp -----
25 extern std::map<std::type_index, MPI_Datatype> MPI__TYPES_MAP;

void MPI__Init_types () {
30     // register common types

    MPI__Type_add<char>(MPI_CHAR);
    MPI__Type_add<int> (MPI_INT);
    MPI__Type_add<long>(MPI_LONG);
35     MPI__Type_add<float> (MPI_FLOAT);
    MPI__Type_add<double>(MPI_DOUBLE);

    // register custom types
40     MPI__Create_type_Particle();

    // [...]
45 }

void MPI__Free_types () {
50     // erase predefined types (because it's not possible to free them)

    MPI__TYPES_MAP.erase( typeid(char) );
    MPI__TYPES_MAP.erase( typeid(int) );
    MPI__TYPES_MAP.erase( typeid(long) );

55     MPI__TYPES_MAP.erase( typeid(float) );
    MPI__TYPES_MAP.erase( typeid(double) );

    // free all others MPI_Datatypes
60     for (auto& elem : MPI__TYPES_MAP) MPI_Type_free(&elem.second);
}

```

Listing 7.12 | Automatic access to MPIData Types

7.4.2 Point-to-point Communications

We saw in the last chapter that all point-to-point communications are part of a more global exchange process, which is why they all are embedded in specific structures. For one exchange process, the mainstream developer disposes of three objects to help him perform point-to-point communications:

- `MessageSend` which provides a method to fill it with data to send;
- `MessageReceive` that has a method to collect data received;
- `MessageCenter` which gives access to `MessageSend` and `MessageReceive` items, and also has methods to start, wait and clean a set of communication (respectively `send()`, `collect()` and `clean()`).

We recall that the fact that there are (or not) several Domains on each Node remains transparent for the mainstream developer. In this section, we will sort out what happens in the background in case of coalesced messages at the Node level; the process is also pictured on Figure 7.6. A strategy pattern is available to ungroup messages should the context look in favor of this mode of communications.

In order to reuse these structures and avoid code duplication, all of them are *template* structure, the template argument being the type of data exchanged (i.e. an atom). For instance, in a standard all-atom simulation, we will have on each Domain a `MessageCenter<Atom>` to exchange moving atoms, and another one to update ghost layers. The template parameter will be denoted *T* for the rest of this section.

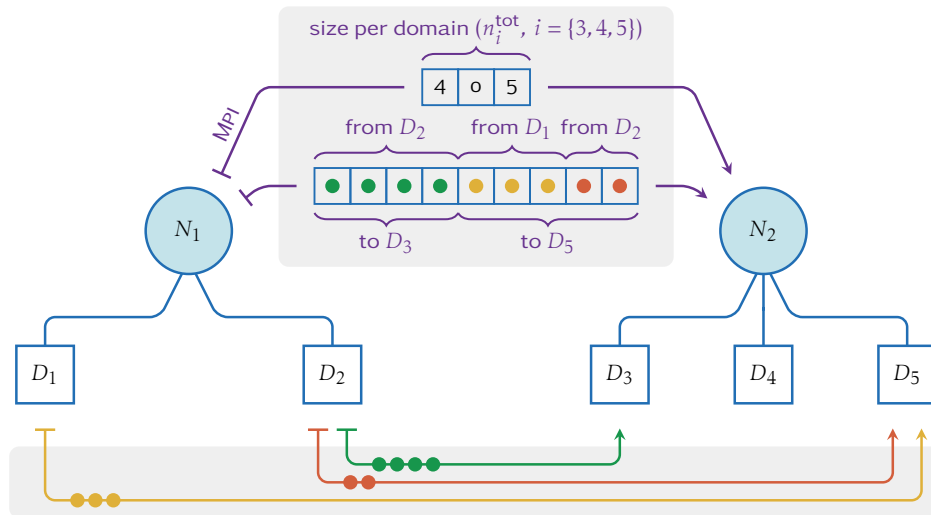


Figure 7.6 | Point-to-point Communications. In this example, domains D_1 and D_2 on node N_1 have to send data to domains D_3 and D_5 , as shown in the bottom gray frame. In a full-MPI approach, we would need to send three different messages, which is exactly what the developer sees. In the background, messages are grouped by nodes. We first have to compute the number of elements that N_1 will send per domain of N_2 , which lets N_2 allocate receive buffers of the right size and easily retrieve data for each domain. Then, the packed message is sent with non-blocking communications.

On Domain: MessageSend

On a Domain, A `MessageSend` can be viewed as a collection of lists: each list corresponds to a message to a set of elements to send to a particular Domain; is implemented as a `std::map<int,`

`std::vector<T>>`. The developer can add an element to a collection with the `push()` method; it takes a Domain index (`int`) and an object to send (`T`) as arguments so that the element to send is placed in the right collection. If the index in a `MessageSend::push()` call does not correspond to one of the Domain's neighbors, nothing is done; an exception mechanism will be implemented later.

Once the `MessageSend` has been filled, the developer uses the `send()` method from the `MessageCenter` from where he pulled the `MessageSend`. In this method, The `MessageCenter` adds the `MessageSend` to the `Session` corresponding to the exchange process (i.e. particle exchange or ghost update). Unlike the `MessageCenter` structure, which is present on each Domain, there is only one instance of a `Session` on a Node. This last action is therefore protected with locks so that only one Domain intends to modify the `Session` at the same time.

On Node: Session part I (send)

The `Session` gathers data to send from all Domains of the Node. It is likely that data for the same Node, or even the same Domain is spread on various `MessageSend` and, in some cases, should be gathered. The goal of a `Session` is to arrange all this data in order to minimize the total number of messages.

First, let us consider that we have knowledge of all our neighbor Domains, the Node to which they belong. These data are indeed on the communication manager. We will denote N_i , $i \in [1, n]$ the neighbor Nodes and $\mathcal{D}_i = \{D_j, j \in [1, d_i]\}$ the list of Domain indexes on each of these Nodes; we assume that these lists are ordered. We go through all `MessageSend` objects to get

- the total number of objects (n_i^{tot}) that we are going to send to each neighbor Node;
- the detail of message sizes for every Domain of each neighbor Node, i.e. another list $\mathcal{N}_i = \{n_j, j \in [1, d_i]\}$, such that $\sum_{j=1}^{d_i} n_j = n_i^{\text{tot}}$.

Note that we may have some n_j , or even some n_i^{tot} equal to zero, and that the neighbor Node set may contain the Node itself, to enclose the intra-Node case. We then exchange with all neighbor Nodes the $\mathcal{N}_i$ lists. This will allow us to correctly allocate receive buffers. This sending/receiving is quite simple as it consists in fixed size integers. We use non-blocking send and receive for this operation.

While communications take place, we can allocate proper buffers to “pack” data to be sent: indeed, they are still spread in `MessageSend` structures. We create n arrays of size n_i^{tot} , then fill them by going once again through all `MessageSends`. In every array, data are arranged so that contents are contiguously grouped by Domains: data for domain D_j start at index $\sum_{k=0}^j d_k$ (see also Figure 7.6).

Once send buffers have been properly filled, we delete `MessageSend` to free some memory, and call `MPI_Wait` to ensure the end of size exchange. We will denote $\mathcal{N}'_i = \{n'_j, j \in [1, d_i]\}$ data received from each neighbor Node, where n'_j corresponds to the subset size from Domain D_j . We continue by allocating, for each neighbor Node, a buffer of size $n_i'^{\text{tot}} = \sum_{j=1}^{d_i} n'_j$ to receive data.

Since send buffers are properly filled and receive ones are allocated, we are eventually ready to exchange our data. In the same way as the sizes earlier, we run through all neighbor Nodes, and use non-blocking MPI send and receive. If one of the neighbor Node rank is equal to the Node own rank, a `memcpy` replaces MPI calls.

Back On Domain

Back on the `Domain` after calling the `send()` method of the `MessageCenter` object, the developer has the possibility to perform several operations while communications are occurring. To ensure the end of these communications, he has to call the `collect()` method (still from the `MessageCenter`).

The `MessageCenter::collect()` method calls the `Session::collect()` method with an address of a `MessageReceive` as an argument. The `Session` is responsible for allocating and filling this structure with received data for every `Domain`.

On Node: Session part II (collect)

The `Session::collect()` method starts with a thread barrier to let each `Domain` complete their tasks.

We then need to count the number of elements received for each `Domain`: right now we just know the message sizes received from each neighbor `Node`, and we may have data for a single `Domain` coming from different `Nodes`. After going through size data to compute the receive size per `Domain`, we allocate memory of `MessageRecv` objects.

Next, we call the `MPI_Wait` to end communications. We go through all receive buffers and copy data in `MessageRecv` structures. As we sorted data coming from a `Node` by `Domain` index, this operation is quite easy to perform. Once `MessageRecv` objects are filled, we delete all buffers.

On Domain: Clean

Every `Domain` has now a `MessageRecv` object which contains elements that were destined to it. The developer can extract these elements using methods `size()` and `data()`, which respectively gives the number of received elements and the address of the first one. When data from `MessageRecv` has been processed, the developer can call the `MessageCenter::clean()` method to deallocate memory from `MessageRecv` structures.

8

Dynamic Load Balancing

IN this chapter, we address issues of load balancing at the domain decomposition level, which is one of the most important causes of poor performance when performing simulations on a large number of cores. We start by an example which shows the importance of a good load balancing in MD simulations, we then present existing solutions that have been made to face the problem in MD software. We continue by discussing about the design and implementation of a load balancing capability for our architecture, before showing some results.

8.1 Load Balancing in MD

8.1.1 Why It Is Important

When using domain decomposition in a MD simulation, the global simulation box is shared between domains, giving each domain an equal amount of space. This solution is perfectly fine as long as the simulated system is large and homogeneous, as the space distribution matches the particle one. However, when density differences or uneven distributions of particles arise, empty domains spend most of their time waiting for overloaded ones.

For instance, let us consider two simulations (with ExaSTAMP) of the same system of particles, with one which entirely fills the simulation box, and the other one in a box which is 25% larger in each dimension (i.e. ≈ 2 times larger total) as shown on Figure 8.1. As expected, both cases run in almost the same time on a single core (less than 2% difference), and roughly have the same performance up to 8 cores¹: indeed, cutting the domain up to two equal parts in each dimension still results in a balanced distribution. Beyond this point, at least one dimension has to be cut in four, which inevitably creates imbalance. Performance drop and barely reach a 50% efficiency on 64 cores.

In MD simulations, particles arrangement can be much worse than the example given in Figure 8.1. For instance, simulations performed by Durand et al. [41], are the perfect illustration of an unbalanced system, with an uneven distribution of particles in space and large density variation (see Figure 8.2). These simulations typically run on 4000 cores, which can only amplify the performance drop we have just observed on “just” 64 cores. If we seek to perform efficient large scale simulations, we cannot go without a proper load balancing system.

¹We used only domain-decomposition/MPI here, no threads are involved.

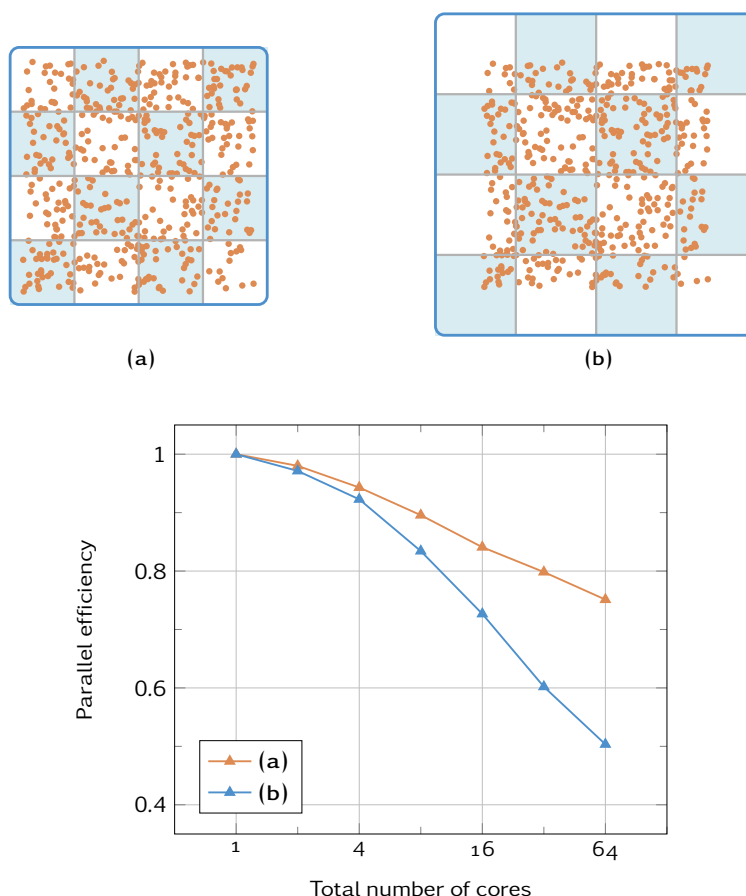


Figure 8.1 | Why load balancing is important (I). We run simulations of the exact same arrangement of particles inside two different simulation boxes. In parallel, using only domain decomposition, the second one (b) results in unbalanced domains and sees its performance drop. Simulations were performed on Airain's standard partition; they consist in 4 million Copper atoms interacting through a LJ potential for 256 iterations.

8.1.2 Load Balancing in MD codes

Most MD software dedicated to biomolecular systems come with a load balancing solution: their simulations tend to have less particles, and their distribution in space is far from homogeneous (in protein folding for instance), which exaggerates the effect of a bad distribution.

As for material-oriented MD codes, very few of them are equipped with a load balancing capability, although we saw that it was almost always necessary to perform large scale simulation. Among the codes cited in Chapter 4, only MOLOCH (though there are no information available about it) and LAMMPS claim to have one. We can also cite Fattebert et al., who proposed in [46] an original method based on Voronoi cells decomposition to perform load balanced MD simulations.

LAMMPS

In order to adjust balance across domains, LAMMPS disposes of two recently implemented (2012) functions, `balance` and `fix_balance`, which are respectively used for static and dynamic load balancing. Both functions rely on two different methods called *shift* and *RCB* (see pictures on Figure 8.3).

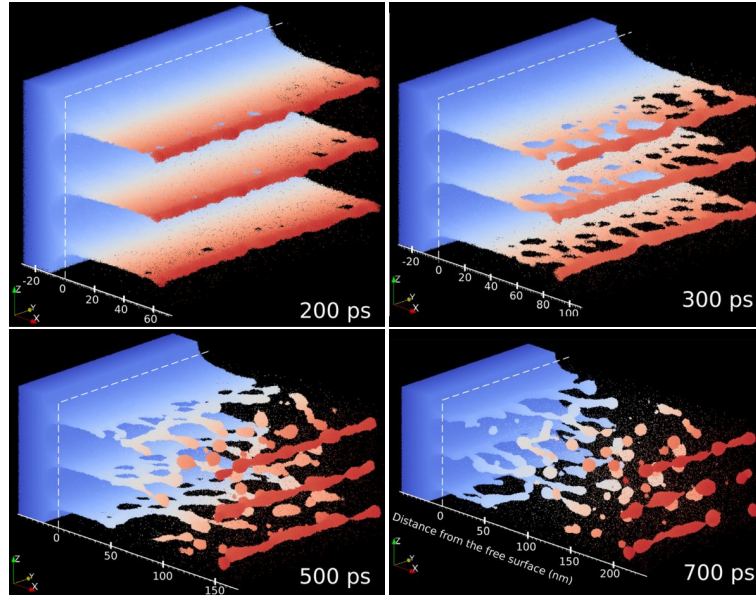


Figure 8.2 | An unbalanced simulation. From [41]. Example of simulation performed at CEA, which definitively could benefit from a load balancing capability. We may notice the evolution of the system size, which has its value multiplied by three between the first and the last picture.

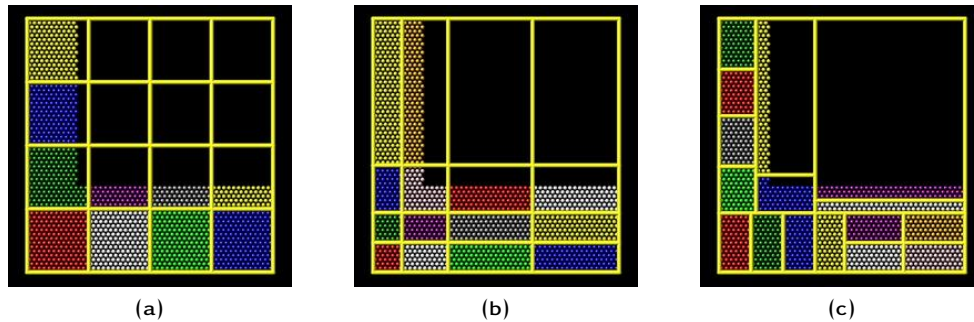


Figure 8.3 | Load balancing with LAMMPS. From http://lammps.sandia.gov/doc/fix_balance.html. (a) Default partitioning, with a 3D regular grid of processors. (b) The *shift* method still produces a grid of processors, but with different sizes. It moves the cutting planes between domains (or 2d) in order to adjust the workload. (c) The RCB method does not produce a grid of processors: domains keep their rectangular shapes, but with different size and shape so that they all have the same number of particles.

Shift. The shift method relies on what LAMMPS calls a *grid* of domains/processors: a grid is the result of the global simulation box partitioning with cutting planes which are perpendicular to the simulation box's axis. When the grid changes, domains keep their rectangular shape, but also their neighbors, which simplifies a lot communication patterns during balancing steps. The shift method balance takes in argument the dimension in which it performs the balancing. Cutting planes positions are adjusted with a density-dependent recursive multisectioning algorithm, so that the number of particles on either side of each plane is evenly distributed. If the counts do not match the target value, cut positions are adjusted based on the local density. Once the balancing is complete and final processor assigned, particles can migrate to their new owning processor as part of the normal re-neighboring procedure, since domains keep the exact same neighbors.

The density-dependent part of this algorithm makes it efficient to re-balance an already nearly balanced system, or to balance a system which has small density variations. According to LAMMPS documentation, it converges more quickly than the RCB method in these cases. However, systems

in which particle distribution changes a lot during the simulation [39] or systems with strong density fluctuations [117] cannot properly benefit from balancing with this method.

RCB. The Recursive Coordinate Bisection method (RCB) [14] was added in August 2014, which explains what there are currently no performance or feedback information about it yet. Hence we just describe the method here. The global simulation box is cut in two parts by a plane perpendicular to the axis corresponding to the global simulation box's longest dimension. The cut is made such that both sides have the same number of particles. The set of processors is also divided and each one is assigned to a side. The process is then repeated in each of the two sub-boxes created, and so on ... Simple variants exist to fit numbers of processors which are not power of two.

With the RCB method, domains keep their rectangular shape but do not form a grid anymore. This arrangement is more complex but also more general, and it should be able to correctly balance cases which are not well supported with the shift method.

8.2 Load Balancing in ExaSTAMP

In hydrodynamic codes, a common technique is to use graph partitioning methods applied to the simulation mesh (which may or may not be structured). Processor therefore exchange mesh elements to balance their workload.

Drawing on this technique, we decided to consider the cell structure used for neighbor search as a mesh to perform load balancing operations. Partitioning over cells allows us to manage particles by block, which is much more convenient for large simulations. Moreover, since the cell structure can be considered as a structured mesh or a graph, we can benefit from existing partitioners, among which we choose ZOLTAN (see next section). However, this also means that we cannot keep going with rectangular-shaped domains. This is why we designed a new domain topology specifically for load balancing, where domains can have any shape.

8.2.1 The ZOLTAN Library

ZOLTAN [18, 19, 17] is a toolkit for HPC applications which provides efficient solutions for issues such as load balancing, data movement, unstructured communications and memory usage. ZOLTAN also has graph coloring and graph ordering capabilities, which can be useful for schedulers, parallel preconditioners and linear solvers.

With its object-based software design, ZOLTAN allows a clear separation between the libraries and application data structures: instead of requiring the application to build data structures dedicated to the library, the application just needs to pass functions which allow the library to access its data. This design has various advantages. First, access functions are easy to implement, and memory usage is also lower compared to the construction of an intermediate data structure. Moreover, potential changes in the library's data structures do not propagate back to the application. The only drawback is that library calls to access function may induce some overhead. In experiments, this cost is very well contained regarding ZOLTAN internal calls.

Load balancing with ZOLTAN

As we saw with LAMMPS, there is no single partitioning strategy which is effective in all situation, even if we restrain ourselves to a specific area (i.e. MD for us). For instance, we might prefer a high-quality balance, regardless of the cost to generate it for expensive potentials when we prefer a fast balance computation and sacrifice some quality on the way on cheaper force computations. Most important, as we do not know in advance which strategy work best in every case, we need a convenient tool to compare balancing methods.

ZOLTAN's core consists in a suite of dynamic load balancing and partitioning algorithms that increase applications' parallel performance by reducing processors idle time. These algorithms are gathered in two classes:

- Geometric methods:
 - Recursive Coordinate Bisection [14];
 - Recursive Inertial Bisection [103, 114];
 - Hilbert Space-Filling Curve Partitioning [90, 122];
 - Refinement Tree Partitioning;
- and Topology-based methods:
 - Graph & Hypergraph partitioning;
 - Graph partitioning (through interfaces to ParMETIS [70], SCOTCH [89] and other famous graph partitioners).

ZOLTAN performs load balancing operations on what it calls *objects*. Each processor has a list of objects, and an associated cost for each one of them. When geometric partitioners are in use, each object is also associated with coordinates, whereas edges have to be specified for graph-based methods. When we call ZOLTAN's partition function, it returns for each processor a list of the objects that are leaving along with their recipients, and/or a list of the objects that are arriving with the respective senders. This allows an easy rebuild of the processor's data to carry on the now balanced computation.

8.2.2 Integration of a Load Balancing Capability in ExaSTAMP

Any Decomposition

ExaSTAMP's default decomposition is called *rectilinear*. Given a number of processors in each dimension, the global simulation box is decomposed such that all domains have the same number of cells (plus or minus one) in each dimension (see Listing 8.1). This decomposition results in an arrangement similar to what LAMMPS calls a (3D) *grid* of processors, which is however static in ExaSTAMP. In order to benefit from load balancing, we need a new domain topology before anything else.

In LAMMPS, the RCB method ensures that domains keep their rectangular shape. As we intend to use all ZOLTAN's capabilities, we decided to design our new domains in the most general possible way. The only common trait that our *any* domain shares with the rectilinear one is that it keeps the cell structure as a basic block: we use the cell as a ZOLTAN *object* to perform load balancing operations.

```

// For 'numItemsToShare' items to share among 'numWorkers', a worker
// identified with its 'workerId' get 'size' items starting at index
// 'start'
void split (int numItemsToShare, int numWorkers, int workerId,
           int& start, int& size) {

    int q = numItemsToShare / numWorkers;
    int r = numItemsToShare % numWorkers;

    if (workerId < r){
        start = workerId*(q+1);
        size = q + 1;
    }
    else{
        start = workerId*q + r;
        size = q;
    }
}

```

Listing 8.1 | Split function used in ExaSTAMP rectilinear decomposition

If we picture the global domain with the cell grid structure over it, a rectilinear domain can be identified with six integers: a 3D coordinate for its origin and another one for its size. This is not the case for *any* domains, which are identified by any set of cells. In particular, they do not even have to be connected (in the topological sense).

Architecture Modifications

Classes `Domain` and `Grid` are not at all impacted by the introduction of an alternative topology for domains. Indeed, this was predictable since we saw in Chapter 6 that `Domain` specifies elementary services to the `Grid` from functions defined in its interface. Moreover, the `Grid` is eventually a base class and is only instantiated through its child, `SingleSpecGrid`.

The `SingleSpecGrid` object is eventually the only one which has to go through some modifications. We separated all the topology-related methods into a new object called `Topology`. This base class can be instantiated either as a `RectilinearTopology` or as an `AnyTopology`, and is integrated as one of `SingleSpecGrid`'s attributes. Figure 8.4 represents the final stage of `Grid` design after the addition of the `Topology` concept (previous version on Figure 6.10 – variant (b)), and a list of `Topology`'s methods is available in Table 8.1.

Avoiding Code Duplication

By going from a very particular type of domains to the most general shapes possible, we might have expected major changes. In the design, we just modified the `SingleSpecGrid` class by extracting the topology-related content in a dedicated object. In particular, all our algorithms based on the `Traversal` object have been left unchanged, since the concept can be adapted to other shapes than the rectilinear one. As we kept the cell structure, we did not touch the `Cell` structure as well.

These elements allow us to validate the overall design in a way, since we managed to keep our modules separated from each others. A strong modification in one of them has only a minor impact on the rest of the code.

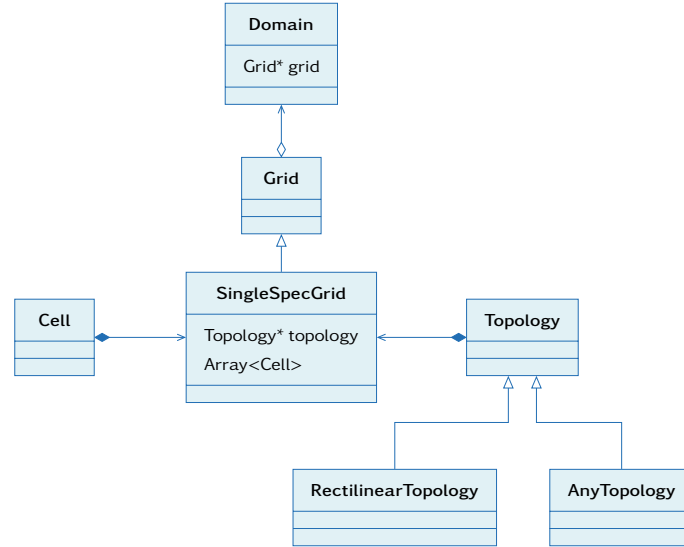


Figure 8.4 | Integration of the Topology concept.

Method	Description
<code>numberOfCells</code>	access number of real cells
<code>totalNumberOfCells</code>	access total number of cells (ghost included)
<code>ghostThickness</code>	access ghost layer thickness (δ)
<code>limitInf/limitSup</code> <code>cellInf/cellSup</code>	access the minimum/maximum coordinates of the domain
<code>coords</code>	given a cell index, return the corresponding coordinates
<code>index</code>	given cell coordinates, return the corresponding index
<code>neighborDomain</code>	given a cell coordinates c and a direction d (+1/0/-1 in each dimension), returns the index of the domain which owns the cell of coordinates $c + d$
<code>inGrid</code>	given a cell or a particle coordinates, returns a boolean telling whether they belong to the domain

Table 8.1 | Topology Methods.

Cost Function

Now that we have ZOLTAN-compatible domains, we must provide a cost function. A common technique is to use timers to measure the cost of a ZOLTAN object, i.e. a `Cell` for us. This is the case in [46], where the load on each processor is estimated by the wall-clock time minus the idle time. The use of time measurements avoids having to build a workload model, but it raises issues in term of accuracy and reproducibility. Indeed, timers can easily be perturbed by external effects on production machines: for instance, if a cluster node is shared between two applications, they may hinder each other by using too much bandwidth or performing I/O. Moreover, as time measurements never are the same twice in a row, there is no chance that the several runs of the same simulation produce identical distributions. This is a major issue for debugging and applications which require reproducibility and a high level of accuracy.

Another method consists in balancing the number of particles among domains, which is what is performed in LAMMPS. This method works well as long as there is only one type of particle

involved in the simulation: two or more types imply three or more potentials (there are $n(n+1)/2$ potentials for n types); if some potentials are much more expensive than others, a perfectly balanced partition (in number of particles) can absolutely be unbalanced in practice.

This is why we decided to use the following workload model as a cost function, which gives reproducible simulation and balance multi-materials simulations. In order to provide the maximum flexibility in our architecture, we nonetheless let any developer the possibility to implement its own cost function at the cell level and use it in ZOLTAN.

Let us consider a volume V with N particles of t different types ($t < N$), such that each type of particle is evenly distributed across V . Interactions between particles of types i and j are computed with short range potentials with a cutoff radius of r_{cut}^{ij} , and we denote N_i the total number of particles of type i (hence we have $\sum_{i=1}^n N_i = N$). We can compute $n_j(i)$ the number of particles of type j in the neighborhood of any particle of type i by multiplying their average density with the volume occupied by the sphere of radius r_{cut}^{ij} :

$$n_j(i) = \frac{N_j}{V} \left[\frac{4}{3} \pi \left(r_{\text{cut}}^{ij} \right)^3 \right]. \quad (8.1)$$

From this we deduct the total number of i - j interactions $N_i n_j(i)$; if we also associate a cost W_{ij} to each potential, the total cost of the force computations in the volume V is given by

$$W = \frac{4\pi}{3V} \sum_{i=1}^t \sum_{j=1}^i N_i N_j \left(r_{\text{cut}}^{ij} \right)^3 W_{ij}. \quad (8.2)$$

This is the expression (without the prefactor $4\pi/3V$) that we use as our cost function. Although the assumption about evenly distributed particles may not be 100% accurate, this cost function contains significant information compared to a balance over the number of particles alone. We prove it by performing a comparison of both methods in Section 8.3.

Trigger a Load Balancing Step

When using the Any decomposition, balance operations are enabled periodically by default. We also anticipate a mode where balance operations are triggered automatically when the global imbalance goes beyond a certain threshold (which is a parameter defined by the user). The expression of the global imbalance $\mathcal{I}$ is given by

$$\mathcal{I} = \frac{\max(w_i) - \bar{w}}{\bar{w}}, \quad (8.3)$$

where w_i the workload on every domain (computed by adding the workload on each cell), and $\bar{w}$ the average workload. For instance, a global imbalance of 10% means that the most loaded domain is 1.1 times more loaded than the optimum load.

8.2.3 Implementation

Any Topology

The implementation *any* topology consists in a list of cells and a neighboring table. More precisely, the cell list is a list of all cells' global IDs, and the neighboring table is an array of the same size of the list: each element is either an array giving neighbor domains in each direction if the cell is on

the domain's edge, or a null pointer if the cell has only neighbor cells on the same domain. This lets the `Grid` know the set of domains to communicate with, for both particle exchange and ghost update processes. Besides, we can point out the fact that the cell list is sorted in order to perform fast search operations (using bisection method) in it.

ZOLTAN Query Functions

The ZOLTAN library requires four access functions to call the geometric-based methods (which are the most adapted for particle simulations according to the documentation), and two more to use graph and hypergraph partitioning. They give access to (1) the total number of dimensions of the problem, (2) the number of objects' (i.e. cells) per domain, (3) the objects coordinates and (4) cost, using the cost function explained in the previous section. To use graph methods, we also have to provide (5) a function giving the number of edges per object, and (6) another one specifying edges and their cost.

Grid Rebuilding After a Balancing

Because of its complexity, the `AnyTopology` and the following algorithm have only been implemented for a single ghost layer. Since we only run simulations with symmetrization enabled, this does not currently limit us. A more general approach will be carried out in the future.

ZOLTAN Partition Function. The balancing step takes place right after a full step of the numerical scheme in use. From the `Domain` class, we call the `Zoltan_LB_Partition` function to get a new balanced partition. This function, whose visual representation is pictured on Figure 8.5, allocates and initializes the following arguments:

- a boolean which indicates whether or not the partitioning changed;
- an array containing the cells' global IDs that have to be sent to the current domain;
- an array which holds the domains' indexes from where these cells arrive;
- an array with the global IDs of the cells that leave the current domain;
- and an array containing the domains' indexes where the leaving cells must be sent.

The goal of this algorithm is to retrieve all necessary data to update the `AnyTopology`, which consists in the list of cells and the neighbor table. If the first part is quite straightforward considering what provides ZOLTAN, rebuilding the neighbor table can be a delicate process. Indeed, there is no direct way to know whether or not one of the current domain's ghost cells has been moved. Moreover, we do not have any information about incoming cells' neighbors.

Save Particles. We are not going to modify the `Grid` object. Instead, it will be deleted and replaced by a new one. The first thing to do is to separate leaving and staying cells on the current domain. We create an array to store staying particles, and a message structure to send the others, then we go through our cells to fill these structures. Once this step is complete, we send leaving particles and delete the cell's content in the grid to save memory. Messages will be retrieved at the very end of the process.

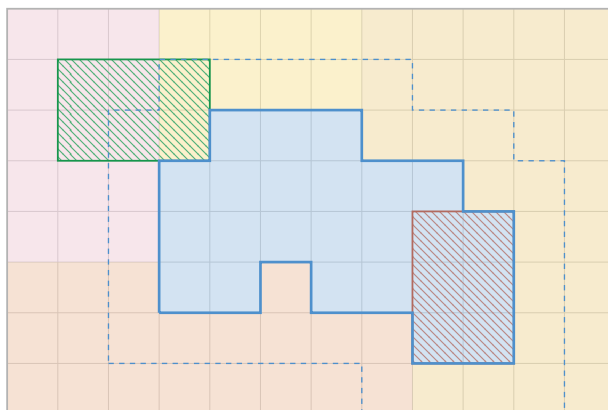


Figure 8.5 | Overview of a domain after calling ZOLTAN partition function. The current domain (blue zone) with its ghost layer (dotted blue line) and neighbor domains. Incoming and leaving cells are respectively represented with the green and red hatched zones.

Update Owners. From here, a cell *owner* designates the index of the domain that owns this cell. This step consists in identifying the future owner of both real and ghost cells of the current domain. In other words, we must find for all cells within the dotted blue line in Figure 8.5, their owner for the next step.

This is quite obvious for the real cells: if they are not in the list given by ZOLTAN, it means that they are staying in the domain; in the other case, we use the list of domain indexes provided by ZOLTAN to identify the new owners. For ghost cells, there is no straight way to get the new owners. Since ghost cells are edge cells of the neighbor domains, we conduct communications similarly to the ghost update process: for every neighbor domain, we fill a message with the new owner of each edge cell. Messages are then sent, and data from communication are processed right after.

From this point, half of the work is done: we are able to recover the neighbor table for each cell which are not incoming cells.

Exchange Moving Cells Environment. At the end of last step, we are in particular able to recover the neighbor table for all leaving cells, since they are obviously still part of the domain. We therefore send for each leaving cell an array containing all its neighbors' future owners. After receiving and processing content for incoming particles, we are eventually ready to rebuild the grid.

Reintegrate Particles. Once the grid has been rebuilt, we add saved particles from the first step, and eventually end the communication of particles initiated at the first step. These particles are also added to the grid, meaning that everything is ready to resume the computation.

8.2.4 Conclusion

We presented a new domain topology which allows the use of a load balancing capability. Partitionnement is performed by the ZOLTAN library, which offers various methods to share the work between domains, and can also be interfaced to use other well-known partitioners.

The algorithm used to rebuild a Grid and its AnyTopology from the new partition generated by mainly relied on already existing elements of the architectures: for instance, we did not write a single MPI call, since all communications have been performed by our dedicated structures.

Besides, we can point out that this reconstruction does not involve any collective communications, which is always a good point for performance.

The introduction of a new topology in ExaSTAMP has been made without changing a single line of code in low-level classes such as the `Cells`, which contain particles. Algorithms on the `Grid` have not been modified either, since they rely on the `Traversal` structure which is independent from the topology. This made the validation of the topology a lot easier, but mostly allowed us to approve our architecture in terms of flexibility.

8.3 Evaluation

This section goes on several simple tests to have an idea about what our load balancing feature can bring. After applying load balancing to our first example, we show the relevance of our cost function. We then perform a comparison of ZOLTAN main methods on a test case, in both static and dynamic partitioning. Applications on a real production simulation will be performed in Chapter 10.

8.3.1 First Tests

Back On First Example

In order to realize what load-balancing can offer, we resumed the example in Section 8.1.1. If it did not meet the speedup of the perfectly balanced case (variant (a) in Figure 8.1), we managed to straighten up the dropping acceleration with only a single partitioning (using the RCB method) at the beginning of the simulation (see Figure 8.6).

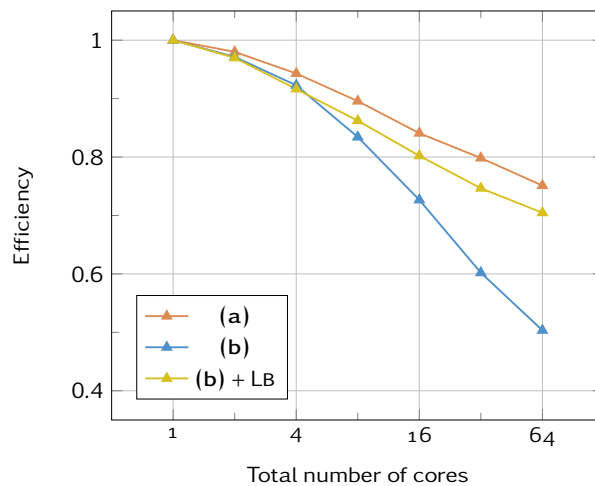


Figure 8.6 | Why load balancing is important (II). If we resume the example in Figure 8.1 with load balancing enabled, we obtain a much better speedup (yellow curve), with a speedup much closer to the perfectly balanced case.

Why our Cost Function is Better than Counting Particles

In this paragraph, we show an example which demonstrates the use of our cost function presented in equation (8.2). Let us consider a cube of copper atoms interacting via a Lennard-Jones potential, except for a small part of it where atoms interact with a Sutton-Chen (EAM) potential as shown in Figure 8.7 (a). In this figure, the blue part corresponds to the LJ potential, the green one is for the

SC potential, and both zones do not interact (ideal gas). The green cube's edge is half the size of the blue one. Therefore, it represents 12.5% of the number of atoms (same density everywhere), but about 30% of the total workload if we consider that the Sutton-Chen potential is three times more expensive than the LJ one.

We ran simulations of this cube on eight domains with both default (rectilinear domains) and any decompositions. For any decomposition, we performed a single partition at the initialization step using the RCB method either with our cost function or with a balance over the number of particles. Figure 8.7 shows resulting decompositions: we observe that with the default partition (b), the whole compute-expensive zone is carried on by a single domain; with load balancing on the number of particles (c), the global domain is split in eight equal parts but once again only two of them are taking care of the Sutton-Chen region. The version with our cost function (d) eventually brings a satisfying solution. If the expensive zone is unequally divided between domains (90% of it is held by four of them), they have different sizes such that domains with cheaper computations have more atoms.

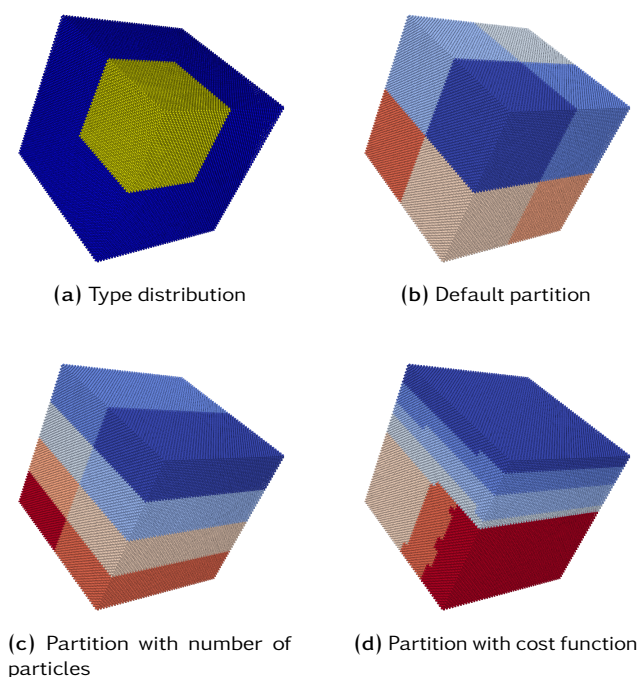


Figure 8.7 | Evaluation of our cost function against a balance on the number of particles. In (a), the green zone correspond to atoms interacting with a SC potential, which is 3–4 times more expensive than the LJ one operating in the blue zone. Both default rectilinear (b) and any with simple balancing (c) partitions are unsuitable to the situation. When balancing is applied with a proper cost function (d), simulations are about 30% faster (500,000 atoms, 8 cores on Airain's standard partition).

Simulations results confirmed what we saw in pictures (of Figure 8.7: on Airain's standard processors and a 500,000 atom system, the version with our cost function is 29% faster than both other versions that ran in the same time (less that 1% difference). This shows that the efficiency of load-balancing is highly dependent on heuristics in use, and that the choice of a good heuristic may depend on the simulated system: for instance, our cost function is perfectly adapted for multi-materials simulation.

ZOLTAN Methods Overview

Among algorithms proposed by ZOLTAN, we focused in particular on four methods: the Recursive Coordinate Bisection (RCB), the Recursive Inertial Bisection (RIB), Hilbert Space-Filling Curve Partitioning (SFC), and Graph partitioning.

The first three methods are *geometric* methods, which means that the partition is based on objects' (i.e. cells in our case) coordinates: object that are physically close are assigned to the same domain. Geometric methods do not have an explicit control of data redistribution costs, but they implicitly achieve low migration costs: for small changes in data, domains' boundaries move only slightly, resulting in little data redistribution. The graph method represents the problem as a weighted graph. In that case, the vertices are the object to be partitioned, and the edges represent dependencies between objects. The partition is made such that different parts have equal vertex weight and that the weight of edges cut by part boundaries remains small.

According to ZOLTAN documentation, geometric methods are easier to use, faster and less expensive than graph-based methods, which however tend to produce better partitions, and have a control on communications through weighted edges. In practice, geometric methods work better on particle simulation, when graph ones are more suited for mesh based simulation, and especially irregular, unstructured ones.

Figure 8.8 represents different partitions obtained with different methods of a million-atom system on eight domains. The atom cube is included in a larger simulation box such that atoms occupy about half of the total volume.

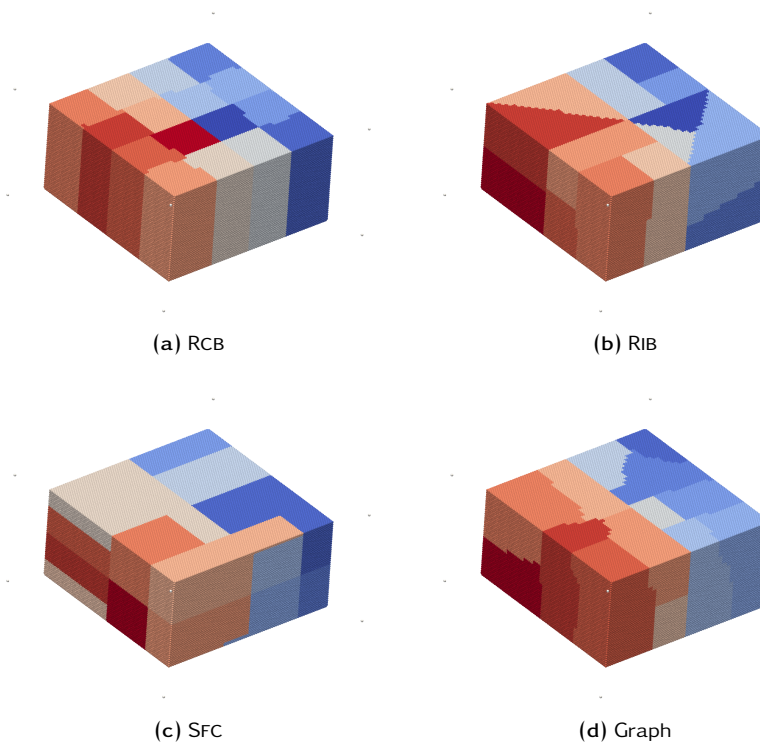


Figure 8.8 | Overview of ZOLTAN methods.

RCB (a). With the RCB algorithm, the global simulation box is first divided into two domains by a cutting plane orthogonal to one of the coordinate axis so that half the work load is in each domains. The splitting direction is determined by computing the direction where the set of objects is most spread out. As a recursive algorithm, domains are then further divided by using the same algorithm until the number of domains equals the number of processors. Although this algorithm was first designed to cut into a number of sets which is a power of two, the set's sizes in a particular cut do not need to be equal, since part sizes can be properly adjusted.

RIB (b). The RIB method is very similar to the RCB one. Indeed, the only difference is that the cutting planes used in the bisection algorithm are not necessarily orthogonal to one of the coordinate axis,

SFC (c). The Hilbert Space-Filling curves are functions that map the $[0,1]$ interval into coordinates (in 1, 2 or 3 dimensions). The SFC partitioning algorithm seeks to divide $[0,1]$ into P intervals (each interval being associated to a domain) such that all intervals contain the same weight of objects, which are associated to these intervals by their Hilbert coordinates. The algorithm creates N bins ($N > P$) to partition $[0,1]$, and weights in each bin are summed across all processors. A greedy algorithm then sums the bins, placing a cut whenever the desired weight for current part interval is reached. This process is repeated as long as the imbalance tolerance is not achieved. If the number of bin remains the same, these which previously contained a cut are refined.

Graph (d). Hypergraph partitioning is a useful partitioning and load balancing method when connectivity data is available. It can be viewed as a more sophisticated alternative to the traditional graph partitioning, where hyperedges (hypergraph consists of vertices and hyperedges) are formed by a vertex and all its neighbors. This angle is well suited for parallel computing, since an hyperedge represents the communications requirements. In ZOLTAN's graph/hypergraph partitioner, we can assign a weight to both vertices and hyperedges.

8.3.2 Static Partitioning

In order to compare ZOLTAN's methods, we ran a larger version of the case in Figure 8.8 with about 18 million atoms on 200 domains of Airain's ivybridge partition. The block of particles does not move during the simulation, and only two balancing operations are performed (one at the initialization, and another at mid-run). For different partitioning methods, we measured the average imbalance rate (defined at equation (8.3)) and the efficient time (time per atom per iteration per step per thread). Results are reported in Figure 8.9.

We observe that geometric methods are doing much better than the graph method with a perfect balance: the imbalance rate is below 10^{-3} (meaning that the most loaded domain has only 0.1% more load than the average), when the graph algorithm has an imbalance rate around 5%. The trend is similar at the performance level: runs using geometric methods have similar efficient times, and are about three times faster than the simulation with the graph/hypergraph algorithm. If we focus on geometric methods, the RCB is used in the fastest run, followed by the RIB (1% slower) and then by the SFC (15% slower).

We then performed a second experiment with the same test case and using the best balancing method (RCB) from the first one. We also kept the same processors and the same number of cores,

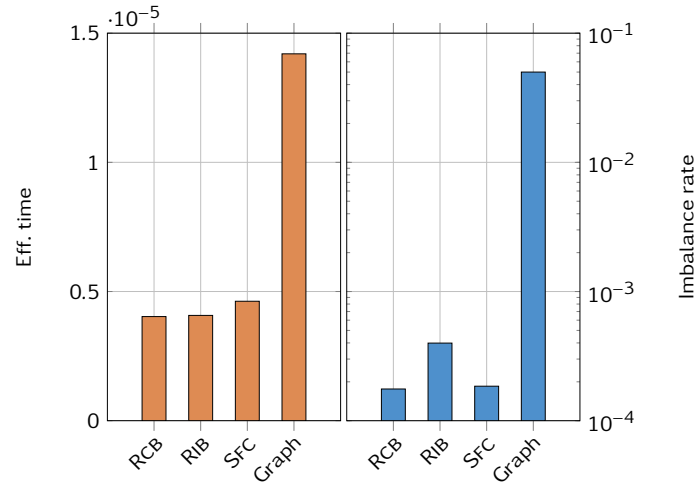


Figure 8.9 | Comparison of ZOLTAN methods (static).

but we changed the number of threads per domain, to see if larger domains could produce a better balancing and/or better performance.

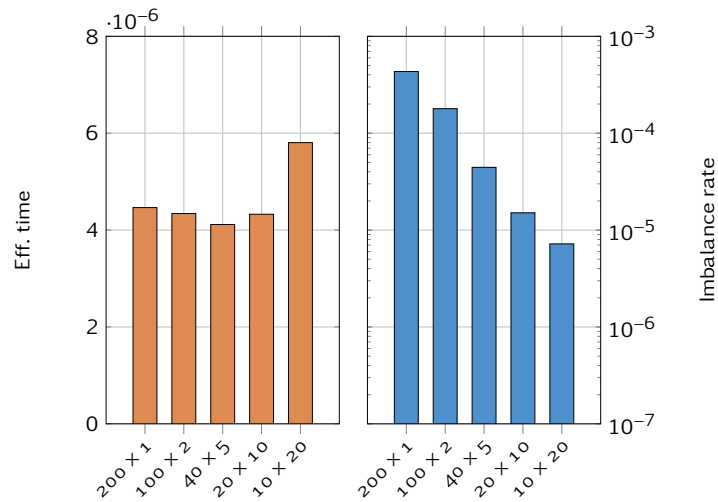


Figure 8.10 | Evaluation of RCB method for different combinations of MPI/TBB (static).

As expected, it is easier to balance a few, large domains than a lot of small ones: in Figure 8.10, we can see that the imbalance rate is reduced every time the number of domains decrease. At the performance level, there is only a slight difference, and as soon as we have one MPI for a full machine node, performance drops due to NUMA effects. In fact, the fastest simulation happens for 40 domains and 20 threads per domains, which corresponds to two MPI processes per socket.

8.3.3 Dynamic Partitioning

In this section, we present experiments using the same system used in static experiments (18 million atoms in a block covering 50% of the global simulation box), except we gave an impulsion to the atom block such that it crosses the global domain during the simulation (using periodic boundary conditions). Once again, we compared ZOLTAN methods, and tried different combinations of MPI/threads with the best one.

The choice of a good balancing frequency is very important in simulations where the particle distribution evolves quickly: with too much balance steps, we end up wasting too much time in domain reorganization instead of computing the actual simulation; but if we do not balance enough, particles move and the actual partition becomes unbalanced, resulting in poor performance. In this case, we improved the computation time from 20 to 50% depending on the balancing method by setting the balancing rate every 100 iterations instead of 500. At this point, time spent in balance operations (i.e. ZOLTAN calls and data migration) is contained to 5% of the total time. In this case, a balance every 100 iteration is the maximum: beyond this rate, time won in a better balance is lost in migration operations.

When comparing the average efficient time of different methods (in Figure 8.11), most of the results are similar to the static case. Geometric methods are still much better than the graph one (which is not represented), with a much better imbalance rate and by being more than three times faster. If we focus on geometric methods, the RCB is again the fastest, followed by SFC (14% slower) and by RIB (13%) slower. Regarding imbalance rates, we observe the same shape than the static case, but with much higher values. Indeed, this case is very unfavorable for a good balance, with the block of particles moving quite fast through the domain. This stress case, which has been established to test our load balancing capability, would not happen in a “real” simulation. We observe nonetheless on Figure 8.12 that right after a balance (every 100 iterations), the imbalance rate is between 10^{-4} and 10^{-3} , which is exactly the value we had in the static case. Besides, this figure attests well the correlation between imbalance rate and efficient time, proving again the necessity of a load balancing in such conditions.

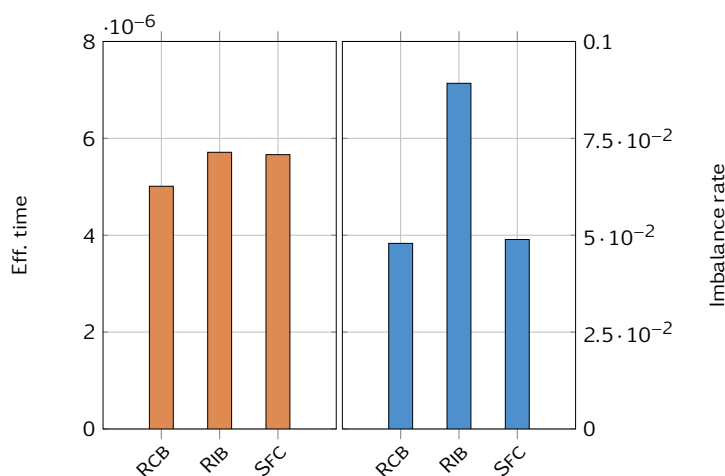


Figure 8.11 | Comparison of ZOLTAN methods (dynamic).

With different combinations of MPI and TBB (in Figure 8.13), results are again similar in shape to the static case. We report a better imbalance rate for runs with less, bigger domains, but here a better imbalance rate does not necessary means a faster simulation, the optimum case being 40 domains with 5 threads per domain.

8.3.4 Conclusion

In this section, we saw that load balancing was an essential capability to obtain a certain level of performance as soon as the particle distribution over the global simulation box becomes irregular. We then showed that a good cost function was necessary in cases where several potentials were

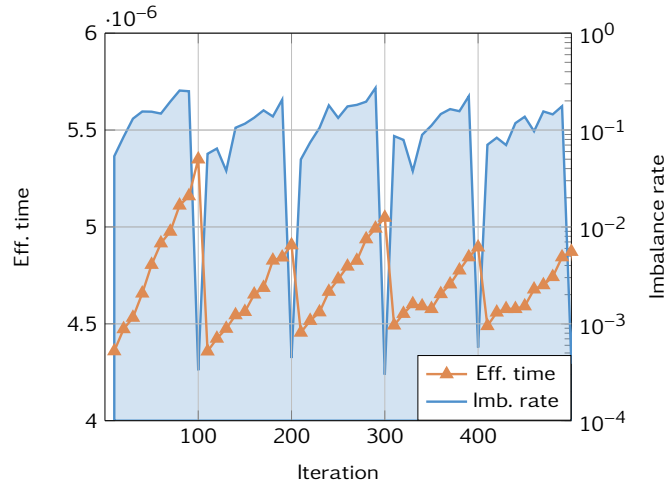


Figure 8.12 | Evolution of efficient time and imbalance rate (dynamic). Case with RCB, 200 MPI $\times$ 1 TBB, zoom on first 500 iterations.

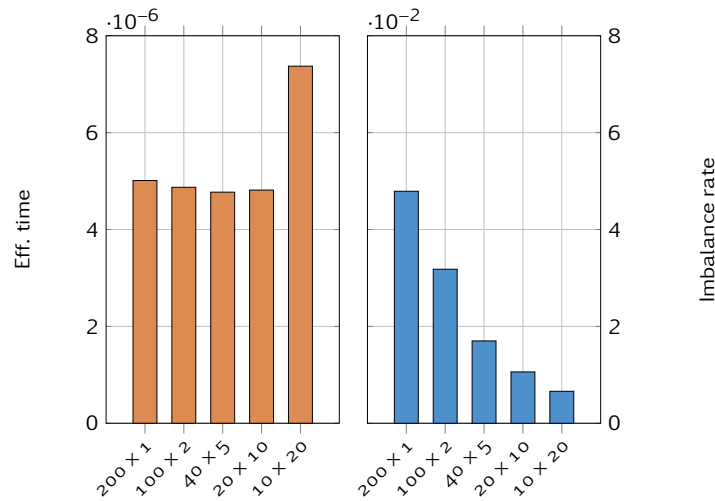


Figure 8.13 | Evaluation of RCB method for different combinations of MPI/TBB (dynamic).

involved. When we compared ZOLTAN algorithms on two test cases, we saw that geometric methods produced much more balanced partitions than the graph/hypergraph method, as specified in ZOLTAN's documentation (though no quantitative information was provided). A focus on geometric methods showed that the average imbalance was not necessarily a strong indicator about performance (RIB on Figure 8.9), but rather gave an estimation of a method's general behavior. Among geometric algorithms, the Recursive Coordinate Bisection is the one which gives the best performance. Our experiments eventually confirmed that the use of shared memory could significantly improve the quality of a partition (few, big domains are better than a lot of small ones). However, the maximum performance has been obtained with 2 MPI processes per socket (the case with one MPI per socket being very close behind – only 1% slower in dynamic case), showing that thread parallelization could be improved for any decomposition.

PART III

Validation

9

Overall Performance

IN Chapter 7, we showed performance of ExaSTAMP on a single core, and the impact of vectorization in particular. This chapter presents an evaluation of our framework in parallel. We first measure the scalability offered by TBB on different processors, then process to weak-scaling tests on a much larger number of cores, before comparing our framework performance against CoMD and STAMP codes.

9.1 On a Single Compute Node

9.1.1 Strong Scaling with TBB

In order to evaluate the scalability of our thread parallelization and to compare MPI-only against hybrid MPI/TBB versions, we conducted strong scaling tests in which we ran simulations of the same perfect crystal of 8 million copper atoms for 500 iterations. We performed these tests on three different architectures (nehalem, haswell and KNC on Cirrus cluster), with both Lennard-Jones and Sutton-Chen interactions, in order to have an evaluation of a cheap potential and a more expensive one. Speedups on nehalem, haswell and KNC are respectively drawn on figures 9.1, 9.2 and 9.2.

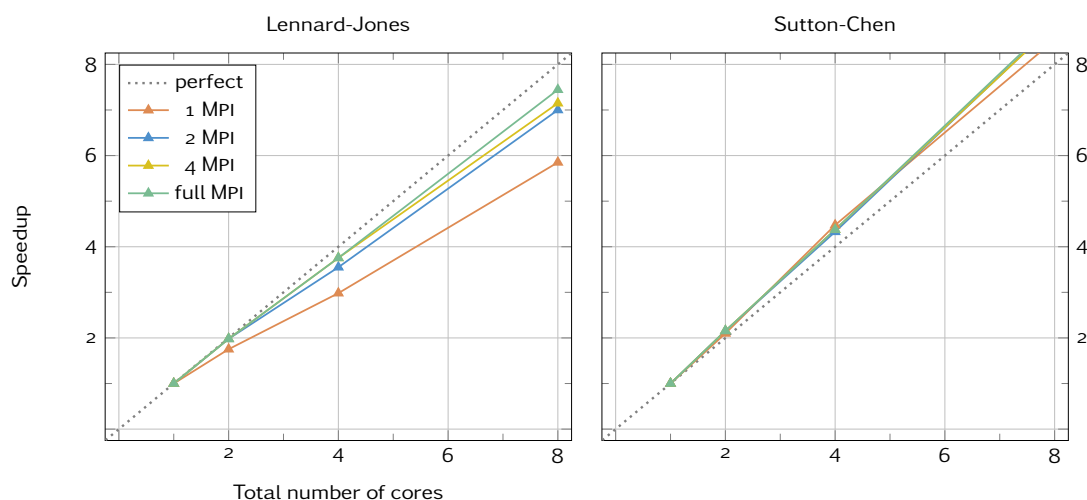


Figure 9.1 | Thread scaling on nehalem.

Performance on nehalem (which is the least recent of processor tested) is not at all in favor of shared memory: if the full-MPI run gets a decent acceleration with an efficiency of 93% in the LJ case, those using threads are systematically slower. In particular, the speedup obtained in the full-TBB case is surprisingly low since we only run on eight cores! Actually, the LJ is a very “cheap” potential, with a very low ratio of computational costs over memory accesses, which makes TBB’s work stealing strategy less efficient. This could partly explain why efforts towards the use of thread based parallelization has been limited until recently. With a much more expensive potential, the acceleration obtained in the full-TBB case is much more closer to the one from the full-MPI, but the use of shared memory is not worth it compared to MPI-only parallelization.

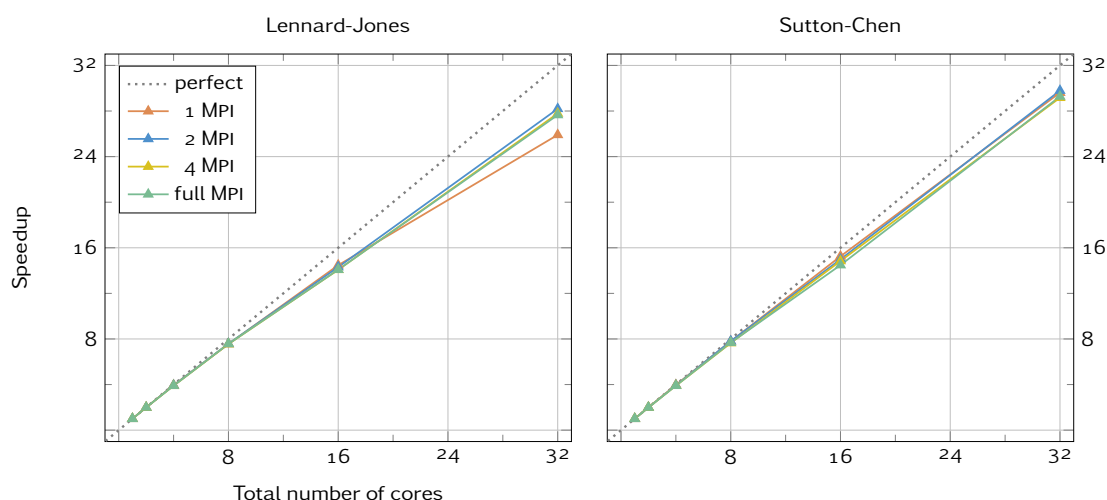


Figure 9.2 | Thread scaling on haswell.

On haswell, results are much closer to what we expected, with a very good overall speedup, which is slightly better with the SC potential (with a parallel efficiency of 93% on 32 cores) than the LJ case (88%). Up to 16 cores, the run with full thread parallelization has the best acceleration in both cases. On 32 cores, NUMA effects are present and it is faster to use one MPI process per socket. The overall difference between full shared-memory, hybrid version and MPI-only remains low.

By looking at plots on Figure 9.3, we immediately realize that the Intel® Xeon Phi™ does not use the same core arrangement as “classic” processors. On this architecture, we must use shared memory to get some performance. We obtain an almost linear speedup on 60 cores with both Lennard-Jones (efficiency of 95%) and Sutton-Chen potentials (98%).

9.1.2 Overall Atom Throughput

On Figure 9.4, we gathered the best case for each figure from the previous paragraph in order to compare raw performance of each processor. We extended our plots to test hyperthreading on haswell and KNC processors.

We observe that performance on the nehalem is very close to the one obtained on a haswell for the LJ potential; this can be explained by the fact that the nehalem has a higher base clock rate, and that the vectorization has a limited effect on cheap potentials. With the Sutton-Chen potential, the haswell is much faster (1.7× faster on the same number of cores).

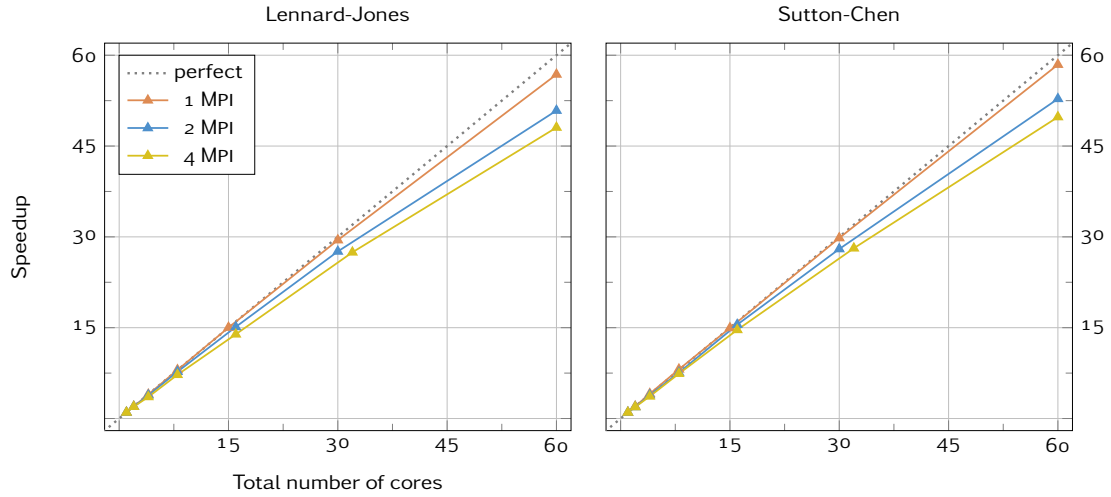


Figure 9.3 | Thread scaling on KNC.

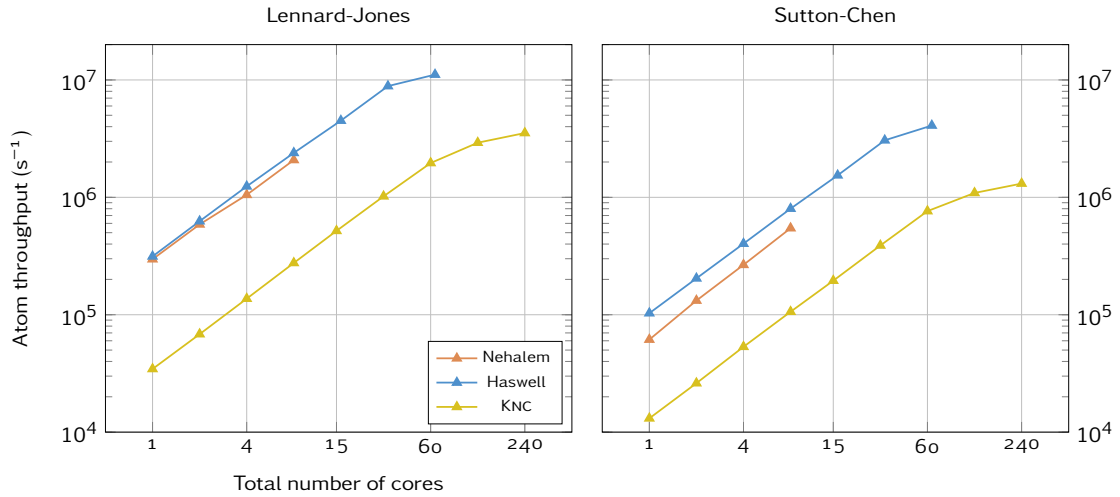


Figure 9.4 | Overall atom throughput. The atom throughput corresponds to the equivalent number of atom that could have been processed per iteration if they lasted a second.

Since the KNC requires hyperthreading to express its full performance, it would be unfair to compare it with the haswell at the same number of cores. At full power (i.e. 64 threads for the haswell and 240 on the KNC), the haswell is three times faster than the KNC with both LJ and SC potential. This gap between haswell and KNC architectures could be reduced with better performance of hyperthreading on the second one, where we could expect a linear speedup up to at least 120 threads. This illustrates well the difficulty of extracting performance on Intel® Xeon Phi™ architectures, whose peak performance is supposed to be twice larger than the haswell's one.

In this context, the development of OpenCL kernels conducted at INRIA Bordeaux in order to perform GPU computation can be adapted when the KNC is considered as an accelerator. Preliminary results show much better performance with a throughput of 3.5×10^7 atoms per second in the LJ case [23], but several adjustments such as atom migration between domains and a complete ghost layer update remain to be implemented for production simulations.

9.2 On a Large Number of Cores

9.2.1 Weak Scaling

Once we validated performance on a single node and showed the use of shared memory (at least on recent processors), we had to check the code scalability on a larger number of cores. In that case, we conducted weak scaling, with the same simulations as above, except that we adapted the system size in order to have 4.4×10^5 atoms per core and used exclusively the Sutton-Chen potential. We also ran simulation for a longer time (2000 iterations) in order to get more precise measurements. Simulations were conducted on haswell (Cirrus), ivybridge (Airain) and nehalem (TERA) processors, and results are respectively reported in figures 9.5, 9.6 and 9.7.

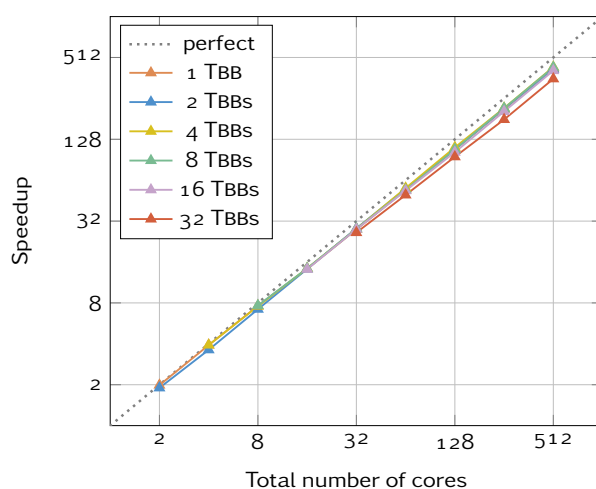


Figure 9.5 | Weak scaling on haswell.

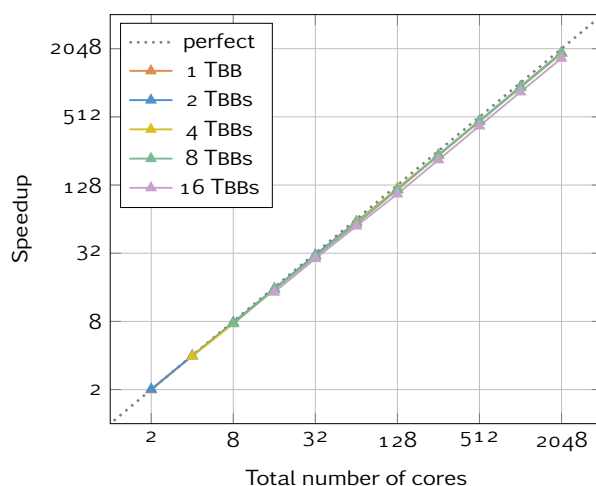


Figure 9.6 | Weak scaling on ivybridge.

All curves show similar results, with very good scalability. If simulations using less than one MPI process per socket are always one step below the others due to NUMA effects, differences between full-MPI and hybrid executions are close to zero.

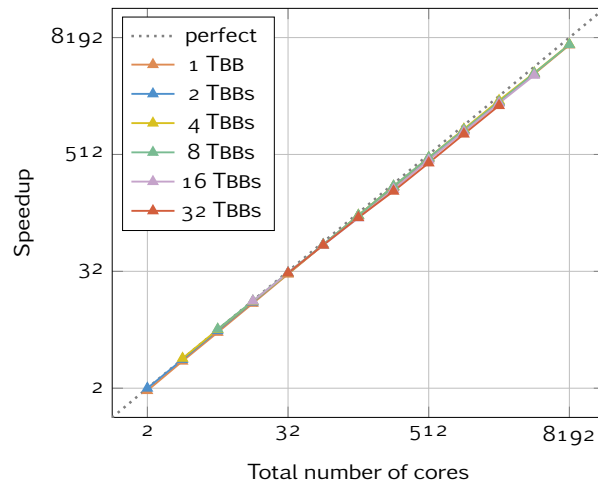


Figure 9.7 | Weak scaling on nehalem.

9.2.2 Benefit of Threads on Memory Usage

Strictly speaking in terms of performance, there is no significant advantage in the use of shared memory over full-MPI for simulations on a large number of cores. This becomes completely false when we consider memory usage over the simulation, since a decomposition running full-MPI has to store much more ghost cells for instance. In order to quantify these differences, we measured for each run of Figure 9.7 the maximum memory usage during the simulation and reported it in Figure 9.8.

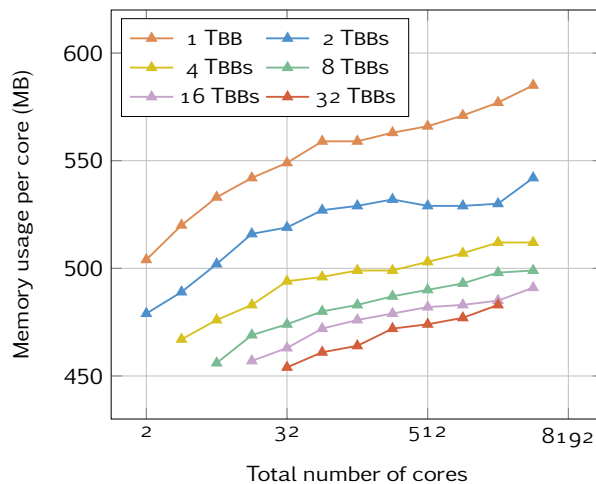


Figure 9.8 | Memory usage.

We observe that whatever the system size, simulations with one MPI process per compute node could save about 100 MB per core compared to full-MPI simulations. This makes full-MPI simulation about 20% more expensive in memory, and cannot be ignored when we see the evolution of available memory per core in the past few years.

9.3 Comparison with other MD Software

The comparison of different MD software is a delicate process since small variations in some parameters (e.g. particle density or a potential cutoff radius) can have a huge impact on performance. Since our framework shares a checkpoint/restart system with STAMP, it has been easy to conduct the exact same simulation with both codes. We eventually managed to get this simulation ready for CoMD, but neither for LAMMPS nor for MiniMD. The test-case consists in the simulation of a perfect crystal of four million atoms for 500 iterations. We ran this case with both Lennard-Jones and Sutton-Chen potentials¹ on a haswell processor. In parallel, we ran CoMD and ExaSTAMP using only shared memory, and STAMP with MPI. After having checked that all three codes produced the same results, we reported efficient-time numbers in Table 9.1.

Potential	Num. cores	Eff. Time (10^{-6} s)		
		CoMD	STAMP	ExaSTAMP
Lennard-Jones	1	2.14	3.20	2.73
	8	2.93	3.80	2.96
	16	3.47	4.32	3.15
	32	4.25	4.49	3.73
Sutton-Chen	1	30.96	35.33	10.13
	8	31.12	35.79	10.55
	16	32.51	36.24	10.92
	32	32.89	36.97	11.29

Table 9.1 | Comparison of CoMD and STAMP against ExaSTAMP.

The comparison of STAMP and ExaSTAMP performance enhances the importance of vectorization and thread parallelization: our framework gets better performance on a single core and a better scaling with the LJ potential. When comparing CoMD against ExaSTAMP, we observe that CoMD is faster to run the LJ potential on one core: we pay here the price of our neighbor lists when CoMD computes the force during the neighbor search. ExaSTAMP however scales better, so that it gets the same performance on 8 cores, and overtakes CoMD on 16 and 32 cores. We can notice in particular that CoMD suffers from NUMA effects even more than ExaSTAMP on 32 cores. For simulations with the Sutton-Chen potential, ExaSTAMP is about three times faster whatever the number of cores, which is due to the lack of vectorization in CoMD.

9.4 Conclusion

Except a “poor” thread scalability on the nehalem processor (which has to be put into perspective with a speedup of 6 on 8 cores), our framework shows an almost linear acceleration up to 8192 cores, as long as we keep one MPI process per socket to avoid NUMA effects.

On recent processors (which have been optimized a shared-memory utilization), thread-based parallelization brings better performance on a single node, which is especially true for the KNC, where the use of MPI within a node induces a critical drop in performance. On a large number of cores, thread-based parallelization allows us to save significant amounts of memory.

¹Since CoMD has only tabulated EAM potentials, we implemented ourselves Sutton-Chen’s analytic formulas in the code.

The comparison with STAMP and CoMD software mostly showed the importance of vectorization. On a haswell, ExaSTAMP is respectively 1.3 and 3.5 times faster than STAMP with Lennard-Jones and Sutton-Chen potentials.

10

Ejecta Simulations of Metal Under Shock

10.1 Context

When a shock is propagating through a metallic crystal in contact with vacuum through an irregular free surface, the reflection of the shock wave at the interface can give rise to the ejection of 2D jets/sheets of atoms which develop and break up, forming ejecta (fragments) of different masses and volumes, as shown on Figure 10.1.

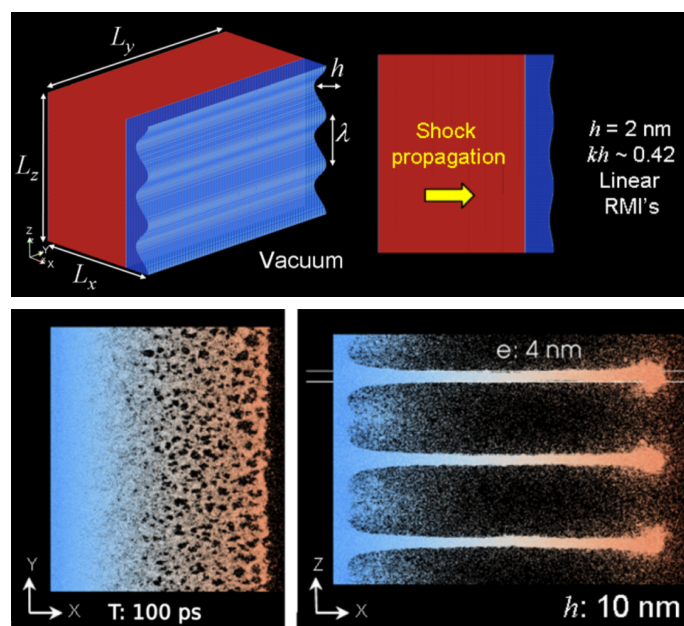


Figure 10.1 | Ejecta Simulation of Metal Under Shock. From [40]. [top] Description of the generic configuration with a sinusoidal free surface roughness. The shock propagates along the O_x axis. [bottom left/right] Top and side views of the Sn crystal at the moment of appearance of holes/cracks in the atom sheets. The sheets begin to break up as their thickness goes down to a value of about 4 nm.

At CEA, this kind of phenomena are the subject of various studies, which provided an understanding of various underlying processes. In [39], simulations showed that the ejection phenomenon tends toward a steady state on long times, and that ejected particles remain spherical with time. Moreover, their size distribution partially exhibits a power law scaling. The investigation on the fragmentation process demonstrated that it is initiated by stochastic and local

fluctuations of the sheet's thickness that cause the creation of pores in the sheet [40]. These pores grow, coalesce, and percolate following a hierarchical process. An analytic model was eventually proposed in [41] in order to describe more precisely mass and size versus velocity distributions of ejected particles.

All simulations in these studies have been performed with STAMP, using systems of copper or tin from 1×10^7 to 2×10^8 million atoms. Simulations are carried on times up to the nanosecond, which still represents 10^6 iterations. The main issue with these simulations is that the particle distribution over the simulation box is very inhomogeneous (see also Figure 8.2), resulting in seriously unbalanced situations since STAMP only uses a grid-based decomposition.

In order to test in real conditions the load balancing capability of ExaSTAMP, we chose two different ejecta simulations for its first production cases: the first one consists in resuming a 200 million atoms simulation that was stopped at one nanosecond, and the second one is a new “large” case with about 700 million atoms, which could confirm the scale invariant property of some phenomena observed in smaller cases.

10.2 Adjustments for Production Runs

10.2.1 A Specific Potential for Tin

Simulations in [39, 40, 41] involve copper or tin atoms. Interactions for copper are treated with a Sutton-Chen potential, when tin systems use a custom EAM potential, developed by Sapozhnikov et al. [99]. This potential, which has been designed to predict properties of shocked tin near its melting point, is described by the three following functions

$$\begin{cases} \phi(r) &= E_0 - 2\frac{E^{(\text{coh})}}{Z} \left[1 + \alpha \left(\frac{r-r_0}{r_0} \right) + \eta \left(\frac{r-r_0}{r_0} \right)^2 + \mu \left(\frac{r-r_0}{r_0} \right)^3 + D \left(\alpha \frac{r-r_0}{r_0} \right)^3 \frac{r_0}{r} \right] \\ F(\bar{\rho}) &= -(1 - \ln(\bar{\rho}))\bar{\rho} \\ \rho(r) &= \frac{1}{Z} S\left(\frac{r_{\max}-r}{r_{\max}-r_{\min}}\right) \exp\left(-\beta \frac{r-r_0}{r_0}\right) \end{cases}, \quad (10.1)$$

where S is a polynomial smooth function which expression is

$$S(x) = \begin{cases} 0 & \text{if } x < 0 \\ 35x^4 - 84x^5 + 70x^6 - 20x^7 & \text{if } x \in [0, 1] \\ 1 & \text{if } x > 1 \end{cases}. \quad (10.2)$$

As for Lennard-Jones and Sutton-Chen potentials, parameters for this EAM potential are available in Appendix B.

Since both production cases for ExaSTAMP involve tin, we added this new potential to our framework. Implementation was pretty straightforward, we inherited a new potential from the EAM class and wrote the functions with our intrinsic generator. When applied to a perfect crystal, this new potential is about 25% more expensive than a Lennard-Jones, when the Sutton-Chen is between 3 and 4 times more expensive (measurements were performed on the full simulation).

10.2.2 Free Boundary Conditions

Ejecta simulations have specific boundary conditions. In the shock direction (O_x), we have a wall condition on one side to initiate the shock, and free boundary condition in the other side. For other directions (O_y and O_z), we have periodic boundary conditions.

If we already had periodic and wall boundary conditions, we had to implement the free one to perform our production simulations. When a particle reaches a global edge of the simulation box where free boundary conditions are in effect, concerned domains are extended by a factor defined by the user (usually by 20 to 50%), and all Grid objects are updated.

10.2.3 Issues with Load Balancing

First attempts in using load balancing on the 200 million case turned in an unexpected way: in most simulations, we ran out of memory after a few balancing steps, and the acceleration provided by load balancing was quite disappointing, despite analyzes showing that domains were perfectly balanced in number of particles.

Indeed, all domains effectively had the same load, but we did not anticipate their distribution: almost all domains were located near the base of the jet, which was spread on one or two domains. In the end, a single domain was covering about 80% of the simulation box. Whereas this would not be a problem on relatively small simulations, our case already has a significant extension, and the global simulation box gathers almost a billion cells. Even empty of particles, a Cell object takes some space; the fact that a single domain gathers 80% of all cells explains the memory problem. Moreover, the poor performance observed can also be explained with this excess of cells: going through ten times more cells than other domains, even empty ones, is not without cost.

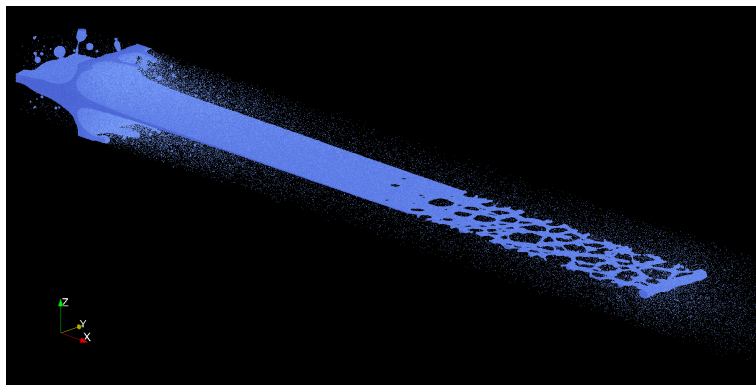


Figure 10.2 | The 200 million atoms case after one nanosecond. Zoom on the jet structure. The part behind the base of the jet has been cropped and is much bigger, such that most domains are assigned to this part, when a few others are in charge of all the sheet, causing disparities in memory usage that slow down the entire simulation.

In order to face this issue, we need each domain to have more or less the same number of cells *and* the same number of particles. We first tried to add a constant to our cost function, so that empty cells were not considered as “free”, but adjusting this constant against potential cost happened to be more complicated than expected. Instead, we had to use one of ZOLTAN’s features which allows to perform balance operations on multi-dimensional weights: each cell now has two costs, one related to the computational time (i.e. our cost function), and the other one related to memory occupancy. Balance operations should now result in partition balanced both in terms of number of cells and computation time.

However, the use of multiple weights does not come without any drawbacks. If we had some guarantees about the existence of an optimal solution for partitioning with a single weight, it does not extend to multiple weights. Furthermore, the multiple weights capability is not supported by all ZOLTAN's algorithms. If we can go with both bisection methods, we nonetheless have to drop the SFC partitioning. In theory, ZOLTAN's graph algorithms support this capability, but only up to a certain number of objects which is surpassed for the 200 million case.

10.3 Results

10.3.1 Accuracy

Before starting production runs, we must verify physical results produced by ExaSTAMP. Simulation in NVE ensemble ensure both energy and moments conservation in time, which we already checked during development steps. Since we are going to resume a simulation initiated by STAMP, we must ensure we produce the exact same results. When we compared both kinetic and potential energies produced by both codes on large systems, they matched up to the 13th decimal on short runs. On smaller systems and longer runs, we compared for each particle positions produced by both codes: histograms of relative differences showed centered gaussian curves, with extrema around 10^{-10} , which definitively validated the exactness of ExaSTAMP's results.

10.3.2 Performance

The 200 million case

We started the 200 million case on 2048 cores of TERA supercomputer (256 domains, 8 threads per domain, which represents 1 MPI per socket), with a balance operation every 5000 iterations. In this configuration, the RIB method proved slightly better than the RCB, which was not expected considering our tests in Chapter 8, and shows that every situation is different and must be tested for the best partition algorithm. First measures showed that simulation ran about 10 times faster than with STAMP (on the same number of cores), with a factor three due to sequential acceleration, the rest being brought by load balancing. However, traces showed that the partitioning quality was still unsatisfying, since we enabled ZOLTAN's multiple weights capability. The use of larger domains would have improved the imbalance factor, but on those nodes, NUMA effects were too important and unfortunately did not let us see any improvements. Despite a perfect scalability on 4096 cores which would have diminished memory occupancy per core, we kept running on 2048 since the job scheduler was much more likely to execute smaller runs. We therefore concentrated our efforts on finding optimization to speed-up our simulations.

Since the issue was brought by an important number of cells, we eventually thought of diminishing this number (without of course changing the physic of the simulation). Indeed, cells are only here to facilitate the neighbor search: with larger cells, neighbor search is more expensive but we will improve memory usage since we have a lot of empty cells. Furthermore, with less cells we may improve the partition's quality and retrieve some performance. We started from a cell size of 6 Å (the cutoff radius from Sapozhnikov potential is about 5.6 Å), and forced different cell sizes up to 12 Å.

This technique proved effective up to a cell size of 9 Å (see results in Table 10.1), with simulation up to 2.5 times more efficient and three times faster when considering compute time and the cost

Cell size (Å)	6.00	6.60	7.00	7.50	8.00	8.50	9.00	10.00
Number of cells (10^6)	804.9	604.8	506.9	412.1	339.6	283.1	238.5	173.9
Memory (MB/core)	1279.2	979.0	837.4	693.4	576.2	488.3	417.5	306.3
Eff. Time (10^{-5} s)	4.23	4.01	3.43	2.84	2.36	2.00	1.71	1.92

Table 10.1 | Evolution of some quantity from the 200 million case with the cell size. We observe that augmenting the cell size allowed us to limit memory usage, which significantly improved performance.

and balance/migrations. The fact that we almost did not lose any time in the neighbor search with cells three times larger can be explained by our projection optimization (see Section 7.2.1), which prevented a lot of unnecessary tests. In the end, our modified version with larger cells is still far from being nicely balanced (the imbalance rate varies between 25 and 50%), but runs 30 times faster than STAMP¹, and scales perfectly on 4096 cores. The case is currently still running and data produced are exploited by physicists.

The 700 million case

Since we started the 700 million atoms case from the start, with the initiation of the shock propagation, we did not encounter any particular issues. During the first hundred thousand iterations, the extension of the jet is still limited as on pictures of Figure 10.3, which still allows good balance rates (around 5%). As a consequence, the efficient time is much better than in the previous case, around 8×10^{-6} (in second per particle per iteration per thread). This case runs on 2048 cores, but scales perfectly on 4096 and 8192, with still one MPI process per socket in order to minimize memory usage and avoid NUMA effects.

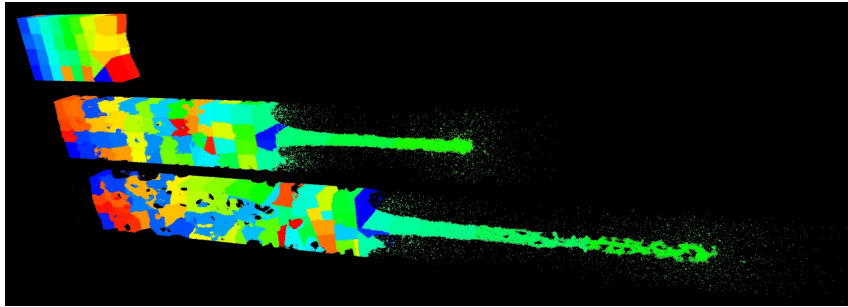


Figure 10.3 | An ejecta simulation with domain distribution at different times. Reduced simulation (about 7 million atoms) similar to the 700 million case.

10.3.3 Conclusion

In this section, we realized that ejecta simulation could be, up to a certain scale and a physical time, very challenging. As long as the atom sheet is not too extended, simulations are well balanced and performance is very satisfying. Around the nanosecond, the simulation box has seen its extension multiplied several times in the O_x direction, just as the number of cells. At this point, ZOLTAN's partitioners still provide balanced distributions in number of particles, but one or two domains found themselves in charge of 80% of the global simulation box and ran out of memory.

¹Without load balancing, we estimated the imbalance rate beyond 150%.

We faced this issue by enabling ZOLTAN's multiple weights capability: this allowed us to take into account the memory cost of cells, but significantly reduced partitions' quality. With partially balanced simulations, we still managed to be much faster than STAMP by a factor of 10. Augmenting the cell size decreased memory usage and improved the balance, making the production case 30 times faster than STAMP, bringing down a month of simulation into only a day.

These good results nonetheless leave the door open to many improvements. For instance, a specific partitioner giving non-connected domains (which is, by construction, not possible with ZOLTAN's geometric algorithms) could provide a good overall balance by separating the global simulation box in a dense zone and a thin one, then perform a partition on both zones and assign a to every domain a part of each zone. Another solution would be to consider a larger entity than cells as a ZOLTAN object, which could lead to better quality partitions.

Supercomputer architectures are evolving towards massively multicore nodes with hierarchical memory structures and equipped with larger vectorization registers. In order to achieve a high level of performance on such architectures, the programmer has to explicit operations that were previously taken in charge by systems or compilers, such as locality of memory allocations or the writing of vectorized code.

Nowadays, very few HPC codes fully support a thread-based parallelization, which is somehow natural given the long development times of such large applications. As for low-level optimization, they are often missing since most developments are carried out by physicists or numerical analysts, except when they are commissioned to specific libraries (e.g. BLAS or LAPACK). Despite being less effective, this kind of application still meets decent performance on Haswell-like architectures, as long as the decrease of memory per core does not disturb the scalability of the most greedy ones. However, the lack of an efficient thread-based parallelization and vectorization will become critical on architectures such as the Intel® Xeon Phi™, which is considered as one of the few choke points to reach the exascale barrier.

When analyzing source code of high performance applications, we realize that in most cases, there is no obvious distinction between parts related to numerical analysis and others needed to express some parallelism. This slows down developments by making the code harder to read and to understand, and also prevents eventual non-expert developers from introducing new features. A clear separation would enable (enlightened) physicists or applied mathematicians to develop their own modules within a high-performance environment maintained by parallelism experts, so that they could fully tap their machines' performance. Moreover, the generalization of such process could exhibit similar patterns between applications, which would ease the spreading of innovating techniques and algorithms between different fields.

Contributions

In this thesis, we propose the design and implementation of ExaSTAMP, a code architecture in the context of Molecular Dynamics simulations. Dedicated to material sciences, this framework will be used as a production tool in order to study the behavior of matter under shock. ExaSTAMP will cover the need to simulate very large systems, and fully exploit the power of future processors. Moreover,

the code must separate numerical modules from parallelism to let non-expert programmers develop their functions and benefit from the framework's performance.

To this end, we studied MD processes and algorithms in a design phase, from where we draw different object hierarchies in order to minimize code duplication. We built our architecture by separating *node* and *domain* concepts to fit NUMA architectures. Communications between domains are either interfaced through the *communication manager*, or hidden in dedicated messages structures. Domain and *grid* objects specify algorithms through *traversals*, which allowed us to hide thread parallelism and to introduce a new topology without changing a single line of numerical modules.

Regarding implementation, our object oriented design allows us to test various features without impacting the standard developer. We focused in particular on efficient data structures which enable both thread scalability and vectorization. About that, we showed that relying on the compiler could be risky for simple kernels, and provided instead a simple tool that generates explicit vectorized code. This intrinsic generator is quite simple to handle and brings a significant acceleration compared to auto-vectorization. After having demonstrated its necessity, we also brought ExaSTAMP with a load balancing capability at the MPI level. Besides, we established a collaboration with INRIA Bordeaux for the development of highly optimized OpenCL kernels in order to target GPUs. Their work as a standalone library is very promising, and its integration into our architecture is expected soon.

ExaSTAMP shows an almost perfect scalability, either with TBBs on one node, or in hybrid mode on several thousand cores. On Haswell and Intel® Xeon Phi™ (KNC), we have a linear speedup except when using hyperthreading, which we interpreted as cache management issues. Besides, we reported a short run with about 15 billion atoms on 32,768 cores, which proves our capability to be used on a large number of cores. Regarding memory, we pointed out the fact that the use of thread parallelization could seriously reduce memory usage compared to a full-MPI implementation, and that this was applicable to most of domain-decomposition based applications. In order to validate our work, we eventually tested ExaSTAMP in production on ejecta simulations. If the space extension of one of the two cases prevented the load balancing capability to provide a perfectly balanced distributions, we managed to be about 30 times faster than STAMP, and are able to run on twice less cores thanks to our better memory management.

Perspectives

The work we described in this thesis has raised various open questions, and several research opportunities remain to be explored.

Multi-domain implementation. One of our biggest regrets is not to have implemented the ability to run several domains on a single node, which constrains us to perform simulations with one MPI process per socket in order to avoid NUMA effects. Within a node, the instantiation of a domain per memory bank and a strict sharing of available threads between these domains should significantly reduce distant access and improve performance. With Intel®'s TBB, the `tbb::task_group` object provides a simple environment which should ease the implementation of this capability without impacting the rest of the code.

Improving load balancing. Despite an important acceleration compared to STAMP version, our choices for load-balancing showed some limits in the context of ejecta simulations. We nonetheless dispose of several leads to improve load balancing efficiency on such cases. By choosing a larger entity that cells as a ZOLTAN object (e.g. a block of cells whose size would be an adjustable parameter), we would significantly reduce the number of degrees of freedom for the partition function, which may result in better quality distributions. Furthermore, with less objects to balance, we may be able to successfully call graph methods, which could get rid of the connectivity constraint imposed by geometric algorithms, which is responsible for our memory problems. Another option would consist in the development of our own load balancing algorithm on the top of ZOLTAN when the situation requires it.

Optimizations for Intel® Xeon Phi™ architectures. If we managed to extract good performance on the KNC, we are still much slower on this architecture than on the Haswell, which however gets half of the KNC's peak performance (about 2 Tflop/s against 1). If the KNL should resolve various issues observed on the KNC, we will need to perform several optimizations in order to entirely fill this gap. First efforts should focus on hyperthreading efficiency. Indeed, if we observe a perfect scalability from 1 to 60 cores, the speedup curve flattens at 120 and 240 threads, which should not happen since on each core, the Intel® Xeon Phi™'s system schedules alternate hardware threads at each alternate cycle. Regarding future KNLs, the management of the specific memory (MCDRAM) raises a lot of questions about its impact on global performance.

Development of new capabilities. In this thesis, we developed an architecture to perform high-performance MD simulations. To this end, we developed basic structures which enable the simulation of atomic systems using pair or EAM potentials. To finish the validation of our framework, we must keep adding new functionalities, such as integration schemes, potentials, and other types of particles (e.g. molecules) ... Beta-developers should be able to introduce these capabilities without modifying high-level layers of the code, and simulation using these new tools should get the same level of performance as the results presented in this work. The development of dynamic multi-scale methods in ExaSTAMP are scheduled to start in a few months in the context of another PhD.

Integration of GPU kernels. Future works also include a significant part about GPU computing. Once we finish the integration of OpenCL kernels developed at INRIA Bordeaux into ExaSTAMP, we will initiate the development of more complex potentials which will be used in production. In parallel, the use of accelerators on supercomputers will raise some matters in terms of load balancing and data movement that will have to be solved to perform very large scale simulations.

Towards Exascale Particle Simulations

Distribution of the intrinsic generator. Our intrinsic generator is not limited to particle simulations. When we see the importance of vectorization on future machines and the lack of solution proposed at this level (BLAS libraries only propose arithmetic operations on vectors), we think that our solution may be adapted in numerous situations. We started to extract our vector classes as an independent library, but we still need to add other functionalities (such as an aligned allocator) in order to have a complete, stable tool to help people generate vectorized code.

A more flexible structure for particle simulations. The cell structure on which we relied to build our particle storage solution owes its efficiency to the fact that we considered condensed matter systems, which guarantees a certain number of particles in each cell. This is why we explicitly separated ourselves from MD applications targeting biomolecular systems.

Following the idea of the cell structure being a perfectly structured mesh, we could imagine a more general arrangement in the style of adaptive mesh refinement: the global simulation box would still be divided in cells, but these cells would be of different size such that they all approximately contain the same number of particles. In that way, dense areas will be treated as we treat cells in ExaSTAMP, but cells in light zones will be gathered in larger entities. Such solution would save a lot of memory in situations with high density disparities (e.g. ejecta simulations), without wasting too much time in neighbor search by limiting the cell object to a certain number of particles. Moreover, by ensuring a minimum number of particles per cell, we can be sure that vectorization will remain efficient.

With the overcoming of this density constraint, we could eventually consider a unified framework for exascale MD simulations covering a wide area of application, including biochemical systems, polymers, liquids and metals.

Data management for exascale computing. The coming of exascale computing and associated data generated from large-scale simulations in various fields will constrain us to change the way we approach these data. Step by step, our ability to produce them has taken over our ability to exploit them. With exascale datasets, we will be creating far more data than we can explore in years with current tools. In order to make exascale computing worthwhile, one of the greatest challenges in science will be to effectively understand and make use of this rapidly growing data, with new theories, techniques, and software that will later become standard tools involved in future discoveries and scientific advances.

Appendix

A

Processors Used in our Tests

This section presents all CPU models used in our tests. In tables A.1 to A.4, fields “Soc.,” “Cps” and “TpC” respectively stand for the number of sockets, the number of cores per sockets and the number of threads per cores. When multithreading has been disabled, the number of threads per cores is preceded by “*”.

- **Table A.1:** Desktop machines have been used for simple, sequential tests.
- **Table A.2:** Inti/Cirrus are small clusters designed to test and evaluate performance of new processors.
- **Table A.3:** TERA 100 was the first european supercomputer to reach the PetaFlop/s and reached the 6th of Top500 in November 2010, and is CEA’s main computing ressource. We used TERA 100 for production runs and compare ExaSTAMP performance against STAMP.
- **Table A.4:** Airain is a CCRT¹ supercomputer that we used for larger and/or longer simulations than these carried on Inti/Cirrus. Except for very large simulations (over 2048 cores), we preferred Airain over TERA 100 as it has more recent processors that support in particular AVX instructions.

Partition	Model Name	Freq. (GHz)		Soc.	CpS	TpC	Cache L3 (MB)	Vectorization
		norm.	turbo					
westmere	Intel®Xeon® CPU X5660	2.8	3.2	2	6	*2	12	SSE
sandybridge	Intel®Xeon® CPU E5-2687W v2	3.4	4.0	2	8	2	25	AVX

Table A.1 | Desktop Machines.

¹See <http://www-ccrt.cea.fr/>.

Partition	Model Name	Freq. (GHz)		Soc.	CpS	TpC	Cache L ₃ (MB)	Vectorization
		norm.	turbo					
atom	Intel®Atom™ CPU C2750	2.4	2.6	1	8	1	4	SSE
haswell	Intel®Xeon® CPU E5-2698 v3	2.3	3.6	2	16	2	40	AVX
ivybridge	Intel®Xeon® CPU E5-2697 v2	2.7	3.5	2	12	*2	30	AVX
knc	Intel® Xeon Phi™ Coprocessor 5120	1.1	–	–	60	4	**30	IMCI
nehalem	Intel®Xeon® CPU X5560	2.8	3.2	2	4	*2	8	SSE

Table A.2 | Inti/Cirrus. ** For the KNC, the cache size concerns the L2 cache.

Partition	Model Name	Freq. (GHz)		Soc.	CpS	TpC	Cache L ₃ (MB)	Vectorization
		norm.	turbo					
standard	Intel®Xeon® CPU X7560	2.3	2.3	1	8	*2	24	SSE

Table A.3 | TERA 100.

Partition	Model Name	Freq. (GHz)		Soc.	CpS	TpC	Cache L ₃ (MB)	Vectorization
		norm.	turbo					
ivybridge	Intel®Xeon® CPU E5-2680 v2	2.7	3.5	2	10	*2	20	AVX
standard	Intel®Xeon® CPU E5-2680 0	2.8	3.6	2	8	*2	25	AVX

Table A.4 | Airain.

B

Atoms and Potentials Parameters

We present here all parameters used in our benchmarks. Unless specified otherwise, parameters for atoms and potentials are respectively in tables B.1 and B.2 to B.4.

Atom	Symbol	Mass (kg)	Lattice	Lattice param. (m)
Copper	Cu	1.055×10^{-25}	FCC	3.540×10^{-10}
Tin	Sn	1.971×10^{-25}	BCC	3.702×10^{-10}

Table B.1 | Atoms.

Parameter	Value	Unit
r_{cut}	5.675×10^{-9}	m
ϵ	9.34×10^{-20}	J
σ	2.27×10^{-10}	m

Table B.2 | Lennard-Jones Potential. For Cu/Cu interactions.

Parameter	Value	Unit
r_{cut}	7.29×10^{-10}	m
c	3.317×10^1	–
ϵ	3.605×10^{-21}	J
a_0	0.327×10^{-9}	m
n	9.050	–
m	5.005	–

Table B.3 | Sutton-Chen Potential. For Cu/Cu interactions.

Parameter	Value	Unit
r_{cut}	5.599×10^{-10}	m
$E^{(\text{coh})}$	2.956×10^{-19}	J
A	1.410	–
n	0.724	–
Z	7.618	–
r_{max}	5.599×10^{-10}	m
r_{min}	1.000×10^{-10}	m
β	6.0	–
r_0	3.437×10^{-10}	m
E_0	5.150×10^{-20}	J
α	3.072	–
D	0.145	–
η	2.72	–
ν	–1.87	–

Table B.4 | Sapozhnikov Potential. For Sn/Sn interactions.

Résumé Détaillé

Introduction

La simulation numérique est un outil relativement récent qui a contribué à l'élaboration de modèles complexes, aux côtés de la théorie et l'expérience. Les méthodes dites *particulaires* constituent un exemple de ses applications ; elles sont notamment utilisées en astrophysique, en biologie ainsi qu'en physique des matériaux. Si la course à la performance a permis le développement rapide des capacités informatiques, les modèles de programmation conçus pour exécuter efficacement des simulations sur des *super-calculateurs* est devenu extrêmement complexe. Ceci sera d'autant plus vrai quand il s'agira de dépasser la barrière de l'*exaflop*.

Dans cette thèse, nous proposons la conception et le développement d'une plateforme de simulation adaptée aux architectures des futurs super-calculateurs ; le domaine d'application choisi est la *dynamique moléculaire*. Cette plateforme, qui sera utilisée en production, devra couvrir une large gamme de situations physiques. En outre, toute la complexité de programmation liée à l'expression du parallélisme devra être séparée et cachée des modules de physique, afin que n'importe quel développeur puisse apporter ses contributions sans se soucier des performances.

La dynamique moléculaire

La dynamique moléculaire (DM) est une méthode de simulation utilisée pour étudier les propriétés d'un système de particules donné. Cette méthode, très utilisée en biologie, chimie et sciences des matériaux, consiste à intégrer numériquement les équations du mouvement qui régissent un ensemble de particules.

Intégration

Les particules pouvant être des molécules et des atomes, les équations à résoudre devraient être celles de la mécanique quantique. L'approximation de Born-Oppenheimer nous permet de négliger le mouvement des électrons par rapport aux noyaux ; en considérant la longueur d'onde de de Broglie, nous pouvons légitimement négliger les effets quantiques pour des systèmes dont la température est suffisamment élevée.

L'évolution d'un système de particules décrites par leur coordonnées et quantités de mouvement est donnée par les équation d'Hamilton. Nous en déduisons facilement une expression plus

familière, la seconde loi de Newton. Pour intégrer précisément cette équation, il faut que les schémas utilisés satisfassent un certain nombre de propriétés (stabilité, symplecticité, ...). Les schémas de type *Verlet* sont généralement utilisés, car ils remplissent ces conditions sans être trop gourmands en moyens de calculs.

Par construction, l'intégration des équations de la DM fournit des solutions dans l'ensemble *micro-canonique*, ce qui signifie que le nombre de particules, le volume global de simulation et l'énergie totale restent constants au cours du temps. Pour appliquer la DM à d'autres configurations où des quantités comme la pression ou la température sont des paramètres au lieu de propriétés, il faut modifier les équations de Newton. Ces modifications prennent la forme de termes de frottements ou de forces stochastiques par exemple.

Potentiels

Dans la plupart des simulations de DM, les particules sont considérées comme ponctuelles et la force d'interaction entre les particules est approchée par le gradient d'un *potentiel*. Ce dernier dépend des positions des particules les unes par rapport aux autres. Le calcul de la force est de loin la partie la plus délicate : le potentiel contient toute la physique de la simulation de DM, et le calcul de la force peut représenter jusqu'à 99% du temps de simulation. Les potentiels utilisés en DM sont empiriques ou semi-empiriques ; ils sont en général calculés à partir d'une formule analytique, mais il arrive qu'ils soient interpolés à partir de tables si les calculs deviennent trop coûteux.

En théorie, les potentiels utilisés en DM ont une portée infinie. Afin d'éviter des temps de calculs rédhibitoires, les interactions sont négligées au-delà d'une certaine distance appelée distance de coupure. Cette approximation, qui n'est valable que pour certains potentiels, permet de passer d'un problème d'ordre N^2 à un problème d'ordre N . Parmi les potentiels les plus utilisés, nous trouvons le potentiel de Lennard-Jones et les potentiels de type EAM. Si le premier est dédié à l'étude des gaz et de certains liquides, les potentiels EAM, beaucoup plus chers en temps de calcul, donnent des résultats bien plus précis pour des systèmes métalliques.

Applications

La DM a vu le jour au début des années 1950, mais cette méthode est longtemps restée associée à la physique théorique du fait de l'absence de moyens informatiques. En 1957, Alder ont réalisé la première simulation de DM en utilisant le modèle des sphères dures. La DM s'est ensuite développée à partir des années 1970 grâce à l'essor des capacités de calcul. La barre du million de particules a été franchie en 1990, et celle du milliard au début des années 2000. Aujourd'hui, la DM est un outil utilisé aussi bien en biologie pour déterminer des structures de protéines qu'en chimie. Au CEA, la DM est un outil de choix pour l'étude du comportement des matériaux dans des conditions extrêmes.

Il existe de nombreux codes pour réaliser des simulations de DM, la majorité étant dédiée aux systèmes bio-moléculaires, dont l'approche est radicalement différente par rapport à la physique de la matière condensée (nombre de particules, complexité des potentiels, ...). Aucun des codes orientés sur l'étude des matériaux ne dispose d'une architecture flexible capable de supporter efficacement le multithreading ou la vectorisation.

Motivations

La course à la performance est motivée par le besoin de modéliser toujours plus finement des situations toujours plus complexes. Si certains phénomènes exhibent des procédés à invariance d'échelle, d'autre bénéficieraient grandement d'un gain d'un ou deux ordres de grandeur au niveau de la taille du système.

Les architectures des processeurs ont connu ces dernières années des changements majeurs : des nœuds massivement *multi-cœurs* ont succédé aux simples SMP, et les accès mémoire au sein de ces nœuds ne sont plus nécessairement uniformes. Avec l'AVX, la *vectorisation* ne peut plus être négligée, et l'utilisation d'accélérateurs (GPU, ...) est devenue courante. Si ces modifications permettent une puissance de calcul toujours plus grande, celle-ci est néanmoins de plus en plus délicate à extraire. Le programmeur doit maintenant gérer explicitement plusieurs niveaux de parallélisme en plus de ses modules de développement.

Considérant la constante augmentation des capacités de calculs pour effectuer des simulations de très grands systèmes, les évolutions récentes mais majeures des architectures des processeurs, mais aussi la conception actuelle de nombreux codes de DM, nous avons décidé de créer ExaSTAMP, une nouvelle plateforme de simulation dédiée à la DM conçue pour être utilisée sur les futurs super-calculateurs. L'objectif de cette thèse est ne consiste évidemment pas en la réécriture d'une centaine de milliers de lignes de code. Nous allons concevoir et développer les composant de base de la plateforme afin que les développeurs puissent ajouter des fonctionnalités au fur et à mesure sans se soucier des performances.

Architecture

Dans ce chapitre, nous détaillons toute la conception d'ExaSTAMP. L'approche orienté-objet nous a permis d'associer à chaque concept de DM un *objet* du point de vue informatique. Chaque objet que nous avons créé possède un signification propre et un champ d'action bien défini. Les relations entre les objets (par exemple, l'héritage) nous ont permis de minimiser la duplication de code, qui est un des principaux obstacles à la maintenabilité du code. Le calcul de la force fait apparaître un schéma commun à tous les potentiels de paire, et un autre pour les potentiels EAM. Ces schémas possèdent plusieurs versions, selon que l'utilisateur souhaite activer ou non le recouvrement calcul-communications, ou la symétrisation.

Pour paralléliser une simulation de DM, la seule méthode scalable est celle de la décomposition de domaine : la boîte de simulation est découpée en autant de parties qu'il y a de processeurs. À chaque itération, il faut alors communiquer les particules qui bougent d'un domaine à un autre. C'est pourquoi on associe traditionnellement un processus MPI à chaque sous-domaine. Comme nous souhaitons pleinement exploiter les architectures *manycore*, nous avons décidé de séparer ces notions. Un objet *Node* représente le processus MPI, et l'objet *Domain* un domaine. Chaque domaine est parallélisé par des threads, et il peut éventuellement y avoir plusieurs domaines par nœud.

En pratique, l'objet qui représente le domaine est une interface dont les implémentations dépendent de la physique (atomes, molécules, méso-particules, ...) considérée. Ces implémentations spécialisent les services requis par les différentes versions du calcul de la force. Le domaine est virtuellement découpé en cellules pour faciliter la recherche de voisins. Les cellules contiennent les particules, et sont regroupées dans un objet appelé *Grid*. Pour faciliter le développement de

nouvelles fonctionnalités, les différentes façons de parcourir une grille sont pré-calculées, et ceci de façon indépendante par rapport à la topologie employée.

Implémentation

L'implémentation est le résultat d'un compromis entre les performances et une séparation explicite entre les modules de physique et le parallélisme. Pour ne pas trop subir le surcout lié à l'héritage, nous avons utilisé le *Curiously Recurring Template Pattern*. Le parallélisme de thread a été implémenté par la bibliothèque TBB d'intel. Afin de maximiser son efficacité sans pénaliser la vectorisation, nous avons étudié plusieurs structures de données pour stocker les particules. Le calcul de force a été l'objet de nombreuses optimisations, notamment pour la recherche de voisins.

Nous avons également montré l'importance de la vectorisation et les limites de l'auto-vectorisation offerte par les compilateurs. La vectorisation explicite est pour l'instant la seule option pour obtenir des performances, mais le code obtenu est lourd et difficilement maintenable. C'est pourquoi nous avons proposé un générateur de vectorisation, qui permet d'obtenir des performances proches des speedups théoriques tout en étant utilisable par des développeurs non-experts.

Par ailleurs, nous avons également mis en place un schéma de communication qui minimise le nombre de messages effectif lorsqu'il y a plusieurs domaines par nœud : si du point de vue de l'utilisateur, les échanges se font toujours entre domaines, en arrière-plan, les messages sont découpés et réarrangés par nœud.

Équilibrage dynamique de charge

Les problèmes d'équilibrage de charge sont une des principales causes de perte de performance lorsque l'on effectue des simulations sur un grand nombre de cœurs. En effet un simple découpage en parallélépipèdes ne peut garantir une bonne répartition de la charge de travail entre tous les domaines. Dans ExaSTAMP, nous avons décidé de considérer la structure en cellule utilisée pour la recherche de voisins comme un maillage. Ainsi, les domaines s'échangent des cellules pour équilibrer leur charge. Cette taille de grain (env. 50 particules par cellule) est beaucoup plus manipulable que les particules elles-mêmes. De plus, cette structure en maillage nous permet d'utiliser des partitionneurs existants. Nous avons porté notre choix sur la bibliothèque ZOLTAN, car elle permet de tester de nombreux algorithmes de partitionnement, et n'impose pas de structure de donnée particulière.

Au niveau de l'implémentation, il nous a fallu concevoir une nouvelle topologie de domaine compatible avec l'équilibrage de charge : un domaine est maintenant un ensemble quelconque de cellules. Nous avons également mis au point une fonction pour évaluer la charge de calcul dans chaque cellule : celle-ci dépend du nombre de particules de chaque type présent dans la cellule, du coût unitaire du potentiel considéré, et de son rayon de coupure. En pratique, l'équilibrage est déclenché dès que l'estimation du déséquilibre dépasse une valeur seuil donnée en paramètre. Lors d'une étape d'équilibrage de charge, l'appel à la bibliothèque retourne une liste de cellules quittant le domaine, et une liste de cellules arrivant sur le domaine. Une fois le domaine reconstruit en tenant compte de ces listes, la simulation peut reprendre normalement.

Les performances obtenues sur des cas simples (statique et dynamique) permettent de valider la fonction de coût, et de dégager les algorithmes de partitionnement les plus efficaces. Dans notre cas, les méthodes dites *géométriques* sortent du lot.

Performances

Nous avons évalué les performances d'ExaSTAMP en conduisant des tests de scaling dits *forts* pour tester la scalabilité sur un nœud en mémoire partagée, et des tests *faibles* sur un très grand nombre de cœurs. Dans tous les cas, nous obtenons des accélérations quasiment linéaires, que ce soit sur des architectures Nehalem, Haswell ou Intel® Xeon Phi™. Nous pouvons toutefois relever une légère baisse de l'efficacité lorsqu'il s'agit d'exploiter l'hyperthreading.

Validation

Un des principaux objectifs de cette thèse était de fournir une plateforme de simulation prête à effectuer des simulations de production. Deux cas de simulation d'ejecta ont été proposés par les physiciens du CEA : une reprise d'une simulation de 200 millions d'atomes, et une simulation à 700 millions d'atomes. Si la validation des résultats physique n'a pas posé de problème particulier, certains ajustements ont été nécessaires au niveau de l'équilibrage de charge pour obtenir de bonnes performances. Au final, ExaSTAMP s'est montré jusqu'à 30 fois plus rapide que le code de DM actuellement utilisé.

Conclusion

Dans la course vers l'*exascale*, les architectures des super-calculateurs évoluent vers des nœuds massivement multi-cœurs, sur lesquels les accès mémoire sont non-uniformes et les registres de vectorisation toujours plus grands. Pour atteindre un bon niveau de performance sur de telles architectures, le programmeur doit maintenant expliciter des opérations qui étaient auparavant prises en charge par les systèmes ou les compilateurs. En analysant le code source de certaines applications de calcul haute-performance, nous observons que, dans la plupart des cas, il n'y a aucune distinction évidente entre les parties liées à l'analyse numérique et celles nécessaires à l'expression du parallélisme. Ceci ralentit l'évolution du code en le rendant plus difficile à lire et à comprendre, mais empêche également les développeurs non experts éventuels de l'introduction de nouvelles fonctionnalités.

Dans cette thèse, nous proposons la conception et l'implémentation d'ExaSTAMP, une architecture de code dans le cadre de simulations de dynamique moléculaire. Dédié à la physique de la matière condensée, il sera utilisé comme un outil de production dans le but d'étudier le comportement de la matière sous choc. Dans une phase de conception, nous avons traduit certains procédés et algorithmes de la DM en une hiérarchie d'*objets* informatiques, ce qui nous a conduit à minimiser la duplication de code. Concernant l'implémentation, nous nous sommes concentrés en particulier sur des structures de données qui permettent à la fois de bonnes performances au niveau du parallélisme de thread et de la vectorisation. À ce propos, nous avons également fourni un outil simple pour produire du code explicitement vectorisée. Ce générateur est très simple à manipuler et apporte une accélération significative par rapport à l'auto-vectorisation. Nous avons également inclus une capacité d'équilibrage dynamique de charge de calcul. Du côté des performances ExaSTAMP a montré une scalabilité presque parfaite, que ce soit en mémoire partagée sur un nœud ou en mode hybride sur plusieurs milliers de cœurs. Lors de tests de production, nous avons relevé une accélération de 30 par rapport au code actuellement utilisé.

Ce travail laisse néanmoins plusieurs questions ouvertes, et de nombreuses pistes sont encore à explorer : amélioration de l'équilibrage de charge avec une structure de cellules à raffinement adaptatif, optimisations pour le KNL, intégrations de noyaux GPU, ...

Bibliography

- [1] Stewart A. Adcock and J. Andrew McCammon. Molecular Dynamics: Survey of Methods for Simulating the Activity of Proteins. *Chemical Reviews*, 106(5):1589–1615, 2006.
➤ Cited on page 28.
- [2] B. J. Alder and T. E. Wainwright. Phase Transition for a Hard Sphere System. *The Journal of Chemical Physics*, 27(5):1208–1209, 1957.
➤ Cited on page 22.
- [3] M.P. Allen and D.J. Tildesley. *Computer Simulation of Liquids*. Clarendon Press, 1987.
➤ Cited on pages 2, 7, 13, and 18.
- [4] Kathleen T. Alligood, Tim D. Sauer, and James A Yorke. In *Chaos: An Introduction to Dynamical Systems*, volume 1 of *Textbooks in Mathematical Sciences*. Springer-Verlag New York, 1996.
➤ Cited on page 9.
- [5] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967.
➤ Cited on page 55.
- [6] Hans C. Andersen. Molecular dynamics simulations at constant pressure and/or temperature. *The Journal of Chemical Physics*, 72(4):2384–2393, 1980.
➤ Cited on page 10.
- [7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Third edition, 1999.
➤ Cited on page 58.
- [8] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In Y. Robert and M. Parashar and R. Badrinath and V. Prasanna, editor, *High Performance Computing (HiPC) 2006*, volume 4297 of *Lecture Notes in Computer Science*, pages 338–352. Springer Verlag, 2006.
➤ Cited on page 34.
- [9] ASCAC. Synergistic Challenges in Data-Intensive Science and Exascale Computing. Technical report, DOE, March 2013.
➤ Cited on page 22.
- [10] N. Attig, M. Lewerenz, G. Sutmann, and R. Vogel-sang. *Molecular Dynamics Algorithms For Massively Parallel Computers*, pages 46–69. 2000.
➤ Cited on page 46.
- [11] M. I. Baskes. Modified embedded-atom potentials for cubic materials and impurities. *Phys. Rev. B*, 46:2727–2742, 1992.
➤ Cited on page 15.
- [12] David M. Beazley and Peter S. Lomdahl. Message-passing Multi-cell Molecular Dynamics on the Connection Machine 5. *Parallel Computing*, 20(2):173–195, 1994.
➤ Cited on page 29.
- [13] H.J.C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A Message-passing Parallel Molecular Dynamics Implementation. *Computer Physics Communications*, 91(1–3):43–56, 1995.
➤ Cited on page 28.
- [14] M. J. Berger and S. H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Trans. Comput.*, 36(5):570–580, 1987.
➤ Cited on pages 88 and 89.
- [15] Bircsak, J. and Craig, P. and Crowell, R. and Cvetanovic, Z. and Harris, J. and Nelson, C.A. and Offner, C.D. Extending OpenMP For NUMA Machines. In *Supercomputing, ACM/IEEE 2000 Confer-*

ence, page 48, 2000.

➤ Cited on page 34.

- [16] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petit, Roldan Pozo, Karin Remington, and R. Clint Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.
➤ Cited on page 58.
- [17] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering, and Coloring. *Scientific Programming*, 20(2):129–150, 2012.
➤ Cited on page 88.
- [18] Erik Boman, Karen Devine, Lee Ann Fisk, Robert Heaphy, Bruce Hendrickson, Courtenay Vaughan, Umit Catalyurek, Doruk Bozdag, William Mitchell, and James Teresco. *Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; User's Guide*. Sandia National Laboratories, 2007. Tech. Report SAND2007-4748W http://www.cs.sandia.gov/Zoltan/ug_html/ug.html.
➤ Cited on page 88.
- [19] Erik Boman, Karen Devine, Lee Ann Fisk, Robert Heaphy, Bruce Hendrickson, Courtenay Vaughan, Umit Catalyurek, Doruk Bozdag, William Mitchell, and James Teresco. *Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; Developer's Guide*. Sandia National Laboratories, 2007. Tech. Report SAND2007-4749W, http://www.cs.sandia.gov/Zoltan/dev_html/dev.html.
➤ Cited on page 88.
- [20] Donald W. Brenner. Empirical potential for hydrocarbons for use in simulating the chemical vapor deposition of diamond films. *Phys. Rev. B*, 42:9458–9471, 1990.
➤ Cited on page 15.
- [21] R. A. Buckingham. The Classical Equation of State of Gaseous Helium, Neon and Argon. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 168(933):264–283, 1938.
➤ Cited on page 13.
- [22] D.A. Case, J.T. Berryman, R.M. Betz, D.S. Cerutti, T.E. Cheatham, III, T.A. Darden, R.E. Duke, T.J. Giese, H. Gohlke, A.W. Goetz, N. Homeyer, S. Izadi, P. Janowski, J. Kaus, A. Kovalenko, T.S. Lee, S. LeGrand, P. Li, T. Luchko, R. Luo, B. Madej, K.M. Merz, G. Monard, P. Needham, H. Nguyen, H.T. Nguyen, I. Omelyan, A. Onufriev, D.R. Roe, A. Roitberg, R. Salomon-Ferrer, C.L. Simmerling, W. Smith, J. Swails, R.C. Walker, J. Wang, R.M. Wolf, X. Wu, D.M. York, and P.A. Kollman. AMBER 2015, 2015.
➤ Cited on page 27.
- [23] Emmanuel Cieren, Laurent Colombet, Samuel Pitoiset, and Raymond Namyst. ExaStamp: A Parallel Framework for Molecular Dynamics on Heterogeneous Clusters. In Luís Lopes, Julius Žilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesús Carretero, Jens Breitbart, and Michael Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 121–132. Springer International Publishing, 2014.
➤ Cited on page 107.
- [24] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of Intel Threading Building Blocks. *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 57–66, 2008.
➤ Cited on page 61.
- [25] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19, 1965.
➤ Cited on page 3.
- [26] James O. Coplien. Curiously Recurring Template Patterns. *C++ Rep.*, 7(2):24–27, 1995.
➤ Cited on page 60.
- [27] Intel Corporation. Intel Advanced Vector Extensions Programming Reference. <https://software.intel.com/file/36945>.
➤ Cited on page 36.
- [28] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1997.
➤ Cited on pages 3 and 33.
- [29] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.
➤ Cited on page 61.
- [30] Murray S. Daw and M. I. Baskes. Embedded-atom Method: Derivation and Application to Impurities, Surfaces, and other Defects in Metals. *Phys. Rev. B*, 29:6443–6453, 1984.
➤ Cited on page 14.
- [31] Murray S. Daw, Stephen M. Foiles, and Michael I. Baskes. The embedded-atom Method: a Review of Theory and Applications. *Materials Science Reports*, 9(7–8):251–310, 1993.
➤ Cited on page 14.

- [32] Louis De Broglie. *Recherches sur la théorie des Quanta*. PhD thesis, Migration - université en cours d'affectation, 1924.
➤ Cited on page 7.
- [33] Ram Devanathan, Nagesh Idupulapati, Marcel D. Baer, Christopher J. Mundy, and Michel Dupuis. Ab Initio Molecular Dynamics Simulation of Proton Hopping in a Model Polymer Membrane. *The Journal of Physical Chemistry B*, 117(51):16522–16529, 2013.
➤ Cited on page 7.
- [34] Karen D Devine, Erik G Boman, Robert T Heaphy, Bruce A Hendrickson, James D Teresco, Jamal Faik, Joseph E Flaherty, and Luis G Gervasio. New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2):133–152, 2005.
➤ Cited on page 28.
- [35] Jack Dongarra. Preface: Basic Linear Algebra Subprograms Technical (Blast) Forum Standard I. *International Journal of High Performance Computing Applications*, 16(1):1–111, 2002.
➤ Cited on page 58.
- [36] Jack Dongarra. Preface: Basic Linear Algebra Subprograms Technical (Blast) Forum Standard II. *International Journal of High Performance Computing Applications*, 16(2):115–199, 2002.
➤ Cited on page 58.
- [37] Jack J. Dongarra, Piotr Luszczek, and Antoine Petit. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.
➤ Cited on page 3.
- [38] Matthieu Dreher and Bruno Raffin. A Flexible Framework for Asynchronous In Situ and In Transit Analytics for Scientific Simulations. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Computer Science Press, 2014.
➤ Cited on page 22.
- [39] O. Durand and L. Soulard. Large-scale molecular dynamics study of jet breakup and ejecta production from shock-loaded copper with a hybrid method. *Journal of Applied Physics*, 111(4):–, 2012.
➤ Cited on pages 88, 113, and 114.
- [40] O. Durand and L. Soulard. Power law and exponential ejecta size distributions from the dynamic fragmentation of shock-loaded Cu and Sn metals under melt conditions. *Journal of Applied Physics*, 114(19):–, 2013.
➤ Cited on pages 113 and 114.
- [41] O. Durand and L. Soulard. Mass-velocity and size-velocity distributions of ejecta cloud from shock-loaded tin surface using atomistic simulations. *Journal of Applied Physics*, 117(16):–, 2015.
➤ Cited on pages 85, 87, and 114.
- [42] Tina Düren, Lev Sarkisov, Omar M. Yaghi, and Randall Q. Snurr. Design of New Materials for Methane Storage. *Langmuir*, 20(7):2683–2689, 2004.
➤ Cited on page 22.
- [43] H. Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. Manycore performance-portability: Kokkos multidimensional array library. *Scientific Programming*, 20, 2012.
➤ Cited on page 30.
- [44] P. P. Ewald. Die Berechnung optischer und elektrostatischer Gitterpotentiale. *Annalen der Physik*, 369(3):253–287, 1921.
➤ Cited on page 11.
- [45] ExMatEx. CoMD Proxy Application. <http://www.exmatex.org/cmd.html>.
➤ Cited on page 29.
- [46] J.-L. Fattebert, D.F. Richards, and J.N. Glosli. Dynamic load balancing algorithm for molecular dynamics based on Voronoi cells domain decompositions. *Computer Physics Communications*, 183(12):2608–2615, 2012.
➤ Cited on pages 86 and 91.
- [47] Tristan Ferroir, Leonid Dubrovinsky, Ahmed El Goresy, Alexandre Simionovici, Tomoki Nakamura, and Philippe Gillet. Carbon polymorphism in shocked meteorites: Evidence for new natural ultrahard phases. *Earth and Planetary Science Letters*, 290(1–2):150–154, 2010.
➤ Cited on page 24.
- [48] S. M. Foiles, M. I. Baskes, and M. S. Daw. Embedded-atom Method Functions for the FCC Metals Cu, Ag, Au, Ni, Pd, Pt, and their Alloys. *Phys. Rev. B*, 33:7983–7991, 1986.
➤ Cited on page 14.
- [49] Peter L. Freddolino, Anton S. Arhipov, Steven B. Larson, Alexander McPherson, and Klaus Schulten. Molecular Dynamics Simulations of the Complete Satellite Tobacco Mosaic Virus. *Structure*, 14(3):437–449, 2006.
➤ Cited on pages 22 and 23.
- [50] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O’ Reilly & Associates, Inc., 2004.
➤ Cited on page 49.
- [51] Timothy C. Germann, Kai Kadau, and Peter S. Lomdahl. 25 Tflop/s Multibillion-atom Molecular Dynamics Simulations and Visualization/analysis on BlueGene/L. In *Proceedings of the 2005 IEEE/ACM Conference on Supercomputing*, 2005.
➤ Cited on pages 22 and 23.
- [52] Timothy C. Germann, Kai Kadau, and Sriram Swaminarayan. 369 Tflop/s Molecular Dynamics Simulations on the Petaflop Hybrid Supercomputer

- 'Roadrunner'. *Concurrency and Computation : Practice and Experience*, 21(17):2143–2159, 2009.
➤ Cited on page 29.
- [53] Josiah Willard Gibbs. *Elementary Principles in Statistical Mechanics*. Cambridge University Press, 2010.
➤ Cited on page 10.
- [54] L. Greengard and V. Rokhlin. A Fast Algorithm for Particle Simulations. *Journal of Computational Physics*, 73(2):325–348, 1987.
➤ Cited on page 12.
- [55] William R. Hamilton. On a General Method in Dynamics. *Philosophical Transactions of the Royal Society of London*, 124:247–308, 1834.
➤ Cited on page 8.
- [56] William R. Hamilton. Second Essay on a General Method in Dynamics. *Philosophical Transactions of the Royal Society of London*, 125:95–144, 1835.
➤ Cited on page 8.
- [57] Kenji Harafuji, Taku Tsuchiya, and Katsuyuki Kawamura. Molecular dynamics simulation for evaluating melting point of wurtzite-type GaN crystal. *Journal of Applied Physics*, 96(5):2501–2512, 2004.
➤ Cited on page 22.
- [58] Amy Henderson and Jim Ahrens. *The Paraview guide : a parallel visualization application*. Kitware, Inc., 2004.
➤ Cited on page 56.
- [59] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving Performance via Mini-applications. Technical report, Sandia National Laboratories, September 2009.
➤ Cited on pages 27 and 29.
- [60] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, second edition, 2002.
➤ Cited on pages 2 and 9.
- [61] P. J. Hoogerbrugge and J. M. V. A. Koelman. Simulating Microscopic Hydrodynamic Phenomena with Dissipative Particle Dynamics. *EPL (Europhysics Letters)*, 19(3):155, 1992.
➤ Cited on page 42.
- [62] William G. Hoover, Anthony J. De Groot, Carol G. Hoover, Irving F. Stowers, Toshio Kawai, Brad Lee Holian, Taisuke Boku, Sigeo Ihara, and J. Belak. Large-scale Elastic-plastic Indentation Simulations via Nonequilibrium Molecular Dynamics. *Phys. Rev. A*, 42(10):5844–5853, 1990.
➤ Cited on pages 22 and 23.
- [63] Intel Corporation. Intel® Threading Building Blocks. <https://www.threadingbuildingblocks.org/>.
➤ Cited on page 61.
- [64] I.T. Todorov and W. Smith. *DL_POLY 4 User Manual*. STFC Daresbury Laboratory, 2015.
➤ Cited on page 27.
- [65] James R. Bulpin and Ian A. Pratt. Multiprogramming Performance of the Pentium 4 with Hyper-Threading. In *Third Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD2004)*, pages 53–62, 2004.
➤ Cited on page 35.
- [66] James Reinders. AVX-512 instructions on Intel Developer Zone. <https://software.intel.com/en-us/blogs/2013/avx-512-instructions>.
➤ Cited on page 36.
- [67] R. A. Johnson. Alloy Models with the Embedded-atom Method. *Phys. Rev. B*, 39:12554–12559, 1989.
➤ Cited on pages 14 and 26.
- [68] J. E. Jones. On the Determination of Molecular Fields. II. From the Equation of State of a Gas. *Proceedings of the Royal Society of London. Series A*, 106(738):463–477, 1924.
➤ Cited on page 12.
- [69] Kai Kadau, Timothy C. Germann, Peter S. Lomdahl, and Brad Lee Holian. Microscopic View of Structural Phase Transitions Induced by Shock Waves. *Science*, 296(5573):1681–1684, 2002.
➤ Cited on page 23.
- [70] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.
➤ Cited on page 89.
- [71] Thomas Kenkmann, Ulrich Hornemann, and Dieter Stöffler. Experimental shock synthesis of diamonds in a graphite gneiss. *Meteoritics & Planetary Science*, 40(9–10):1299–1310, 2005.
➤ Cited on page 24.
- [72] Benedict J. Leimkuhler and Sebastian Reich. *Simulating Hamiltonian dynamics*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2004.
➤ Cited on page 9.
- [73] Benedict J. Leimkuhler, Sebastian Reich, and Robert D. Skeel. Integration Methods for Molecular Dynamics. In Jill P. Mesirov, Klaus Schulten, and De Witt Sumners, editors, *Mathematical Approaches to Biomolecular Structure and Dynamics*, volume 82 of *The IMA Volumes in Mathematics and its Applications*, pages 161–185. Springer New York, 1996.
➤ Cited on page 9.
- [74] Michael Levitt and Arie Warshel. Computer Simulations of Protein Folding. *Nature*, 253:694–698, 1975.
➤ Cited on page 22.

- [75] Peter S. Lomdahl, Pablo Tamayo, N. Gronbech-Jensen, and D.M. Beazley. 50 GFlops molecular dynamics on the Connection Machine-5. In *Supercomputing '93. Proceedings*, pages 520–527, 1993.
➤ Cited on page 29.
- [76] F. London. The general theory of molecular forces. *Trans. Faraday Soc.*, 33:8b–26, 1937.
➤ Cited on page 12.
- [77] Jan H. Los, Luca M. Ghiringhelli, Evert Jan Meijer, and A. Fasolino. Improved long-range reactive bond-order potential for carbon. I. Construction. *Phys. Rev. B*, 72:214102, 2005.
➤ Cited on page 24.
- [78] Jan H. Los, Luca M. Ghiringhelli, Evert Jan Meijer, and A. Fasolino. Erratum: Improved long-range reactive bond-order potential for carbon. I. Construction [Phys. Rev. B 72, 214102 (2005)]. *Phys. Rev. B*, 73:229901, 2006.
➤ Cited on page 24.
- [79] Edward A. Mason. Transport Properties of Gases Obeying a Modified Buckingham (Exp-Six) Potential. *The Journal of Chemical Physics*, 22(2):169–186, 1954.
➤ Cited on page 13.
- [80] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.2. Technical report, 2009.
➤ Cited on pages 49 and 57.
- [81] Philip M. Morse. Diatomic Molecules According to the Wave Mechanics. II. Vibrational Levels. *Phys. Rev.*, 34:57–64, 1929.
➤ Cited on page 12.
- [82] Mark T. Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant V. Kalé, Robert D. Skeel, and Klaus Schulten. NAMD: a Parallel, Object-oriented Molecular Dynamics Program. *International Journal of High Performance Computing Applications*, 10(4):251–268, 1996.
➤ Cited on page 27.
- [83] Shuichi Nosé. A molecular dynamics method for simulations in the canonical ensemble. *Molecular Physics*, 52, 1984.
➤ Cited on page 10.
- [84] Shuichi Nosé. A unified formulation of the constant temperature molecular dynamics methods. *The Journal of Chemical Physics*, 81(1):511–519, 1984.
➤ Cited on page 10.
- [85] Marc N. Offman, Marcin Krol, Israel Silman, Joel L. Sussman, and Anthony H. Futerman. Molecular Basis of Reduced Glucosylceramidase Activity in the Most Common Gaucher Disease Mutant, N370S. *Journal of Biological Chemistry*, 285(53):42105–42114, 2010.
➤ Cited on page 22.
- [86] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0. <http://openmp.org/>, 2013.
➤ Cited on page 61.
- [87] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
➤ Cited on page 36.
- [88] M. Parrinello and A. Rahman. Crystal Structure and Pair Potentials: A Molecular-Dynamics Study. *Phys. Rev. Lett.*, 45:1196–1199, 1980.
➤ Cited on page 11.
- [89] François Pellegrini. Scotch and libScotch 5.1 User's Guide, 2008.
➤ Cited on page 89.
- [90] J. R. Pilkington and S. B. Baden. Partitioning with spacefilling curves. Technical report, CSE Technical Report CS94-349, Dept. Computer Science and Engineering, University of California, San Diego, CA, 1994.
➤ Cited on page 89.
- [91] N. Pineau, L. Soulard, L. Colombet, T. Carrard, A. Pellé, Ph. Gillet, and J. Cléroutin. Molecular dynamics simulations of shock compressed heterogeneous materials. II. The graphite/diamond transition case for astrophysics applications. *Journal of Applied Physics*, 117(11), 2015.
➤ Cited on pages 24 and 25.
- [92] Steve Plimpton. Fast Parallel Algorithms for Short-range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.
➤ Cited on pages 19 and 28.
- [93] Top500 Project. Top500 – The List. <http://www.top500.org>.
➤ Cited on pages 3, 4, and 38.
- [94] Sander Pronk, Szilárd Páll, Roland Schulz, Per Larsson, Pär Bjelkmar, Rossen Apostolov, Michael R. Shirts, Jeremy C. Smith, Peter M. Kasson, David van der Spoel, Berk Hess, and Erik Lindahl. GRO-MACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29(7):845–854, 2013.
➤ Cited on page 28.
- [95] A. Rahman. Correlations in the Motion of Atoms in Liquid Argon. *Phys. Rev.*, 136:A405–A411, 1964.
➤ Cited on page 22.
- [96] Thomson Reuters. Web of Science. <http://www.webofscience.com/>.
➤ Cited on page 21.
- [97] Rockwell International. Rockwell R65Coo/21 Dual CMOS Microcomputer and R65C29 Dual CMOS

- Microprocessor. <http://www.datasheetarchive.com/d1/Scans-055/DSAIH000103824.pdf>.
➤ Cited on page 34.
- [98] Sandia National Laboratories. LAMMPS Molecular Dynamics Simulator. <http://lammps.sandia.gov>.
➤ Cited on page 28.
- [99] F. A. Sapozhnikov, G. V. Ionov, V. V. Dremov, L. Soulard, and O. Durand. The Embedded Atom Model and large-scale MD simulation of tin under shock loading. *Journal of Physics: Conference Series*, 500(3):032017, 2014.
➤ Cited on page 114.
- [100] F.A. Sapozhnikov, V.V. Dremov, G.V. Ionov, I.V. Derbenev, and N.E. Chizhkova. MOLOCH computer code for molecular-dynamics simulation of processes in condensed matter. *EPJ Web of Conferences*, 10, 2010.
➤ Cited on page 30.
- [101] J. David Schall, Paul T. Mikulski, Kathleen E. Ryan, Pamela L. Keating, M. Todd Knippenberg, and Judith A. Harrison. Reactive Empirical Bond-Order Potentials. In Bharat Bhushan, editor, *Encyclopedia of Nanotechnology*, pages 2210–2221. Springer Netherlands, 2012.
➤ Cited on page 15.
- [102] Tamar Schlick. In *Molecular Modeling and Simulation: An Interdisciplinary Guide*, volume 21 of *Interdisciplinary Applied Mathematics*. Springer-Verlag New York, 2 edition, 2010.
➤ Cited on pages 9, 10, and 38.
- [103] H.D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2–3):135–148, 1991.
➤ Cited on page 89.
- [104] W. Smith and T.R. Forester. DL_POLY_2.0: A general-purpose parallel molecular dynamics simulation package. *Journal of Molecular Graphics*, 14(3):136–141, 1996.
➤ Cited on page 27.
- [105] L. Soulard. Application de la dynamique moléculaire classique à la physique des chocs. Mecamat 2015, Aussois.
➤ Cited on page 38.
- [106] L. Soulard. Molecular Dynamics Study of the Microspallation. *The European Physical Journal D*, 50(3), 2008.
➤ Cited on page 30.
- [107] L. Soulard, J. Bontaz-Carion, and J.P. Cuq-Lelandais. Experimental and numerical study of the tantalum single crystal spallation. *The European Physical Journal B*, 85(10), 2012.
➤ Cited on pages 25 and 26.
- [108] L. Soulard, N. Pineau, J. Cléroutin, and L. Colombet. Molecular dynamics simulations of shock compressed heterogeneous materials. I. The porous case. *Journal of Applied Physics*, 117(11), 2015.
➤ Cited on page 38.
- [109] William A. Steele. The physical interaction of gases with crystalline solids: I. Gas-solid energies and properties of isolated adsorbed atoms. *Surface Science*, 36(1):317–352, 1973.
➤ Cited on page 17.
- [110] Frank H. Stillinger and Thomas A. Weber. Computer Simulation of Local Order in Condensed Phases of Silicon. *Phys. Rev. B*, 31:5262–5271, 1985.
➤ Cited on page 15.
- [111] B. Stroustrup. *A Tour of C++*. The C++ in-depth series. Addison-Wesley, 2014.
➤ Cited on page 59.
- [112] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.
➤ Cited on page 59.
- [113] A. P. Sutton and J. Chen. Long-range Finnis-Sinclair Potentials. *Philosophical Magazine Letters*, 61(3):139–146, 1990.
➤ Cited on pages 14 and 15.
- [114] Valerie E. Taylor and Bahram Nour-Omid. A study of the factorization fill-in for a parallel implementation of the finite element method. *International Journal for Numerical Methods in Engineering*, 37(22):3809–3823, 1994.
➤ Cited on page 89.
- [115] J. Tersoff. Empirical Interatomic Potential for Carbon, with Applications to Amorphous Carbon. *Phys. Rev. Lett.*, 61:2879–2882, 1988.
➤ Cited on page 15.
- [116] J. Tersoff. New Empirical Approach for the Structure and Energy of Covalent Systems. *Phys. Rev. B*, 37:6991–7000, 1988.
➤ Cited on page 15.
- [117] Trott, Christian R. and Hammond, Simon D. and Thompson, Aidan P. SNAP: Strong Scaling High Fidelity Molecular Dynamics Simulations on Leadership-Class Computing Platforms. In Kunkel, Julian Martin and Ludwig, Thomas and Meuer, Hans Werner, editor, *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 19–34. Springer International Publishing, 2014.
➤ Cited on page 88.
- [118] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.
➤ Cited on page 27.

- [119] Loup Verlet. Computer "Experiments" on Classical Fluids. II. Equilibrium Correlation Functions. *Phys. Rev.*, 165(1):201–214, 1968.
➤ Cited on page 9.
- [120] Jeffrey S. Vetter. *Contemporary High Performance Computing: From Petascale Toward Exascale*. CRC Press, 2013.
➤ Cited on page 57.
- [121] T. Wainwright and B.J. Alder. Molecular Dynamics Computations for the Hard Sphere System. *Il Nuovo Cimento*, 9(1):116–132, 1958.
➤ Cited on page 23.
- [122] Michael S. Warren and John K. Salmon. A parallel hashed oct-tree N -body algorithm. pages 12–21, 1993.
➤ Cited on page 89.
- [123] D. Wolff and W. G. Rudd. Tabulated Potentials in Molecular Dynamics Simulations. *Computer Physics Communications*, 120(1):20–32, 1999.
➤ Cited on page 11.

List of Tables

2.1	Parameters for Lennard-Jones potential	13
2.2	Parameters for Sutton-Chen potential	15
6.1	Grid Methods	54
6.2	1D representation of Grid Traversals	54
7.1	C RTP vs. Dynamic Polymorphism (I)	61
7.2	Impact of communication overlap	72
7.3	Auto-vectorization limits	75
7.4	Overview of our intrinsic generator	76
7.5	Performance of our intrinsic generator (I)	77
7.6	Performance of our intrinsic generator (II)	77
8.1	Topology Methods	91
9.1	Comparison of CoMD and STAMP against ExaSTAMP	110
10.1	Evolution of some quantity from the 200 million case with the cell size	117
A.1	Desktop Machines	125
A.2	Inti/Cirrus	126
A.3	TERA 100	126
A.4	Airain	126
B.1	Atoms	127
B.2	Lennard-Jones Potential	127
B.3	Sutton-Chen Potential	127
B.4	Sapozhnikov Potential	128

List of Figures

1.1	Connection between experiment, theory and computer simulation	2
1.2	Performance development of Top500 Supercomputers over time	4
2.1	Thermal de Broglie wavelength for different masses and temperatures	8
2.2	Lennard-Jones potential	13
2.3	Truncated and shifted Lennard-Jones potentials	16
2.4	Periodic boundary conditions	16
2.5	Potential for a “wall” boundary condition	17
2.6	Verlet neighbor list and linked-list cell methods	18
2.7	Domain decomposition	20
3.1	Amount of publications for the period 1950–2014	21
3.2	Examples of MD simulations in material science and biology	23
3.3	Sample model of graphite inclusion in copper matrix	24
3.4	Structural evolution of the carbon inclusion structure	25
3.5	Temperature map from MD simulation	26
5.1	SMP architecture	33
5.2	NUMA architecture	34
5.3	A multicore processor	35
5.4	Average number of cores per socket for Top500 supercomputers	35
5.5	Ratio of Top500 supercomputers with an accelerator	37
6.1	ExaSTAMP architecture overview	42
6.2	UML representation of particle-related classes	43
6.3	Decomposition of Verlet Integration Schemes	44
6.4	Classes for time-integration schemes	45
6.5	Potential classes	46
6.6	Ghost layers for pair and EAM potentials	48
6.7	Node and Domain	50
6.8	Communication Manager	52
6.9	Interface and implementation for a domain object	53

6.10	Evolution of Grid design	56
6.11	Input-Output Manager	58
6.12	Specific Grid for GPU accelerated simulations	58
7.1	CRTP vs. Dynamic Polymorphism (II)	62
7.2	Particle List vs Cell Array	64
7.3	AOS vs SOA	65
7.4	AOSOA	66
7.5	Optimizations of neighbor search	70
7.6	Point-to-point Communications	81
8.1	Why load balancing is important (I)	86
8.2	An unbalanced simulation	87
8.3	Load balancing with LAMMPS	87
8.4	Integration of the Topology concept	91
8.5	Overview of a domain after calling ZOLTAN partition function	94
8.6	Why load balancing is important (II)	95
8.7	Evaluation of our cost function against a balance on the number of particles	96
8.8	Overview of ZOLTAN methods	97
8.9	Comparison of ZOLTAN methods (static)	99
8.10	Evaluation of RCB method for different combinations of MPI/TBB (static)	99
8.11	Comparison of ZOLTAN methods (dynamic)	100
8.12	Evolution of efficient time and imbalance rate (dynamic)	101
8.13	Evaluation of RCB method for different combinations of MPI/TBB (dynamic)	101
9.1	Thread scaling on nehalem	105
9.2	Thread scaling on haswell	106
9.3	Thread scaling on KNC	107
9.4	Overall atom throughput	107
9.5	Weak scaling on haswell	108
9.6	Weak scaling on ivybridge	108
9.7	Weak scaling on nehalem	109
9.8	Memory usage	109
10.1	Ejecta Simulation of Metal Under Shock	113
10.2	The 200 million atoms case after one nanosecond	115
10.3	An ejecta simulation with domain distribution at different times	117

List of Algorithms

6.1	Force computation for a pair potential	46
6.2	Force computation for an EAM potential	47
6.3	Neighbor lists construction	47

Listings

7.1	An example of the Curiously Recurring Template Pattern	60
7.2	Inheritance version of Listing 7.1	61
7.3	Parallel region	63
7.4	The ExtArray container: public elements	67
7.5	Neighbor search with symmetrization (I)	69
7.6	Neighbor search with symmetrization (II)	70
7.7	Test Kernels for explicit vectorization	74
7.8	Intrinsic implementation of push kernel for double-precision and SSE instructions .	76
7.9	Implementation of push kernel with our intrinsic generator	77
7.10	Standard prototype for MPI_Gather function	79
7.11	Custom prototype for our MPI_Gather equivalent	79
7.12	Automatic access to MPIData Types	80
8.1	Split function used in ExaSTAMP rectilinear decomposition	90

Contents

Abstract	v
Résumé	vii
Remerciements	ix
Summary	xi
1 Introduction	1
I Molecular Dynamics	5
2 MD Background	7
2.1 Time Integration	7
2.1.1 Classical Molecular Dynamics Equations	7
2.1.2 Verlet Schemes	9
Verlet Integration	9
2.2 Thermodynamical Ensembles	10
2.2.1 NVE	10
2.2.2 NVT	10
2.2.3 NPT	11
2.3 Potentials	11
2.3.1 Short and Long Range Potentials	11
2.3.2 Pair Potentials	12
Lennard-Jones Potential	12
Morse Potential	12
Buckingham and Exp-6 Potentials	13
2.3.3 The Embedded Atom Model	14
2.3.4 Other Potentials	14
2.4 MD Specifics	15
2.4.1 Truncated/Shifted Potentials	15
2.4.2 Boundary Conditions	15

2.4.3	Neighbor Search	18
	Verlet Lists	18
	Linked-Cell Lists	18
2.5	MD on Parallel Computers	19
3	MD Applications: Then and Now	21
3.1	General Applications	22
3.2	Shock Physics	22
3.3	MD at CEA	24
3.3.1	MD Simulations of Shock Compressed Heterogeneous Materials	24
3.3.2	Experimental and Numerical Study of the Tantalum Spallation	25
4	MD Tools	27
4.1	DL_POLY	27
4.2	GROMACS	28
4.3	LAMMPS	28
4.4	MD in Mantevo Project	29
4.4.1	CoMD	29
4.4.2	MiniMD	29
4.5	MOLOCH	30
4.6	STAMP	30
II	ExaSTAMP Project	31
5	Motivations	33
5.1	Computer Evolution	33
5.1.1	From SMP to Hierarchical Memory Systems	33
5.1.2	Multicore/Manycore Processors	34
5.1.3	Vectorization Returns	36
5.1.4	Accelerators	36
	GPU–GPGPU	36
	Hybrid Manycore Processors	36
5.2	Need for Bigger and More Complex Systems	38
5.3	Objectives	38
6	Framework Architecture	41
6.1	Particles	41
6.1.1	Particles	41
6.1.2	Particle Types	42
6.2	Time Integration	43
6.2.1	Decomposition of Verlet Schemes	44
6.2.2	Classes for Integration Schemes	44
6.3	Force Computation	44
6.3.1	Potentials	45
6.3.2	Force Computation Patterns for Potentials	45
	Pair Potential	45

	EAM Potential	46
6.3.3	Neighbor Search	46
6.3.4	Keeping Edges Up to Date	47
6.3.5	Overlapping Communications	48
6.3.6	Symmetrization	49
6.4	Parallel	49
6.4.1	Node and Domain	50
6.4.2	Communication Manager	50
	Collective Communications	51
	Point-to-point Communications	51
6.5	Domain	51
6.6	Grid	52
6.6.1	Cell Traversal	53
6.6.2	Particle Storage	55
	Particle List	55
	Cell Array	55
6.7	Cells	55
6.8	I/Os	56
6.8.1	Quick Diagnostics for Debug	56
6.8.2	A First Checkpoint-Restart System	57
6.8.3	Advanced CR & Outputs	57
	HERCULE and LOVE	57
	I/O Manager	57
6.9	Specific Kernels for GPU Accelerated Simulations	58
7	Implementation and Preliminary Results	59
7.1	Domain, Grid and Particles	59
7.1.1	Domain	59
7.1.2	Grid	59
	Avoiding Overhead Induced by Multiple Virtual Calls	59
	Thread Parallelization	61
7.1.3	Particle List and Cell Array	62
	Particle List	62
	Cell Array	64
7.1.4	Particle Storage	64
	Particle List: AOS vs. SOA	64
	Cell Array: AOSOA	65
7.1.5	Container Used in Cells	67
7.2	Force Computation	68
7.2.1	Neighbor Search	68
	Simple Implementation	68
	Neighbor Cell Differentiation	69
	Test Refactoring in Symmetrization Case	69
	Other Optimizations	69
7.2.2	Force Computation	70

7.2.3	Communications Overlap	71
7.2.4	Enabling Symmetrization	72
7.3	Explicit Vectorization	73
7.3.1	Test Kernels	73
7.3.2	Auto-Vectorization Limits	73
7.3.3	A Template Intrinsic Generator	75
	Intrinsics	75
	Intrinsic Wrapping	76
	Performance of the Intrinsic Generator	77
7.4	Communications	78
7.4.1	Towards More Intuitive MPI Prototypes	78
	A Tool to Handle MPI Data Types	79
	Adding New Types	79
7.4.2	Point-to-point Communications	81
	On Domain: MessageSend	81
	On Node: Session part I (send)	82
	Back On Domain	83
	On Node: Session part II (collect)	83
	On Domain: Clean	83
8	Dynamic Load Balancing	85
8.1	Load Balancing in MD	85
8.1.1	Why It Is Important	85
8.1.2	Load Balancing in MD codes	86
8.2	Load Balancing in ExaSTAMP	88
8.2.1	The ZOLTAN Library	88
8.2.2	Integration of a Load Balancing Capability in ExaSTAMP	89
	Any Decomposition	89
	Architecture Modifications	90
	Avoiding Code Duplication	90
	Cost Function	91
	Trigger a Load Balancing Step	92
8.2.3	Implementation	92
	Any Topology	92
	ZOLTAN Query Functions	93
	Grid Rebuilding After a Balancing	93
8.2.4	Conclusion	94
8.3	Evaluation	95
8.3.1	First Tests	95
	Back On First Example	95
	Why our Cost Function is Better than Counting Particles	95
	ZOLTAN Methods Overview	97
8.3.2	Static Partitioning	98
8.3.3	Dynamic Partitioning	99
8.3.4	Conclusion	100

III	Validation	103
9	Overall Performance	105
9.1	On a Single Compute Node	105
9.1.1	Strong Scaling with TBB	105
9.1.2	Overall Atom Throughput	106
9.2	On a Large Number of Cores	108
9.2.1	Weak Scaling	108
9.2.2	Benefit of Threads on Memory Usage	109
9.3	Comparison with other MD Software	110
9.4	Conclusion	110
10	Ejecta Simulations of Metal Under Shock	113
10.1	Context	113
10.2	Adjustments for Production Runs	114
10.2.1	A Specific Potential for Tin	114
10.2.2	Free Boundary Conditions	115
10.2.3	Issues with Load Balancing	115
10.3	Results	116
10.3.1	Accuracy	116
10.3.2	Performance	116
	The 200 million case	116
	The 700 million case	117
10.3.3	Conclusion	117
11	Conclusion	119
	Appendix	123
A	Processors Used in our Tests	125
B	Atoms and Potentials Parameters	127
	Résumé Détaillé	129
	Bibliography	135
	List of Tables	143
	List of Figures	145
	List of Algorithms	147
	Listings	149
	Contents	151



In the exascale race, supercomputer architectures are evolving towards massively multi-core nodes with hierarchical memory structures and equipped with larger vectorization registers. These trends tend to make MPI-only applications less effective, and now require programmers to explicitly manage low-level elements to get decent performance.

In the context of Molecular Dynamics (MD) applied to condensed matter physics, the need for a better understanding of materials behaviour under extreme conditions involves simulations of ever larger systems, on tens of thousands of cores. This will put molecular dynamics codes among software that are very likely to meet serious difficulties when it comes to fully exploit the performance of next generation processors.

This thesis proposes the design and implementation of a high-performance, flexible and scalable framework dedicated to the simulation of large scale MD systems on future supercomputers. We managed to separate numerical modules from different expressions of parallelism, allowing developers not to care about optimizations and still obtain high levels of performance. Our architecture is organized in three levels of parallelism: domain decomposition using MPI, thread parallelization within each domain, and explicit vectorization. We also included a dynamic load balancing capability in order to equally share the workload among domains.

Results on simple tests show excellent sequential performance and a quasi linear speedup on several thousands of cores on various architectures. When applied to production simulations, we report an acceleration up to a factor 30 compared to the code previously used by CEA's researchers.

Molecular Dynamics for Exascale Supercomputers

Thèse CEA/DAM/DIF – Université de Bordeaux

Emmanuel CIEREN

Sous la direction de Raymond NAMYST & Laurent COLOMBET

2012 – 2015

Dans la course vers l'*exascale*, les architectures des supercalculateurs évoluent vers des nœuds massivement multicœurs, sur lesquels les accès mémoire sont non-uniformes et les registres de vectorisation toujours plus grands. Ces évolutions entraînent une baisse de l'efficacité des applications *homogènes* (MPI simple), et imposent aux développeurs l'utilisation de fonctionnalités de bas-niveau afin d'obtenir de bonnes performances.

Dans le contexte de la dynamique moléculaire (DM) appliqué à la physique de la matière condensée, les études du comportement des matériaux dans des conditions extrêmes requièrent la simulation de systèmes toujours plus grands avec une physique de plus en plus complexe. L'adaptation des codes de DM aux architectures exaflopiques est donc un enjeu essentiel.

Cette thèse propose la conception et l'implémentation d'une plateforme dédiée à la simulation de très grands systèmes de DM sur les futurs supercalculateurs. Notre architecture s'organise autour de trois niveaux de parallélisme: décomposition de domaine avec MPI, du *multithreading* massif sur chaque domaine et un système de vectorisation explicite. Nous avons également inclus une capacité d'équilibrage dynamique de charge de calcul. La conception orienté objet a été particulièrement étudiée afin de préserver un niveau de programmation utilisable par des physiciens sans altérer les performances.

Les premiers résultats montrent d'excellentes performances séquentielles, ainsi qu'une accélération quasi-linéaire sur plusieurs dizaines de milliers de cœurs. En production, nous constatons une accélération jusqu'à un facteur 30 par rapport au code utilisé actuellement par les chercheurs du CEA.

université
de BORDEAUX