Combining Task-based Parallelism and Adaptive Mesh Refinement Techniques in Molecular Dynamics Simulations

Raphaël Prat, Laurent Colombet, Raymond Namyst

Combining Task-based Parallelism and Adaptive Mesh Refinement Techniques in Molecular Dynamics Simulations

Raphaël Prat
Commissariat à l'Energie Atomique
Arpajon, France
raphael.prat@cea.fr

Laurent Colombet
Commissariat à l'Energie Atomique
Arpajon, France
laurent.colombet@cea.fr

Raymond Namyst
University of Bordeaux, Inria
Talence, France
raymond.namyst@u-bordeaux.fr

ABSTRACT

Modern parallel architectures require applications to generate massive parallelism so as to feed their large number of cores and their wide vector units. We revisit the extensively studied classical Molecular Dynamics N-body problem in the light of these hardware constraints. We use Adaptive Mesh Refinement techniques to store particles in memory, and to optimize the force computation loop using multi-threading and vectorization-friendly data structures. Our design is guided by the need for load balancing and adaptivity raised by highly dynamic particle sets, as typically observed in simulations of strong shocks resulting in material micro-jetting. We analyze performance results on several simulation scenarios, over nodes equipped by Intel Xeon Phi Knights Landing (KNL) or Intel Xeon Skylake (SKL) processors. Performance obtained with our OpenMP implementation outperforms state-of-the-art implementations (LAMMPS) on both steady and micro-jetting particles simulations. In the latter case, our implementation is 4.7 times faster on KNL, and 2 times faster on SKL.

KEYWORDS

Adaptive Mesh Refinement, Task Parallelism, Molecular Dynamics

ACM Reference Format:
Raphaël Prat, Laurent Colombet, and Raymond Namyst. 2018. Combining Task-based Parallelism and Adaptive Mesh Refinement Techniques in Molecular Dynamics Simulations. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225085

1 INTRODUCTION

The High Performance Computing community is currently witnessing a significant increase of the number of cores in parallel supercomputers, which are already featuring more than 10 millions of cores. This increase of the parallelism degree is a clear trend that unfortunately comes with a decrease of the amount of memory per core, as well as a strong pressure on the use of highly vectorized code. The Intel Knights Landing processor (KNL) [36] and Intel Skylake processor (SKL) [10], for instance, require respectively from 64 to 256 and 48 to 96 threads to utilize the cores at their full capacities, as well as 512-bit wide SIMD instructions (i.e. 16 floats). This explains why many ongoing efforts are devoted to designing new implementations of parallel simulations to meet these hardware constraints, which require to generate more parallelism, more vectorized code and to enforce stronger data locality.

Molecular Dynamics (MD) N-body simulations, which represent the most accurate method to simulate complex materials by operating at a microscopic level, are naturally good candidates for exploiting most of the underlying hardware's potential, because they are intrinsically data-parallel. Optimization of such simulations over parallel supercomputers has indeed been extensively studied, and very efficient implementations are available over a wide range of modern hardware. One of the key aspect of an efficient implementation of an N-body simulation is to speedup, for each particle, the lookup of neighbor particles which potentially exerts a force on the considered particle. To this end, several methods are used by state-of-the-art software, most notably the Verlet list [37] method or the cell-list one [1]. These methods perform indeed very efficiently under regular conditions, that is, when particles are uniformly spread over the domain. For extreme shock simulations where many particles have high velocities, these methods lead to either frequently recompute neighbor lists or to cope with a massive number of empty cells.

This paper investigates the use of Adaptive Mesh Refinement techniques [3, 4] in N-body simulations. In order to cope with regions of highly variable particle density, we partition the domain in several subdomains organized as OCTREES, which are periodically refreshed. Each OCTREE is intended to be processed sequentially on a single core, and we carefully enforce the execution order so that no adjacent OCTREES can concurrently access a shared cell, leading to a completely lock-free algorithm. To maximize the efficiency of particles interactions computation over recent manycore architectures, such as Intel Xeon Phi KNL, we use a careful data layout for particle attributes that improves data locality and cache utilization, and maximizes code vectorization opportunities. We evaluate our approach using an extended implementation of the EXASTAMP MD simulator [31] over a Intel Xeon Phi and Intel Xeon. We use different input configurations and show that our approach outperforms state-of-the-art solutions while using a similar memory footprint.

The main contributions of this paper are:

- We introduce a new software architecture for MD including AMR techniques;
- We present a thread parallelization strategy adapted to the AMR structure;
- We evaluate our implementation of the AMR on two different architectures using both a static and a highly dynamic test case.

The remainder of this paper is organized as follows. In the next Section, we briefly recall the main concepts used in Molecular

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225085

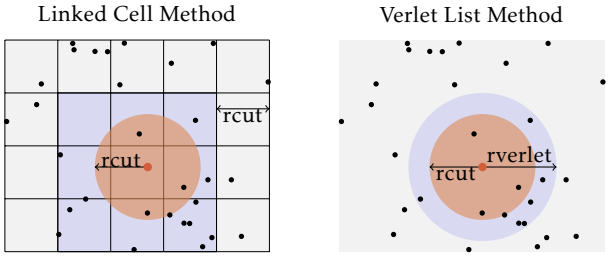

Figure 1: Standard methods to build atom neighbors list.

Dynamics simulations and in Adaptive Mesh Refinement based applications. Section 3 describes the main ideas behind our approach. We demonstrate the relevance of our approach in Section 4 comparing our implementation with state-of-the-art software on several benchmarking scenarios. We present related work in Section 5, and dress concluding remarks in Section 6.

2 BACKGROUND

2.1 Molecular Dynamics Simulations

The classical MD is a computational method used to study dynamical properties of a systems made of particles. This method consists in numerically integrating Newton's equation of motion $\vec{\mathbf{f}} = \mathbf{m}.\vec{\mathbf{a}}$ [1], where the force on a particle depends on the interactions with all others [1]. The force on particles is given by the gradient of a potential $\vec{\mathbf{f}} = -\nabla\vec{\mathbf{U}}$. MD simulations typically follow an iterative process, where each iteration (time step) consists in computing forces applied to each particle, deducing each particle's acceleration, updating particles velocities, and finally updating particles positions. To do so, the most widely adopted scheme is the *Velocity Verlet* integrator [21].

2.1.1 Particle interactions. In most MD simulations, particles are processed as points and the interacting force between particles is approximated as a gradient of a potential that depends on the particles relative positions. The force computation is obviously the most challenging part: it contains all the physics of the simulation and can take up to 95% of the total time. Potentials from classical MD are empirical or semi-empirical, computed from an analytical formula or interpolated from tabulated values [38].

In this paper, we focus on short-range interactions neglected beyond a given distance **RCUT** called the cutoff distance. This approximation is relevant for solid materials, in which distant atoms are "screened" by closer atoms.

2.1.2 Lists of Neighbors. To speed up the force computation step, most MD simulations maintain a list of neighbors for each atom. To build such lists efficiently, two methods (see figure 1) are typically used and can even be combined. The first is based on Verlet lists [37]. A radius of Verlet (RVERLET) is added to RCUT. As a consequence, the list of neighbors contains "useless" atoms which are too far from the considered atom. However, as long as no atom has moved from its original position[2] by more than $\frac{1}{2}$ RVERLET, there is not need to recompute the lists. This method is therefore the most efficient when atoms are moving very slowly, because the Verlet lists are not frequently updated. This method have a complexity of $O(N^2)$, and can easily be performed in parallel. The other one is the linked cells method [1]. The domain is divided into RCUT sized cells on a grid. For all atoms, the neighborhood of an atom is included in the (27 in 3D) neighboring cells. This method avoids the cost of maintaining lists of neighbors per atom. However, it comes at the cost of considering a significant amount of atoms which are beyond the RCUT distance. This method has a complexity of $O(N)$. The strategy of building neighbor lists can influence the simulation performance because it requires a lot of memory accesses.

2.1.3 Pair Potentials. Pair potentials are a particular but extremely widely used case of potentials, where interactions between particles are reduced to pairwise interactions. Therefore, the potential between two particles will only depend on the distance between those particles. Pair potentials are quite effective to describe low-density materials such as liquids and gases. Some common pair potentials are for instance, the Lennard-Jones potential (LJ) [19], the Morse potential [28] and the Exponential-six potential [24]. Although it was first designed to study gases, the Lennard-Jones potential (LJ) has been used in a large part of materials science and has become a standard benchmark for MD codes.

2.2 Adaptive Mesh Refinement (AMR)

The first adaptive mesh refinement was developed by Berger [3, 4] for hydrodynamics application in 1984 to optimize time computation. In hydrodynamics, the domain is discretized into a Cartesian grid where the solution is computed in each point of the grid. The spacing of the grid points determines the local error and hence the accuracy of the solution. AMR consists in coarsening the grid points where the points do not need to be spaced to keep their stability. The error is negligible at these points. The coarsening is determined by a criterion based on density or on a Richardson extrapolation [4, 33]. Note that different levels of refinement may be obtained for a simulation. This method is interesting for simulations with heterogeneous density to reduce computational cost for some parts of the simulation.

AMR methods have already been introduced in N-body simulations in the past. In such simulations, the domain is decomposed into a grid of cells, which are equivalent to points in hydrodynamics. The AMR consists in coarsening the cells containing only a few atoms, using criteria such as the number of particles or average density. A large amount of AMR software have been developed [13] for various physical problems such as astrophysics with Enzo [7] or BoxLib [22], Chombo [8] to solve PDE.

3 ADAPTIVE MESH REFINEMENT FOR MOLECULAR DYNAMICS SIMULATIONS

To meet the needs of highly dynamic particles simulations over many core architectures, we introduce a new approach inherited from Adaptive Mesh Refinement Methods (AMR) to perform MD simulations. We first describe the main characteristics of the ExaStamp framework (Section 3.1) that has served as a basis for our implementation. Section 3.2 presents the main data structures used

[1] $\vec{\mathbf{f}}$ force, $\mathbf{m}$ mass, $\vec{\mathbf{a}}$ acceleration, $\vec{\mathbf{U}}$ potential energy
[2] The original position is the one captured when updating the list of neighbors.

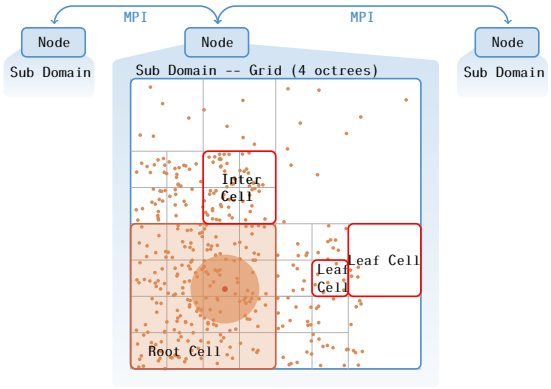

Figure 2: MD simulation with AMR architecture and two levels of refinement. A node is divided into OCTREES and the OCTREES are refined in leaf or inter cells.

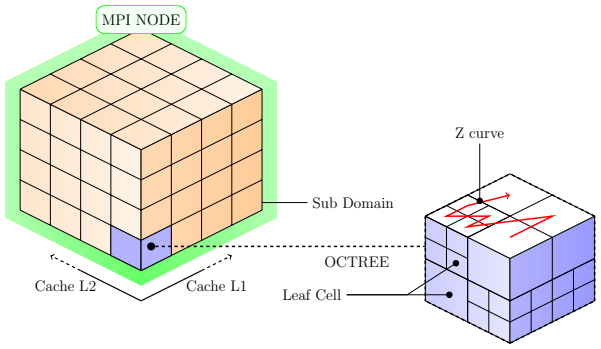

Figure 3: The domain simulation (3D representation of Figure 2) is decomposed into sub-domains that are distributed across MPI nodes, each sub-domain being a grid of OCTREES.

to efficiently cope with highly heterogeneous sets of particles. In Section 3.3, we detail how the workload is dynamically assigned to the underlying computing resources.

3.1 The ExaStamp MD Framework

EXASTAMP is a MD code which has been recently developed at CEA for the upcoming generation of supercomputers. The main goal of EXASTAMP is to cope with simulations featuring up to several billions of atoms on thousands of cores. The code is written in C++ and uses hybrid parallelization mixing MPI and threads (OpenMP/ INTEL TBB). EXASTAMP kernels are written by physicists using high-level types and functions, and the generated code features INTRINSICS instructions to minimize missed vectorization opportunities. These techniques have most notably proved to be effective on the Intel XEON PHI KNIGHTS CORNER processor.

Like many MD software packages, EXASTAMP relies on the linked-cell method to organize particles, that is, a static mesh of cells that does not evolve during the simulation. Although this strategy exhibits high performance when particles are homogeneously distributed across the simulation domain, it is not suitable for extreme dynamic micro-jetting [12] configurations. Indeed, between two time steps, most atoms move from one cell to another, leaving a large number of cells almost empty. Dealing with those empty cells incurs significant memory and performance overheads, because their number can sometimes exceed the number of atoms.

3.2 Software Architecture

Our approach combines the use of several techniques such as the octree's structure, the Structure-of-Array, the Morton index and a vector-friendly handling in order to optimize access to L1 and L2 caches.

3.2.1 Adaptive Mesh Refinement. We apply AMR techniques to the management of cells, so as to deal with cells of different sizes, depending on the particles density. Figure 2 illustrates how the spatial domain is decomposed into cells [32]. The domain is first decomposed into sub-domains, each sub-domain being assigned to one MPI process. Sub-domains are in turn decomposed into a grid of OCTREES. The reason for using a grid of OCTREE instead of a large, unique OCTREE is that an OCTREE serves a unit of work that can be exchanged between MPI processes for load balancing purpose. OCTREES are built and refreshed using a refinement algorithm: each cell is recursively divided into 8 cells of same volume if some refinement criteria are fulfilled. The maximal depth D^{max} limits the number of times OCTREES can be refined. Therefore each OCTREE contains between 1 and $8^{D^{max}}$ cells. As shown in Figure 2, a cell can either be a ROOT cell (also called level 0 cell), a LEAF cell or an INTER cell. Note that the minimal size of a LEAF cell corresponds to the size that would be used in a classical linked-cell method, in accordance with the maximum cut-off distance.

The way OCTREES are assigned to MPI processes is beyond the scope of this paper. Inside MPI processes, thread-level parallelism is used to explore the grid of OCTREES, as it will be detailed in the following Subsection. Given that the number of OCTREES is larger than the number of cores by several orders of magnitude, one important aspect of our strategy is that each OCTREE is assigned to a single thread and is thus processed sequentially. This allows us to optimize the OCTREE traversal with respect to cache usage [14, 15]: LEAF cells are explored along a Z-order curve to minimize cache misses, as illustrated in Figure 3 (red path).

3.2.2 Data layout. Inside an octree, all atoms' attributes are stored in the ROOT cell, in a Structure-of-Arrays (SOA) layout. For the sake of clarity, we simply refer to this data structure as the "array of atoms" in the remaining of the paper. INTER and LEAF cells only store indexes indicating the position of atoms in this array, as illustrated in Figure 4. This strategy prevents the copy of atoms during the refinement, improves the data locality between cells inside an OCTREE and reduces data movements because transfers are performed between OCTREES instead of cells [11]. In addition, because the number of arrays is reduced, they are filled with more atoms, which improves vectorization efficiency.

For each atom, we compute a Morton index [29] out of the cartesian position of its smallest bounding cell. This index gives the path from the ROOT cell to the LEAF containing the atom. The array of atoms is sorted by the atoms' Morton index, to improve cache locality [25].

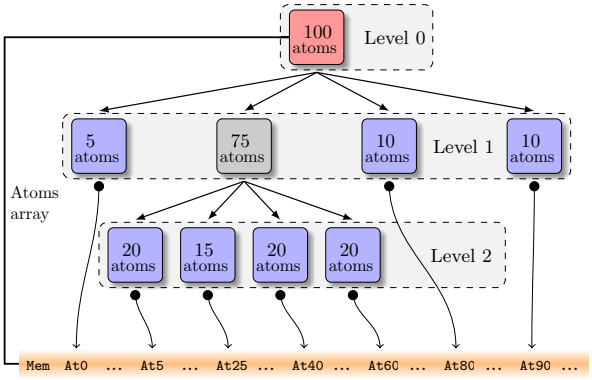

Figure 4: Data management in a OCTREE. The array containing all atoms information (orange color) is allocated into the ROOT cell (red) and atoms (At) are sorted by their morton index. The refined cells INTER (gray) and LEAF (blue) point on this atoms array. This strategy avoids the copy of data during refinement/coarsening steps.

3.2.3 Vector-friendly Particles Handling. To maximize vectorization opportunities inside the computation step, one technique consists in using a vectorization buffer to pack information about neighboring atoms in contiguous areas of memory. While this allows the compiler to generate efficient, vectorized instructions to process data stored in such buffers, the cost of introducing an extra copy of data has a negative impact on performance in the case of low cost potentials (e.g. Lennard Jones). Another technique is to access the data through in a vector-friendly manner. More precisely, we divide the set of atoms into a series of blocks (for which the size is a multiple of the processor's vector width), which is illustrated in Figure 5. We only process blocks that contain at least one atom included in the Verlet radius, while others are skipped. In the worst case, we could end up with blocks containing only one useful atom, which would lead to process only one single atom at a time. In practice, however, we observe that vector registers are filled with more than 50% of useful values. This implementation variant of EXASTAMP will be called AMR Vec in the remaining of this paper.

3.3 Parallelization Strategy

Inside each MPI node, we use OpenMP to process cells in parallel. As mentioned previously, since the number of OCTREES is significantly larger than the number of cores, each OCTREE is assigned to a single OpenMP thread, and thus is processed sequentially. In our current implementation, we spawn one OpenMP task per OCTREE.

When the potential has symmetrical properties (third newton's law), forces exerted between two particles are equal and are thus computed once only. This saves some precious computing cycles, but requires as a counterpart to use costly synchronization tools (e.g. mutexes) to avoid that multiple threads update the same target particle. Another strategy is to process the cells in an order that ensures that no particle can be simultaneously updated by different threads. The wave method was introduced by Meyer et

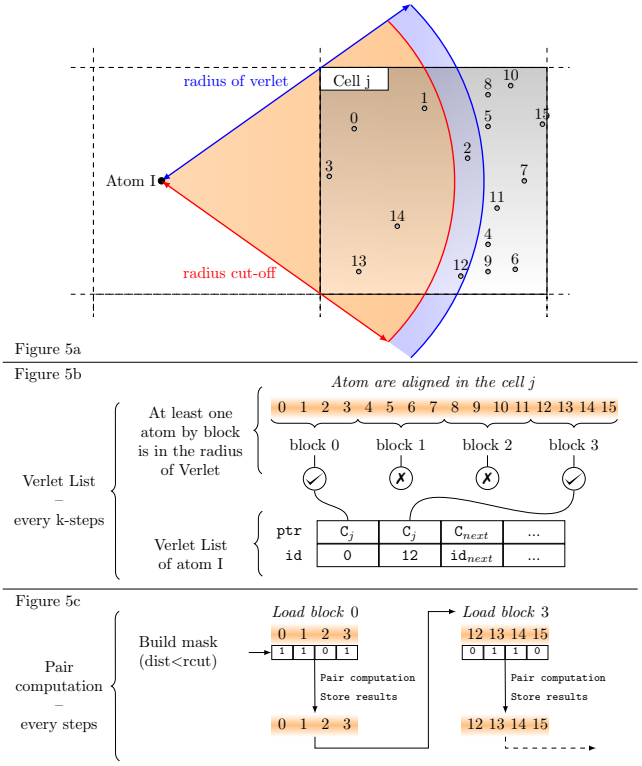

Figure 5: Atom I interacts with some of the atoms inside cell j (a). The Verlet lists are filled (b) by the first indexes of blocks of a vector register size (4 doubles in avx2 256-bit) and containing a minimum of one atom in the radius of Verlet. During the potential step (c), the atoms are loaded by blocks and could be pre-fetched.

al. [23, 26, 34] as a mean to avoid these synchronizations. The idea is to partition cells – or OCTREES in our case – into a set of waves such that two cells of the same wave are far enough from each other to have no neighbor in common. Thus, during the computation of a given wave, neighbor particles can be updated without synchronization (see Figure 6).

Note that there is no need to strictly execute waves one after the other: their execution can actually overlap if it relies on a task-based scheduler able to take inter-task dependencies into account (such as Intel TBB or OpenMP). During the execution of wave n, tasks of wave $n+1$ will be progressively unlocked. Using dependency graph allows to unlock tasks as quickly as possible. We use OpenMP 4.5 to generate explicit tasks with dependencies. In our case, each task represents an OCTREE to compute (see Figure 7), and not just a cell. This method is equivalent to the cell task blocking method [34] and limits the number of waves to 8 but requires that D^{max} is superior to one. This approach is similar to the Particle and Separate-Calculation (PSC) [17].

Finally, compared to the original version of the wave method, OCTREES helps to control the granularity of tasks, and thus helps to minimize the overhead of generating OpenMP tasks.

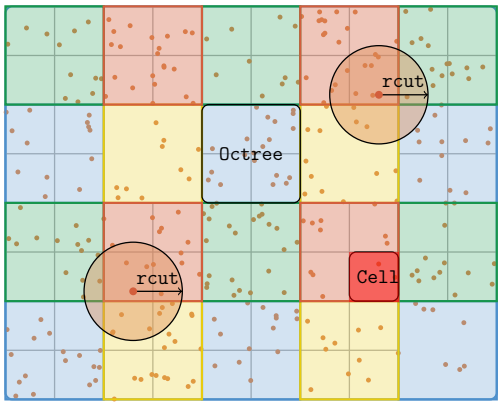

Figure 6: 2D representation of a part of the MD simulation. The atoms are distributed into the cells themselves included into octrees with a tree depth of 1. The octrees are colored in 4 colors such that two atoms included in two octrees of the same color do not interact together.

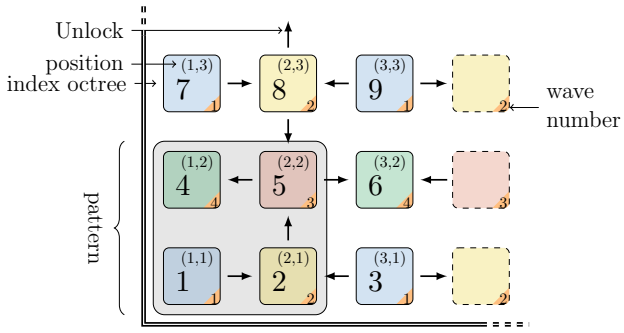

Figure 7: Task dependency graph in 2D. octree are classified into 4 waves (8 in 3D), cells classified in the wave $(n)_{n>=0}$ are unlocked by specific cells in the wave $(n-1)_{n>=1}$.

4 EVALUATION

To assess the effectiveness of our approach, we present a performance study using two test cases: *bulk material*, a steady configuration of atoms, and *micro-jetting*, a highly dynamic one. We compare our implementation (ExaStamp extended with amr) against the Lammps [20] reference package and its mini-application miniMD [16].

4.1 Experimental Framework

Our experimental framework is composed of two architectures. The first is a knl processor featuring 68 cores running at 1.4GHz (4 of which being dedicated to the operating system). Each core has a 2-wide out-of-order pipeline with support for 4 hardware threads and 512-bit vector units. Its memory system consists of a 96 GB DDR4 off-chip memory and a 16 GB MCDRAM stacked memory. The second architecture is a skl-based machine that contains 2 sockets of 24 cores running at 2.7GHz. Each core has a 2-wide out-of-order pipeline with support for 2 hardware threads and 512-bit vector units. It is equipped with 380 GB DDR4 off-chip memory. The same Intel ICC v18 compiler was used on both platforms. We used the following compile flags: -O3, avx 512-bit flag, and `-qopt-zmm-usage=high` on skl. We launched the programs in native mode using the `numactl -p 1` command on knl. The KMP affinity environment was set to granularity=thread,scatter for ExaStamp amr and to granularity=core,scatter for miniMD and Lammps.

Lammps is one of the most relevant classical MD simulator designed for parallel computers. The numerous built-in features for 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 billion atoms on tens of thousands cores. Although the standard Lammps version is parallelized using MPI, many packages have been developed to integrate thread parallelization such as USER-OMP or USER-INTEL. The KOKKOS package has been integrated to optimize Lammps on the accelerators (Xeon Phi and GPU). A load balancing has been implemented using the shift method [30] or the recursive coordinate bisection [5] (rcb) method. Despite of specific optimizations for the knl architecture, the OpenMP version of Lammps is 10 to 15% slower than the full-MPI version [18]. Note that Lammps is compiled using the flags recommended by the documentation on Intel Xeon Phi (knl) and Xeon (skl). We have notably used the following Kokkos-runtime flags: *-k on t nbThread -sf kk -pk kokkos newton on neigh half comm no*. Newton's third law (`neigh half`) is used by all codes to take advantage of the symmetry of lj potential.

miniMD is a simplified version of Lammps written in C++. Due to its simplicity (near 5000 lines), it is possible to explore potential optimisations much more rapidly. Despite supporting only the Lennard-Jones (lj) and Embedded-Atom model (Eam) potentials, miniMD's performance and scaling behavior is representative of much larger and more complex codes with hybrid parallelization MPI/OpenMP. Simulation parameters are defined with a simple input file. miniMD is compiled using the flags recommended in the Intel Makefile, and the following runtime flags: *–half_neigh=1 -t nbThread*.

We tried to faithfully use the same experimental setup for miniMD, Lammps and ExaStamp amr. Most notably, each experiment features a warm-up phase of 10 time steps, followed by 90 iterations. An amr refinement/coarsening process is required for the initialisation step. Two test cases were simulated in our experiments. The first one is a *bulk material* corresponding to an ideal crystal (4,000,000 atoms) with an homogeneous density. In constrat, the second one simulates a *micro-jetting* (3,535,584 atoms) occurring after a shock on a material. The number of atoms in these two test cases represents the quantity assigned to a node by the ExaStamp code during simulations with several hundreds of millions of atoms. The lj potential is used because its low computational cost better highlights performance issues related to suboptimal memory accesses or bus contentions (e.g. memory-bound code regions and threads synchronizations). This makes it easier to accurately identify performance issues and evaluate the effectiveness of code optimizations.

Simulation code	million atoms processed per second	
Architecture	Knights Landing	Skylake
Lammps	7.2	36.9
miniMD	8.0	42.9
ExaStamp amr	24.1	52.0
ExaStamp amr Vec	45.4	68.8

Table 1: Number of atoms processed per second (higher is better) by ExaStamp amr with and without Vec option, miniMD and Lammps (mean). Software performed on 64 cores on knl (left) and 48 cores on skl (right).

Software	Potential	Refinement
Lammps knl	96.4%	NA
Lammps skl	88.3%	NA
miniMD knl	97.5%	NA
miniMD skl	91.2%	NA
ExaStamp amr knl	94.8%	0.9%
ExaStamp amr skl	81.1%	1.9%
ExaStamp amr Vec knl	91%	1.7%
ExaStamp amr Vec skl	77.3%	2.6%

Table 2: Time ratio measured for different parts of the bulk of copper simulation: potential kernel (Potential) and related to the amr structure (Refinement). Time ratio of other parts are not reported (Scheme, Reorganisation, Update Ghost). Note that the Verlet lists are not updated.

4.2 Bulk Material: a Steady-State Simulation

The first benchmark consists in simulating a bulk material featuring 4 millions of atoms in thermodynamic equilibrium. This simulation has many properties such as an homogeneous density, which also remains constant throughout the duration of the simulation. The bulk of copper is obtained by replicating an atom pattern named lattice. A Face Cubic Center (FCC) lattice of 3.54Å is used with a suitable lj potential. For each atom, 39 interactions with neighboring atoms are processed due to the 5.68Å cut-off and symmetric property of the lj potential. A radius of Verlet of 0.3Å is added to avoid re-build Verlet lists. Throughout this simulation, the tree depth D^{max} is set to 2 to obtain best performance.

A classical way to evaluate performance of MD simulations is to measure the number of atoms processed per second. The results in Table 1 report performance obtained with Lammps, miniMD and ExaStamp amr on Intel knl and skl architectures. As expected, we observe that miniMD and Lammps exhibit similar performance. ExaStamp amr is 3 times faster than miniMD, while ExaStamp amr Vec is 5.5 times faster than miniMD on the knl architecture. The difference is reduced on the skl from 3 to 1.21 and from 5.5 to 1.6 respectively. This significant difference between ExaStamp amr and the two other codes comes from the performance penalty incurred by OpenMP atomic operations used in the miniMD and Lammps kernels when updating neighboring atoms. In ExaStamp amr, the use of a wave-based task scheduling, presented in Section 3.3, avoids using such costly synchronization instructions. Moreover, the use of atomic operations in miniMD and Lammps prevents the code from being properly vectorized by the compiler. Note that performance reported for this bulk configuration, which features 4 millions of atoms, only evolves marginally when increasing the number of atoms. For instance, with 216 millions of atoms and a memory usage of about 100GB, the ExaStamp amr Vec throughput is 60 millions of atoms processed per second on the skl. The measurements show that the slight drop in performance is due to an increase of NUMA effects.

Table 2 reveals that the cost of rebuilding the amr data structures is low: under 2% on knl and 3% on skl. It also shows that the percentage of time spent in the potential computation is the lowest for ExaStamp amr, because this part has been significantly optimized: the optimization with memory blocks for vectorization

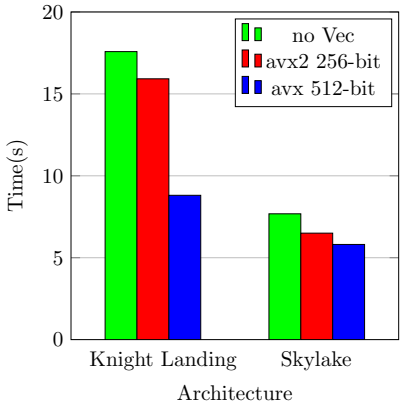

Figure 8: Time simulation for different vectorization flags on ExaStamp amr Vec, the version no Vec corresponding to ExaStamp with avx-512 bits and no option Vec. Software performed on 64 cores on knl (left) and 48 cores on skl (right).

reduces the time ratio of this kernel by 3-4%. Regarding the benefits gained using our cache-friendly octree traversal, we used the Linux perf profiler to finely tune the octree depth (D^{max}). We observed on knl that the number of cache misses of the L1 cache fluctuated according to the maximum depth of the octree. After 100 iterations, for D^{max} =1, we observed 8.8 billions of L1 cache misses. For D^{max} =2, the number of misses dropped to 2.6 billions, and for D^{max} =3, it topped 9.7 billions.

On processors such as knl and skl, special attention must be paid to the vectorization of the code, because its impact on overall performance is expected to be high. Figure 8 shows the evolution of ExaStamp amr performance according to the auto-vectorization flag used at compile time (-mavx, -xMIC-AVX512, -xCore-AVX512). The estimated speedup on the potential kernel reported by vec-report=5 option are: 2.11 with avx 512-bit knl, 1.54 with avx 512-bit skl and 1.3 with avx2 256-bit. Measurements on the full application show an acceleration of 1.88 with avx-512 on knl while it is 2.11

for the potential kernel. Similarly, we measure 1.32 with avx 512-bit skl instead of 1.54 for the single kernel. The difference in the impact of vectorization on both machines is explained on the one hand by the time ratio spent in the potential kernel that is lower on the skl (see Table 2) and on the other hand because the avx 512-bit instructions set is not exactly the same.

Figure 9 describes the evolution of the simulation time of the three codes as function of the number of cores used on a processor. On both processors, the different versions of ExaStamp amr are faster than Lammps and miniMD when the number of cores is greater than 1. Note that the sequential versions of the algorithms used by Lammps and miniMD are different from the parallel versions, which explains the performance between using one core or two. In the parallel versions, there is notably the addition of OpenMP atomic instructions in the potential kernel. We can also notice in Figure 9 that on the knl, versions of ExaStamp amr have the same scalability (almost ideal) however on the skl they suffer more of NUMA effects between 24 and 48 cores than miniMD and Lammps. These effects could be reduced by using one MPI process per socket or by forcing a distribution of tasks on the sockets according to the octree position. These assumptions will constitute one of the purpose of our future developments.

Finally, this example highlights the advantages to extend ExaStamp with amr capabilities using cache blocking, task dependency graph and vectorization even for homogeneous systems. In all cases, it is faster than miniMD and Lammps.

4.3 Micro-Jetting: a Highly-Dynamic Simulation

Simulation code	million atoms processed per second	
Architecture	Knights Landing	Skylake
Lammps	3.3	14.8
ExaStamp amr	9.2	22.3
ExaStamp amr Vec	15.5	29.1

Table 3: Number of atoms processed per second (higher is better) by Lammps, ExaStamp amr with and without Vec option (mean). Software performed on 64 cores on knl (left) and 48 cores on skl (right).

The second benchmark consists in simulating a micro-jetting [12] phenomenon, which is a highly dynamic case. The micro-jetting results of a shock on a sinusoidal default in a tin material. This case performs over 3.5 million atoms with a suitable lj potential. Our experiment is calibrated so that the shock happens from the very first time steps. Thus, the density of atoms is strongly heterogeneous and the velocity of atoms is quite high. For such a highly dynamic configuration, it is expected that our amr solution behaves better than classical MD methods. Throughout this simulation, the best performance was achieved with a maximum tree depth D^{max} equal to 2. Unfortunately, we could only compare our solution against Lammps, because miniMD is not able to cope with complex input files.

Software	Potential	Verlet Lists	Refinement
Lammps knl	81.2%	11.52.%	NA
Lammps skl	72.43%	14.2%	NA
ExaStamp amr knl	71.7.%	20.7%	3.2%
ExaStamp amr skl	61.5%	23.8%	4.7%
ExaStamp amr Vec knl	62.9%	24.6%	4.9%
ExaStamp amr Vec skl	62.6%	15.0%	6.1%

Table 4: Time ratio measured for different parts of the micro-jetting simulation: potential kernel (Potential), building Verlet lists (Verlet Lists) and related to the amr structure (Refinement). Time ratio of other parts are not reported (Scheme, Reorganisation, Update Ghost.)

Table 3 reports the measurements of atoms processed per second and provides some insights into performance obtained on knl and skl processors. First of all, with an equivalent potential kernel, ExaStamp amr is 2.8 faster on knl and 1.5 faster on skl compared to Lammps. In addition, with the Vec optimization, these differences increase respectively from 2.8 to 4.7 and 1.5 to 2. As described in the previous benchmark, these differences are partly explained by a better efficiency of ExaStamp amr, especially in terms of vectorization. Throughout the simulation, 90% of cells are empty or almost empty, which have a negative performance impact on all versions except ExaStamp amr. Note that the previous generation of the ExaStamp software, which is on a par with Lammps on the bulk benchmark, suffers from a performance drop to only 1.4 million atoms per second. We have also tested the MPI version of Lammps with the rcb dynamic load balancing option. This MPI version of the code is more efficient than the OpenMP version with 6.2 million atoms per second instead of 3.3 on knl and 23.1 instead of 14.8 on skl. However, ExaStamp amr remains better in terms of CPU and memory performance (more details in Subsection 4.4). Moreover, using only one MPI process per processor using OpenMP inner parallelism will help to reduce communication costs when moving to large clusters.

Throughout this simulation, the Verlet lists are regularly updated because the atoms move in the direction of the shock. In our case, these are updated approximately every 6 time steps. As shown in the Table 4, the rebuilding of the Verlet lists costs around 20% of the simulation time for ExaStamp amr and approximately 12.5% for Lammps because it spends more time in the potential kernel and is twice slower than ExaStamp amr Vec. The time related to refinement is more significant because, at each update of the Verlet lists, the arrays of atoms within the octrees are sorted and the pointers of the cells are adjusted on these arrays. This time ratio remains proportionately cheap compared to the benefit that it provides. In addition, adapting the refinement during simulation, i. e. coarsening/refining the cells, takes around 7.5% of an iteration with construction of neighbor lists. For our case of micro-jetting, it is interesting to adapt the grid approximately every 400 time steps.

As in the bulk benchmark, we observe the impact of various compile time vectorization flags. The Figure 10 describes the simulation

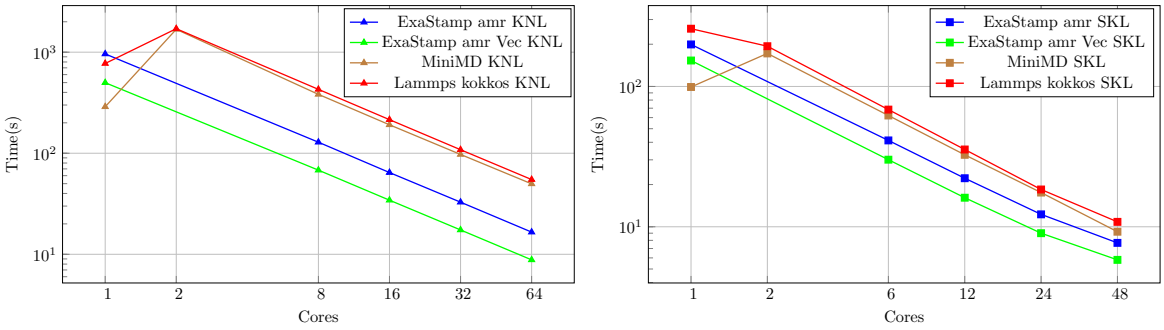

Figure 9: Simulation time (lower is better) with log axis for miniMD, Lammps, ExaStamp amr and ExaStamp amr Vec. For both architectures, the codes keep the same ranking regarding simulation time, with ExaStamp amr Vec being the fastest, then ExaStamp amr, then miniMD and finally Lammps.

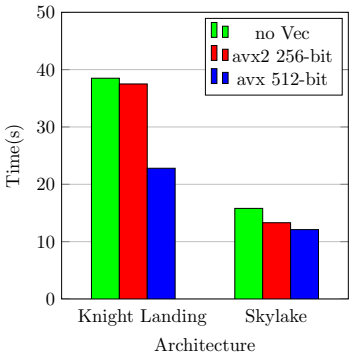

Figure 10: Simulation time for different vectorization flags on ExaStamp amr Vec, the version no Vec corresponding to ExaStamp with avx-512 bits and no option Vec. Software performed on 64 cores on knl (left) and 48 cores on skl (right).

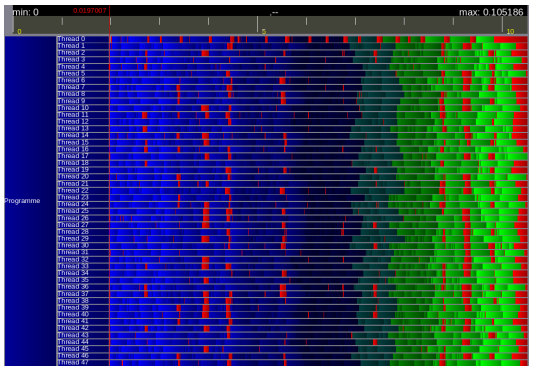

Figure 12: Visualization of tasks execution throughout a time step of a micro-jetting simulation on skl on 48 cores. The colors represent the type of task, from light green to light blue through black corresponding to the tasks from wave 1 to 8. The red portions correspond to the moment when a thread is idle.

time with the same compilation options as used for the Figure 8. Both codes being identical, except for the potential kernel, we get the same pattern as for the Figure 8. Regarding Lammps, vectorization is more impactful thanks to the use of neighbor lists. However, this part of the code represents only 12.5% of total computation time, so it has only a moderate impact on overall performance.

A more in-depth study of the behavior of the codes is conducted in two steps, first by analyzing the times obtained at variable number of cores and second by observing the trace of the tasks with 48 cores on skl. As observed in the Figure 11, for both architectures, there is a significant difference between the sequential version and the parallel version of Lammps. As observed in the homogeneous case, ExaStamp remains much better and always shows a good adaptation to the number of cores despite the load imbalance between the different octree due to the heterogeneity of the atom distribution.

To better analyze the performance of ExaStamp amr on a large number of cores, it is worthwhile to study the task scheduling behavior induced by the wave method. Figure 12 shows a tasks execution trace on the 48 cores skl[3], using one thread per core. The trace was obtained using the VITE software [9]. One can observe that only few idle slots are observed. They account for less than 5% of the simulation time. The most relevant thing to observe is that waves of tasks do not start at the same time: they overlap thanks to the use of fine-grain task dependencies. Compared to a version using a global synchronization barrier between waves, we have measured a gain of 5% on skl (48 cores) and 6% on knl (64 cores). In our implementation, thread 0 is the master thread in charge of task distribution and this is the reason why it is more often idle. At the end of the simulation, there are many threads waiting for, because there are no more ready tasks for the current iteration. This situation could be further improved by taskifying the other parts of the code, to allow the force computation step to overlap with the amr refinement, for instance.

To conclude this evaluation part, we showed that the use of octrees coupled to the wave method significantly reduces the simulation time compared to Lammps on modern processors. This is

[3]The analysis presented focuses on the use of the skl with 48 cores but the comments are also relevant for the knl.

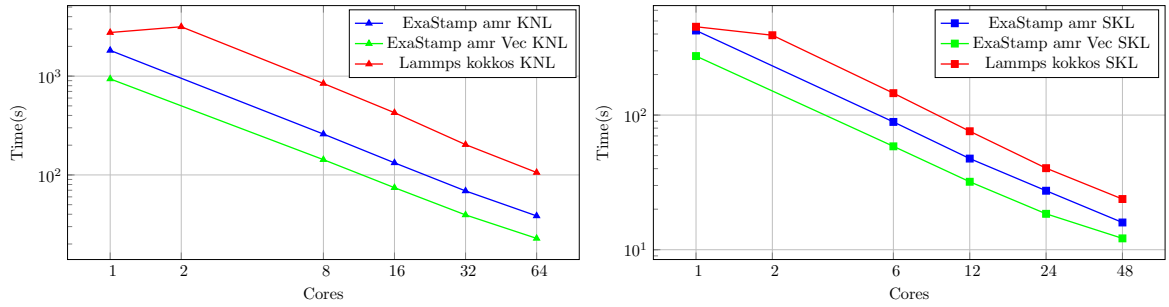

Figure 11: Simulation time (lower is better) of a micro-jetting with log axis for LAMMPS, EXASTAMP AMR and EXASTAMP AMR Vec. For both architectures, the codes keep the same ranking regarding simulation time, with EXASTAMP AMR Vec being the fastest, then EXASTAMP AMR, then MINIMD and finally LAMMPS.

explained by the good use of cache blocking techniques, vectorization and task based scheduling.

4.4 Memory footprint

SIMULATION CODE / DATASET	MEMORY FOOT PRINT (GB) BULK	MICRO-JETTING
LAMMPS	2.3	6.4
LAMMPS (48 MPI processes)	3.4	13.8
MINIMD	2.1	NA
EXASTAMP CLASSIC	3.3	48
EXASTAMP AMR	3.7	8
EXASTAMP AMR Vec	1.9	3.2

Table 5: Memory footprint (lower is better) of the different codes for both datasets: Bulk and Micro-jetting.

In this section we study the overall memory footprint with getrusage in Giga Bytes of both bulk and micro-jetting simulations. We expect EXASTAMP AMR to behave well in that respect, because one of the first expectations of AMR is used to overcome the shortcomings of the linked cell method.

First, by analyzing the bulk case results in Table 5, we notice that the memory is equitably divided between the cells for EXASTAMP classic and the OCTREES for EXASTAMP AMR. Regarding LAMMPS memory storage strategy, the atom information are stored in a structure of arrays and an array delimiting the cells with the number of atoms in each cell which minimizes memory usage. In addition to these, a storage of Verlet lists for each atom is added (more than 75% of the total storage) for each code. In the Verlet lists, MINIMD and LAMMPS store the position of atoms while EXASTAMP needs the cell index and the neighbor positions in that cell.

For the micro-jetting case, we showed the positive impact of AMR on the memory footprint because a large part of the cells are mostly empty or almost empty (more than 90%). Contrary to the bulk test case, the cell storage strategy induced by the linked cell method reveals its weakness. As attested by the second column of the Table 5, OCTREES bear this default. Indeed, a maximum depth tree of 2 reduces by 64 the storage of empty cells. We observe that the OCTREES divide the memory usage by 8, even by 15 with the use of Vec option. These results are relatively close to the memory footprint of LAMMPS.

Thirdly, using many MPI processes, 48 in our case, on one processor can generate twice as much memory. This is due to the ghost atoms that the subdomains must communicate to each others.

AMR structure drastically reduces memory footprint whenever some cells contains very few atoms, while maintaining high performance. Additionally, memory footprint may be further reduced by not storing the Verlet lists (occupying up to 75% of total memory) and recomputing them on demand. This can be achieved at the expense of a performance penalty and is required only in case of very strong memory constraints.

5 RELATED WORK

Many research efforts have been devoted to designing efficient MD simulations on supercomputers. LAMMPS and its associated MINIMD mini application are probably the most widely used software in this domain, and have already been detailed previously. In the following, we carry on a short survey of significant MD software.

DL_POLY_4 [35] brings a package of subroutines, programs and data files, written in fortran 90 and designed to facilitate MD simulations and is used for production. This software is designed to performs on CPU with OpenMP and MPI, as well as on GPU.

GROMACS [2] is written in C with intrinsics functions for SIMD instructions and CUDA for GPU. As LAMMPS, GROMACS contains algorithms for neighbor search based on Verlet lists. Whereas it is designed for biochemical molecules like proteins, lipids and nucleic acids that have a lot of complicated bonded interaction, an all-atom version provides non-bonded interactions such as Lennard Jones or Buckingham potentials.

As MINIMD, several simplified versions of complex MD codes have recently been developed to serve as a testbed for various optimizations over manycore architectures. The CoMD [27] software was designed to study the dynamical properties of various liquids and solids in the context of the ExMatEx project. The reference implementation of COMD uses OpenMP, MPI, OPENCL, and can use either neighbor lists or cell-lists methods.

Moreover LAMMPS proposes other optimizations in the INTEL package [6] such as mixed precision arithmetics, cache optimizations (tiling) and kernel offloading on KNL accelerators. Optimization for GPU are also included in the KOKKOS package.

6 CONCLUSION

Performing Molecular Dynamics N-body simulations over supercomputers is a major step towards a better understanding of physics in many scientific fields.

In this work we propose a combination of Adaptive Mesh Refinement, cache-blocking and task-based execution techniques to significantly increase the efficiency of N-body simulation on modern multicore machines, while decreasing the memory footprint. We have implemented these techniques into the EXASTAMP production code, which runs on CEA's supercomputing facilities. We have evaluated the relevance of our approach by comparing performance obtained by our implementation against the state-of-the-art LAMMPS software over two different multicore hardware. Our experiments show that AMR achieves significant performance gains over existing approaches, even for a LJ potential that has low computational requirements. In the case of a dynamic, imbalanced particles configuration, EXASTAMP AMR can be up to 4.7 faster than LAMMPS on a KNL processor and 2 on a SKL processor. Moreover, we observed that our approach is also very effective on steady-state configurations, thanks to our cache blocking and vectorization optimizations.

Future work include the development of a hybrid MPI/OpenMP version of EXASTAMP AMR, focussing on a dynamic load balancing system able to move OCTREES between MPI processes on large-scale clusters.

ACKNOWLEDGMENT

This work was funded by the French Programme d'Investissements d'Avenir (PIA) project SMICE.

REFERENCES

[1] M.P. Allen and D.J. Tildesley. 1987. *Computer Simulation of Liquids.* Clarendon Press.
[2] Herman JC Berendsen, David van der Spoel, and Rudi van Drunen. 1995. GROMACS: a message-passing parallel molecular dynamics implementation. *Computer Physics Communications* 91, 1-3 (1995), 43–56.
[3] M. Berger and P. Colella. 1989. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.* (1989).
[4] M. Berger and J. Oliger. 1984. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys. 53* (1984).
[5] Marsha J Berger and Shahid H Bokhari. 1987. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comput.* 5 (1987), 570–580.
[6] W Michael Brown, Jan-Michael Y Carrillo, Nitin Gavhane, Foram M Thakkar, and Steven J Plimpton. 2015. Optimizing legacy molecular dynamics software with directive-based offload. *Computer Physics Communications* 195 (2015), 95–101.
[7] Greg L Bryan, Michael L Norman, Brian W O'Shea, Tom Abel, John H Wise, Matthew J Turk, Daniel R Reynolds, David C Collins, Peng Wang, Samuel W Skillman, et al. 2014. Enzo: An adaptive mesh refinement code for astrophysics. *The Astrophysical Journal Supplement Series* 211, 2 (2014), 19.
[8] P Colella, DT Graves, TJ Ligocki, DF Martin, D Modiano, DB Serafini, and B Van Straalen. 2000. Chombo software package for amr applications-design document.
[9] K Coulomb, M Faverge, J Jazeix, O Lagrasse, J Marcoueille, P Noisette, A Redondy, and C Vuchener. 2009. *Visual trace explorer (vite).* Technical Report. Technical report.
[10] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro* 37, 2 (2017), 52–62.
[11] Marie Durand, Bruno Raffin, and François Faure. 2012. A packed memory array to keep moving particles sorted. In *9th Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS).* The Eurographics Association, 69–77.
[12] Olivier Durand, S Jaouen, L Soulard, Olivier Heuze, and Laurent Colombet. 2017. Comparative simulations of microjetting using atomistic and continuous approaches in the presence of viscosity and surface tension. *Journal of Applied Physics* 122, 13 (2017), 135107.
[13] A. Dubey et al. 2014. A survey of high level frameworks in block-structured adaptive mesh refinement packages. *J. Parallel Distrib. Comput.74* (2014).
[14] Dennis Gannon, William Jalby, and Kyle Gallivan. 1988. Strategies for cache and local memory management by global program transformation. *J. Parallel and Distrib. Comput.* 5, 5 (1988), 587 – 616. https://doi.org/10.1016/0743-7315(88)90014-7
[15] Dennis B Gannon and William Jalby. 1987. *The influence of memory hierarchy on algorithm organization: Programming FFTs on a vector multiprocessor.* University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development.
[16] 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. 2009. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574* 3 (2009).
[17] Changjun Hu, Xianmeng Wang, Jianjiang Li, Xinfu He, Shigang Li, Yangde Feng, Shaofeng Yang, and He Bai. 2017. Kernel optimization for short-range molecular dynamics. *Computer Physics Communications* 211 (2017), 31–40.
[18] J.Reinders J. Jeffers and A. Sodani. 2016. *Intel Xeon Phi Porcessor High Performance Programing.* Chapter 20, 443–470.
[19] J. E. Jones. 1924. On the Determination of Molecular Fields. II. From the Equation of State of a Gas. *Proceedings of the Royal Society of London. Series A*, 463–477.
[20] Sandia National Laboratorie. [n. d.]. LAMMPS Molecular Dynamics Simulator. http://lammps.sandia.gov.
[21] Benedict J. Leimkuhler, Sebastian Reich, and Robert D. Skeel. 1996. Integration Methods for Molecular Dynamics. In *Mathematical Approaches to Biomolecular Structure and Dynamics*, Jill P. Mesirov, Klaus Schulten, and De Witt Sumners (Eds.). The IMA Volumes in Mathematics and its Applications, Vol. 82. Springer New York, 161–185.
[22] M Lijewski, A Nonaka, and J Bell. 2011. Boxlib.
[23] Chris M Mangiardi and Ralf Meyer. 2017. A hybrid algorithm for parallel molecular dynamics simulations. *Computer Physics Communications* (2017).
[24] Edward A. Mason. 1954. Transport Properties of Gases Obeying a Modified Buckingham (Exp-Six) Potential. *The Journal of Chemical Physics 22*, 2 (1954), 169–186. https://doi.org/10.1063/1.1740026
[25] Simone Meloni, Mario Rosati, and Luciano Colombo. 2007. Efficient particle labeling in atomistic simulations.
[26] Ralf Meyer. 2014. Efficient parallelization of molecular dynamics simulations with short-ranged forces. In *Journal of Physics: Conference Series*, Vol. 540. IOP Publishing, 012006.
[27] J Mohd-Yusof. 2012. CoDesign Molecular Dynamics (CoMD) Proxy App Deep Dive. In *Exascale Research Conference.*
[28] Philip M. Morse. 1929. Diatomic Molecules According to the Wave Mechanics. II. Vibrational Levels. *Phys. Rev.* 34 (1929), 57–64. Issue 1. https://doi.org/10.1103/PhysRev.34.57
[29] Guy M Morton. 1966. *A computer oriented geodetic data base and a new technique in file sequencing.* International Business Machines Company New York.
[30] Yehoshua Perl and Stephen R Schach. 1981. Max-min tree partitioning. *Journal of the ACM (JACM)* 28, 1 (1981), 5–15.
[31] E. Cieren L. Colombet S. Pitoiset and R. Namyst. 2014. ExaStamp: A Parallel Framework for Molecular Dynamics on Heterogeneous Clusters, Lecture Notes in Computer Science (Ed.), Vol. 8806. Euro-Par 2014: Parallel Processing Workshops, Springer International Publishing, 121–132.
[32] Steve Plimpton. 1995. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics* 117, 1 (1995), 1–19.
[33] Lewis Fry Richardson. 1911. The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character* 210 (1911), 307–357.
[34] R.Meyer. 2013. Efficient parallelization of short-range molecular dynamics simulation on many-core system. *PHYSICAL REVIEW* (2013).
[35] W Smith, TR Forester, and IT Todorov. 2012. The DL POLY Classic user manual. *STFC, STFC Daresbury Laboratory, Daresbury, Warrington, Cheshire, WA4 4AD, United Kingdom, version* 1 (2012).
[36] Avinash Sodani. 2015. Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE.* IEEE, 1–24.
[37] Loup Verlet. 1967. Computer" experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Physical review* 159, 1 (1967), 98.
[38] D. Wolff and W. G. Rudd. 1999. Tabulated Potentials in Molecular Dynamics Simulations. *Computer Physics Communications* 120, 1 (1999), 20–32.