Computer Physics Communications 253 (2020) 107177

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

AMR-based molecular dynamics for non-uniform, highly dynamic particle simulations☆

Raphaël Prat [a,*], Thierry Carrard [a], Laurent Soulard [a], Olivier Durand [a], Raymond Namyst [b], Laurent Colombet [a]

[a] CEA, DAM, DIF, F-91297 Arpajon, France
[b] University of Bordeaux, Inria Bordeaux Sud-Ouest, Talence, France

ARTICLE INFO

Article history:
Received 20 March 2019
Received in revised form 13 January 2020
Accepted 17 January 2020
Available online 23 January 2020

Keywords:
Molecular dynamics
Micro-jetting
AMR
MPI
OpenMP
Shockwave

ABSTRACT

Accurate simulations of metal under heavy shocks, leading to fragmentation and ejection of particles, cannot be achieved by simply hydrodynamic models and require to be performed at atomic scale using molecular dynamics methods. In order to cope with billions of particles exposed to short range interactions, such molecular dynamics methods need to be highly optimized over massively parallel supercomputers. In this paper, we propose to leverage Adaptive Mesh Refinement techniques to improve efficiency of molecular dynamics code on highly heterogeneous particle configurations. We introduce a series of techniques that 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. We analyze performance results on several simulation scenarios, such as the production of an ejecta cloud from shock-loaded metallic surfaces, using a large number of nodes equipped by Intel Xeon Phi Knights Landing processors. Performance obtained with our new Molecular Dynamics code achieves speedups greater than 1.38 against the state-of-the-art LAMMPS implementation.

1. Introduction

Recent Molecular Dynamics (MD) simulations open up interesting perspectives in understanding the behavior of condensed matter under shock [1]. This approach is complementary to and offers several advantages over the classical continuum approaches such as hydrodynamics [2]. In particular, it overcomes any mesh effect while allowing any physics model but the models describing the inter-atomic interactions.

MD simulations involve N-body computations that provide the properties of a system made of particles by numerically solving the Newton's equation of motion during several timesteps: $\mathbf{F}_i = m_i.\mathbf{a}_i$ with $\mathbf{F}_i$ the forces applied on the particle i, and m_i and $\mathbf{a}_i$ respectively the mass and the acceleration of the particle i. Forces applied on a particle are derived from the interactions with other particles modeled by a given potential $\mathbf{F}_i = -\nabla U_i$, U_i being the potential energy. In practice, potential is either computed by analytical formula or interpolated from tabulated values [3].

In this paper, we mainly focus on short-range interactions, where the influence of neighboring particles is neglected beyond a given cut-off distance *rcut*. For simple materials such as noble gases, these interactions are modeled by pair potentials such as the Lennard-Jones potential (LJ) [4], the Morse potential [5] or the Exponential-six potential [6]. The Lennard-Jones potential (LJ) is usually used as a standard benchmark for MD codes due to its low computation cost. For metal materials, the Embedded Atom Model potential (EAM) or the Modified EAM (MEAM) potential [7] are more accurate, but require more computation cycles.

In theory, computing the potential for a given particle P should only involve the subset of particles included in a sphere of radius *rcut* centered in P. However, given that particles can move at each simulation timestep, determining such a subset for each particle P is costly. To save computing cycles, MD simulations use algorithms which approximate these subsets by considering super-sets containing more particles than those included in the *rcut* sphere. Two well-known techniques are used by most existing MD software: the linked cells method [8], and the Verlet lists method [9]. The linked cells method maintains neighbor lists for each particle at each timestep with a complexity of $O(n)$. The Verlet lists method extends the linked cells method by using a greater radius (the Verlet radius) to catch a larger neighborhood. This decreases the rate at which the lists must be refreshed.

Designing highly efficient MD simulations over parallel hardware has been a hot research topic for decades. These efforts have

☆ The review of this paper was arranged by David W. Walker.
* Corresponding author.
E-mail address: raphael.prat@inria.fr (R. Prat).

https://doi.org/10.1016/j.cpc.2020.107177
0010-4655/

led to efficient and scalable software codes such as LAMMPS [10] or Gromacs [11]. Recently, these codes have evolved to cope with the evolution of supercomputers architectures towards an increasing number of cores per node, the expanding width of vector registers and the usage of non-uniform memory architectures. Current implementations of MD codes typically exploit three levels of parallelism by using a hybrid MPI+X (e.g. MPI+OpenMP) programming model together with carefully designed vector-friendly data structures. Despite these efforts, current parallelization techniques struggle with MD simulations involving large temporal variations in particle density over the simulation domain. For instance, when a material is under extreme conditions (pressure, temperature, deformation, shock, etc.) and is subject to micro-jetting or dislocation, the atom density becomes highly heterogeneous and large parts of the simulation domain suddenly become empty. In such cases, existing MD codes typically exhibit significant overhead. Moreover, their efficiency is reduced due to the sparse nature of many cells, limiting opportunities of exploiting vector capabilities of modern processors.

To compensate for such inhomogeneous particle distributions, we introduce Adaptive Mesh Refinement (AMR) techniques [12,13] into MD simulations. AMR is a well-known technique used in many fields to cope with such physical phenomena. It consists in dynamically adapting the mesh to better capture heterogeneity. AMR was first introduced in hydrodynamics simulations in order to reduce the computation time by refining the discretized domain (grid) where the points need to be spaced to keep their stability properties [13]. AMR is currently used in a large amount of software [14] for various physical problems such as astrophysics with Enzo [15] or to solve PDE with Chombo [16].

This paper introduces an efficient hybrid MPI/OpenMP parallelization taking advantage of AMR data structures to efficiently cope with highly dynamic MD simulations by reducing the imbalance between the threads and MPI processes. We leverage our previous work, based on the use of AMR techniques [17] to improve memory locality, multi-threading parallelism and we address the division of the domain simulation into subdomains assigned to the MPI processes. We demonstrate that the introduction of AMR techniques does not add overhead to communications between nodes. We also extend our task dependency graph to cope with irregular subdomain pattern. We validate our approach using an extended implementation of the ExaSTAMP [18] MD simulator over a cluster of 512 Intel® Xeon Phi Knight Landing (KNL). 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 the following:

- We introduce a new MD simulation code using AMR techniques able to efficiently exploit modern supercomputers;
- We present a task-based dependency graph strategy for AMR in order to optimize thread-based parallelization and an MPI parallelization strategy adapted to the AMR structure;
- We evaluate our implementation of the AMR using large scale production-cases and traditional benchmarks.

This paper is structured as follows. In the next Section, we briefly recall the main concepts used in MD and AMR based applications then we describe the main contribution of the presented work. We validate the relevance of our approach in Section 3 by comparing our implementation with state-of-the-art software on several benchmarking scenarios. We present related work in Section 4, and dress concluding remarks in Section 5.

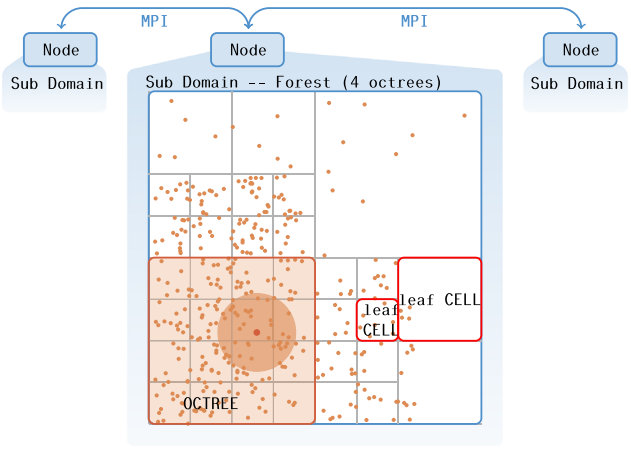

Fig. 1. MD software architecture adapted for AMR (2D). Particles are distributed over a number of subdomains. Each subdomain is in turn decomposed into octree cells.

2. Adaptive mesh refinement for molecular dynamics simulations

Traditional cell-based methods can hardly cope with highly-dynamic MD simulations because they were not designed to handle huge disparities of particle density across the simulation domain. We thus propose to use AMR-based techniques to recursively subdivide the simulation domain into trees of cells, the leaves of which may be of different volumes, depending on some application-specific refinement criteria. The overall idea is to end up with small cells only where it is necessary (e.g. high density of particles), and to deal with large cells in the low density areas. We now present the most relevant features of this approach that aims at achieving high hardware efficiency as well as dynamic load balancing.

2.1. Introducing AMR in MD simulations

As depicted in Fig. 1, in our AMR-based architecture, the simulation domain is divided into subdomains using the spatial domain decomposition method [10] and the subdomains are distributed among MPI processes. As a rule of thumb, simulations spawn one MPI process per multicore node, but other deployment schemes are possible. Unlike classic linked cells methods that split subdomains into a grid of cells, our approach splits subdomains into grids of multiple octrees called *forests of octrees*.

Octrees are refined using an algorithm that depends on MD-specific criteria. The chosen criteria may depend on the number of particles per cell weighted by the cost of the used potential. Other criteria can be implemented according to the numerical specificities of the simulation. In 3D, each cell is recursively divided into 8 subcells of the same size, until the refinement criteria are met. The refinement must be stopped before getting the size of subcells smaller than the Verlet radius.

In each octree, the atoms' information is stored in a structure of arrays into the root cell, in order to maximize the effectiveness of compiler vectorization. Particles are sorted by their Cartesian positions (Morton index [19]) and the leaf cells correspond to contiguous sections of these arrays. During the computation of the potential energy, all the particles of an octree are processed within an OpenMP task. In other words, each octree is processed in a sequential way. Such an octree traversal strategy has proven to be very effective both on static and dynamic test cases on multicore KNL and Intel® Xeon Skylake (SKL) platforms [17].

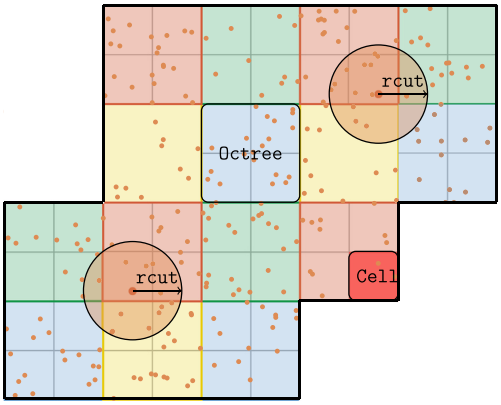

Fig. 2. 2D representation of a subdomain of the MD simulation. The atoms are distributed into the octrees themselves included into octrees with $D^{max} = 1$.

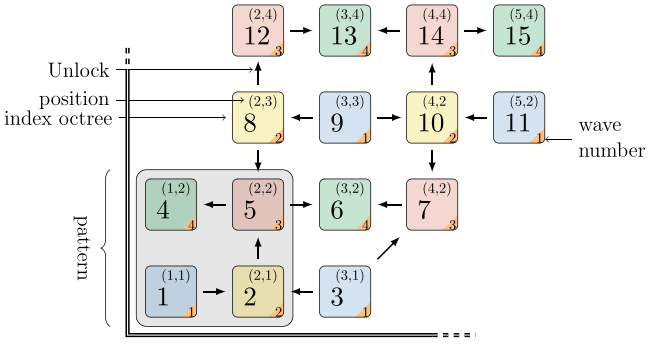

Fig. 3. Task dependency graph in 2D. Octrees are classified into 4 waves (8 in 3D), octrees classified in the wave $(n)_{n\geq 1}$ are unlocked by specific octrees in the wave $(n-1)_{n\geq 1}$. In this case, a dependency between octrees 3 and 7 is added to avoid that both running at the same time.

To achieve maximal performance a tradeoff must be found between the number of octrees per MPI process (i.e. the more octrees, the more parallelism) and the size of octrees (i.e. small octrees do not exploit caches enough). Thus, at initialization time, we first evaluate the size of the smallest leaf cell, and then we try to determine a "good root cell size". This algorithm therefore searches for the optimal maximum depth of the octrees, called D^{max}. For this purpose it uses several parameters that are mainly the Verlet radius, the minimum number of octrees over the entire domain and the maximum number of atoms per Verlet cell.

2.2. Inner-node parallelization

Inside each MPI process, we have developed two OpenMP parallelization strategies to process octrees forests over multi-core hardware. The first strategy is based on a task dependency graph to limit the number of threads synchronization: octrees are processed in "waves" in such a way that neighbor octrees are never processed at the same time. This strategy is particularly well-suited when the number of octrees to process is very high with respect to the number of cores. An octree corresponds to an OpenMP task in the graph. When the number of octrees is too low, we opt for a second strategy, which uses a more synchronous approach. The overall loop that iterates on the list of octrees is parallelized using the `omp for` construct. Since all octrees can potentially be executed at the same time, additional thread synchronizations are needed when updating particles.

In both strategies, octrees are sequentially processed to optimize the caches reused by a cache blocking effect and to obtain sufficient workload to be able to create tasks.

2.2.1. Task-based dependency graph

To take advantage of the symmetric properties of most potentials (third Newton's law: $\mathbf{F}_{i\to j} = -\mathbf{F}_{j\to i}$ with $\mathbf{F}_{i\to j}$ the force applied by the particle i on the particle j), the forces applied between two particles are computed once for the first particle, and then reported to the second particle. This reduces the computation cost of the simulation but generates costly thread synchronizations (e.g. use of mutexes) to ensure that two particles are not updated at the same time. To address this problem, we have introduced an adaptation of the wave method [17,20]. Octrees are processed in successive waves in such a way that two adjacent octrees are never processed simultaneously. This is illustrated in Fig. 2, where octrees are classified on 4 waves (8 in 3D) blue, yellow, green and orange. Octrees with $D^{max} > 0$ are processed wave by wave. This ensures that a particle processed by a thread is not in the neighborhood of any other particle processed by another thread. If $D^{max} = 0$, there is no AMR structure and we fall back to a classical system of linked cells method. In this particular case, the number of waves may be higher, particularly because of the potential used. For example, the need to have access to neighbors' neighbors greatly increases the number of waves. It should be noted that, unlike the more traditional task blocking method [21], our wave method guarantees a number of waves equal to 8 if $D^{max} > 0$. A straightforward approach consists in generating an OpenMP parallel loop for each wave because all tasks in the same wave can be concurrently performed without any conflict. In this case, there is always an efficiency loss at the end of each parallel loop due to the OpenMP barrier between two waves. But this method does not require to process waves in a specific order.

A solution to relax synchronizations is to overlap the execution of consecutive waves while enforcing some dependencies between neighbor octrees. This can be implemented by explicitly generating an OpenMP task graph. The construction of this graph is based on the use of dependencies between tasks with respect to their spatial location (see Fig. 3). The waves are generated sequentially, but their executions can overlap. Indeed the execution of an octree belonging to the nth wave can start as soon as each adjacent octree belonging to a preceding wave has been processed.

2.2.2. Loop parallelization

The task graph parallelization method exhibits high efficiency as long as the number of ready tasks is large enough to feed the cores of the underlying architecture. One solution to increase the number of tasks is to artificially reduce D^{max}. However, coping with small octrees usually has a negative impact on parallel overhead and cache usage. The second solution is to fall back to the simultaneous execution of all octrees using an OpenMP parallel loop. In this case, thread synchronizations are mandatory to preserve particle data integrity. When the number of tasks is still insufficient for both solutions, the last possibility is to iterate with OpenMP parallel loop on the leaf cells of the octrees.

2.3. Inter-node parallelization

As mentioned at the beginning of Section 2.1, the simulation domain is divided into subdomains, each subdomain being assigned to a single MPI process. There is generally a *one-to-one* mapping between the MPI processes and the cluster nodes. This allows our fine-grain OpenMP parallelization approach to fully tap into the potential of the underlying shared-memory architecture. Moreover, it also keeps the number of MPI processes

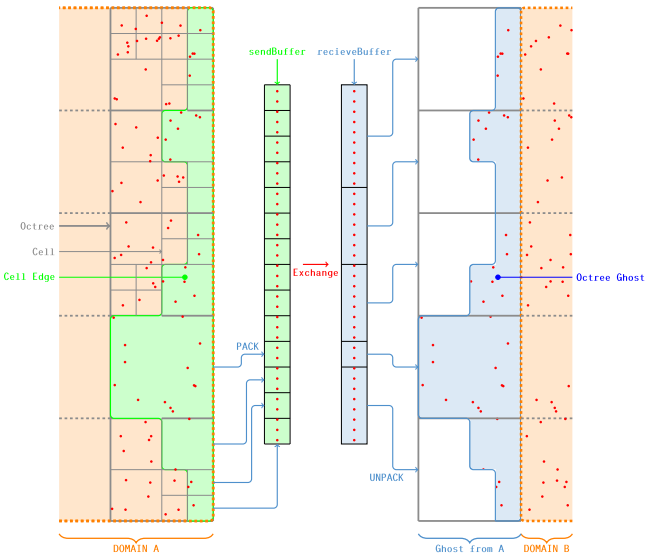

Fig. 4. Update of a portion of the ghost octrees of domain B by domain A. In order to minimize particle transfer, only the leaf cells of the data on the border of the domain A are selected (green). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

as small as possible in order to decrease the communication cost between nodes.

Since not all neighbors of a given particle are stored locally, our approach uses the well-known *ghost cells* technique to grab information about missing particles. This approach is further described in the following subsection. In Section 2.3.2 we present how octrees are redistributed among MPI processes when load balancing is required.

2.3.1. Ghost communications

We use ghost cells around each subdomain to cache information about particles located on the border of neighbor subdomains. Such cells, which correspond to leaf cells on neighbor subdomains, do not necessarily have the same size, as illustrated in Fig. 4.

Updating ghost cells requires data exchanges between nodes implemented by MPI Send/Recv communications. To speed up communications, buffers are assembled in parallel. Ghost cells are packed concurrently into send buffers (Fig. 4). To avoid the usage of locks, the memory destination of each cell into a given buffer is precalculated. A similar approach is used on the receiving side to unpack ghost cells.

In contrast with the traditional approach where all the ghost cells have the same size (i.e. determined by the Verlet radius), our AMR approach transfers cells of bigger size, and thus transfers unneeded sets of particles that are actually located too far from the border of their subdomain. However, bigger-size cells do not contain large amount of particles, and transferring contiguous blocks of potentially unneeded particles has proven more effective than filtering particles to transfer only parts of some cells.

2.3.2. Partitioning

To evenly distribute the workload among all MPI processes, a computational weight is evaluated for each octree based on the number of particles, their type, their depth inside the octree, the potentials used, etc. As illustrated in Fig. 5, each process is in charge of a set of cells forming a contiguous space of arbitrary topological shape.

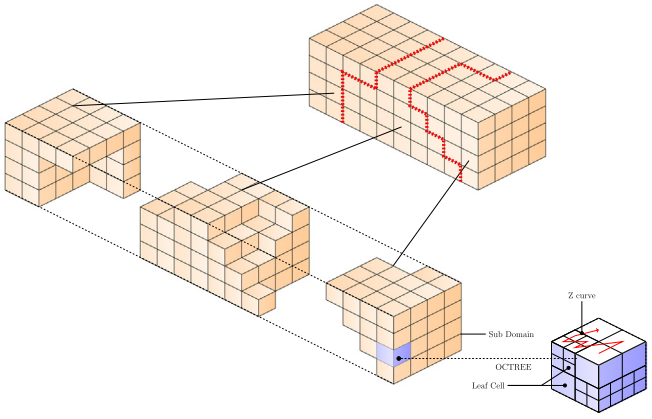

Fig. 5. The simulation domain is partitioned into different subdomains that can be of any topological shape.

The possibility of using subdomains of arbitrary shape provides more flexibility when partitioning the global domain into subdomains of equivalent computational load. The set of weighted octrees is used as an input to the Zoltan [22] partitioning library. The next section presents evaluations obtained using various partitioning techniques such as Shift method, Recursive Coordinate Bisection (RCB), Recursive Inertial Bisection (RIB), Space Filling Curving (SFC) and ParMetis.

3. Evaluation

To evaluate the effectiveness of our approach, we present a performance study using two test cases: a stable *bulk copper* configuration, and a *micro-jetting* simulation. This will allow us to compare the performance of our implementation in ExaSTAMP (called AMR) with LAMMPS and with a non-AMR version of ExaSTAMP. Like LAMMPS, the non-AMR version uses a classical linked cell method combined with Verlet lists.

In this paper, we report on the performance of our MPI + OpenMP parallelization strategy, as well as on the associated load balancing mechanisms. Additional performance details about the inner-node optimizations (C++ internal structures, memory footprint, caches usage and vectorization) are presented in [17].

3.1. Experimental framework

To validate the relevance of our approach, we evaluate the performance of our implementation against the LAMMPS package on the two selected test cases. LAMMPS is one of the most relevant classical MD simulators designed for parallel computers. It has proved to perform efficiently on benchmarks featuring several billions of atoms over 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 also been integrated to optimize LAMMPS on accelerators (KNL and GPU). Load balancing can be performed by using either the shift method [23] or the recursive coordinate bisection [24] (RCB) one. Despite specific optimizations for Intel KNL processors, our experiments revealed that the OpenMP version of LAMMPS remains 10 to 15% slower than the full-MPI version [25]. We have notably used the following Kokkos-runtime flags recommended by LAMMPS documentation: *mpirun -np nbMPI executable -k on t nbThread -sf kk -pk kokkos newton on neigh half comm no -in in.lj* with *newton on* to use third newton law, *neigh half* to store a pairwise interaction in one neighbors list and *comm no* to use non-threaded communication.

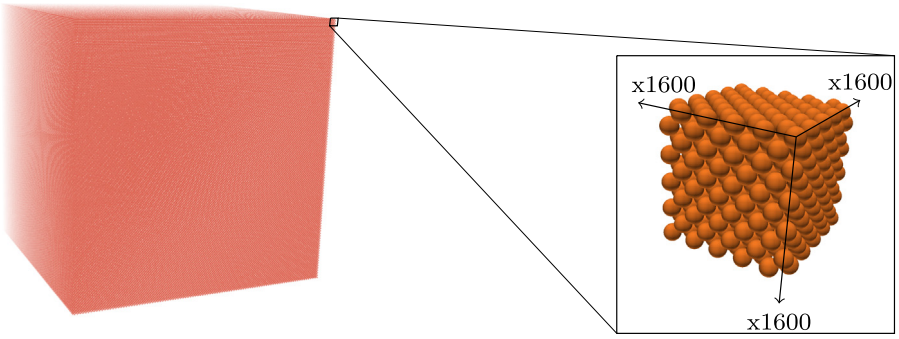

Fig. 6. Bulk of copper with 16,384,000,000 atoms of copper generated with 1600 ∗ 1600 ∗ 1600 FCC lattices.

To avoid overloading figures with uninstructive plots, we retained the hybrid version of LAMMPS in our experiments because it was the most efficient one on our target configuration.

All our experiments were run on a machine consisting of 512 KNL processors featuring 68 cores with a frequency of 1.4 GHz. Note that 4 cores (per processor) are dedicated to for the operating system leaving 64 cores per node for user applications. Each core has a 2-wide out-of-order pipeline with support for 4 hardware threads and 512-bit vector units.

The remaining of this section is organized as follows. The first phase of our experiments consists in evaluating the performance of the two versions of ExaSTAMP against LAMMPS. We use two different test cases: the simulation of a bulk of copper (Section 3.2), which is a steady state configuration, and the simulation of a micro-jetting of tin (Section 3.3) which represents a more dynamic configuration. Both test cases use the Lennard Jones (LJ) potential. This potential is widely used to benchmark MD codes. Due to its low computational cost, the LJ potential highlights performance issues of non-computational parts of the code such as threads synchronizations, memory accesses or MPI communications.

In a second phase, we demonstrate the benefits of our approach on a more realistic test case featuring the Modified Embedded Atom Model (MEAM) potential on the micro-jetting configuration (Section 3.4). Although MEAM is rarely used in benchmarks due to its very high computational cost, it is one of the best potentials regarding the modeling of interactions between metallic atoms. Unfortunately, LAMMPS does not include an analytical implementation of MEAM, so only comparisons between our two versions of ExaSTAMP are reported.

3.2. Bulk of copper

The first benchmark illustrated in Fig. 6 consists in simulating 100 timesteps ($\Delta t = 1$ fs) of a bulk material featuring 2.048 and 16.348 billion of atoms in thermodynamic equilibrium. This simulation has many properties such as the homogeneous density, which also remains constant throughout the whole duration of the simulation. The bulk of copper is obtained by replicating an atom pattern named lattice. A Face Centered Cubic (FCC) lattice of 3.54 Å is used with a suitable LJ potential ($\epsilon = 9.340$ x 10^{-20} J and $\sigma = 2.27$ Å). For each atom, 39 interactions with neighboring atoms are processed due to the 5.68 Å cut-off and the symmetric property of the LJ potential. A Verlet radius of 0.3 Å is added to avoid rebuild Verlet lists. Such a homogeneous case is not a configuration able to demonstrate the advantages of AMR in MD simulations, and we merely expect ExaSTAMP AMR to exhibit decent performances thanks to its carefully optimized code design.

The simulation runs on a cluster of 512 KNL (32,768 cores) and the simulation domain is decomposed into 512 subdomains assigned to 512 MPI processes with 256 threads each (4 hyper-threads per core). Throughout this simulation, the tree depth D^{max} is set to 2, which leads to the best results.

Table 1
Bulk of copper: number of atoms processed per second (**Higher is better**).

Software	2.048 × 10^9 atoms	16.384 × 10^9 atoms
ExaSTAMP	4.0 × 10^9	6.5 × 10^9
ExaSTAMP AMR	16.5 × 10^9	22.6 × 10^9
LAMMPS	6.2 × 10^9	NA

Overall performance numbers are reported in Table 1. The measurements show that ExaSTAMP AMR is 2.61 times faster than LAMMPS and 4.05/3.48 times faster than ExaSTAMP. The bulk configuration with 16.348 billion of atoms cannot be generated with LAMMPS due to the limitation of 32-bit integers during the velocity initialization loop, even with the `BIGBIG` option.

Fig. 7 describes the average time spent during potential computation, including updates of ghost areas. The time corresponding to the Verlet lists is not reported because those lists are only built during the initialization step because of the atoms moving around their initial position. Similarly, since the refinement functions are only called during the initialization step, their duration is negligible and does not appear in Fig. 7.

Most of the overall simulation time is spent inside the potential kernel. Although the number of flops computed by each version is identical, the implementation of LJ in the AMR version gives better results with 3.78 times faster than LAMMPS and 3.28 times faster than ExaSTAMP. The superiority of the AMR version on this configuration is mainly due to the avoidance of costly thread synchronizations thanks to our task-based wave execution model (see Section 2.2.1). Another reason is the optimization of the L1 cache by grouping the cells into octrees (cache blocking and Z-curve), which is only implemented in ExaSTAMP AMR. Regarding the average time consumed in updating the ghost areas, we observe that our strategy of exchanging cells of arbitrary size is not costly: the AMR and LAMMPS versions exhibit the same performance. Note that ExaSTAMP, which uses a MPI + OpenMP strategy to update ghost areas, is much slower.

Although the AMR version involves a set of techniques and algorithms specifically designed to deal with highly dynamic and heterogeneous simulations, we observe that its implementation is nonetheless efficient on a steady, uniform and memory-bound test case such as Bulk of Copper.

3.3. Micro-jetting using a Lennard Jones potential

The second test case aims at evaluating the behavior of our approach on a more dynamic and non-uniform test case: the simulation of a system generating micro-jetting.

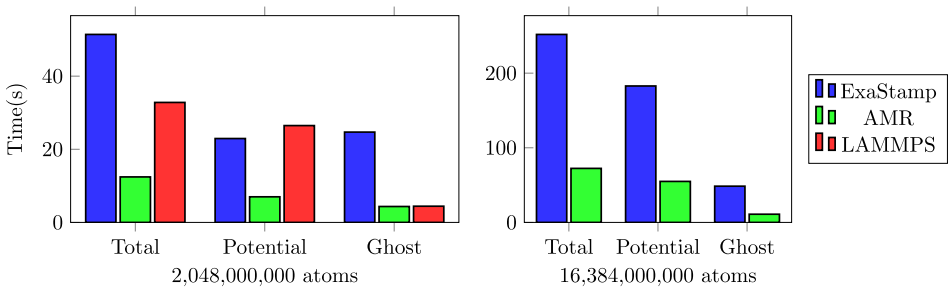

Fig. 7. Distribution of costs between potential computation and message exchanges between sub-domains (**Lower is better**).

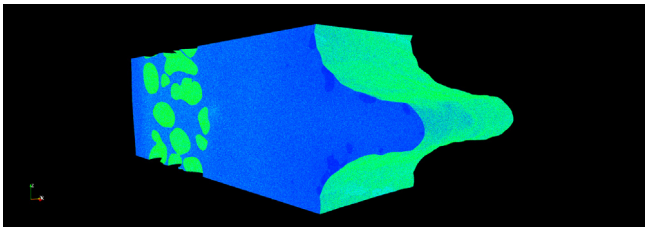

Fig. 8. Micro-jetting of 589,228,457 atoms of tin obtained with an MEAM potential after 1,550,000 iteration of time ($\Delta t = 1$ fs).

The micro-jetting results from the reflection of a strong shock wave on a surface containing geometrical defects. The micro-jetting involves microscopic processes (break out) that are important to understand and to model for in hydrodynamics codes, for example in inertial confinement fusion simulations.

In this test case, the micro-jetting results from a shock on a tin material of 589,228,457 atoms, as illustrated in Fig. 8, which shows the position of atoms after 1,550,000 timesteps of $\Delta t = 1$ fs computed using a MEAM potential. In order to make comparisons with LAMMPS, all codes use the same atom positions at timesteps 1,550,000 (by reading the same dump file) and use a LJ potential instead of the MEAM potential to study the impact of the non-uniform distribution of atoms. The LJ potential uses $\epsilon = 3.003 \times 10^{-20}$ J and $\sigma = 2.99$ Å for tin with a 10 Å cut-off. The Verlet radius is set to 1 Å for Verlet lists. The Verlet lists are rebuilt almost every 17 timesteps. In addition, the refinement algorithm is applied to each rebuilt Verlet lists.

In this test case, a high number of atoms move from one subdomain to another, increasing the volume of MPI communications. Moreover, in the AMR version, cells no longer have the same computation weight nor the same size, which challenges the underlying task scheduling system.

To obtain reliable performance numbers, we run several times a simulation of 2000 timesteps of 1 fs and the raw numbers are gathered and extracted every 100 time steps. All of the three simulations use the efficient partitioning method (Recursive Coordinate Bisection) each 500 timesteps to redistribute the octrees or cells between the subdomains.

In Fig. 9, we can see that ExaSTAMP AMR performs overall 1.38 times faster than LAMMPS and 3.51 times faster than ExaSTAMP. To further understand this behavior, we report the amount of time spent in potential computation, ghost cells exchanges and Verlet lists updates. The average and maximum (scratched) time of these parts are also presented.

Regarding the potential computation, we observe that AMR method clearly outperforms both the classical versions of ExaSTAMP and LAMMPS. We also note that the cost of updating octree cells in our AMR implementation is very low. As evidenced by the average time to build Verlet lists, the use of AMR techniques slightly degrades the neighbor search by introducing additional checks. The leaf cells being bigger than the cells of LAMMPS and ExaSTAMP, the number of distances checked is indeed higher.

The most expensive phase of the simulation comes from the updating ghosts cell areas. Unlike the "classic" ExaSTAMP and AMR ExaSTAMP, LAMMPS does not provide an option to deactivate the overlapping of communications, so it was necessary to find a method to determine the cost of communications in this code. To obtain a comparable communication time, we measure the time of the potential computation that we subtract from the time of the global function in order to obtain the non-overlapped communication time. LAMMPS and AMR times being very close, we can conclude that the complexity of the exchanges of ghost octree cells, which heavily relies on the subdomains shape, does not degrade the overall performance of ExaSTAMP AMR. The relative high cost of MPI communications in LAMMPS is explained by the low overlapping of these communications in the presence of a low computational potential (LJ). In addition, the AMR version is faster than ExaSTAMP because the reduction of the number of empty cells greatly reduces the time spent in the ghost cell query when building MPI messages. LAMMPS is the fastest regarding this part even if the cost of rebuilding Verlet lists has to be weighted against the total cost of the simulation, which remains low even with an inexpensive potential.

The "Other" section is the sum of the times required to compute and display logs and to implement load balancing. The AMR method has a positive impact on the time of the load balancing phase due to the low number of octrees in comparison with the number of cells.

Overall, this test case shows that ExaSTAMP AMR provides better performances than LAMMPS and confirms the relevance of introducing adaptive mesh refinement techniques in MD simulations to tackle dynamic and non-uniform configurations.

Note that our implementation could be further enhanced by allowing different D^{max} to be used across subdomains. Currently, the number of octrees per subdomain can be highly variable, which can generate some load imbalance between MPI processes.

3.4. Micro-jetting using a MEAM potential

We now consider using our approach on a micro-jetting production data set that requires to use the MEAM potential instead of the low cost LJ potential. The arithmetical complexity of the MEAM potential is almost 100 times greater than the complexity of the LJ potential: computing MEAM on a particle involves neighbors of its direct neighbors, and takes the possible screening of particles into account.

In our test case, the *rcut* is set to 4.17 Å to reduce the number of interactions. The Verlet radius remains set to 1 Å. As previously stated, LAMMPS does not provide an analytical MEAM potential and only comes with a tabulated one, whose accuracy does not meet the requirements of the simulation. Therefore

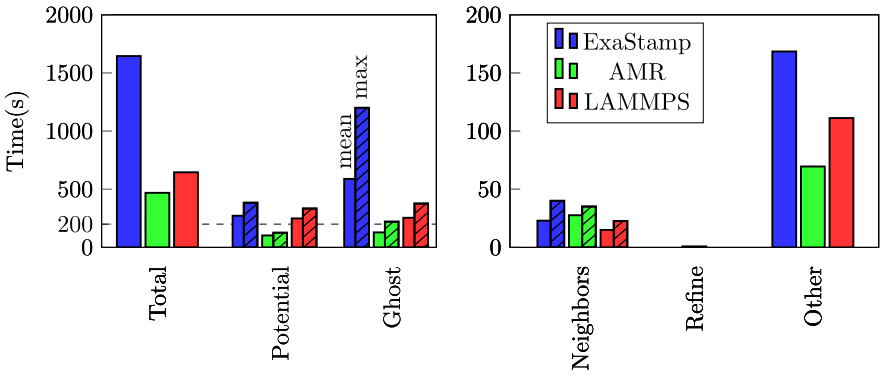

Fig. 9. Time for different sections of each software during the micro-jetting simulation. For the sake of clarity, times over 200 s are reported on the left and the others on the right. The mean (no stripe) and maximum (stripes) time of MPI processes are given for LJ potential computation, ghost update and neighbors update. (**Lower is better**).

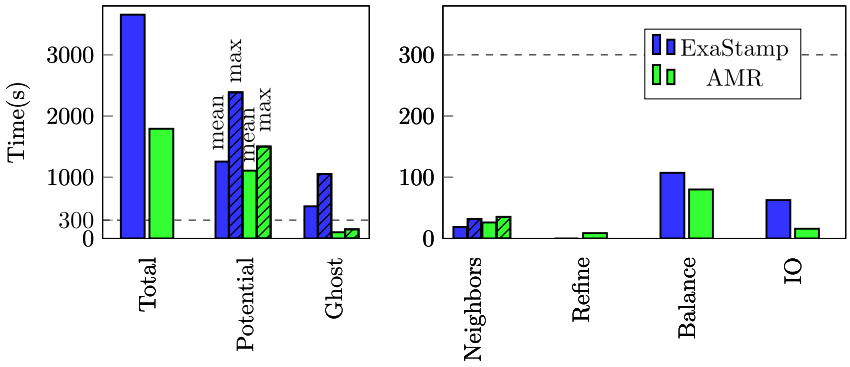

Fig. 10. Time for different sections of the micro-jetting simulation with MEAM: Total (without init), MEAM potential computation, ghost updates, Neighbors update, refinement and coarsening for AMR, dynamic load balancing and IO. (**Lower is better**).

we only compare the AMR and "classic" versions of ExaSTAMP. The experimental setup is identical to the one used in previous experiments.

Performances achieved by both ExaSTAMP versions are reported in Fig. 10. We can observe that the overall performance of ExaSTAMP AMR is much higher: AMR is able to process 0.657×10^9 atoms per second while the "classic" version of ExaSTAMP is limited to 0.322×10^9 atom/s. The AMR version is most notably able to update ghosts more efficiently. Additionally, when the number of octree cells is too small, the AMR version is able to automatically switch to the traditional parallel strategy, which exploits inner-cell parallelism. The time spent in load balancing is lower for AMR because the algorithm has fewer objects to balance: ExaSTAMP has to deal with 81,180,164 cells while AMR only has to cope with 10,087,476 octree cells.

Note that the reconstruction of Verlet lists is more expensive in the AMR version. However, the difference with the "classic" method remains negligible and does not impact the overall performance of the code.

Finally, we have successfully used ExaSTAMP AMR to perform a complete simulation of a microjetting composed of 1.3 billion of tin atoms using MEAM (Fig. 11). The simulation took almost 17 days to complete 2 millions of iterations over 512 KNLs with one post-processing output every 1000 iterations.

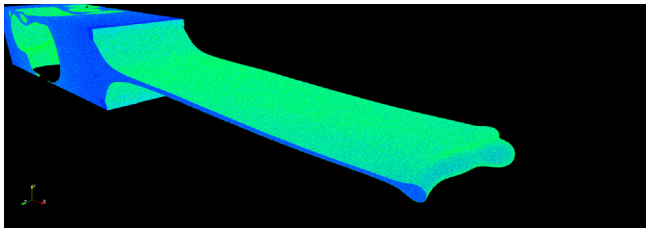

Fig. 11. Micro-jetting simulation: the shock, which propagates through the crystal, causes a phase change. This results in the ejection of a liquid tin sheet with a discontinuous edge.

4. 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 [26] brings a package of subroutines, programs and data files, written in fortran 90 and designed to make MD simulations easier and is used for production. This software is designed to perform on CPU with OpenMP and MPI, as well as on GPU. In addition there are no load balancing algorithms in dl_poly_4to compensate for a bad density distribution.

Gromacs [11] 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 bonded interactions, an all-atom version provides short-range interactions such as

Lennard Jones or Buckingham potentials. To balance the workload between MPI processes, Gromacs uses [27] the Eighth Shell Method [28].

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 [29] 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.

A solution used in many other fields is to dynamically and locally adapt the mesh according to the physical phenomenon. The first adaptive mesh refinement technique [12,13] has been introduced for hydrodynamics applications in order to reduce the computation time by refining the discretized domain (grid) where the points need to be spaced enough to keep their stability properties. The refinement criteria are based on the density or on a Richardson extrapolation [30]. The AMR has been declined for N-body simulations by recursively subdividing a space into equal cells as long as the refinement criteria are fulfilled. Bodies are stored in a k-d-tree data layout, which is a special case of the binary space partitioning (BSP) trees [31]. It is used to find the nearest neighbors of a body or to improve the collision detection between two bodies. AMR is currently used in a large amount of software [14] for various physical problems such as astrophysics with Enzo [15] or to solve PDEs with BoxLib [32] and Chombo [16]. To the best of our knowledge, our approach is the first to introduce AMR in molecular dynamics simulations.

5. Conclusion

Performing MD simulations over supercomputers is a major step towards a better understanding of physics in many scientific fields. In this work, we propose an efficient MPI, OpenMP and vectorization parallelization using the advantages of Adaptive Mesh Refinement structures to solve non-uniform large scale MD simulations by reducing the imbalance inside and between multicore cluster nodes. The octree-based strategy enables effective optimizations by reducing the number of empty cells, improving data reuse by cache blocking, and by using a lock-free task-based algorithm. Our experiments show that our implementation outperforms the LAMMPS state-of-art implementation by a factor of 1.38 on a micro-jetting and by a factor of 2.61 on a steady scenario.

Future works include auto-tuning to find the optimal maximum depth tree of octrees per subdomain, so as to reach the best tradeoff between the amount of parallelism and the efficiency of per-thread code sections. We are also investigating improvements to our load balancing algorithm to automatically identify the most appropriate triggering points. In order to deal with simulations with long-range interactions, one solution could be to combine our AMR structure with the octree-based architecture used by the fast multipole method [33]. In addition, a solution, proposed by Agullo et al. [34], included a task-based parallelism with StarPU [35] runtime that matches with our task-based dependency graph. Finally, we intend to further validate our approach on new highly-dynamic configurations.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

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

References

[1] O. Durand, L. Soulard, J. Dyn. Behav. Mater. 3 (2) (2017) 280–290.
[2] O. Durand, S. Jaouen, L. Soulard, O. Heuze, L. Colombet, J. Appl. Phys. 122 (13) (2017).
[3] D. Wolff, W. Rudd, Comput. Phys. Commun. 120 (1) (1999) 20–32.
[4] J.E. Jones, Proc. R. Soc. A 106 (738) (1924) 463–477.
[5] P.M. Morse, Phys. Rev. 34 (1) (1929) 57.
[6] E.A. Mason, J. Chem. Phys. 22 (2) (1954).
[7] M.S. Daw, M.I. Baskes, Phys. Rev. B 29 (12) (1984) 6443.
[8] M. Allen, D. Tildesley, Computer Simulation of Liquids, Clarendon Press, Oxford, 1987.
[9] L. Verlet, Phys. Rev. 159 (1) (1967) 98.
[10] S. Plimpton, J. Comput. Phys. 117 (1) (1995) 1–19.
[11] H.J. Berendsen, D. van der Spoel, R. van Drunen, Comput. Phys. Comm. 91 (1–3) (1995) 43–56.
[12] M.J. Berger, P. Colella, J. Comput. Phys. 82 (1) (1989) 64–84.
[13] M.J. Berger, J. Oliger, J. Comput. Phys. 53 (3) (1984) 484–512.
[14] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, et al., J. Parallel Distrib. Comput. 74 (12) (2014) 3217–3227.
[15] G.L. Bryan, M.L. Norman, B.W. O'Shea, T. Abel, J.H. Wise, M.J. Turk, D.R. Reynolds, D.C. Collins, P. Wang, S.W. Skillman, et al., Astrophys. J. Suppl. Ser. 211 (2) (2014) 19.
[16] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, B. Van Straalen, Chombo software package for amr applications-design document, 2000.
[17] R. Prat, L. Colombet, R. Namyst, Proceedings of the 47th International Conference on Parallel Processing, 2018, p. 48.
[18] E. Cieren, L. Colombet, S. Pitoiset, R. Namyst, European Conference on Parallel Processing, Springer, 2014, pp. 121–132.
[19] G.M. Morton, A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing, International Business Machines Company, New York, 1966.
[20] C.M. Mangiardi, R. Meyer, Comput. Phys. Comm. (2017).
[21] R. Meyer, Phys. Rev. E 88 (5) (2013).
[22] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, C. Vaughan, Comput. Sci. Eng. 4 (2) (2002) 90–97.
[23] Y. Perl, S.R. Schach, J. ACM 28 (1) (1981) 5–15.
[24] M.J. Berger, S.H. Bokhari, IEEE Trans. Comput. (5) (1987) 570–580.
[25] J.J. Jeffers, A. Sodani, Intel Xeon Phi Processor High Performance Programing, Morgan Kaufmann, 2016, pp. 443–470, Ch. 20.
[26] W. Smith, I.T. Todorov, Mol. Simul. 32 (12–13) (2006) 935–943.
[27] B. Hess, C. Kutzner, D. Van Der Spoel, E. Lindahl, J. Chem. Theory Comput. 4 (3) (2008) 435–447.
[28] S. Liem, D. Brown, J.H. Clarke, Comput. Phys. Comm. 67 (2) (1991) 261–267.
[29] J. Mohd-Yusof, Exascale Research Conference, 2012.
[30] L.F. Richardson, Philos. Trans. R. Soc. A 210 (1911) 307–357.
[31] H. Fuchs, Z.M. Kedem, B.F. Naylor, ACM Siggraph Computer Graphics, Vol. 14, ACM, 1980, pp. 124–133.
[32] M. Lijewski, A. Nonaka, J. Bell, Boxlib, 2011.
[33] L. Greengard, V. Rokhlin, J. Comput. Phys. 73 (2) (1987) 325–348.
[34] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, T. Takahashi, SIAM J. Sci. Comput. 36 (1) (2014) C66–C93.
[35] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, Concurr. Comput.: Pract. Exper. 23 (2) (2011) 187–198.