The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Hybrid parallelization of molecular dynamics simulations to reduce load imbalance

Julian Morillo · Maxime Vassaux · Peter V. Coveney · Marta Garcia-Gasulla

Received: date / Accepted: date

Abstract The most widely used technique to allow for parallel simulations in molecular dynamics is spatial domain decomposition, where the physical geometry is divided into *boxes*, one per processor. This technique can inherently produce computational load imbalance when either the spatial distribution of particles or the computational cost per particle is not uniform. This paper shows the benefits of using a hybrid MPI+OpenMP model to deal with this load imbalance. We consider LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator), a prototypical molecular dynamics simulator that provides its own balancing mechanism and an OpenMP implementation for many of its modules, allowing for a hybrid setup. In this work, we extend the current OpenMP implementation of LAMMPS and optimize it and evaluate three different setups: MPI-only, MPI with the LAMMPS balance mechanism, and hybrid setup using our improved OpenMP version. This comparison is made using the five standard benchmarks included in the LAMMPS distribution plus two additional test cases. Results show that the hybrid approach can deal with load balancing problems better and more effectively (50% improvement versus MPI-only for a highly-imbalanced testcase) than the LAMMPS balance mechanism (*only* 43% improvement) and improve simulations with issues other than load imbalance.

Julian Morillo
Pl. Eusebi Güell, 1-3, 08034 Barcelona (Spain)
Tel.: (+34) 93 413 72 48
Fax: (+34) 93 413 77 21
E-mail: julian.morillo@bsc.es

Maxime Vassaux
E-mail: m.vassaux@ucl.ac.uk

Peter V. Coveney
E-mail: p.v.coveney@ucl.ac.uk

Marta Garcia-Gasulla
E-mail: marta.garcia@bsc.es

Keywords load balance · parallel computing · molecular dynamics · MPI · OpenMP · hybrid programming model

1 Introduction and Related Work

Molecular dynamics is ubiquitous for the theoretical investigation of all ranges of materials from their structure to their properties. LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator, [47] [36]) is a classical molecular dynamics code with a focus on materials modeling. It has potentials for solid-state materials (metals, semiconductors) and soft matter (biomolecules, polymers), and coarse-grained mesoscopic systems. It can be used to model atoms or, more generically, as a parallel particle simulator at the atomic, meso, and continuum scales. It is a prototypical molecular dynamics simulation engine [36].

Molecular dynamics simulations of molecular systems beyond the micrometer and microsecond scales remain prohibitive even though parallelization and high-performance computing resources are routinely employed. Parallelization of molecular dynamics simulation can be achieved using MPI (Message Passing Interface) and a spatial decomposition of the simulation domain. The basic idea of a spatial decomposition method is to divide the physical geometry of the system under simulation into small boxes, one per processor. Each processor will compute primarily on atoms within its box. This may induce load imbalance in problems with non-uniform atom densities. The problem of load imbalance in MPI programs is well known [24] and in particular, in molecular dynamics simulations, it is widely recognized [23,38].

The challenge of the MPI load imbalance problem comes from the nature of MPI programming, where each process has its own data that can only be shared by explicit message passing. Simultaneously, the nature of load imbalance is dynamic and affected by many factors, therefore difficult to predict. Figure 1 presents a taxonomy of the different solutions proposed to cope with load imbalance. Traditionally, such solutions can be divided into two groups: the ones that are applied *before* execution and the ones applied *during* execution. In the first group, we can consider different mesh partitioners [50,32]. These solutions are static and cannot address load changes during the execution. Moreover, they need to be tuned for new architectures, algorithms, or simulations.

The approaches applied during the execution can be classified as solutions that "move" data and solutions that change computational resources. The methods that redistribute data [29,42] usually execute a load balancing algorithm with a given frequency. This algorithm determines if there is a load imbalance problem, when necessary, computes a new partition, and finally redistributes the data as needed. These approaches are not able to deal with very dynamic load imbalance. They also need to be able to measure load and decide how frequently the load balancing algorithm is executed because the cost of redistributing the data is not negligible. Usually, these solutions are

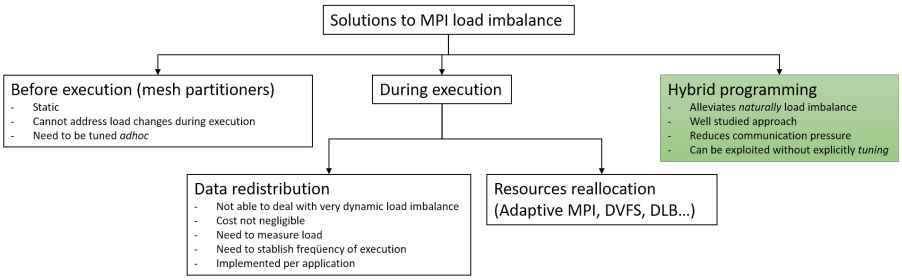


Fig. 1: Taxonomy of solutions to load imbalance. This paper falls conceptually into the box marked in green.

implemented within each application; LAMMPS provides its own *balancing* mechanisms for both *before* [5] and *during* execution [7].

In the category of solutions applied during execution that change the computational resources, we find different approaches. Adaptive MPI [30], for example, relies on virtualized processes, and the runtime is in charge of scheduling them to achieve a good load balance. They run on top of CHARM++ [18] which implies a change in the programming language and model, so although this might be an interesting comparison, the changes needed in the code [1] leave it out of the scope of this work. Etinski et al. [25] propose to use the Dynamic Voltage and Frequency Scaling (DVFS) reducing the frequency of less loaded processes to save power. Also, in this category, we find the Dynamic Load Balancing library [27,28]; this library changes the computational resources assigned to the different MPI processes to help balance their load.

We propose to use the hybrid programming model MPI+OpenMP [39, 41] to alleviate the load balance problem. Hybrid parallelization is a well-studied area that has already shown its benefits when compared to other approaches [22]. The idea of using an OpenMP/MPI hybrid approach for improving the performance of molecular dynamics simulations has already been explored in the past [33,31,34]. Kunaseth et al. [33], for example, take benefit of the hybrid approach to propose and analyze two data-privatization thread scheduling algorithms focusing on the memory footprint they pose. Jung et al. [31] develop a scheme by combining a cell-wise version of the midpoint method with pair-wise Verlet lists based on a hybrid approach. Its evaluation is limited to long-range interactions, while we cover all short-range, mid-range, and long-range interactions. Anirban et al. [34] discuss computational bottlenecks and challenges in MD to present a hybrid scheme for systems with short-range interatomic interactions.

The hybrid programming model approach offers the advantages of a hybrid code: improves the load balance, and at the same time reduces the pressure on the communication between MPI processes. In contrast with other approaches that need to be programmed *adhoc* for each input or architecture, an OpenMP [14] parallelization can be exploited in many situations without needing to tune the code specifically. Deng et al. [23], for example, describe

an adaptive method for achieving load balance in parallel computations that is tested on standard short-ranged parallel molecular dynamics calculations. Our proposal, in contrast, is to use a hybrid (MPI+OpenMP) approach. We argue that the use of OpenMP can help alleviate MPI scaling issues, especially the ones related to load balance, and that this can be done straightforwardly by leveraging on the OpenMP characteristics. Moreover, our evaluation is not limited to short-ranged molecular dynamics calculations: mid-range and long-range simulations are also considered, including all the benchmarks provided by the LAMMPS distribution, together with two extra testcases with quite different characteristics regarding load balance.

We use LAMMPS to demonstrate the benefits of hybrid parallelization (as a generic parallelization paradigm), and therefore the results apply not only to LAMMPS but to all MD simulation codes (NAMD [12], GROMACS [4], etc.). Many LAMMPS modules have OpenMP versions for shared-memory parallelism, allowing for hybrid setups in which MPI+OpenMP configurations can be run. Although there are OpenMP versions of many LAMMPS modules, many of them lack an OpenMP implementation [6]. Other ones present a parallelization pattern [16] that is not optimum for performance or programmability. This leaves MPI as the only parallel option for these parts of the code. Nonetheless, the code is designed to be easily modified or extended with new functionality. In this paper, we use such a feature to parallelize with OpenMP some code regions that lack this parallelism and improve the original OpenMP implementation in other sections of code. As it will be shown in Section 4, the actual changes to the implementation are relatively simple, pointing out another benefit of the hybrid programming when compared to other approaches. A more comprehensive parallelization, i.e., eliminating all the sequential parts when running with OpenMP is totally out of the scope of this paper.

The main contributions of this paper are:

1. we present some enhancements to the LAMMPS OpenMP implementation;
2. we provide an extensive evaluation of this improved LAMMPS hybrid version against the MPI-only version as a baseline case but also against the LAMMPS balance mechanism mentioned previously;
3. we show how the hybrid approach can deal with imbalance issues better than the balance mechanism and, furthermore, it can improve performance in cases where load imbalance is not the main problem.

This paper is organized as follows. Section 2 presents a description of LAMMPS and the benchmarks/testcases used for the evaluation together with the performance analysis tools and efficiency metrics employed. In Section 3 we explain why MPI load imbalance is a problem, the difficulties to address it, and why it is very common in molecular dynamics simulations, together with the two compared approaches to solve it: the LAMMPS balancing mechanism and the use of a hybrid model. Section 4 includes a description of our proposed additions to the LAMMPS OpenMP implementation. Section 5 contains the environment employed for the evaluation together with a characterization of

the benchmarks/testcases used. Finally, a complete performance comparison of the three evaluated scenarios is done for all the considered benchmarks/testcases. Section 6 concludes our study with comments and remarks.

2 Background

2.1 LAMMPS

The Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS, [9,36]) is a highly parallelized code for the simulation of classical molecular dynamics. LAMMPS is widely used by the materials science community. LAMMPS is intended for parallelism and runs as well on single processors as in parallel using the Message-Passing Interface (MPI) and a spatial decomposition of the simulation domain. Many LAMMPS modules provide accelerated performance on CPUs, GPUs, and Intel Xeon Phis. LAMMPS is distributed by Sandia National Laboratories as an open-source code under the terms of the GPL.

LAMMPS is used to simulate the dynamics and the properties of a wide range of systems including amorphous and crystalised materials, proteins, and much more. The need for computational chemists to simulate larger systems for longer periods of time has continuously pushed the improvement of LAMMPS scalability. Besides, LAMMPS is now also frequently coupled with other tools such as machine-learning or continuum model simulators for scale-bridging purposes. As a result, large ensembles of molecular dynamics simulations can be simulated simultaneously. LAMMPS has already been used to simulate the dynamics of tens of billions of atoms. On what is referred to as the "Lenard-Jones" benchmark, the highest throughput recorded was 4.34 TFlops in 2005 on a $40x10^9$ atoms simulation. In a more recent study LAMMPS was shown to reach $2.35x10^{-8}$ s/atom/timestep [13].

2.2 Benchmarks

For evaluation purposes, we consider in this paper a combination of LAMMPS standard benchmarking scenarios and a couple of additional scenarios triggering more specifically load-balancing issues (Table 1). LAMMPS features a set of five standard benchmarks representative of the diversity of systems that can be simulated. We assume that parallel efficiency is highly impacted by the range of interatomic potential interactions. We, therefore, can classify the five scenarios into one of the following three classes of problems:

1. Short-range interacting systems: each particle interacting on average with respectively 7 and 5 neighbours.
2. Mid-range interacting systems: each particle interacting on average with respectively 45 and 55 neighbours.

3. Long-range interacting system: the "Rhodopsin" scenario [17] integrates long-range Coulomb interactions, resulting in each particle interacting on average with 440 neighbours.

Each scenario simulates the dynamics of 32,000 atoms. The data required for the simulation of these benchmarks is included in the distribution of LAMMPS. Further details on the constraints applied during the simulation of the scenarios can be found on the LAMMPS Benchmarks page website (https://lammps.sandia.gov/bench.html).

Benchmarks	Short-range	**Granular chute** [3]: convective flow of falling particles interacting via a frictional history potential **Polymer chain** [15]: thermal fluctuations of hundred monomers long chains
	Mid-range	**EAM** [2]: thermodynamic fluctuations of a metallic copper bulk solid which atoms interact via the embedded atom method (EAM) potential **Lennard-Jones** [10]: thermodynamics of an atomic fluid
	Long-range	**Rhodopsin** [17]: conformation changes of the rhodopsin protein in a solvated lipid bilayer, the CHARMM force-field is used to describe atoms pairwise and multi-body interactions.
Additional scenarios		**Epoxy**: non-equilibrium dynamics of highly-crosslinked epoxy polymer chains under applied stretching **CG-GO**: coarse-grained (CG) graphene-oxide (GO) sheet embedded in a polymer precursor

Table 1: Benchmarks.

In addition to LAMMPS standard benchmark scenarios, we introduce the simulation of an epoxy resin [48] and the simulation of a graphene-based nanocomposite [45] (Table 1). The epoxy resin is constrained with a fixed number of atoms and temperature. Meanwhile, the volume is controlled throughout the simulation and varied at a fixed strain rate. In the second scenario, the graphene-oxide (GO) sheet is a dense, two-dimensional packing of carbon atoms, while polymer precursors consist in a disordered phase of poly(methyl methacrylate) (PMMA) precursors. The "CG-GO" scenario simulates the dynamics of GO during annealing, the number of atoms and the volume of the system are fixed and the temperature is increasing from 300K to 500K.

These two custom systems face specific computational efficiency issues which justified the improvement of the current load balancing methods available in LAMMPS. We will perform efficiency measurements that highlight existing bottlenecks and propose load balancing improvement based on the analysis of the measurements.

2.3 Performance Tools and Efficiency Metrics

In this paper, we go one step further and aim at gaining insight on the reasons for the performance achieved in the different situations at study. For this, we rely on performance analysis tools and the performance methodology promoted by the Center of Excellence (CoE) for Performance Optimization and Productivity (POP) [1].

The performance analysis tools used in this work are the following:

Extrae: To obtain traces of the different executions, it supports MPI and OpenMP among other parallel programming models [20,43]. Extrae uses PAPI [46] to collect information regarding performance hardware counters.

Paraver: To visualize the traces obtained with Extrae. It allows us to analyze in detail the behaviour of the program and also to compute the performance metrics [21,35].

The POP CoE has defined a systematic methodology for performance analysis. This methodology is independent of the tool being used for the analysis and defines a set of performance metrics. These metrics are well defined, accepted by the community, and meaningful, pointing the analysts to the main factors affecting the performance and scalability of the code [49,19]. In this paper, we use some of these metrics as they allow us to compare the different LAMMPS benchmarks using a common ground.

The POP metrics are hierarchical and multiplicative, meaning that the parent metric can always be computed as the product of its child metrics. Each metric can get values between 0 and 100, and the metric indicates how well that indicator is performing. For example, a load balance of 70% indicates that 30% of the CPU time used is lost due to load imbalance, and also that by addressing the load imbalance problem we will be able to improve the execution by at most 30%. Specifically, we are going to use 3 metrics from the POP methodology: Parallel Efficiency, Load Balance, and Communication efficiency.

These efficiency metrics are based on the simplification of a process into two states: the state in which it is performing computation, which is called *Useful* (blue), and the state in which it is not performing computation, e.g., communicating to other processes, which is called *Not useful* (red).

An example of this simplification can be seen in Figure 2 where we can see two processes, named p_1 and p_2 running from left to right (horizontal axis represents time). We can see how their execution changes between the two states from *Useful* to *Not useful* and vice versa during their execution.

We call $P = \{p_1, \ldots, p_n\}$ the set of MPI processes, and n the number of MPI processes. For each MPI process p, we define the set $U_p = \{u_1^p, u_2^p, \ldots, u_{|U|}^p\}$ of the time intervals where the application is performing useful computation (the set of blue intervals). We define the sum of the durations of all useful time intervals in a process p as shown in Equation 1, and we call it the useful duration of a process.

[1] https://pop-coe.eu/

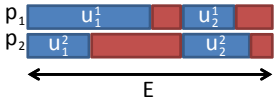


Fig. 2: State evolution of two processes

$$D_{U_p} = \sum_{U_p} \blacksquare = \sum_{j=1}^{|U_p|} u_j^p \tag{1}$$

Similarly we can define $\overline{U}_p$ and $D_{\overline{U}_p}$ for the red intervals.

We also define the *elapsed time* E as $E = max_{p=1}^{n}[D_{U_p} + D_{\overline{U}_p}]$. The elapsed time is the total duration of the execution (as depicted in Figure 2).

Parallel Efficiency. The *Parallel Efficiency* (PE) indicates the amount of time that is being lost due to the parallelization of the code. Or, equivalently, the ratio between the time consumed on useful computation and the total consumed CPU time. As we said the *Parallel efficiency* PE can be computed as the product of its children, in this case the *Load balance* LB and *Communication efficiency* CE and is defined as shown in Equation 2.

$$PE = \frac{D_{U_p}}{E * n}; PE = LB * CE \tag{2}$$

Load Balance. *Load balance* measures the efficiency loss due to different loads (useful computation) for each process. Its definition can be seen in Equation 3.

$$LB = \frac{\sum_{i=1}^{n} D_{U_i}}{n * \max_{i=1}^{n} D_{U_i}} \tag{3}$$

Communication efficiency. Finally, the *Communication efficiency* is the efficiency loss for communicating data; it can be divided into two child metrics *Serialization efficiency* and *Transfer efficiency*. Serialization corresponds to time lost due to synchronizations between different processes, i.e. when one process needs to wait for another one. Transfer is the time lost in any kind of MPI overhead, it includes different factors such as network bandwidth, communication latency, or implementation overheads. The definition of *Communication efficiency* can be found in Equation 4.

$$CE = \frac{\max_{i=1}^{n} D_{U_i}}{E} \tag{4}$$

3 Challenges and Proposed Approaches to Load Imbalance

3.1 Imbalance in Molecular Dynamics Simulations

Molecular dynamics (MD) is a commonly used tool for simulation of the structural, thermodynamic, and transport properties of biological and polymeric systems on the picosecond to nanosecond timescale. During a timestep of the MD simulation, forces are computed on each atom due to its interaction with other atoms, and atoms move by integrating simple Newtonian equations of motion [38].

The parallel nature of MD simulations has long been recognized [38,26]. The overall calculation on P processors should scale as N/P, N being the total number of atoms in the simulated system. For general molecular systems simulated on message-passing machines, most parallel implementations have used the $replicated-data$ technique [44] where a copy of all N atomic positions is stored on each of P processors. This enables easy computation and load-balancing. However, at each timestep the interprocessor communication needed to globally update a copy of the N-vector of atom positions scales as N, independent of P. Thus replicated-data methods do not scale to large numbers of processors. An alternative known as $force-decomposition$ scales as $N/\sqrt{P}$ but is still sub-optimal [37]. For large N/P ratios, spatial-decomposition methods are clearly the best algorithmic choice. By subdividing the physical volume among processors, most computations become local and communication is minimized so that optimal N/P scaling can be achieved. Such a method is used by LAMMPS.

The basic idea of a spatial decomposition method for MD is to divide the physical geometry into small boxes, one per processor. Each processor will compute primarily on atoms within its box. This may induce load imbalance in problems with non-uniform atom densities.

3.2 Balancing

To alleviate the balancing problem, LAMMPS provides the `balance` command [5]. This command adjusts the size and shape of processor sub-domains within the simulation box, to attempt to balance the number of atoms or particles and thus indirectly the computational cost (load) more evenly across processors. The load balancing is "static" in the sense that this command performs the balancing once, before, or between simulations. The processor sub-domains will then remain static during the subsequent run. To perform "dynamic" balancing, LAMMPS provides the `fix balance` command, which can adjust processor sub-domain sizes and shapes on the fly during a run.

Load-balancing is typically most useful if the particles in the simulation box have a spatially-varying density distribution or when the computational cost varies significantly between different particles. For example, a model of a vapor/liquid interface, or a solid with an irregular geometry containing void

regions. In these cases, LAMMPS default of dividing the simulation box volume into a regular-spaced grid of 3d bricks, with one equal-volume sub-domain per processor, may assign numbers of particles per processor in a way that the computational effort varies significantly. This can lead to poor performance when the simulation is run in parallel ([5],[47]).

The balancing can be performed with or without per-particle weighting. With no weighting, the balancing attempts to assign an equal number of particles to each processor. With weighting, the balancing attempts to assign an equal aggregate computational weight to each processor, which typically induces a different number of atoms assigned to each processor. The weight assigned to a particle is defined *a priori* by the user based on his knowledge of the particle, for example, the expected number of neighbours and interactions.

3.3 Hybridization

It is not a trivial task to determine the optimal model (pure MPI vs MPI + OpenMP) to use for some specific application. Although pure MPI can sometimes outperform hybrid, it is not less true that lots of counterexamples do exist and results tend to vary with input data, problem size, etc. even for a given code. In order to get optimal scalability one should in any case try to implement the following strategies:

- Reduce synchronization overhead
- Reduce load imbalance
- Reduce computational overhead and memory consumption
- Minimize MPI communication

Works like [40] pinpoint cases where hybrid programming model (MPI + OpenMP) can indeed be the superior solution because of reduced communication needs and memory consumption, or improved load balance.

Hybridizing the code can help alleviate MPI scaling issues, especially the ones related to load balance as the load balance within OpenMP is addressed straightforwardly when using a dynamic schedule with work-sharing or the tasking model (i.e. generating explicit tasks that will be dynamically executed by threads when they become idle).

To perform the tests with the *hybrid* versions of the benchmarks we have made use of OpenMP features already provided by the LAMMPS library. Version *lammps-20Nov19* [8] has been used. However, and guided by the *Epoxy* testcase, some modifications have been added to the OpenMP implementation that are described in the following section.

4 Improvements and extensions to the LAMMPS OpenMP implementation

This section presents a comprehensive description of all the changes performed within LAMMPS OpenMP implementation. As stated in Section 1, LAMMPS

is a huge piece of code so a complete parallelization with OpenMP is totally out of the scope of this paper: this section presents code modifications to optimize the execution of just one of the testcases (in this case *epoxy*). Other testcases/benchmarks may, or may not, use the affected LAMMPS modules. Nonetheless, the applied analysis and solutions are sufficiently general so that they could be spotted in other parts of LAMMPS, in other MD codes or in any code in general.

The use of the performance tools described in Section 2.3 to trace and analyze the execution of the *epoxy* testcase allowed us to find a source of load imbalance in `void NPairHalfBinNewtonTri::build(NeighList *list)` function (blue regions in Figure 3). The bottom part of Figure 3 shows a timeline with the execution of two LAMMPS iterations for 48 OpenMP threads. The upper timeline represents the execution with 48 MPI ranks (included for reference, both timelines are at the same timescale). It can be seen that the OpenMP parallelization of `void NPairHalfBinNewtonTri::build(NeighList *list)` alleviates the load imbalance, i.e. the differences in the blue bars representing the running time of each thread are reduced. Note, however, that the MPI execution is almost two times faster than the OpenMP one due to the sequential parts (i.e. the parts not parallelized by OpenMP) in the latter.

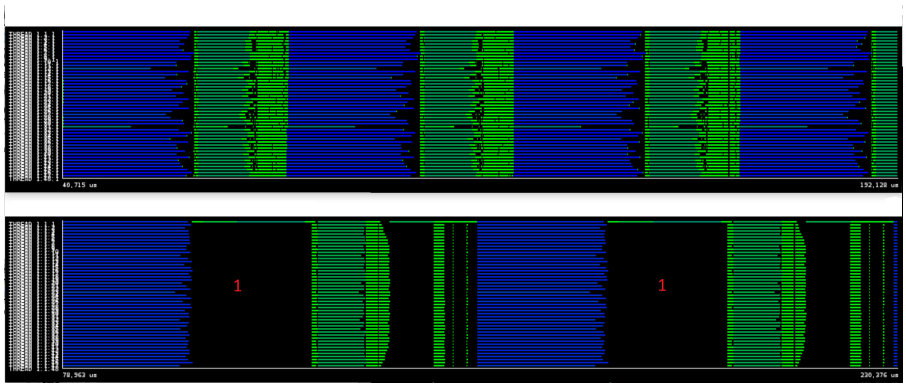


Fig. 3: MPI-only (top) vs OpenMP-only (bottom) execution: the load imbalance is alleviated (blue region) but there are significant parts not OpenMP parallelized. (Duration=151.41 ms)

4.1 OpenMP taskification

Given the timeline in Figure 3, our first work was to parallelize with OpenMP the biggest sequential part (marked with a red 1 in the timeline). This part corresponds to `Neighbor::build_topology()` function.

This function consists of 4 calls to 4 different functions `build` (each one from a different class). Each of these 4 functions have a very similar structure

that consists in a computation phase that ends up with an `MPI_Allreduce` of an `int` calculated in this computation phase (among other things). The 4 functions work with different data structures, therefore they are independent of each other and can be run in parallel.

The objective was to annotate these 4 calls to `build` functions with OpenMP tasks in order to allow their execution in parallel by different threads. Besides, the issue associated with the `MPI_Allreduce` command remains at the end of each function. The solution consists in moving these communications out of the 4 functions and putting them one level above in the `Neighbor::build_topology()` function. The idea, then, is to have at the end 4 tasks with the computation of the 4 `build` functions and one final task with the 4 communications. In this way, we delay MPI communications as much as possible, preventing unnecessary waiting times if there is imbalance between MPI ranks in some of the 4 computation phases.

In order to allow the MPI communications to be executed at the end of the 4 `build` computations we need to move them outside of each function. To do that, a change in the signature of the functions is needed: we need them to return an `int` (the value shared in the `MPI_Allreduce`) instead of `void`. Then, in each of the `build` functions, the calculated `int` in the computation phase is returned by the function instead of being directly shared with other MPI ranks through the `MPI_Allreduce` call. These returned values are then used in the new `MPI_Allreduce` calls located in `void Neighbor::build_topology()` function. As future work, it could be considered to overlap computation and communication through the use of `MPI_Iallreduce`. However, enough parallelism has already been extracted in this code region (see Figure 4). Section 4.4 presents an example of how to overlap computation and communication.

Listing 1 presents a skeleton of the final implementation of `Neighbor::build_topology()` function. As it can be seen, 4 new local variables are declared: the variables will be used to store the values returned by each of the `build` functions and, in turn, to honor the dependencies between the 4 computation tasks and the communications task.

```
void Neighbor::build_topology()
{
int nmissing_bond, nmissing_angle, nmissing_dihedral,
    nmissing_improper, all;
#pragma omp parallel
#pragma omp single
{
#pragma omp task depend(out:nmissing_bond)
  if (force->bond) {
    nmissing_bond = neigh_bond->build();
    ...
  }
#pragma omp task depend(out:nmissing_angle)
  if (force->angle) {
    nmissing_angle = neigh_angle->build();
    ...
  }
#pragma omp task depend(out: nmissing_dihedral)
```

```
  if (force->dihedral) {
    nmissing_dihedral = neigh_dihedral->build();
    ...
  }
#pragma omp task depend(out: nmissing_improper)
  if (force->improper) {
    nmissing_improper = neigh_improper->build();
    ...
  }
#pragma omp task depend(in:nmissing_bond,nmissing_angle,
    nmissing_dihedral,nmissing_improper)
{
  MPI_Allreduce() x 4
    ...
} //end task
#pragma omp taskwait
} //end single and parallel
}
```

Listing 1: Modified `void Neighbor::build_topology()` code including OpenMP taskification

The present modifications significantly reduce the execution time of the biggest sequential part and work efficiently for a small number of threads. Note, however, that we are generating only 5 OpenMP tasks and only 4 of them can run in parallel as the communications one needs to wait for the execution of the others. So, when moving to the extreme case of using 48 threads (the number of cores on the target machine), more parallelism is needed. To accomplish that, the loops that make the calculations inside each of the 4 build functions have also been taskified: this allows the generation of sufficient tasks to feed all threads.

The results of these modifications can be appreciated in Figure 4 where the red lines mark explicitly the region of code affected by these changes and the reduction in execution time (upper part of the figure corresponds to the original LAMMPS OpenMP implementation and the bottom timeline corresponds to our improved OpenMP version). Note that quantifying how much potential benefit is expected from each code modification before applying it is a difficult job as it depends on many factors like the executed benchmark, the input used, or even the machine where it is being executed. So we rather prefer to show these benefits through the use of traces of real executions before and after applying the changes, as in Figure 4. This does not imply a lack of methodology: code changes are applied in the order in which more benefits are expected. As an example, note how the first region to be parallelized is the biggest one (i.e. the most time-consuming, marked with a red 1 in Figure 3).

4.2 Use of OpenMP dynamic scheduler

The second code modification has been done in the previous mentioned function `NPairHalfBinNewtonTri::build(NeighList *list)`. It has been shown

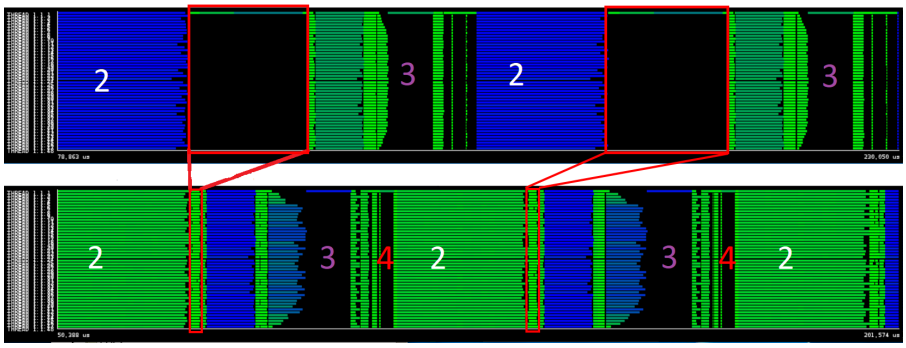


Fig. 4: Improved version (bottom timeline) compared to the LAMMPS original OpenMP implementation (upper timeline): red squares mark the region of code affected by the parallelization. (Duration=151.41 ms)

how by simply using the LAMMPS OpenMP implementation the detected imbalance was alleviated. A close look at the source code shows, however, that the LAMMPS OpenMP implementation does a `static` partition of the workload (like MPI does) so there is still some room for improvement in this part of the code. This static partition of the workload is done through the use of 3 macros defined in `npair_omp.h` These 3 macros are widely used along all the LAMMPS OpenMP code so the same code refactoring done in this section could be done in many other parts of the code.

The macro actually performing the workload partition is `NPAIR_OMP_SETUP(num)`. It does so by dividing `num` among the number of available threads (i.e. in a loop of `num` iterations, defines the starting and end iteration that must be executed by each thread by setting `ifrom` and `ito` variables).

Once it is understood how these macros work, it is quite straightforward to implement the proposed approach. As it can be seen in Listing 2 the proposed change simply consists in substituting the original `for` that uses `ifrom` and `ito` variables by another one that, instead, starts at 0 and ends at `nlocal` (i.e. the value used in `NPAIR_OMP_SETUP` in this case). Of course, the new `for` is surrounded by a `#pragma omp for schedule(dynamic)` to do the worksharing (note that a `#pragma omp parallel` is not needed as it is already present at the beginning of the function in the original code, see Listing 2).

```
void NPairHalfBinNewtonTriOmp::build(NeighList *list)
{
  ...
  NPAIR_OMP_INIT;
#if defined(_OPENMP)
#pragma omp parallel default(none) shared(list)
#endif
  NPAIR_OMP_SETUP(nlocal);
  ...
  #pragma omp for schedule(dynamic,50)
```

```
  for (i = 0; i < nlocal; i++) {
//  for (i = ifrom; i < ito; i++) {
        ... (Computation)

  }
  NPAIR_OMP_CLOSE;
}
```

Listing 2: Sketch of `NPairHalfBinNewtonTriOmp::build(NeighList *list)` code including the dynamic OpenMP schedule

For our guiding testcase, we found out (just by trying different values) that a chunk size of 50 (as shown in Listing 2) gave us a good trade-off between load balance and overhead. So it does not pretend to be the optimum value. Note, in any case, that the specific optimum value would depend on many factors like the testcase, the input set, the number of threads being used on a given execution, or the level of load imbalance between iterations. The important take-away from this subsection is to let the OpenMP runtime manage any potential load balancing issue instead of distributing statically the work.

4.3 Enabling more OpenMP parallelism

Looking into the generated traces showed that there was a relatively large portion of code not OpenMP parallelized just before the execution of `void NPairHalfBinNewtonTriOmp::build(NeighList *list)`. This is shown in Figures 4 and 5 marked with a red 4. The bottom timeline of Figure 5 represents (at the same timescale) the same part of the execution after parallelizing some functions in this region of code. The red lines going from one timeline to the other show the reduction in execution time achieved when using the newly implemented parallel regions. Three different functions have been parallelized in this section of code but let us focus on the most important in terms of execution time: `void NBinStandard::bin_atoms()` (see Listing 3).

```
void NBinStandard::bin_atoms()
{
  ...
#if defined(_OPENMP)
#pragma omp parallel for
#endif
  for (i = 0; i < mbins; i++) binhead[i] = -1;

  ...

  if (includegroup) {
    int bitmask = group->bitmask[includegroup];
    for (i = nall-1; i >= nlocal; i--) {
      if (mask[i] & bitmask) {
        ...
      }
    }
    for (i = atom->nfirst-1; i >= 0; i--) {
```

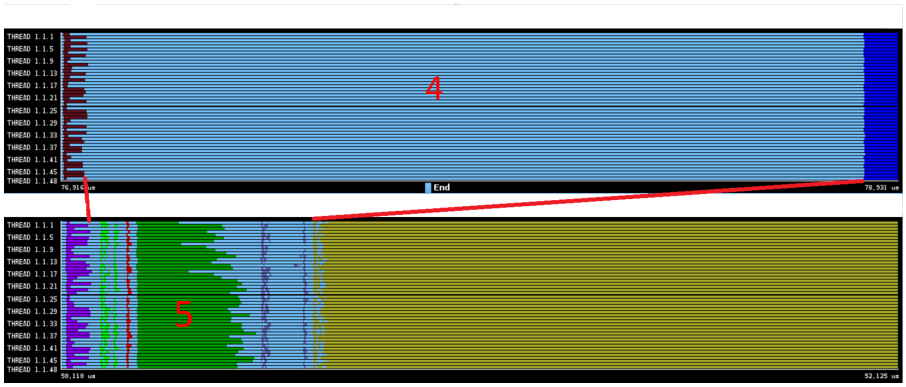


Fig. 5: Influence of the newly implemented parallel regions: a comparison before (top) and after (bottom) parallelization; the compute time associated with region 4 (red) is reduced by a 70% factor as shown by the red lines. (Duration = 2.02 ms)

```
      ...
    }

  } else {
#if defined(_OPENMP)
#pragma omp parallel for private(ibin)
#endif
    for (i = nall-1; i >= 0; i--) {
      ibin = coord2bin(x[i]);
      atom2bin[i] = ibin;
      bins[i] = binhead[ibin];
      binhead[ibin] = i;
    }
  }
}
```

Listing 3: Sketch of `void NBinStandard::bin_atoms()` code including the new added parallelization

Two `parallel for` worksharing constructs have been defined: the first one is not very relevant in terms of execution time and it corresponds to the red area in the bottom timeline in Figure 5. The important one is located at the bottom of the Listing 3 which corresponds to the green region (marked with a red 5). As it can be seen it is a very simple `parallel for` that just needs to privatize `ibin` to work correctly.

Another important region of code candidate for enabling more OpenMP parallelism is the one marked with a purple 3 in Figure 4. This area corresponds to the execution of `PPPM::poisson_ik_triclinic()` method.

The method consists in three differentiated parts (one for each `x`, `y` and `z` direction) with identical structure: an initial loop, a call to `fft2->compute()`

and, last, three nested loops. Unfortunately, the use of a variable (`n`) prevents a direct parallelization of both the initial and the three nested loops.

The solution for the first loop is to incorporate the management of variable `n` (initialization and increment) to the control structure of the loop (see Listing 4). Once this is done, a simple `pragma omp parallel for` suffices.

```
void PPPM::poisson_ik_triclinic()
{
  int i,j,k,n;

  // x direction gradient
  #pragma omp parallel for
  for (i = 0, n = 0; i < nfft; i++, n = n+2) {
    work2[n] = fkx[i]*work1[n+1];
    work2[n+1] = -fkx[i]*work1[n];
  }

  fft2->compute(work2,work2,-1);

  int BS = (nyhi_in - nylo_in + 1) * (nxhi_in - nxlo_in + 1) * 2;
  #pragma omp parallel
  {
  #pragma omp for private(n,j,i) nowait
  for (k = nzlo_in; k <= nzhi_in; k++) {
    n = (k - nzlo_in) * BS;
    for (j = nylo_in; j <= nyhi_in; j++) {
      for (i = nxlo_in; i <= nxhi_in; i++) {
        vdx_brick[k][j][i] = work2[n];
        n += 2;
      } //i
    } //j
  } //k

  // y direction gradient
  #pragma omp for
  for (i = 0, n = 0; i < nfft; i++, n=n+2) {
    work2[n] = fky[i]*work1[n+1];
    work2[n+1] = -fky[i]*work1[n];
  }
  } //parallel

  ... //(rest of code omitted)

}
```

Listing 4: Sketch of parallelized version of `void PPPM::poisson_ik_triclinic()` method

The solution for the second case (the three nested loops) is trickier: we need to privatize variable `n` and to do that we need to *manually* calculate the initial value for `n` at each iteration of the outer-most loop. This is easilly done through the use of the added variable `BS` that stores the increments of the variable `n` in the two inner-most loops. Once all of these is done (see Listing 4), the outer-most loop can be parallelized by simply privatizing `n`, `j` and `i`.

As a last comment, the parallel region opened for the nested loops of *x direction* is used for the first loop of *y direction* as depicted in Listing 4. The same is done between *y direction* and *z direction* (not shown in the Listing).

4.4 Overlapping computation and communication

Last, but not least, we show here how to effectively overlap computation with MPI communication. More precisely, we have worked in `remap_3d` function, which is called several times in the same region of code mentioned in the last part of the previous subsection. The function consists of 4 differentiated parts:

1. A sequence of `MPI_Irecv` calls to receive data from other processes.
2. A sequence of (`pack`, `MPI_Send`) calls that packs and sends data to other processes.
3. A call to `pack` and `unpack` to manage the data of the calling process.
4. A sequence of (`MPI_Waitany`, `unpack`) to wait for the corresponding `MPI_Irecv` to get the data and put it in the required memory location.

The changes done in the code to allow computation and communication overlapping can be seen in Listing 5 and are summarized in the following items:

1. All code is wrapped by a `parallel` and a `single` constructs to create the parallel OpenMP region and allow only one thread to enter the code to create tasks.
2. The loop corresponding to point number 2 of the original code has been split into two loops: one loop doing all the packs and the other doing all the `MPI_Send`. The loop doing the packs has been moved to the very beginning of the function and each pack has been defined as a task.
3. To allow the previous taskification, `plan->sendbuf` has been redefined (not shown) and now is a *buffer of buffers* indexed by `isend`: this allows for all pack tasks to be independent.
4. As it is independent of the rest of the communications, the self-data management of point 3 of the original code has been moved next and taskified.
5. A `taskwait` is needed just after the loop with `MPI_Irecv` because the following `MPI_Send` needs the tasks with packs defined in point 2 to be finished.
6. Finally, the unpacks of the last loop have been defined as tasks: in this way, the following `MPI_Waitany` does not need to wait for the previous unpack to finish.

```
void remap_3d(FFT_SCALAR *in, FFT_SCALAR *out, FFT_SCALAR *buf,
              struct remap_plan_3d *plan)
{
... // (omitted code)

#pragma omp parallel
#pragma omp single
```

```
{
    for (isend = 0; isend < plan->nsend; isend++) {
      #pragma omp task firstprivate(isend)
      plan->pack(&in[plan->send_offset[isend]],
                 &plan->sendbuf[isend*plan->sendbuf_size],&plan->
                     packplan[isend]);
    }

    // copy in -> scratch -> out for self data
    if (plan->self) {
      isend = plan->nsend;
      irecv = plan->nrecv;
      #pragma omp task firstprivate(isend,irecv)
      {
      plan->pack(&in[plan->send_offset[isend]],
                 &scratch[plan->recv_bufloc[irecv]],
                 &plan->packplan[isend]);
      plan->unpack(&scratch[plan->recv_bufloc[irecv]],
                   &out[plan->recv_offset[irecv]],&plan->
                       unpackplan[irecv]);
      }
    }

    // post all recvs into scratch space
    for (irecv = 0; irecv < plan->nrecv; irecv++) {
      MPI_Irecv(&scratch[plan->recv_bufloc[irecv]],plan->recv_size
          [irecv],
                MPI_FFT_SCALAR,plan->recv_proc[irecv],0,
                plan->comm,&plan->request[irecv]);
    }
#pragma omp taskwait
    // send all messages to other procs
    for (isend = 0; isend < plan->nsend; isend++) {
      MPI_Send(&plan->sendbuf[isend*plan->sendbuf_size],plan->
          send_size[isend],MPI_FFT_SCALAR,
               plan->send_proc[isend],0,plan->comm);
    }

    // unpack all messages from scratch -> out
    for (i = 0; i < plan->nrecv; i++) {
      MPI_Waitany(plan->nrecv,plan->request,&irecv,
          MPI_STATUS_IGNORE);
      #pragma omp task firstprivate(irecv)
      plan->unpack(&scratch[plan->recv_bufloc[irecv]],
                   &out[plan->recv_offset[irecv]],&plan->
                       unpackplan[irecv]);
    }
} //parallel and single

... // (omitted code)
}
```

Listing 5: Sketch of the modified code in `remap_3d()` method, including the code reordering and the taskification of packs and unpacks

Figure 6 shows how computation and communication have been effectively overlapped. The timelines correspond to a trace of a run with 8 MPI processes

with 6 OpenMP threads each. The upper timeline represents the MPI calls (being pink → MPI_Irecv, blue→ MPI_Send, and green → MPI_Waitany) and the bottom timeline represents task execution. Figure 6 is a zoom of just one invocation of `remap_3d` method for the first 2 processes (12 threads in total) used in the execution. In this case, `isend = irecv = 3` so each process executes 3 pack tasks + the pack/unpack task corresponding to the self data (the tasks on the left part), and 3 unpack tasks (on the right). It can be seen how the packs are now overlapped with the `MPI_Irecv` calls at the beginning and how the `MPI_Waitanys` at the end do not need to wait for the execution of the previous unpack.

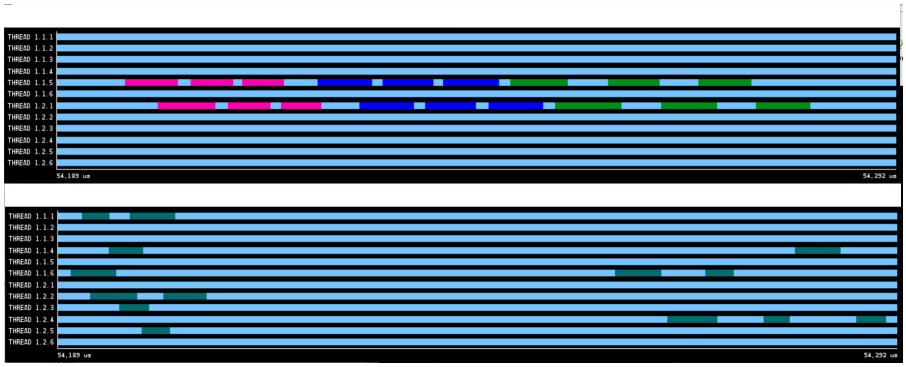


Fig. 6: Overlapping computation and communication in the `remap_3d` method. MPI calls (top timeline) and task execution (bottom timeline) are overlapped during 40% of communication time. (Duration =103 μs)

5 Evaluation

5.1 Environment

The experiments have been performed on MareNostrum4 [11]. This supercomputer is based on Intel Xeon Platinum processors from the Skylake generation. It is a Lenovo system composed of SD530 Compute Racks, an Intel Omni-Path high-performance network interconnect, and running SuSE Linux Enterprise Server as the operating system. Compute nodes are equipped with:

- 2 sockets Intel Xeon Platinum 8160 CPU with 24 cores each @ 2.10GHz for a total of 48 cores per node.
- L1d 32K; L1i 32K; L2 cache 1024K; L3 cache 33729K.
- 96 GB of main memory 1.88 GB/core.
- 100 Gbit/s Intel Omni-Path HFI Silicon 100 series PCI-E adapter.
- 10 Gbit Ethernet.

	Testcases:		Short-range:		Mid-range:		Long-range:
	Epoxy	CG-GO	Granular chute	Polymer chain	EAM	Lennard-Jones	Rhodopsin
Parallel efficiency	0,69	0,40	0,38	0,38	0,58	0,58	0,64
|-- Load balance	0,85	0,42	0,85	0,80	0,81	0,95	0,94
|-- Communication efficiency	0,81	0,95	0,45	0,48	0,72	0,62	0,69

Table 2: Efficiency metrics of all testcases and benchmarks using 48 MPI ranks. Colors in both this table and Table 3 represent a gradient scale going from dark green (1, perfect efficiency) to dark red (poor efficiency).

– 200 GB local SSD available as temporal storage during jobs.
– The processors support well-known vectorization instructions such as SSE, AVX up to AVX-512.

The software environment used is as follows:

– LAMMPS 20Nov19
– Intel 17.0.4 20170411 compiler

5.2 Benchmark Characterization

Table 2 shows efficiency metrics for all the testcases and benchmarks studied in this work. These performance metrics correspond to MPI-only executions and using 48 MPI ranks in all cases. Most of them present such a poor parallel efficiency at this core count, that considering using more ranks does not make much sense, and for this reason, we conducted our experiments on a single node. Particularly bad are the cases of short-range interactions benchmarks and the CG-GO testcase. The reasons for poor parallel efficiency are diverse. While the main problem arising in short-range interactions benchmarks is *communication efficiency*, the most limiting factor of the CG-GO benchmark is *Load Balance* with an extremely low value (in contrast with the rest of the benchmarks). Mid-range interactions benchmarks have very similar characteristics: although with slightly different weights on the two components, they both have the same *parallel efficiency* value. Finally, "rhodopsin" is the best performing benchmark. Note that, the "epoxy" testcase presents quite different characteristics when compared with CG-GO as discussed later.

So with all these benchmarks, we cover very different scenarios both in terms of the type of simulation and in terms of performance metrics characteristics.

As a reference, we also include the same metrics for hybrid setups in Table 3. A direct comparison between both tables is impossible since the load balance metric in the case of hybrid setups is biased by non-exhaustive OpenMP parallelization (i.e., implicitly increasing the load imbalance, as previously discussed). This also explains the fact that the best hybrid setup is (24x2) in most cases. Nonetheless, it is worth noting the 50% improvement in load balance in the CG-GO testcase (i.e., the one where the load balance is really the most

	Testcases:		Short-range:		Mid-range:		Long-range:
	Epoxy (24x2)	CG-GO (8x6)	Granular chute (24x2)	Polymer chain (24x2)	EAM (24x2)	Lennard-Jones (24x2)	Rhodopsin (24x2)
Parallel efficiency	0,65	0,69	0,37	0,32	0,55	0,52	0,58
|-- Load balance	0,85	0,82	0,75	0,63	0,83	0,89	0,85
|-- Communication efficiency	0,77	0,84	0,49	0,50	0,66	0,58	0,67

Table 3: Efficiency metrics of all testcases and benchmarks for the best performing hybrid setups (the actual setup, i.e., number of MPI processes and OpenMP threads is indicated in parenthesis in each column).

limiting factor), or the improvement in terms of communication efficiency in the short-range benchmarks. A more comprehensive study in terms of performance is done in the following section.

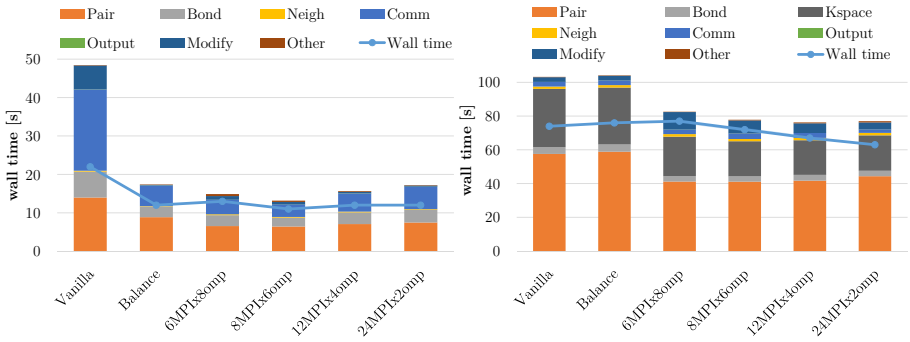
5.3 Execution Time

In this section, we present the wall time execution of all testcases and benchmarks for different setups including the so-called *Vanilla* (i.e. the regular MPI-only execution), *Balance* (i.e. MPI-only execution but including the balancing mechanisms provided by the LAMMPS implementation) and different hybrid configurations. So, no OpenMP features are being used in the Vanilla nor in the Balance experiments. 48 cores are used in all cases and, for the hybrid configurations, only configurations up to the point where using more OpenMP threads translates into worse performance than the MPI-only case are shown. In all benchmarks, the number of timesteps has been increased to have an execution wall time of at least 1 minute for the *Vanilla* case in order to better appreciate execution time differences (i.e. the wall times reported are the ones provided by LAMMPS and they only have a precision of seconds). Results are averaged over 5 different runs in each case, although variability was less than 5% in all cases.

5.3.1 Testcases

Let us start with the CG-GO testcase as it is the one that shows the greatest load imbalance. Figure 7 (left part) presents the execution times for different setups of this benchmark. The bars represent the maximum time spent by an MPI rank on a given code section as reported by LAMMPS in its performance execution report while the blue line that traverses the figure represents the total wall time of the different executions. So the difference between both heights gives an idea of the load imbalance that affects a given configuration. As it can be seen, this is huge for the *Vanilla* case. The *Balance* version reduces this difference a lot (mainly by reducing the "Comm" maximum execution time). Note, however, how a wide range of hybrid configurations (from 24 to 8 MPI processes) do this better, achieving also lower wall execution times.

Figure 8 represent 10 timesteps of the executions using the *Vanilla*, *Balance* and *Hybrid* versions at the same timescale. As it can be seen, the *Vanilla*



(a) CG-GO performance results (b) Epoxy performance results

Fig. 7: Performance results for test cases inputs

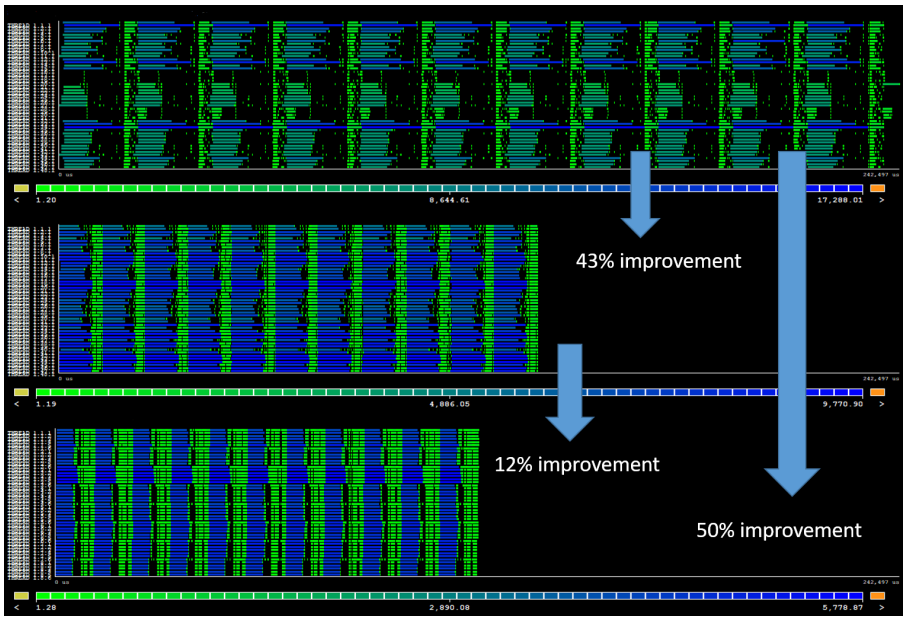


Fig. 8: Useful duration timelines for the CG-GO testcase (upper: *Vanilla*, middle: *Balance*, lower:*Hybrid*). (Duration = 242.50 ms)

version presents a heavy load imbalance: this allows the *Balance* version to achieve a 43% improvement in execution time (timeline in the middle). But, more interestingly, the *Hybrid* version (bottom timeline) is 12% faster than the *Balance* version and it achieves a 50% improvement when compared with the *Vanilla* version.

Figure 9 shows the parallel functions executed by the *Hybrid* version. The time spent in OpenMP parallel regions, in this case, is 80,6% of the total execution time.

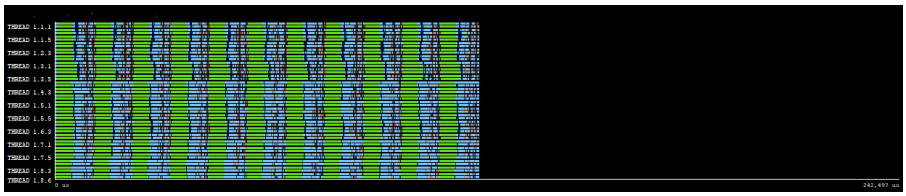
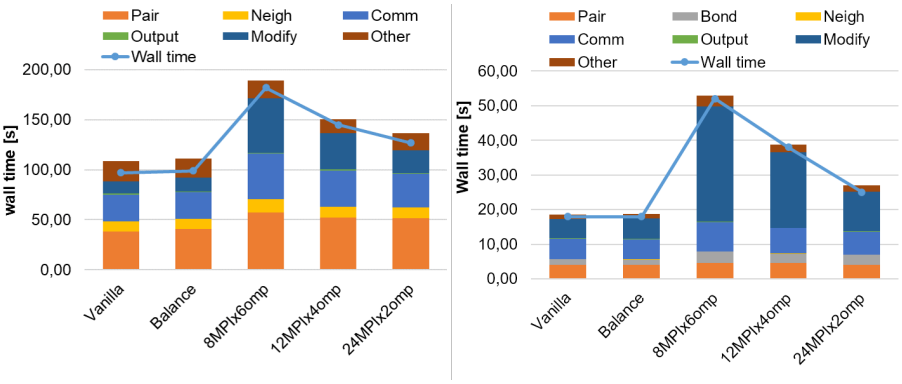


Fig. 9: Parallel functions timeline for the CG-GO testcase (*Hybrid* version). (Duration = 242.50 ms)

A very different scenario is shown in the right plot of Figure 7, which presents the execution time of the Epoxy resin testcase. Note the different characteristics of this testcase when compared with the CG-GO. First, the imbalance is not so relevant for the *Vanilla* scenario. In fact, it can be seen how the use of the balance mechanisms provided by LAMMPS actually makes the execution slower (i.e. it adds overhead without any improvement). Hybrid configurations from 24 to 8 MPI ranks perform better in terms of both load balance and execution time, mainly due to the better performance of the pairing ("Pair", orange in Figure 7) of the hybrid cases, meaning that the OpenMP parallelization is more efficient than the MPI one. This testcase clearly shows how the *Hybrid* version is able to improve the *Vanilla* even when the *Balance* version is not, demonstrating that hybridization provides other benefits than just load balance.

5.3.2 Short-range interactions benchmarks

Figure 10 presents the results for the short-range interactions benchmarks. For both benchmarks the analysis is actually the same: neither the *Balance* mechanism nor the *Hybrid* solution are able to improve the *Vanilla* setup.



(a) Granular chute flow performance results (b) Polymer chain melt performance results

Fig. 10: Short-range interactions benchmarks results

For the *Balance* mechanism, the explanation is clear: the problem of the *Vanilla* setup, if any, is not load imbalance. For the *Hybrid* configuration, we will take the *Polymer chain* benchmark as a representative but a similar analysis could be done for the *Granular chute*. Figure 11 represents 10 timesteps of the *Polymer chain* benchmark. The timelines, of course at the same timescale, show two main reasons that explain why these short-range interactions cases do not benefit from the use of a *Hybrid* implementation:

1. The most computationally intensive part (dark blue) is not OpenMP parallelized so the execution time is increased.
2. Only very small parts of the less computational intensive part (light green) are parallelized with OpenMP, leading also to an increased execution time. This can be perfectly seen in Figure 12 where the OpenMP parallel regions are depicted (meaning light blue no parallel region at all, i.e. sequential execution). Actually, the percentage of time of the whole execution spent in OpenMP parallel regions is only 23%. This suggests that there is a lot of room for improvement by parallelizing other parts of the code used by this benchmark in similar ways as explained in Subsections 4.1 and 4.3. One would expect a similar level of opportunities and difficulties (including data dependencies) to parallelize these other parts like the ones depicted in Subsections 4.1 and 4.3.

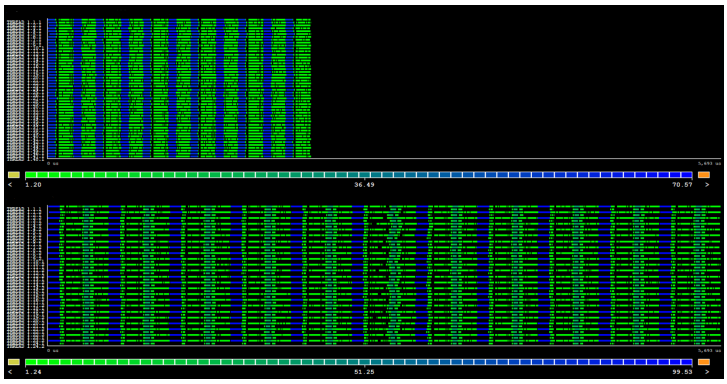


Fig. 11: Useful duration timelines for the Polymer chain melt benchmark (Upper part: *Vanilla* execution, lower part: *Hybrid* version). (Duration = 5.69 ms)

5.3.3 Mid-range interactions benchmarks

Figure 13 (left) presents the execution times of the *EAM* benchmark for different configurations. In this case, all the versions perform quite similar. It is noticeable, however, that the best performing version is *Hybrid* for the 24MPIx2omp case: 70 seconds in contrast with the 76 seconds of the *Vanilla*

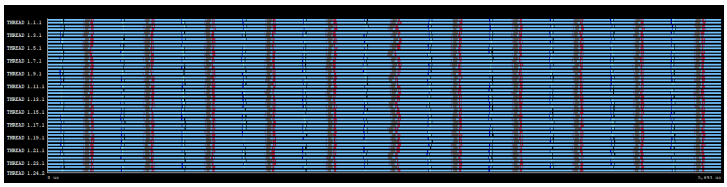
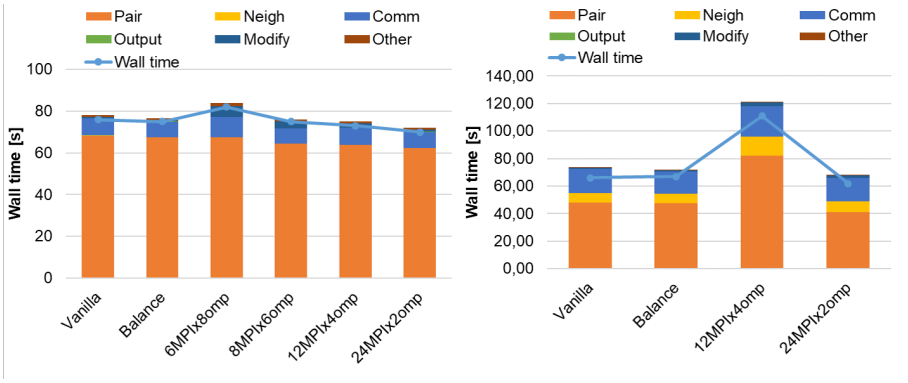


Fig. 12: Parallel functions timeline for the Polymer chain melt benchmark (Hybrid version). (Duration = 5.69 ms)

or the 75 seconds of the *Balance* version. A similar analysis can be done for the *Lennard-Jones* benchmark (Figure 13 (right)): the only noticeable difference is that in this case *Balance* is a bit worse than *Vanilla* (just one second) while the 24MPIx2omp *Hybrid* configuration is still able to improve by 4 seconds the *Vanilla* case.



(a) EAM metallic solid performance results (b) Lennard-Jones liquid performance results

Fig. 13: Mid-range interactions benchmarks results

5.3.4 Long-range interactions benchmarks

Figure 14 presents the execution time of the *Rhodopsin* benchmark for different configurations. The three *Hybrid* configurations on the right are able to outperform both *Vanilla* and *Balance* versions. The pairing process (orange bar) is much faster in the *Hybrid* configurations.

Figure 15 represents two timelines of 10 timesteps of the *Rhodopsin* benchmark execution at the same timescale. As it can be visually noted, the execution of the *Hybrid* case is significantly faster. This gain in performance comes mainly from the pairing phase (blue sections in the timelines) done in the `compute` function in the `PairLJCharmmCoulLongOMP` module of LAMMPS which is faster for the *Hybrid* case.

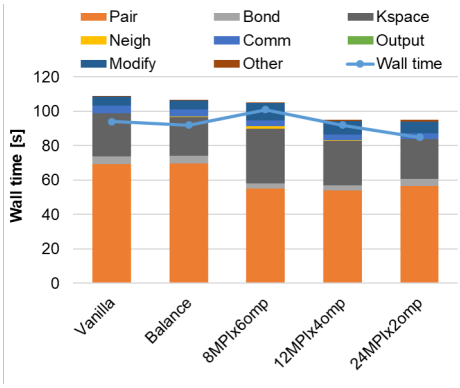


Fig. 14: Rhodopsin protein benchmark results.

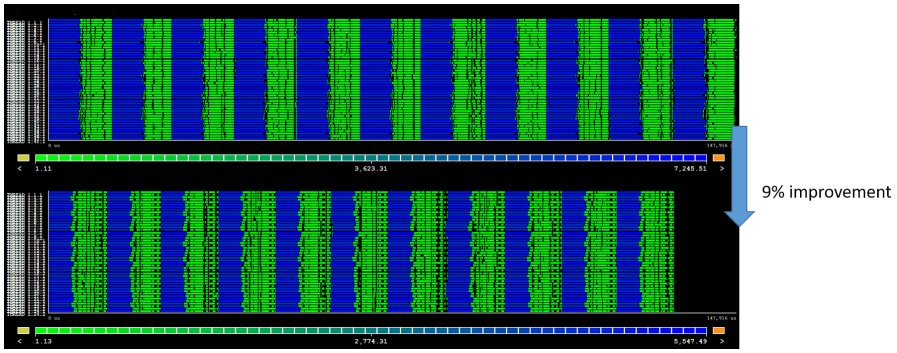


Fig. 15: Useful duration timelines for the Rhodopsin protein benchmark (top: *Vanilla*, bottom: *Hybrid*). (Duration = 147.92 ms)

As it can be seen on the red parts of Figure 16, this function is fully OpenMP parallelized. Note, however, that there are still other parts of the code not parallelized with OpenMP (light blue in Figure 16 and black in bottom timeline of Figure 15), making the execution of the green areas in Figure 15 being slightly faster for the *Vanilla* case. But even so, the improvement achieved by the OpenMP implementation of `compute` subroutine is able to compensate by far this loss in performance.

All in all, the ratio of time in parallel regions with respect to the whole execution time is pretty high: 76,6%. We can now compare short-range with long-range interactions benchmarks and explain why *Hybrid* implementation is able to improve the performance of the latter but not of the formers. Comparing Figures 12 and 16 one can see how the percentage of time outside any parallel OpenMP region (light blue areas) for both benchmarks is drastically different. Actually, the percentage of time inside OpenMP parallel regions of the *Polymer chain* benchmark is only 23,7%. This is exacerbated if we focus on the pairing phases: while in the case of "rhodopsin" benchmark this corre-

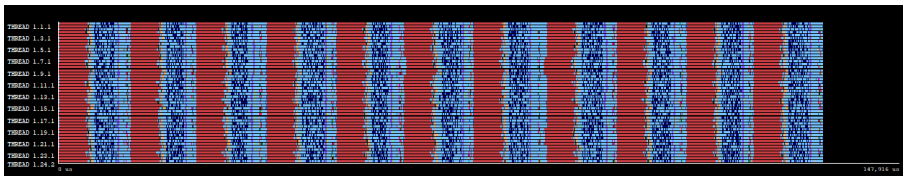


Fig. 16: Parallel functions timeline for the Rhodopsin protein benchmark (*Hybrid* version). (Duration = 147.92 ms)

sponds to the red parts in Figure 16, so it represents a high percentage of the whole execution, this is not the case for the *Polymer chain* benchmark where the pairing (light brown in Figure 12) corresponds to a very residual part of the whole execution (in this benchmark, subroutine `compute` of `PairLJCutOMP` module).

So, in the end, the reason that ultimately explains the different behaviour of the two kinds of benchmarks is that the weight of the pairing process (which is the most important part parallelized with OpenMP in all benchmarks) in the whole execution is very high for the long-range interactions benchmarks (45,6% in the *Rhodopsin* benchmark) while it is insignificant for the short-range interactions benchmarks (8,4% for the *Polymer chain* benchmark).

6 Conclusions

This paper has shown the potential of the proposed hybrid MPI+OpenMP approach to effectively alleviate performance problems such as load imbalance in LAMMPS simulations. Although we have used LAMMPS to demonstrate the benefits of hybrid parallelization, the same approach could be applied to any MD simulation code such as NAMD or GROMACS.

LAMMPS provides a ready-to-use balancing mechanism to partially solve load balancing problems in molecular dynamics simulations with non-uniform atom densities. The LAMMPS balancing mechanism shows high efficiency compared to MPI-only simulation in case of high load imbalance (CG-GO testcase). We have introduced the use of an MPI+OpenMP hybrid implementation of LAMMPS as a third option. Furthermore, we have complemented the current partial OpenMP implementation of LAMMPS with additions and modifications, driven by *epoxy* testcase.

Our proposed modified version of LAMMPS has been extensively compared against the baseline MPI case and against the use of the LAMMPS balance mechanism. Five benchmarks present in the LAMMPS distribution with varying ranges of interaction (short, mid, and long-range) together with two testcases with very different characteristics were used for the comparison.

For the short-range interactions benchmarks, the regular MPI-only version was the best performing. As long as they do not present a load imbalance problem, the balancing mechanism does not provide any benefit in this case.

The problem with the hybrid setup in this case is that only a small fraction of the simulations (∼20%) runs in OpenMP parallel regions. This suggests that more additions similar to the ones proposed in Section 3.3 could be done to the OpenMP LAMMPS implementation.

For the mid-range interactions benchmarks, the hybrid option was the best in all cases. The balance mechanism only improves a bit the EAM simulation while it is the worst option for the Lennard-Jones benchmark. These results indicate that the hybrid implementation is able to improve performance metrics other than load balance, such as communication efficiency.

In the "rhodopsin" benchmark (long-range interactions) the execution time is mainly dominated by the LAMMPS pairing process. The obtained results show that the OpenMP parallelization of the pairing is much faster than the MPI one, making the hybrid approach the best option also for this benchmark.

Regarding the highly-imbalanced testcase (CG-GO), the balancing mechanism shows its potential by achieving a 43% improvement with respect to the regular MPI simulation. Interestingly, the hybrid version is able to improve even further, up to 50%.

In the case of the "epoxy" resin testcase (the one that motivated the implementations explained in Section 3.3), the use of the balance mechanism only adds overhead (the execution is slower than for the regular MPI version). The hybrid implementation, on the contrary, is the best option again showing that it is able to improve simulations where load balance is not the main problem.

So, the overall conclusion is that LAMMPS hybrid setups are able to handle scenarios with very high load imbalance at least as well (if not better) as the LAMMPS balance mechanism while also providing benefits in other scenarios where load balance is not the main performance bottleneck.

Our suggestion to LAMMPS developers, and MD in general, is to put effort into hybridizing the code with an MPI+OpenMP strategy instead of implementing *ad hoc* balancing methods. This is because the hybrid code not only can address more dynamic load imbalance but also improve parallel efficiency by reducing communication.

Acknowledgements This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology (TIN2015-65316-P), by the Generalitat de Catalunya (2017-SGR-1414), and by the European POP CoE (GA n. 824080). This work is also funded as part of the European Union Horizon 2020 research and innovation programme under grant agreement nos. 800925 (VECMA project; www.vecma.eu) and 823712 (CompBioMed2 Centre of Excellence; www.compbiomed.eu), as well as the UK EPSRC for the UK High-End Computing Consortium (grant no. EP/R029598/1).

References

1. Adaptive MPI - Using Existing MPI Codes with AMPI. https://charm.readthedocs.io/en/latest/ampi/03-using.html. [Online; accessed 04-November-2021]
2. EAM metallic solid benchmark. https://www.lammps.org/bench.html#eam. [Online; accessed 08-November-2021]

3. Granular chute flow benchmark. https://www.lammps.org/bench.html#chute. [Online; accessed 08-November-2021]
4. GROMACS. https://www.gromacs.org/. [Online; accessed 04-November-2021]
5. LAMMPS balance command. https://docs.lammps.org/balance.html. [Online; accessed 03-November-2021]
6. LAMMPS documentation, OpenMP section. https://docs.lammps.org/Speed_omp.html. [Online; accessed 04-October-2021]
7. LAMMPS fix balance command. https://docs.lammps.org/fix_balance.html. [Online; accessed 03-November-2021]
8. LAMMPS release 20 Nov 2019. https://github.com/lammps/lammps/releases/tag/patch_20Nov2019. [Online; accessed 08-November-2021]
9. LAMMPS website. https://www.lammps.org/. [Online; accessed 08-November-2021]
10. Lennard-Jones liquid benchmark. https://www.lammps.org/bench.html#lj. [Online; accessed 08-November-2021]
11. Marenostrum4. https://www.bsc.es/marenostrum/marenostrum. [Online; accessed 03-November-2021]
12. NAMD Scalable Molecular Dynamics. https://www.ks.uiuc.edu/Research/namd/. [Online; accessed 04-November-2021]
13. Official LAMMPS website, benchmark section: Billion-atom LJ benchmarks. https://www.lammps.org/bench.html#billionl. [Online; accessed 29-September-2021]
14. OpenMP. https://www.openmp.org/. [Online; accessed 03-November-2021]
15. Polymer chain melt benchmark. https://www.lammps.org/bench.html#chain. [Online; accessed 08-November-2021]
16. POP (Performance Optimisation and Productivity, A Centre of Excellence in HPC. Patterns, Loop iterations manually distributed. https://co-design.pop-coe.eu/patterns/loop-manual-distribution.html. [Online; accessed 04-October-2021]
17. Rhodopsin protein benchmark. https://www.lammps.org/bench.html#rhodo. [Online; accessed 08-November-2021]
18. Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., et al.: Parallel programming with migratable objects: Charm++ in practice. In: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 647–658. IEEE (2014)
19. Banchelli, F., Peiro, K., Querol, A., Ramirez-Gargallo, G., Ramirez-Miranda, G., Vinyals, J., Vizcaino, P., Garcia-Gasulla, M., Mantovani, F.: Performance study of hpc applications on an arm-based cluster using a generic efficiency model. In: 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 167–174. IEEE (2020)
20. Barcelona Supercomputing Center: Extrae. https://tools.bsc.es/extrae. [Online; accessed 03-November-2021]
21. Barcelona Supercomputing Center: Paraver. https://tools.bsc.es/paraver. [Online; accessed 03-November-2021]
22. Berger, R., Kloss, C., Kohlmeyer, A., Pirker, S.: Hybrid parallelization of the LIGGGHTS open-source DEM code. Powder Technology **278**, 234–247 (2015). DOI https://doi.org/10.1016/j.powtec.2015.03.019. URL https://www.sciencedirect.com/science/article/pii/S0032591015002144
23. Deng, Y., Peierls, R.F., Rivera, C.: An Adaptive Load Balancing Method for Parallel Molecular Dynamics Simulations. Journal of Computational Physics **161**(1), 250 – 263 (2000). DOI https://doi.org/10.1006/jcph.2000.6501. URL http://www.sciencedirect.com/science/article/pii/S002199910096501X
24. Devine, K.D., Boman, E.G., Heaphy, R.T., Hendrickson, B.A., Teresco, J.D., Faik, J., Flaherty, J.E., Gervasio, L.G.: New challenges in dynamic load balancing. Applied Numerical Mathematics **52**(2-3), 133–152 (2005)
25. Etinski, M., Corbalan, J., Labarta, J., Valero, M., Veidenbaum, A.: Power-aware load balancing of large scale mpi applications. In: 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–8. IEEE (2009)
26. Fincham, D.: Parallel computers and molecular simulation. Molecular Simulation **1**(1-2), 1–45 (1987). DOI 10.1080/08927028708080929. URL https://doi.org/10.1080/08927028708080929

27. Garcia, M., Corbalan, J., Labarta, J.: LeWI: A Runtime Balancing Algorithm for Nested Parallelism. In: Proceedings of the International Conference on Parallel Processing (ICPP09) (2009)
28. Garcia-Gasulla, M., Mantovani, F., Josep-Fabrego, M., Eguzkitza, B., Houzeaux, G.: Runtime mechanisms to survive new hpc architectures: a use case in human respiratory simulations. The International Journal of High Performance Computing Applications **34**(1), 42–56 (2020)
29. Harlacher, D.F., Klimach, H., Roller, S., Siebert, C., Wolf, F.: Dynamic load balancing for unstructured meshes on space-filling curves. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, pp. 1661–1669. IEEE (2012)
30. Huang, C., Lawlor, O., Kale, L.V.: Adaptive mpi. In: International workshop on languages and compilers for parallel computing, pp. 306–322. Springer (2003)
31. Jung, J., Mori, T., Sugita, Y.: Midpoint cell method for hybrid (mpi+openmp) parallelization of molecular dynamics simulations. Journal of Computational Chemistry **35**(14), 1064–1072 (2014). DOI https://doi.org/10.1002/jcc.23591. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.23591
32. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on scientific Computing **20**(1), 359–392 (1998)
33. Kunaseth, M., Richards, D., Glosli, J., Kalia, R., Nakano, A., Vashishta, P.: Analysis of scalable data-privatization threading algorithms for hybrid mpi/openmp parallelization of molecular dynamics. The Journal of Supercomputing **66**, 406–430 (2013). DOI 10.1007/s11227-013-0915-x
34. Pal, A., Agarwala, A., Raha, S., Bhattacharya, B.: Performance metrics in a hybrid mpi–openmp based molecular dynamics simulation with short-range interactions. Journal of Parallel and Distributed Computing **74**(3), 2203–2214 (2014). DOI https://doi.org/10.1016/j.jpdc.2013.12.008. URL https://www.sciencedirect.com/science/article/pii/S0743731513002505
35. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: transputer and occam developments, vol. 44, pp. 17–31 (1995)
36. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. Journal of computational physics **117**(1), 1–19 (1995)
37. Plimpton, S., Hendrickson, B.: A new parallel method for molecular dynamics simulation of macromolecular systems. Journal of Computational Chemistry **17**(3), 326–337 (1996). DOI https://doi.org/10.1002/(SICI)1096-987X(199602)17:3¡326::AID-JCC7¿3.0.CO;2-X
38. Plimpton, S., Pollock, R., Stevens, M.: Particle-mesh ewald and rrespa for parallel molecular dynamics simulations. Proc. 8th SIAM Conf. on Parallel Processing for Scientific Computing (2000)
39. Rabenseifner, R., Hager, G., Jost, G.: Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In: 2009 17th Euromicro international conference on parallel, distributed and network-based processing, pp. 427–436. IEEE (2009)
40. Rabenseifner, R., Hager, G., Jost, G.: Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 427–436 (2009). DOI 10.1109/PDP.2009.43
41. Rabenseifner, R., Wellein, G.: Communication and optimization aspects of parallel programming models on hybrid architectures. The International Journal of High Performance Computing Applications **17**(1), 49–62 (2003)
42. Schloegel, K., Karypis, G., Kumar, V.: A unified algorithm for load-balancing adaptive scientific simulations. In: SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, pp. 59–59. IEEE (2000)
43. Servat, H., et al.: Framework for a productive performance optimization. Parallel Computing **39**(8), 336–353 (2013)
44. Smith, W.: Molecular dynamics on hypercube parallel computers. Computer Physics Communications **62**(2), 229 – 248 (1991). DOI https://doi.org/10.1016/0010-4655(91)90097-5. URL http://www.sciencedirect.com/science/article/pii/0010465591900975

45. Suter, J.L., Sinclair, R.C., Coveney, P.V.: Principles governing control of aggregation and dispersion of graphene and graphene oxide in polymer melts. Advanced Materials **32**(36), 2003213 (2020). DOI https://doi.org/10.1002/adma.202003213. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/adma.202003213
46. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting performance data with papi-c. In: M.S. Müller, M.M. Resch, A. Schulz, W.E. Nagel (eds.) Tools for High Performance Computing 2009, pp. 157–173. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
47. Thompson, A.P., Aktulga, H.M., Berger, R., Bolintineanu, D.S., Michael Brown, W., Crozier, P.S., in 't Veld, P.J., Kohlmeyer, A., Moore, S.G., Nguyen, T.D., Shan, R., Stevens, M., Tranchida, J., Trott, C., Plimpton, S.J.: Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. Computer Physics Communications p. 108171 (2021). DOI https://doi.org/10.1016/j.cpc.2021.108171. URL https://www.sciencedirect.com/science/article/pii/S0010465521002836
48. Vassaux, M., Sinclair, R.C., Richardson, R.A., Suter, J.L., Coveney, P.V.: The role of graphene in enhancing the material properties of thermosetting polymers. Advanced Theory and Simulations **2**(5), 1800168 (2019). DOI https://doi.org/10.1002/adts.201800168. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/adts.201800168
49. Wagner, M., Mohr, S., Giménez, J., Labarta, J.: A structured approach to performance analysis. In: International Workshop on Parallel Tools for High Performance Computing, pp. 1–15. Springer (2017)
50. Walshaw, C., Cross, M.: Mesh partitioning: a multilevel balancing and refinement algorithm. SIAM Journal on Scientific Computing **22**(1), 63–80 (2000)