# P1.7 Project Report on $N$-Body LJMD Refactoring, Optimization and Parallelization

Zakaria Dahbi, Francesco Andreucci, Behzad Salmassian

This concise report presents a statistical analysis of various benchmarking scenarios applied to the serial ljmd.c code. We show by results the impact of using different optimization flags and the introduction of OpenMP and MPI parallelization. We clearly report the time-to-solution regarding the problem sizes (number of atoms in the system), typically for 108, 2916, and 78732 atoms, and we collect the performance statistics of each case. Furthermore, we also used the GTest library to test the slices of the code after each implementation of OpenMP and MPI.

## I. INTRODUCTION

We firstly split the original ljmd.c source file into several source .c files containing the various functions (with respective .h headers). In particular, we have:

- types.h : this file contains the definition of the sys structure, that contains the physical quantities (positions, velocities, forces) and parameters of the simulation
- constants.h : this file contains the values of the physical constants used in the code, namely the Boltzmann constant in kcal/mol/K and a factor to express energies in kcal/mol
- input.c : this file contains functions related to the reading of the simulation parameters from provided files. "read_from_file" reads the input file argon_*.inp, where * stands for the number of atoms. The input file contains the main parameters of the simulations, that are stored as elements of the sys structure. The "read_restfile" function reads the restart file argon_*.res that contains the initial values of position and velocities of the atoms.
- memory.c : this file contains functions related to the allocation and deallocation of memory for the structure ("memalloc" and "cleanup").
- output.c : this file contains the function "output" that prints the number of atoms, temperature, kinetic, potential and total energy of the system on the terminal. Also, it fills the .erg and .traj files.
- verlet.c : this file contains the two functions that implement verlet algorithm for evolution of velocities and positions ("velverlet1" and "velverlet2")
- force_comp.c : this file contains the function "force" that computes the forces between the atoms based on the LJ potential
- utitlities.c : this files contains the remaining functions necessary for running the code. "wallclock" gets the current time in seconds, "pbc" implements periodic boundary conditions, "ekin" computes the kinetic energy of the system and "azzero" sets to zero the components of an array that is passed as an argument.

We also created unit tests using google test to test for: 1) Computing forces for a two-atom system in three different scenarios (test_force.cpp): the interparticle distance is larger than the potential cut-off (and therefore the forces are zero); the interparticle distance is smaller than the cut-off (and therefore the forces are non-zero); the interparticle distance is larger than the cut-off on two directions (but smaller than the box's size) and larger than the box's size in the third one. When taking into account pbc, the component of the force along this latter direction is non-zero. 2) Computing the kinetic energy of a system of two atoms in two cases (test_ekin.cpp): one for zero kinetic energy and one for non-zero kinetic energy. 3) Test for the correct reading of a test input and restart file (test_input.cpp). 4) Test for the velverlet functions for different initial velocities and positions. (test_verlet.cpp).

We then divided the tasks in the following way:

1. Serial optimization: Behzad Salmassian
2. MPI implementation: Francesco Andreucci
3. OpenMP implementation: Zakaria Dahbi
4. Hybrid MPI-OpenMP: Francesco and Zakaria

## II. HARDWARE SPECIFICATIONS

The benchmarks shown in this report are collected using CINECA Leonardo supercomputer,

FIG. 1. Leonardo supercomputer.

which has 99% of its computational power based on GPUs. In the Leonardo supercomputer, there are 3456 compute nodes with where each node has the following configurations, shown in the following figure:

```
*******************************************************************************
* Atos Bull Sequana XH21355 "Da Vinci" Blade                                  *
* Red Hat Enterprise Linux 8.6 (Ootpa)                                        *
*                                                                             *
* 3456 compute nodes with:                                                    *
*       - 32 cores Ice Lake at 2.60 GHz                                       *
*       - 4 x NVIDIA Ampere A100 GPUs, 64GB                                   *
*       - 512 GB RAM                                                          *
*                                                                             *
* Internal Network: Nvidia Mellanox HDR DragonFly++                           *
* SLURM 22.05.9                                                               *
*                                                                             *
* For a guide on Leonardo:                                                    *
* https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.2%3A+LEONARDO+UserGuide *
* For support: superc@cineca.it                                               *
*******************************************************************************
```

FIG. 2. Configuration within each node.

## III. SERIAL CODE OPTIMIZATION

We optimized the original serial code incrementally, we were able to extract the following timing results for different scenarios as reported in the following tables:

| # natoms | Time (s) |
|---|---|
| 108 | 15.355 |
| 2916 | 283.470 |
| 78732 | 2174.571 |

TABLE I. Benchmarks of unoptimized serial code.

| # natoms | Time (s) |
|---|---|
| 108 | 12.711 |
| 2916 | 173.384 |
| 78732 | 790.545 |

TABLE II. Benchmarks of (`-O3`) optimized serial code.

| # natoms | Time (s) |
|---|---|
| 108 | 2.990 |
| 2916 | 106.628 |
| 78732 | 790.539 |

TABLE III. Benchmarks of optimized serial code with flags `-ffast-math -fexpensive-optimizations -msse3`.

| # natoms | Time (s) |
|---|---|
| 108 | 1.290 |
| 2916 | 50.108 |
| 78732 | 427.181 |

TABLE IV. Benchmarks of optimized serial code with Newton's Third Law.

| # natoms | Time (s) |
|---|---|
| 108 | 1.141199 |
| 2916 | 48.315378 |
| 78732 | 430.492133 |

TABLE V. Benchmarks of optimized serial code with all above and math outside loop.

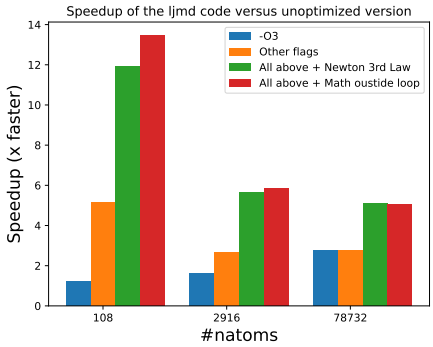



FIG. 3. Speedup of different optimization levels versus unoptimized one.

By looking at the results in the different Tables. II and Fig. 3. We have the following statistics to each case with the unoptimized serial version of the code. However we also notice that the effect of the various optimization

steps on the code vary a lot based on the size of the input. In particular, for the larger datasets the -O3 flag seems to be the one bringing more benefits, while the other flags do not bring much improvement (for reference, on Leonardo the C and CXX compiler we used are gcc and g++ version 8.5.0). Regarding the code optimizations, the one that brings the most benefit is the use of Newton third law, especially for large datasets. A possible explanation for this is that the L3 cache in Leonardo is roughly 48MB. This means that, for the first dataset, the arrays for the forces, positions and velocities all comfortably fit into the cache. This is not true for the other datasets, so maybe the benefit of taking the math functions outside the loops is washed away by the spatial non-locality of the code.

#natoms: 108

| Case | $-O3$ | other flags | 3rd Law | Math outside |
|---|---|---|---|---|
| × faster | 1.21 | 5.13 | 11.90 | 13.45 |

TABLE VI. Gained speedup for 108

#natoms: 2916

| Case | $-O3$ | other flags | 3rd Law | Math outside |
|---|---|---|---|---|
| × faster | 1.63 | 2.66 | 5.66 | 5.87 |

TABLE VII. Gained speedup for 2916

#natoms: 78732

| Case | $-O3$ | other flags | 3rd Law | Math outside |
|---|---|---|---|---|
| × faster | 2.75 | 2.75 | 5.09 | 5.05 |

TABLE VIII. Gained speedup for 78732

From the Tables VI, VII, and VIII, we observe that the compiler flag were able to generate a more optimized code that runs with a reduced run-time for the three considered sizes.

### A. Profiling the serial code

We profiled the serial code using gprof at the various stages of the optimization for 108 atoms: the results are reported in 4. As we can see from panel *a*) the most expensive function in the code is the "force" functions, that computes the forces between atoms. The other panels refer to the case where optimization flags are turned on: for some reason on Leonardo this conflicts with gprof profiling, but nonetheless we get some interesting informations. First of all, the O3 flag, while substantially reducing the overall run-time of the code, does not drastically change the pecentages of time spent in "force" and "pbc" functions. On the other hand, when we add the other flags we see that the time spent in "pbc" function is drastically reduced, it could be that the msse3 flag introduced a vectorization of the while loop inside the pbc function.
When we introduce Newton third law, the number of calls to "pbc" is halved, as expected, and also the cumulative times spent in force and pbc are halved. Finally, taking out the expensive math functions outisde of the loops, the cumulative time for the "force" function reduces while the time spent in the "pbc" function remains practically unchanged. We also note that the percentages of time spent in "force" and "pbc" went back close to the original values, although the overall run-time is drastically reduced.

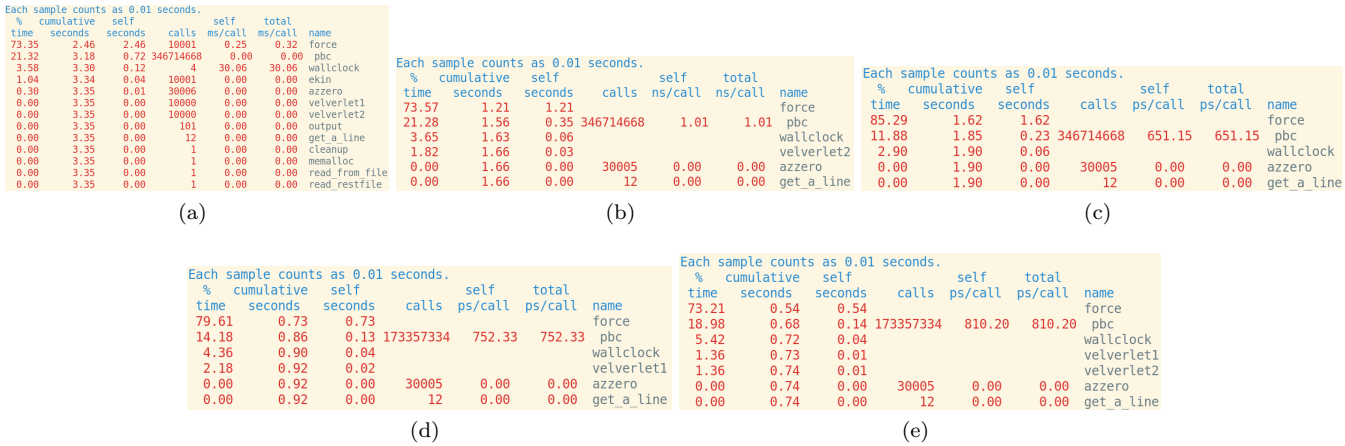



FIG. 4. Gprof results for the various incremental steps in the optimization of the serial code: from *a*) to *e*) we have: unoptimized serial, compiled with "-O3", compiled with "-ffast-math -fexpensive-optimizations -msse3", implemented Newton 3rd law to halve the processed number of pairs and took expensive math functions outside for loops

## IV. OPENMP PARALLELIZATION

In this scenario, we implemented OpenMP parallelization on the optimized splitted (with all compiler optimization flags) serial source code presented in Sec. III. The OpenMP parallelization is applied using the traditional way by re-indexing the loops to consider the accessible number of threads and the thread ID. Then, we also used OpenMP compiler pragmas to provide hints to the compiler about which sections of the code can be parallelized to generate parallel machine code.

To do so, we carefully followed the hint from the LJMD lecture, and we applied OpenMP only on the *src/-force_comp.c* as it is the most expensive function affecting the performance of the code, as reported by gprof. To parallelize, we used compiler pragmas to ensure that the code will still run as serial optimized when OpenMP is not called. We re-organized the optimized serial code by moving some code blocks that do not depend on the loop on to the top of the force_omp.c file, such as the math constants:

```
    int tid, start, end;
    int i, j;
    double rx, ry, rz, rsq;
    double r6, rinv, ffac;

    double c12 = 4.0 * sys->epsilon * pow(sys->
    sigma, 12.0);
    double c6 = 4.0 * sys->epsilon * pow(sys->
    sigma, 6.0);
    double rcsq = sys->rcut * sys->rcut;

    double epot = 0.0;
```

Listing 1. Math constants and variables outside parallel region.

Then, we defined a parallel region with reduction of epot to compute the global potential energy ($sy \rightarrow epot$) by summing up the private values from each thread. We pass all the variables outside the parallel region using firstprivate clause so that each thread has its copies:

```
#ifdef _OPENMP
    #pragma omp parallel reduction(+:epot)
    firstprivate(c12, c6, rcsq, i, j, tid, start
    , end, rx, ry, rz, rsq, r6, rinv, ffac)
#endif
```

Listing 2. Parallel region with reduction on epot.

As OpenMP threads are asynchronous, we had to use a barrier pragma to define a synchronization point to ensure that all threads reach this point before continuing, which also prevents the race condition to happen.

```
#ifdef _OPENMP
    #pragma omp barrier
#endif
```

Listing 3. Threads synchronization.

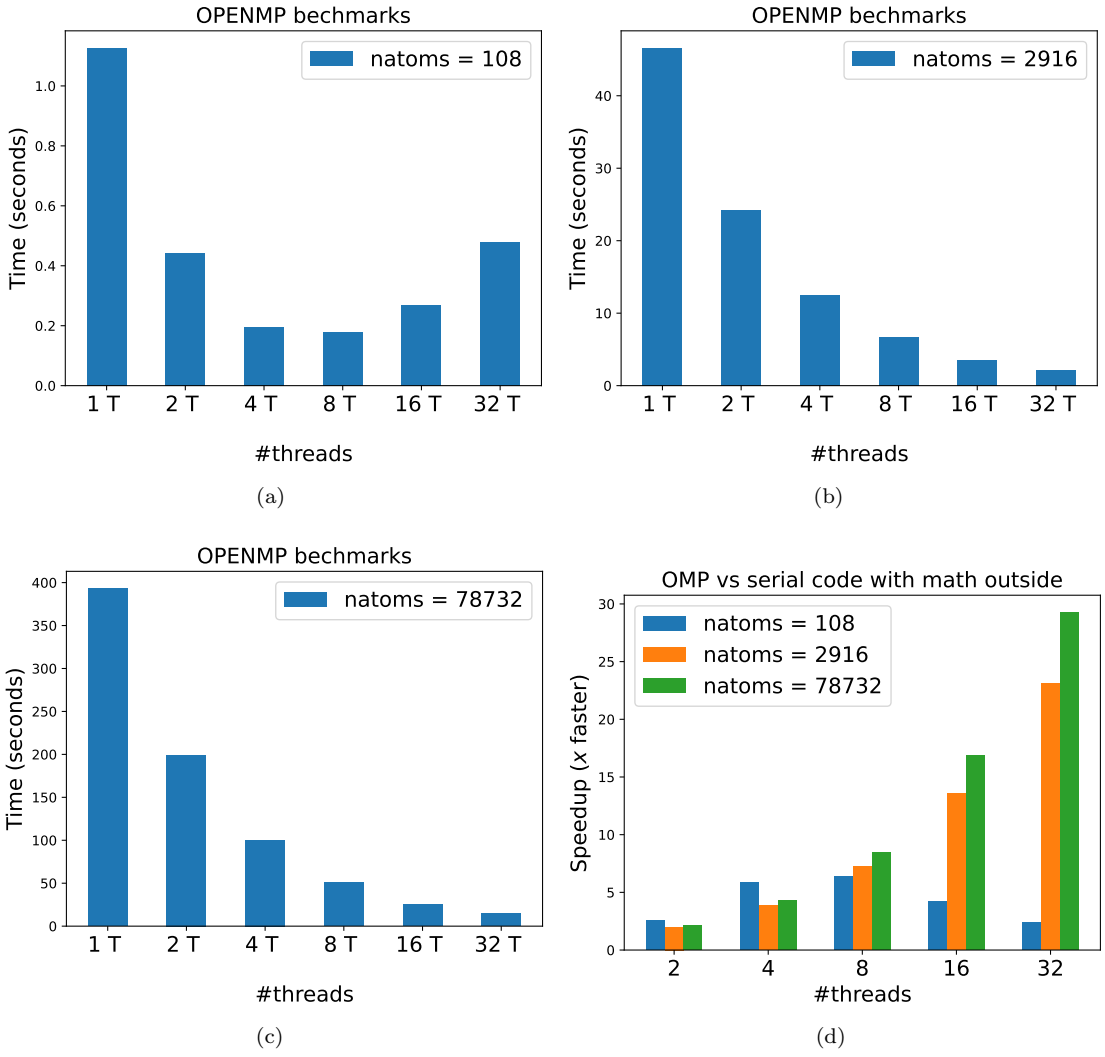


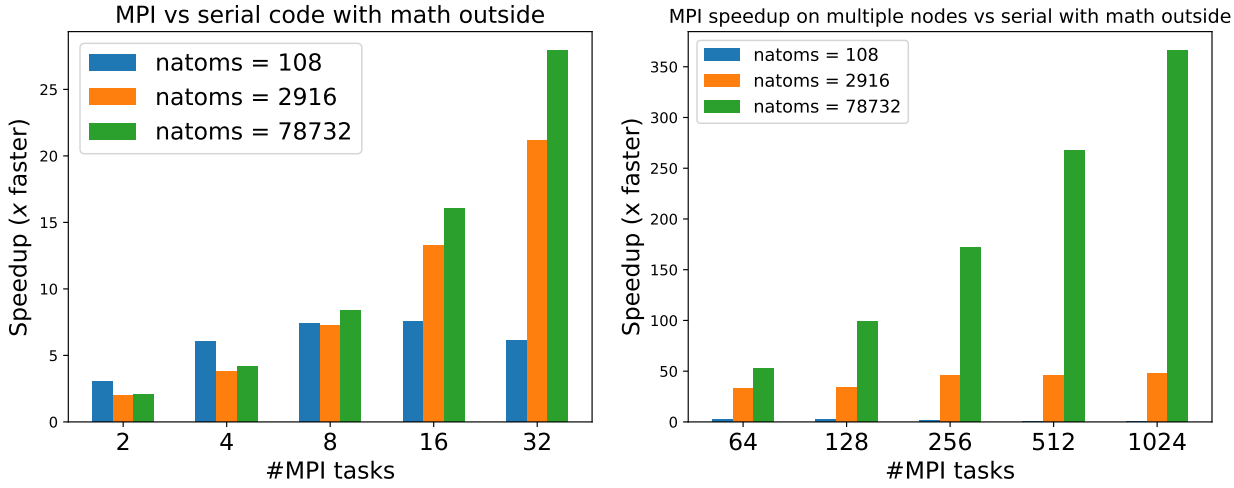

FIG. 5. OpenMP recorded time and speedup for the three considered sizes of atoms, and for an increasing number of threads. 5(a): natoms=108. 5(b): natoms=2916. 5(c): natoms=78732. 5(d): speedup gain.

By analyzing 5(b) and 5(c) for the sizes of atoms 2916 and 78732. The observations indicate that the time-to-solution scales up to 32 threads very well, to half. This suggests that doubling the number of OpenMP threads would lead to a halving of the overall execution time. This pattern demonstrates effective parallelization, as the computational load is evenly distributed among the additional threads, resulting in enhanced computational efficiency. However, for the small size (108), we observe that the time scales very well by half up to 4 threads and then the saturation effect starts to appear, meaning that going beyond 4 threads is very bad as it causes an overhead. For the other sizes (2916 and, 78732), it is expected that the time will stop to scale by increasing the threads to some thresholds. This is normal, as optimal scaling often requires ensuring a balanced distribution of computational work among threads.

## V. MPI PARALLELIZATION

We use a replicated data approach. If LJMD_MPI is defined, the members of the structure **sys**, **sys.mpirank** and **sys.nsize** are initialized to the rank of the MPI task and the comm size in the main, respectively. In the main, only process 0 does the I/O, and it broadcasts the parameters it reads from the files to the other processes. In the "force" function, the positions of the atoms are broadcast to all the processes. Every process then computes the forces for a subset of the particles (this is done by iterating over atoms jumping by the number of processes starting to count from the rank of the process) and stores them in the auxiliary "c" arrays (which are defined as members of the **sys** structure if MPI support is ON). Similarly, the contribution of that subset of atoms to the potential energy is also computed. Finally, both forces

and potential energy are reduced in the process 0: the latter is printed at each step, while the formers are used to update the positions of the atoms using the **velverlet** functions. The updated positions are then broadcasted to all the MPI tasks at the next call of force at the following step of the simulation, and so on.
The results of the benchmark for MPI are reported in figure 6 where we report the speed-up for the three datasets for several numbers of MPI tasks. In panel, *a*) we report the results for numbers of MPI tasks that fit a single node, while in panel *b*) we report the results for higher numbers of MPI tasks which require the allocation of multiple nodes. For the smallest dataset, the optimal run point is with 8 tasks, since for larger number of tasks the speed-up saturates and the decrease due to the communication overhead. For 2916 atoms, it seems the optimal point is, strictly speaking, with 256 processes (so 8 full nodes). However, the speed-up for 64 tasks is, 33 while the one for 256 is 45, so the efficiency for this latter case is actually pretty bad. For the largest dataset of 78732 atoms, we see and increase in the speed-up up to 1024 processes, but once again the parallel efficiency is actually decreasing.




(a) (b)

FIG. 6. Speed up for pure MPI: on the left we have the speed-up for tasks going from 2 to 32 (so they fit into a node), while on the right we have tasks going from 64 to 1024, corresponding to 2 to 32 nodes. The speed-up for 108 atoms is reported only up to 256 tasks.

## VI. HYBRID PARALLELIZATION

Since both MPI and OpenMP use a replicated data approach in basically the same way, we can combine the two. Now we have to be careful to replicate the allocation of the forces among threads as well for the auxiliary forces $c_x$, $c_y$, and $c_z$. Futhermore, to properly process the pairs of atoms now we need to jump by the total number of workers, which is:

$$(sys \to tmax) \times (sys \to nsize),$$

and the counting must start from:

$$t_{id} + (sys \to tmax) \times (sys \to mpirank),$$

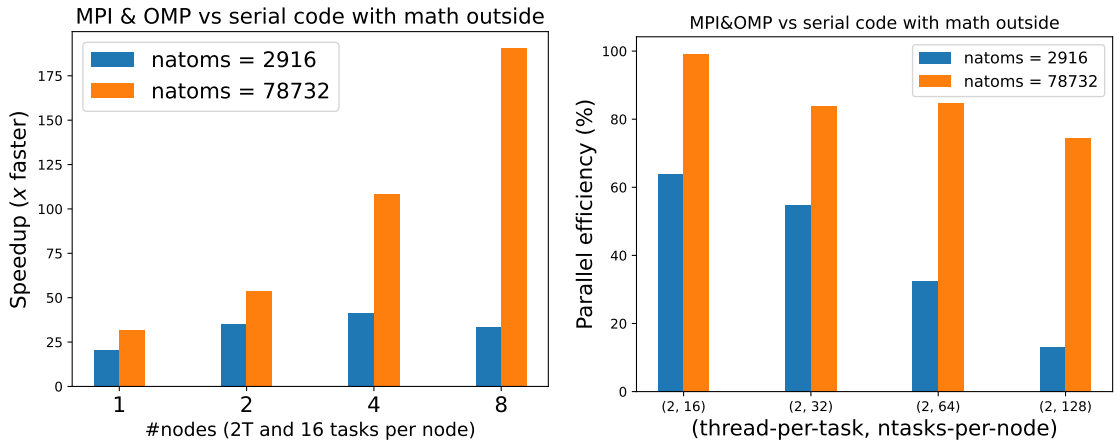
where $(sys \to tmax)$ is the total number of threads, $(sys \to nsize)$ is the total number of MPI processes, $t_{id}$ is the thread ID, and $(sys \to mpirank)$ is the MPI process ID. So that every worker only processes a unique subset of the atoms. By inserting "**#ifdef**" and defining the correct default values for the variables storing the rank, thread ID, number of MPI tasks and number of threads, we get a code which can be compiled with MPI support or OpenMP support or both. In this paragraph, we are interested in what happens if both are turned on. The results of our benchmarks are reported in figuremade seehere are several comments to be madesee,irst of all, as we can see in a single node the runtime of the code is roughly constant: this makes sense because, as we described, MPI tasks and OpenMP threads do the same thing in the code (e.g., process replicated data). Therefore, we can expect that the relevant variable that influences the time-to-solution is the total number of "workers". Furthermore, in a single node (that in Leonardo contains a single socket) both threads and MPI tasks are using the same communication channels. To further corroborate this point, we report the results for running the code without allocating the whole node: as we can see

from the following table, only the total number of workers is relevant.

| #Threads | #Tasks-per-node | Time |
|---|---|---|
| 1 | 1 | 44.910889 |
| 2 | 1 | 22.982478 |
| 1 | 2 | 22.944728 |
| 4 | 1 | 11.949066 |
| 1 | 4 | 11.933567 |
| 2 | 4 | 6.242787 |
| 4 | 2 | 6.285452 |

TABLE IX. OPENMP threads versus MPI tasks.

We then benchmarked the code allocating more than one node: the results are reported in figure 7. Interestingly, also in this case the total number of workers seems to be what influences the time-to-solution, indicating that the internode communication channels in the Leonardo machine are very efficient. (Indeed, there are small differences in the times because the code is not exactly the same for MPI and OpenMP, in particular the reduction mechanisms are different). This also entails that increasing the number of nodes should make the code scaling. This exactly what we see in the speed up plot: the best speed up for the 2196 dataset is obtained by allocating 4 nodes, while the large dataset is still scaling at 8 nodes. We also report the parallel efficiency, which is the speed-up divided by the number of workers. If we consider a 50% cut-off, we can see that for 2916 atoms, while the speed-up is a bit larger for 4 nodes regarding the speed-up obtained for 2 nodes, it is actually much less efficient. The efficiency for the larger dataset is also decreasing with the increasing of the worker numbers, but remains well above 50%.
In order to see a difference between the behavior of MPI workers and OpenMP ones, we would need much bigger datasets to highlight the different nature of the communication of MPI and OpenMP.




(a) (b)

FIG. 7. Speed up (in panel $a$) and parallel efficiency (in panel $b$) for the 2916 and 78732 atoms datasets.

Finally, we report the best configurations to run the code for each dataset for the three parallelization schemes we have considered:

| Dataset | MPI | OpenMP | Hybrid |
|---|---|---|---|
| 108 atoms | 8 tasks | 4 threads | # |
| 2916 atoms | 64 tasks | 32 threads | 4 full nodes (128 workers) |
| 78732 atoms | 512 tasks | 32 threads | 8 full nodes (512 workers) |

TABLE X. Best configuration for running each dataset with MPI, openMP or hybrid approach

For the small dataset there is actually no improvement in doing the hybrid approach. For 2916 the best speed up in the hybrid approach is between 2 nodes (21) and 8 nodes (35), although the second one is much less efficient then the first, as can be seen form the efficiency plot in 7. For what concerns the larger datatset, the time-to-solution scales up to the number of workers we considered, but we can see from the efficiency plot that the parallel efficiency is slowly decreasing. Further increasing the number of the nodes will lead at a certain point at the saturation of the speed-up also for this case. We also remind that, given Leonardo efficiency in internode communications, to actually see a difference in the behaviour of MPI and OpenMP workers we would need a much bigger dataset than the ones we have been con-

sidering.