University of California Berkeley

Computer Science Department

CS 267

Assignment 2.2: Parallel Particle Simulation using MPI

Submitted by
Cristobal Pais
Alexander Wu
CS 267
March 2nd
Spring 2018

Contents

Assignment 2.2: Parallel Particle Simulation using MPI

C.Pais, A.Wu

March 2nd 2018

Abstract

An optimized code for a single and multi-threaded particle simulation algorithm in the context of efficient algorithms implementations in C is developed for the NERSC's Cori supercomputer The multi-threaded version is programmed using the Message Passing Interface (MPI). The optimization process is performed in a cumulative approach based on a theoretical analysis of the vanilla implementation provided with complexity $O(n^2)$ identifying its bottlenecks and potential improvements for decreasing its complexity up to a linear complexity $O(n)$. The solution approach consists of a simple mapping between the particles system and a grid/matrix structure allowing us to decrease the number of interaction checks performed per simulation step by defining the concept of neighbors cells, checking only a limited amount of interactions. A matrix class that maps the particles into a certain cell of the mesh is implemented as the main data structure. Best serial implementation reaches a linear complexity $O(n)$. The optimal multi-threaded MPI implementation developed reaches average factors up to 0.92 and 0.88 for strong and weak scaling respectively, beating our previous shared memory implementation using OpenMP. Results of the experiments in a different machine are provided. Further improvements are discussed.

1 Introduction

A simple two-dimensional bounded particle system where particles can interact with each other as well as hit the walls of the bounded regions is implemented. This small but challenging problem is useful for understanding more complex systems implementations used in different areas such as biological sciences, physics, and engineering, as well as test our abilities to decrease the complexity of a naive simulation approach and exploit the potential parallelism of the code via the Message Passing Interface (MPI), a message-passing standard designed for parallel computations.

In this project, a discretized time simulation of a box with a constant average density of particles (scaled) is implemented. Each simulation consists of $T = 1000$ time steps where each particle interacts with other particles that are within a particular radius/zone per time step. The vanilla implementation provided has two main components: (1) Compute forces: forces are calculated for each particle taking into account the interactions with its neighbors, calculating its acceleration, after this step (2) Move particles: the movement of each particle is calculated based on its acceleration and current velocity including the potential effect of the existing walls of the region (inelastically bouncing).

The challenge of the current assignment is to implement an efficient distributed memory multi-threaded version of the particle simulation system by using optimal data structures, eliminating unnecessary steps/-computations, and parallelization tricks reaching a linear complexity performance $O(n)$. On the other hand, we seek for high-performance results with respect to the strong and weak scaling of the parallel approach: based on the number of threads p, the theoretical goal - lower bound - is to achieve a running time of $O(Tn/p)$, a perfect scaling of the parallel implementation.

C programs containing MPI instructions will be developed by the team using the first part of the project as its main starting point: the optimized serial and OpenMP versions using the efficient grid data structure that allowed the team to obtain a linear complexity $O(n)$ in the problem.

The main solution developed by the team is based on keeping a data structure that allows us to check as the minimum number of particles as possible when computing the forces and moving the particles inside the

simulation. Once the MPI version of the code is optimized and the data structure parallelization is studied, a comparison between the distributed and shared memory approach will be performed.

The structure of the report is as follows: In section 2, the main optimization methodology is described as well as the member's contribution and hardware description. Section 3 describes the tested optimization approaches and their potential impact. In section 4, results are discussed for the different optimization techniques, as well as defining and describing the main algorithm developed. The algorithm is tested on a different machine, comparing its performance to the original platform. Finally, section 5 contains the conclusions and future work of the project.

2 Methodology

The main methodology of this project consists of performing an optimization of the original particle simulation code (in C) provided with the course material, analyzing the different programming techniques/tricks that can be implemented in order to obtain a better performance, allowing us to reach an $O(n)$ complexity for the serial version while seeking an $O(Tn/p)$ running time for the parallel code. The performance is measured using the *autograder* code provided with the assignment when running instances with different number of particles and/or number of threads allowed, simplifying our analysis for both implementations: the slope of the serial version as well as the strong and weak scaling factors of the parallel code (as defined in the assignment statement, where for the *strong scaling we keep the problem size constant but increase the number of processors while in weak scaling we increase the problem size proportionally to the number of processors so the work/processor stays the same*) are automatically calculated from the summary of our results.

In the previous report, we analyzed the "AS-IS" performance of the simulator original files provided with the statement of the assignment. Once the initial results were obtained, different data structures were tested in order to achieve the $O(n)$ linear running time for both the serial and OpenMP versions. In this case, an MPI C code version is implemented using the previous codes as a starting point.

Several experimental instances differing in the n value are tested, with $n \in [500, 1600000]$ for testing the performance of our code. Depending on the techniques applied for optimizing the original algorithm, a series of parameters are optimized for each approach, performing the experiments - mainly following a brute force rule - and keeping the best solutions achieved.

In addition, all reported implementations are tested in a daily (common-user) machine for comparison purposes, being able to understand the importance of developing specific algorithms implementations depending on the hardware characteristics. A series of plots are generated for each experiment in order to visualize the performance of the different approaches.

2.1 Contribution

During the development of the project, each member of the team develops its own experiments, sharing his results in a common Github repository for checking the current state of the implementations while re-using code from other members code. In addition, cross-testing of other members' code is performed. After performing a series of experiments, the best implementation is selected as the main submission file.

Based on the experience and preferences of the members of the group, the members' contributions and tasks can be summarized as follows:

- **Cristobal Pais**
  - Serial $O(n)$ implementation.
  - Wrote library of grid data structures.
  - Initial writer of the MPI version (Naive approach) and Shared-memory (RMA) version.
  - Editor and main writer of the current report.

- **Alexander Wu**
  - MPI optimized version.
  - Cleaned up data structures for optimal parallel performance.
  - Changed grid updating strategy.
  - Contribution to the report.

2.2 Hardware & Software

The optimization of the particle simulation algorithm is developed for a specific hardware and runtime environment from The National Energy Research Scientific Computing Center (NERSC). In addition, for comparison purposes, the same code is tested in a common-user machine (e.g. a simple laptop). All experiments, benchmarks, and performance results are implemented using the following hardware and software:

2.2.1 Hardware

1. **NERSC's Cori supercomputer**
   - Intel® Xeon™ Processor E5-2698 v3 ("Haswell") at 2.3 GHz
     - 2 sockets per chip (32 cores per node)
     - Supports SSE, SSE2, SSE3, SSE4, AVX, AVX2 (among others).
   - Memory
     - 64 KB 8-way set Level 1 cache (32KB instructions, 32 KB data)
     - 256 KB 8-way set Level 2 cache
     - 40 MB 20-way set Level 3 cache (shared per socket)
   - Software
     - SUSE Linux version 4.4.74-92.38-default
     - Built with PrgEnv-intel and PrgEnv-gnu/ gcc version 4.8.5
     - Default Compiler flags: -O3
2. **Personal computer: Laptop**
   - Intel(R) Core(TM) i7-4510MQ CPU @ 2.00GHz
     - 2 cores, 4 threads
     - Supports SSE, SSE2, SSE3, SSE4, AVX, AVX2 (among others).
   - Memory
     - 32 KB Level 1 cache
     - 256 KB Level 2 cache
     - 4 MB Level 3 cache (shared per socket)
   - Software
     - Ubuntu 16.04.2 LTS
     - gcc/mpcc version 4.8.5
     - Default Compiler flags: -O3

2.3 Assumptions

Based on the statement of the problem, our implementation of the particle simulator program satisfies the following assumptions:

- A bounded square region contains the entire system. Interactions between particles and walls are inelastic.
- Density of the whole particle system is kept constant when modifying the number of particles inside the grid, scaling its size.
- Interactions between particles occur within a bounded region.
- Linear scaling between work and processors.

3 Solution Description

In this section, we present the main strategies applied for optimizing the original particle simulator. First, as done in the first part of this project, the strategy applied to obtain a serial version of the code with an asymptotic complexity of $O(n)$ is summarized, consisting of a matrix/grid data structure for modeling the relevant interactions between particles and their neighbors instead of checking the $n-1$ particles for every element. Then, a specific analysis of the distributed memory MPI implementation is performed, indicating the main modifications and techniques used for getting the best possible performance.

3.1 Single threaded (Serial)

In order to obtain the $O(n)$ complexity implementation, we divided our analysis into two main components:

3.1.1 Grid optimization

We decided to use a grid to convert our serial routine's runtime from $O(n^2)$ to $O(n)$. This $\approx \sqrt{n} \times \sqrt{n}$ grid divided the simulation space into $\approx n$ subspaces, and would partition the particles by its location. Rather than interact each particle with each other, because the strength of the force between particles is inversely proportional the square of the distance between particles, particles which are close together affect each other much more than particles that are far apart. We choose to keep the grid dense relative to the number of particles ($O(n)$ total spaces), and only interact each particle with the particles in the 8 neighboring spaces on top of the particle's own space, rather than interacting each particle with all n other particles. By doing so, we achieve $O(1)$ runtime for a single particle (because there is on expectation 1 particle per grid space), and $O(n)$ runtime for all n, with negligible error relative to the "true" simulation of interacting all $O(n^2)$.

We represent our grid as an array of dynamically sized vectors (for storing particles). For a detailed comparison between vector types and similar data structures performance, we perform a series of tests as well as checking articles/studies covering the topic (see [2]).

In order to implement the selected data structure, we developed a class *matrixCells* containing the main logic and the mapping information between the particles and the mesh as well as an auxiliary class *matrixIter* for performing the operations related to iterations across the vectors. These two classes contain all private and public parameters and methods relevant for the particle simulator. The most important ones of our initial implementation are:

1. *push2mesh*(): pushes the n particles inside the mesh object, generating the map between the underlying dynamic vector data structure and the two-dimensional simulation system.
2. *insert*(): inserts particles inside the mesh via pointers using the *push_back*() native function.
3. *clear*(): particles' pointers are eliminated from the underlying mesh structure for a new time step.

3.1.2 Updating the Grid

Initially, at the end of every time step, we would clear the entire grid of its particles, and repopulate it based on the new location state of the particles using the *clear*() method. However, we found this to be very computationally intensive and took a lazy approach instead. For each particle, after we update the particle's x and y coordinates for the timestep, we also check to see if it has changed grid locations. If so, we remove it from the old bucket and add it to the new one.

This comes at a slight computation cost to remove something from a vector, but because our vectors are small on expectation, we prefer to keep the simplicity of using an array-based data structure. Thanks to this improved implementation, we are able to obtain an $O(n)$ asymptotic complexity, as we already discussed in the first part of this assignment (and will check in section 4 for the MPI implementation).

3.2 Distributed memory (MPI)

Following the successful serial and shared memory implementations, we use them as the basic structures for our multi-threaded code. In this case, we already include all the optimization techniques applied in the first part of this assignment (lazy grid update, locks, etc., as briefly discussed in the previous section). Therefore, we develop a series of MPI implementations until we obtain the desired performance level. The analysis is the following:

3.2.1 Naive MPI approach

Mixing the structure of the vanilla MPI code provided with the assignment as well as the OpenMP and serial version already implemented in the first part of the project, a naive MPI code is developed. The main steps of this implementation are the following:

1. The parallel environment is initialized via the $MPI\_INIT()$ command. The size of the "world" and the rank of the current process are queried.
2. The data type associated with the particle objects are declared for the MPI world.
3. Data is partitioned across processors: evenly distributed for getting a balanced load, being aware of potential leftovers. Memory is allocated for the local partitions.
4. The root process ($rank = 0$) initializes the particles. Once initialized, these are broadcasted using the $MPI\_Bcast()$ command. Therefore, each thread will have a full set of n particles (for simplicity in this Naive implementation).
5. A grid is initialized in each thread. Main simulation loop starts by pushing the particles objects inside the generated mesh.
6. Particles' forces and movements are calculated by each thread for its relevant particles (based on the previously calculated indexes/offsets). Thus, each process updates its own particles inside the mesh.
7. Finally, all particles status are shared among all processes using the $MPI\_AllGatherv()$ command, allowing us to have exactly the same state of the simulation across all threads.

The major advantage of this implementation is its simplicity: only a few modifications of the vanilla MPI and the optimized serial implementation from part 1 are required. However, as we will see in section 4, its performance is not particularly optimized. Copying the particles and grids among all threads is clearly not efficient since each process is only checking the neighbors' cells for its assigned particles, not needing to carry with all the data from the beginning (just the adjacent cells to the ones assigned), like in the already implemented shared-memory approach using OpenMP. Therefore, the optimized implementation will tackle this significant flaw of the naive implementation.

Using this simple implementation, we are able to boost the performance of the vanilla's scaling factors from 0.18 (strong) and 0.08 (weak) - with respect to the optimized serial version - up to 0.37 and 0.31, mainly due to the underlying grid structure that allows our algorithm to reach a linear asymptotic complexity $O(n)$.

3.2.2 Shared-memory like approach: one-sided MPI

Based on the contents from lecture 10 "*Advanced MPI and Collective Communication Algorithms*" and using our previously implemented OpenMP code, we tested the one-sided MPI calls implementation in order to emulate a shared-memory approach in an attempt to obtain a very similar performance to our OpenMP formulation without needing to perform a significant amount of changes in both our original data structure and main execution code. The main idea consists of creating two shared objects/windows associated with the particles and the underlying mesh structure containing them. Sharing these objects in memory, allows us to not need to carry individual copies of them with each thread saving a significant amount of memory usage and communication time for broadcasting the simulation results across all workers, and also, since the read/write operations are performed directly inside the window and each worker takes care of a particular segment, no duplicated tasks would be performed.

However, this approach has its own drawbacks as described during lectures and in [3] *MPI's one-sided routines take a very symmetric view of processes: each process can access the window of every other process (within a communicator). Of course, in practice there will be a difference in performance depending on whether the origin and target are actually on the same shared memory, or whether they can only communicate through the network.* For this reason, MPI allows us to group different processes into shared memory domains by using the $MPI\_Comm\_split\_type$ command.

Hence, the implementation is as follows:

1. An MPI_WIN is initialized for the particles object as well as the mesh generated via the $matrixCells$ class. These objects will emulate the shared-memory environment during the simulation, allowing each thread to access (read and write) them without performing communication/synchronization steps.
2. The root process ($rank = 0$) allocate and shares the object via the $MPI\_Win\_allocate\_shared()$ function, recording the address in memory of the object via a base pointer.
3. The rest of the threads perform a query using $MPI\_Win\_shared\_query()$ to the windows object in order to get access to its memory address.
4. Specific working indexes are calculated for each thread based on the even distribution load balance approach provided with the Vanilla MPI implementation. Hence, each process will perform operations with only their relevant objects: the "private" particles and the associated part of the mesh.
5. Finally, a barrier is located at the end of the main simulation loop in order to assure that each iteration starts with the current simulation state (where all particles have been updated from the last iteration). This is implemented using the $MPI\_Barrier$ command.

Based on the OpenMP implementation, we expect that the performance of this implementation should be better than the Vanilla and Naive approaches since we are emulating the efficient OpenMP shared-memory like structure. In addition, we are decreasing the amount of communication between workers, a usual bottleneck when using MPI. However, this was not the case: no significant improvements - and even worse results - are obtained in comparison to the Naive formulation.

Besides the difficulties of coding this approach due to our lack of experience with these level of MPI instructions (never worked with them before), the overhead produced by the addition of the barrier at the end of the main loop significantly impacts the overall performance of the code, not allowing us to obtain the expected (better) results. About 80% of the time is spent in the barrier command, indicating us that unless we modify this step, the code will not be able to be improved in any significant way. Different workarounds were tested: sending individual messages, use of locks, $MPI\_PUT()$ and $MPI\_GET()$ commands, etc., however, no improvements were obtained. In addition, important is to note the fact that in this case, the shared

memory objects are actually being shared through the network (between nodes) and thus, the performance cannot be as good as with the real shared memory approach applied with our OpenMP version.

Therefore, we discard this approach, mainly due to our lack of experience and complete understanding of how to exploit its powerful commands: we would focus in those features/characteristic that we know we are able to take advantage for obtaining an optimized version of the code. Although it took a considerable amount of time (coding and testing), the experience was very useful as a first approach to this MPI programming style, allowing us to learn new commands that were not known by the team members, giving us new options for tackling future projects.

3.2.3 Optimized MPI approach

Noting and analyzing the main drawbacks from the Naive MPI approach and the Vanilla implementation provided with the statement, an optimized MPI code is developed. In order to be able to obtain a better performance with the distributed memory approach, we decided to modify the current data structure defined by the $matrixCells$ class used with the serial and OpenMP versions already implemented since its core and data manipulation logic is not as flexible as we need for performing the interaction and communication between the different threads in the distributed memory fashion.

Therefore, the main structure of the algorithm is as follows:

1. *Initialization*: The parallel environment is initialized ($MPI\_INIT()$).
2. *Partitioning*: Data is partitioned column-wise and the root process initializes the particles objects. These are broadcasted to all processes using $MPI\_Bcast()$. This step required a modification of the original data structure, including new methods and parameters for manipulating the particles.
3. *Grid*: A mesh is initialized and particles are pushed into it using the modified version of the $push2mesh()$ function that takes into account the ranking of the process as well as "owned" set of particles.
4. *Halo Zones*: The main simulation loop starts with an MPI barrier for synchronizing all workers. We start communicating the halo zone of each node to the corresponding neighboring processors. We choose to use an asynchronous $MPI\_Isend()$ followed by $MPI\_Recv()$ and a Barrier to synchronize communication. These particles are added to the grid. Thanks to this implementation, we both avoid possible deadlocks as well as improve the performance of the communication overhead due to less blocking operations.
5. *Compute Forces*: Each process calculates the forces of its own particles. The halo zone particles are removed from the grid.
6. *Move particles*: Each process updates the position of its own particles within the mesh. Particles that have to "migrate" to a different node are removed from the grid altogether and moved to a separate migration buffer.
7. *All-to-All communication*: An all-to-all gather of the migrating particles is performed with $MPI\_Allgatherv()$. Each node looks in the gather for particles that belong to itself, and add them to its mesh.

Using this implementation, we will be able to obtain a significant performance boost in comparison to both the Naive and Vanilla implementations. As we will discuss in section 4, a slight variation of this optimized version will be developed, obtaining our final optimal MPI implementation, noticing that it is possible to improve the performance of the communication steps by reducing the size of the messages sent.

Therefore, we proceed to the experimental stage in order to test our implementations, performing a series of simulations for different instance sizes.

4 Results & Discussion

In this section, we present the main results obtained from a series of experiments including the optimization techniques introduced in section 3.

4.1 Preliminary results: Vanilla and Naive versions

Using the code from the first part of this project as the basic structure of our MPI implementation as well as the vanilla implementation provided with the assignment statement, we obtain a series of preliminary results for benchmarking our optimized code.

4.1.1 Vanilla MPI

When running the AS-IS/Vanilla version of the code provided with the assignment, we can clearly see in Figure 1 that the implementation is not very efficient and taking advantage of the parallelism: adding more threads p is not always better, event obtaining worse performance with 32 than 24 threads. We can easily compare and analyze these results by looking at Figure 2 where the associated speedup plot is presented, for the reduced instance with 500 particles (due to our quota constraint in Cori we do not include larger instances in detail for the Vanilla version since they take too long to simulate). Clearly, when comparing the Vanilla version attained performance with respect to the ideal benchmark, we can see in Figure 2 how this original implementation is not able to reach any value similar to the optimal performance after the 6 threads threshold, with very low-efficiency values as well as speed-up factors (not even five times the single-threaded solving time). Therefore, it is clear that adding more threads is not impacting - at all - the performance of the code. The explanation behind this pattern consists of the amount of time spent in communication, the quadratic complexity of the original code $O(n^2)$, and synchronization operations.

Looking at the strong scaling factor obtained in Cori, we obtain 0.57, and 0.23 factors for the weak scaling performance metric with respect to the Vanilla serial version and poor 0.18 and 0.08 factors when using the optimized serial version. Therefore, the code is clearly non-scalable and does not take advantage of the resources available in Cori. Similar poor results are obtained when testing the code with larger instances $n \geq 50000$, where running times become too large (due to the underlying $O(n^2)$ complexity).

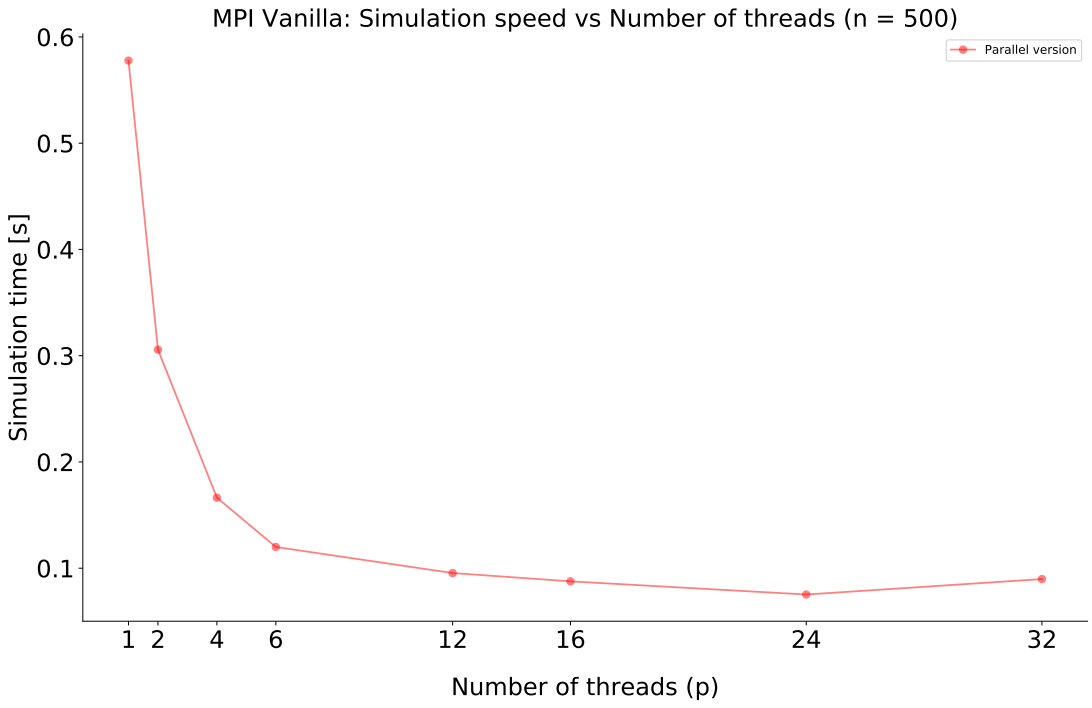


Figure 1: Vanilla MIP multi-threaded performance increase with respect to number of threads ($n = 500$).

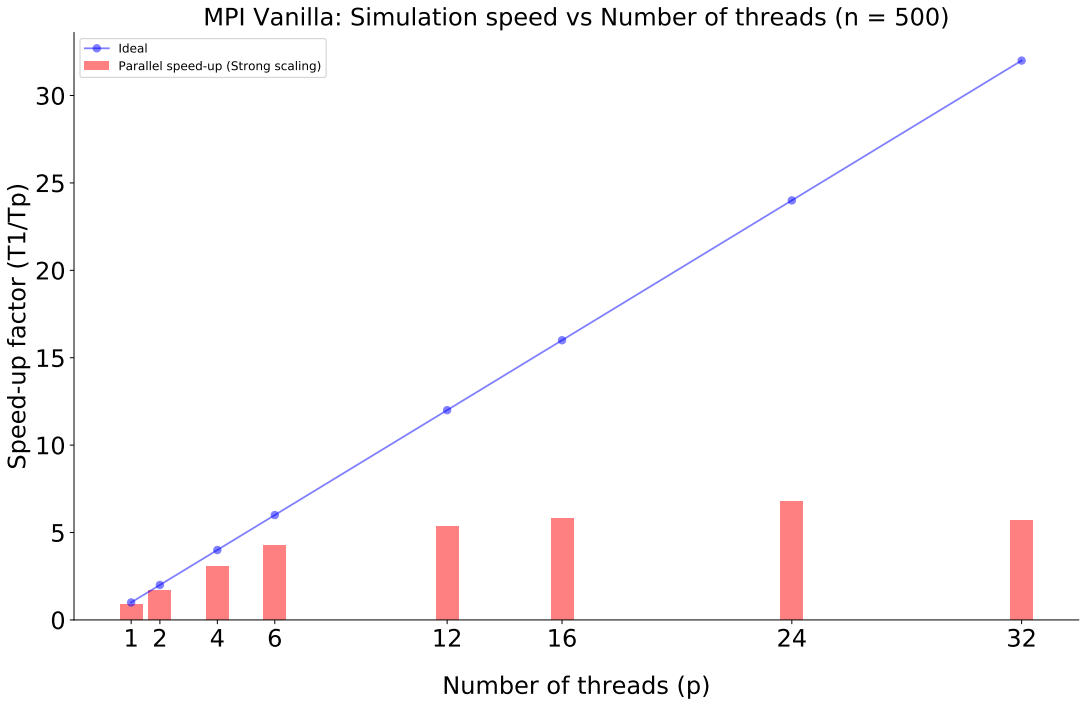


Figure 2: Vanilla MPI multi-threaded speed-up factor versus number of threads for $n = 500$ (w.r.t Vanilla serial version).

4.1.2 Naive MPI

Although to its inherent - and known - limitations as described in section 3.4.1, the Naive MPI implementation using the mesh structure developed for the serial and OpenMP versions of the first part of this project is performing better than the Vanilla version in terms of solving times. Looking at the results for small instances ($n = 500$, tested up to $n = 16000$) in Figures 3 and 4 we can observe better simulation times than the Vanilla implementation as well as a non-erratic pattern when adding more threads to perform the simulation: adding more threads leads to better solving times, although with a decreasing rate - as expected based on our previous sections discussion - due to its unoptimized structure. In a similar way, the strong scaling performance is better than the one obtained with the Vanilla implementation w.r.t. the optimized serial implementation: 0.28 vs 0.18.

In Figure 3, we can see significant improvements in the obtained simulation times when including up to 12 cores, however, after this threshold, we can clearly see how the performance of the code is no longer significantly improved, being almost a flat line when using more than 16 cores. These results are consistent with the speed-up plot provided in Figure 4 where we can see that the speed-up factor obtained with 16, 24, ad 32 cores are almost identical. In addition, it is possible to see how far our results are from the ideal benchmark factor: the average strong scaling factor is around 0.28 while the average weak factor is no more than 0.3[1].

Better results are obtained when running the same implementation with larger instances $n \geq 50,000$. In Figures 5 and 6 we present the results obtained when $n = 50,000$. From these plots, we can see that the simulation speed versus the number of threads pattern is very similar to the one obtained with the smaller instances where adding more threads lead to better simulation times, but following a decreasing rate trend. On the other hand, we have that the speed-up factors are slightly improved when dealing with a bigger instance, being able to obtain an average strong scaling factor around 0.37 (and an average of 0.3 for weak scaling), a pattern that is consistent with the speed-up factors presented in Figure 6.

In this case, as opposed to the Vanilla implementation and the current approach implemented with small instances we can see a significant improvement when adding threads, also after the 16 cores threshold: the current Naive implementation is able to better exploit the resources in Cori. However, it is still not efficient

[1] Average strong and weak factors are reported by running each implementation 10 times and computing the relevant averages.

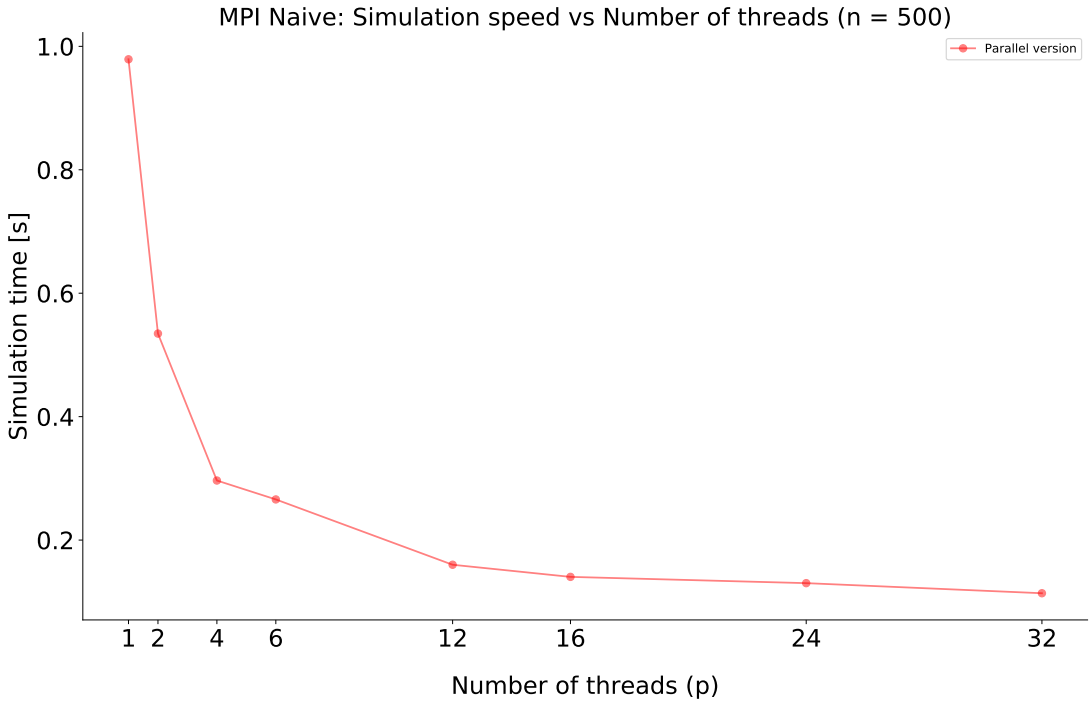


Figure 3: Naive MIP multi-threaded performance increase with respect to number of threads ($n = 500$).

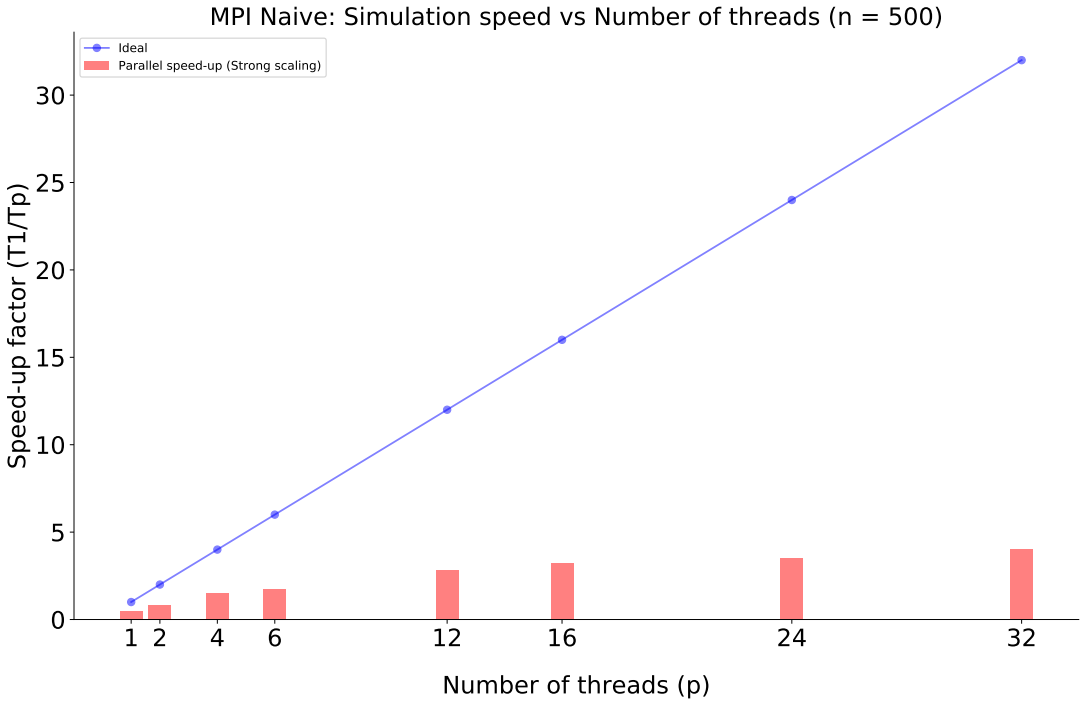


Figure 4: Naive MPI multi-threaded speed-up factor versus number of threads for $n = 500$.

enough to be a useful distributed memory implementation. Remembering that our shared memory using an OpenMP implementation reached average strong and weak scaling factors around 0.80 and 0.70 respectively, we have a large margin of improvement by tackling the known performance bottlenecks of this first Naive implementation.

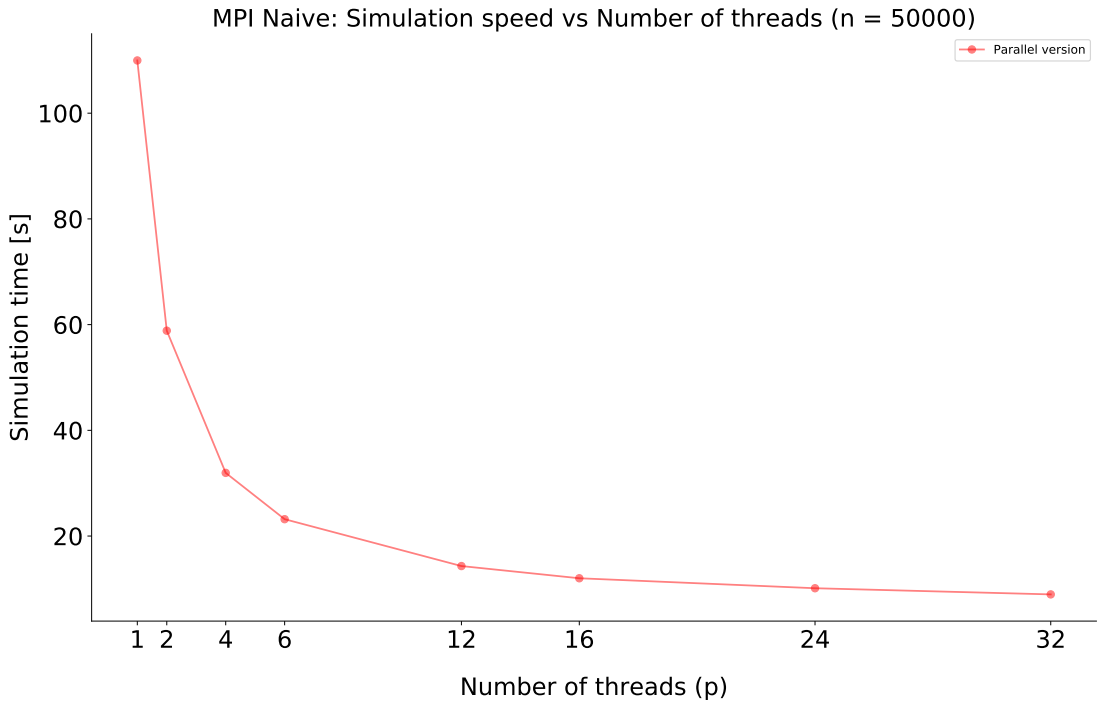


Figure 5: Naive MIP multi-threaded performance increase with respect to number of threads ($n = 50000$).

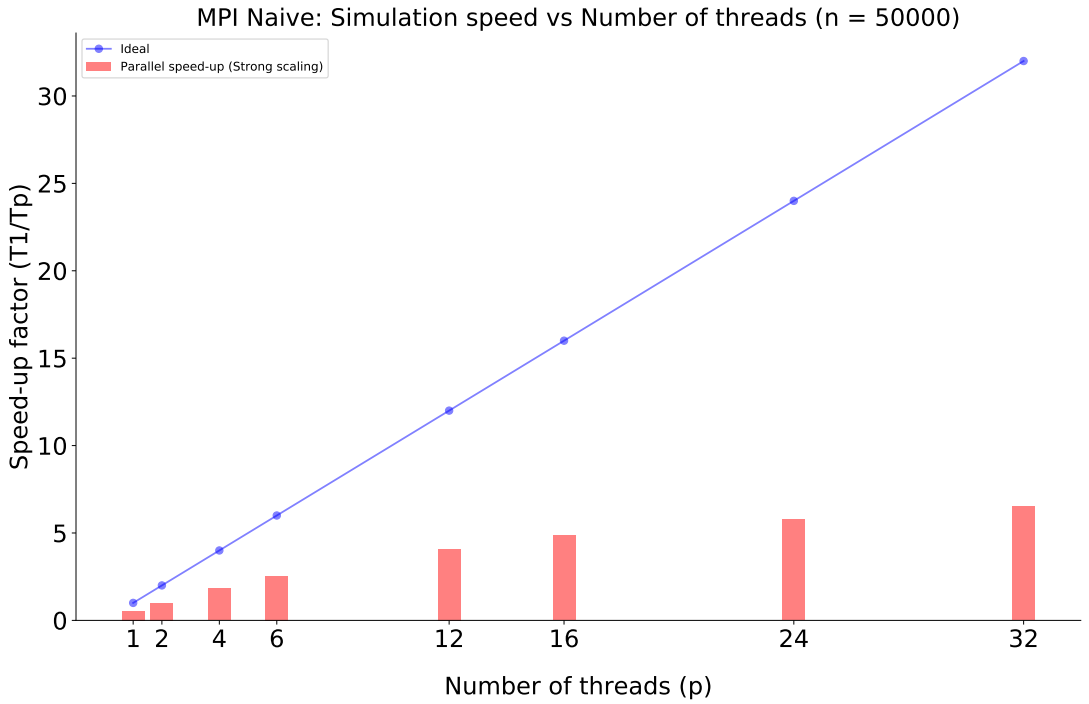


Figure 6: Naive MPI multi-threaded speed-up factor versus number of threads for $n = 50000$.

4.2 Optimized MPI versions: Performance & Processors

Based on the single-threaded and OpenMP versions of the previous report, and following the techniques discussed in section 3, two optimized MPI multi-threaded version are developed. The main difference between both implementations lies in the number of columns from the mesh that are being sent by the workers: the second optimized version is just communicating the relevant ones, as described in section 3 and thus, we expect to obtain the best performance with this final implementation. For completeness, both versions are analyzed and compared, explicitly showing the evolution of our implementations.

Two main performance metrics are analyzed: (1) complexity $O(n)$, and (2) weak and strong scaling factors. The FLOPS analysis is discarded following the information posted in Piazza: *Haswell processors simply don't have some of the CPU counters that Ivy and Sandy Bridge processors had, so performance monitoring software like CrayPat and PAPI isn't able to provide FLOP counts.*

4.2.1 Complexity

As with the single-threaded version of the code, we want our parallel code to reach an $O(n)$ asymptotic complexity. In order to check this performance metric, we fix the number of threads p to a certain number and then we vary the number of particles to be simulated.

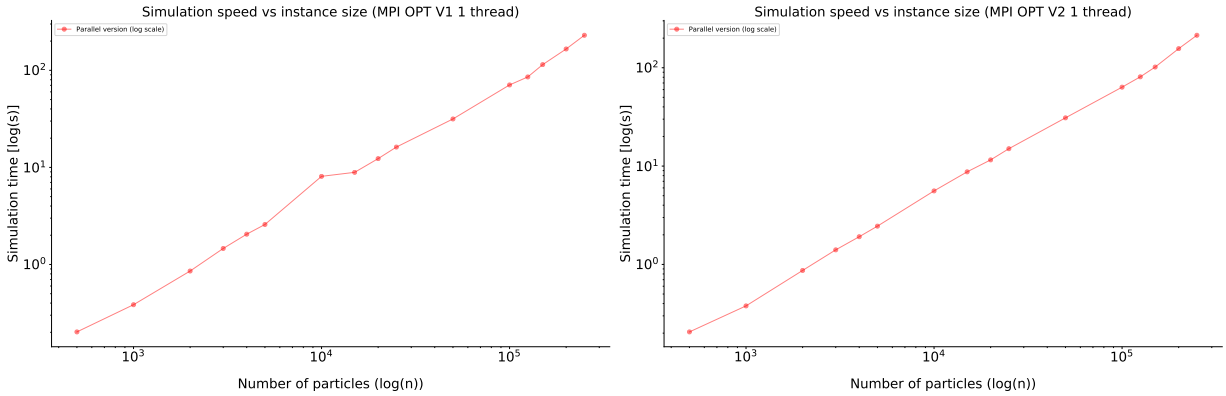
In Figures 7, 8, and 9 we can clearly see that our MPI implementations are able to reach the desired $O(n)$ complexity. Looking at the slope of the curves we can easily see that the slope tends to one. Checking the results obtained from the *autograder* utility provided with the assignment, we can check that the estimated asymptotic complexity (slope) is very close to 1 (Table 1). Furthermore, in the case of 4 and 16 threads, we can see that the estimated slope value is below the unit, allowing us to conclude that the MPI implementation is satisfying the complexity $O(n)$ requirement.

In addition, it is possible to see that - as expected - the second optimized version (V_2) obtains better running times than version 1, indicating us that the implementation of the optimization techniques mentioned in section 3 has been successful.

Table 1: Summary results from parallel implementation using 1, 4, and 16 threads with $n \in [500, 250000]$

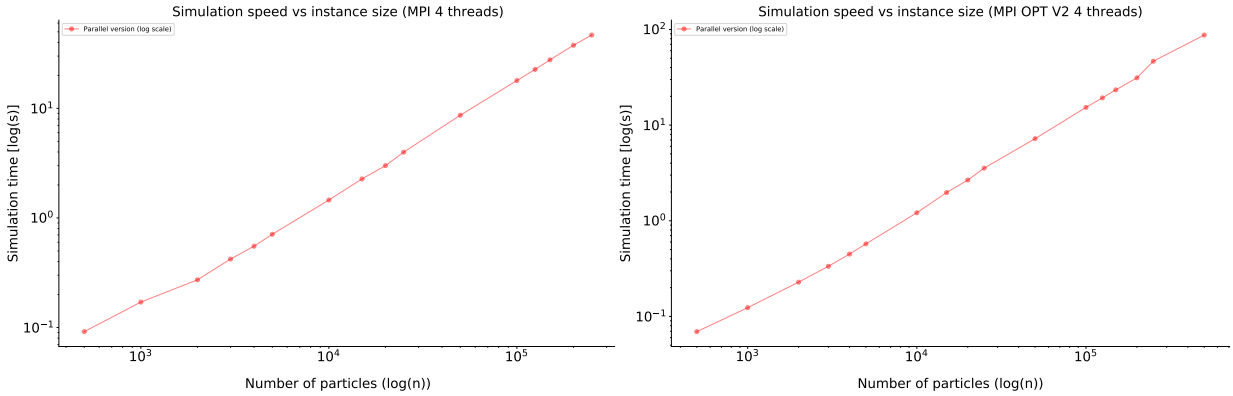
Size	**Time 1C [s]**		**Time 4C [s]**		**Time 16C [s]**	
n	V_1	V_2	V_1	V_2	V_1	V_2
500	0.201	0.205	0.080	0.069	0.088	0.064
1000	0.384	0.378	0.204	0.123	0.103	0.081
2000	0.855	0.868	0.269	0.228	0.318	0.118
3000	1.461	1.406	0.470	0.334	0.171	0.134
4000	2.051	1.913	0.542	0.448	0.222	0.197
5000	2.581	2.450	0.684	0.573	0.273	0.190
10000	8.089	5.593	1.415	1.212	0.512	0.324
15000	8.893	8.742	2.231	1.974	0.629	0.470
20000	12.315	11.563	3.007	2.665	0.801	0.637
25000	16.195	15.026	3.986	3.550	0.984	0.874
50000	31.550	30.941	7.930	7.215	1.977	1.754
100000	70.702	63.481	17.649	15.329	4.053	3.573
125000	85.467	80.848	21.627	19.269	5.045	4.537
150000	114.435	101.684	27.476	23.378	6.125	5.522
200000	165.943	156.402	37.971	31.203	8.749	11.532
250000	229.659	213.935	47.121	46.577	10.568	9.302
Estimated Slope	1.121	1.110	1.073	1.062	0.820	0.890

Therefore, we are able to see the benefits of our modified serial implementation's data structure in the parallel



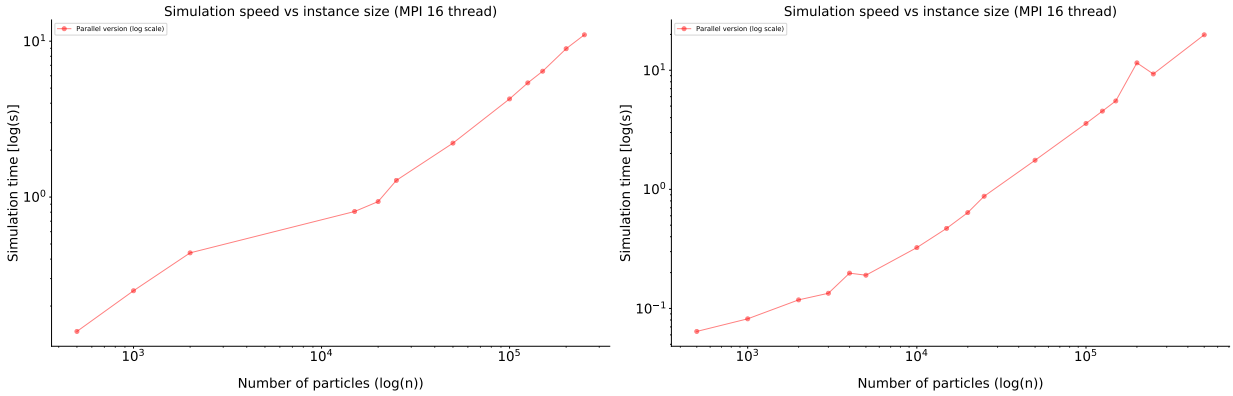
(a) MPI OPT V1 results (b) MPI OPT V2 results

Figure 7: MPI multi-threaded (1 thread) implementation log-log scale plot for instances $n \in [500, 500000]$.



(a) MPI OPT V1 results (b) MPI OPT V2 results

Figure 8: MPI multi-threaded (4 thread) implementation log-log scale plot for instances $n \in [500, 500000]$.



(a) MPI OPT V1 results (b) MPI OPT V2 results

Figure 9: MPI multi-threaded (16 thread) implementation log-log scale plot for instances $n \in [500, 500000]$.

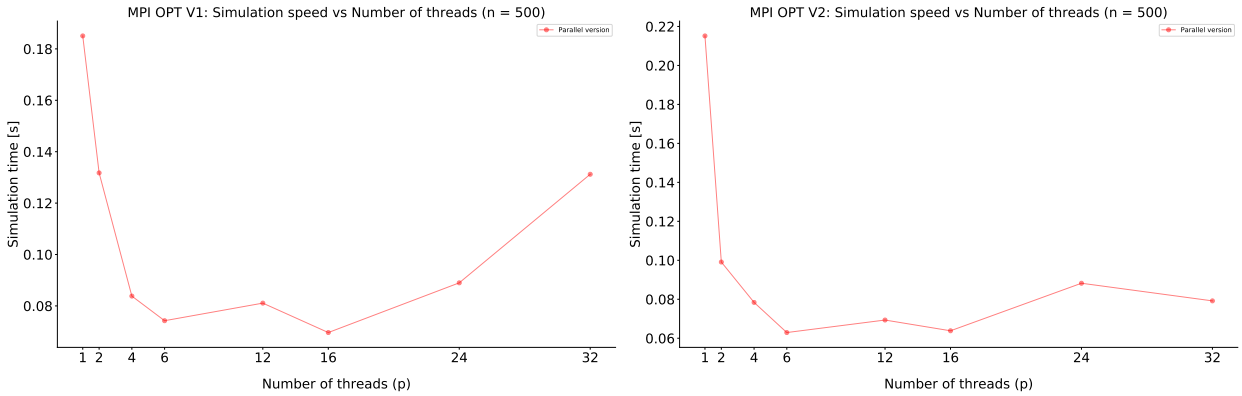
version. As stated above, we needed to modify/add/reformulate some of its basic components/methods in order to obtain the best performance of the distributed memory approach, as we discovered by analyzing the results obtained by our initial Naive implementation.

4.2.2 Speedup analysis

One of the classic challenges when implementing a parallel approach for a particular problem is to be able to reach good values for both the weak and strong scaling factors, key indicators of the performance of the overall parallel implementation. Therefore, the main objective of our implementation is to be as close as possible to the idealized *p-times speedup*, when we reach a perfect scaling factor (efficiency is equal to one).

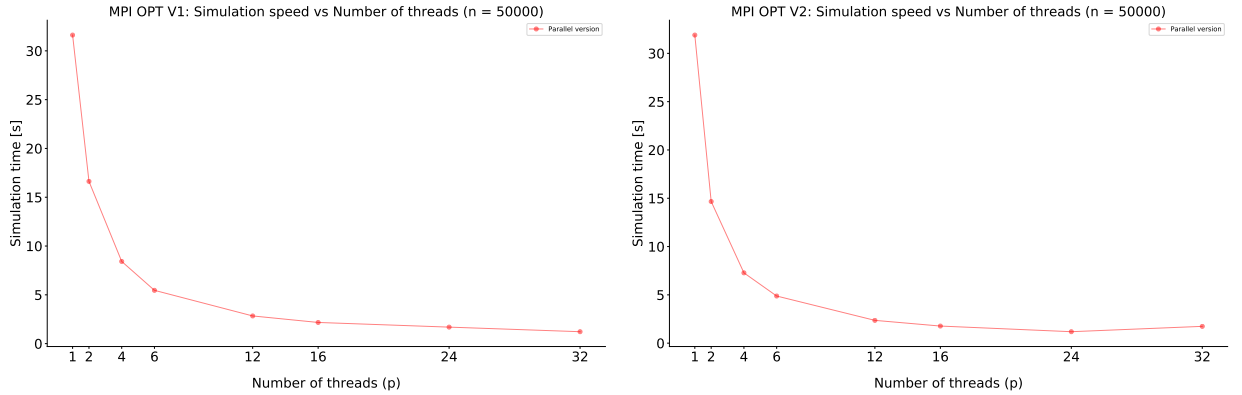
When running our Naive version - as well as the Vanilla code provided -, the MPI implementation obtains very poor results for both scaling factors: 0.37 of strong scalability and 0.3 of weak scalability. Therefore, these implementations are not able to obtain a significant performance boost when adding more threads and thus, they are not really useful for large-scale problems due to their lack of ability to take advantage of the underlying hardware/resources available.

With our final parallel implementations V_1 and V_2, we are able to obtain significantly better results. As can be seen for instances with $n = 500$ and $n = 50000$ in Figures 10 and 11, we can see how the performance of the simulator is significantly boosted for the larger instance when adding more and more threads to the computation, indicating that the strong scaling factor is improved. However, it is also clear that there exists a slight decreasing performance factor as well as an erratic performance with $n = 500$ (too small instance for exploiting the parallel implementation performance) allowing us to expect that the efficiency is around $75\% - 80\%$ (with respect to our best optimized serial version) due to the bad scaling after including more than 16 threads.



(a) MPI OPT V1 results (b) MPI OPT V2 results

Figure 10: Optimized MPI multi-threaded speed-up factor versus number of threads for $n = 500$



(a) MPI OPT V1 results (b) MPI OPT V2 results

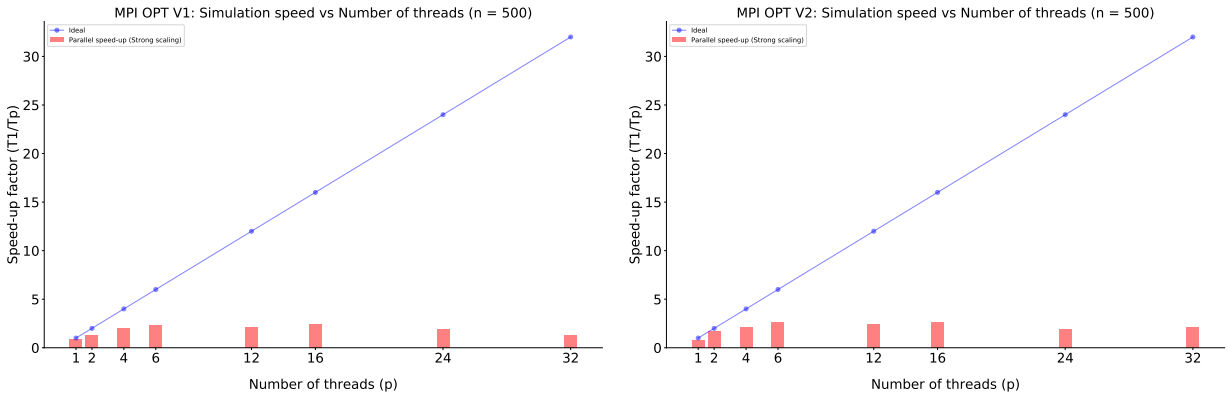
Figure 11: Optimized MPI multi-threaded speed-up factor versus number of threads for $n = 50000$

When comparing the attained performance with respect to the ideal benchmark, we can see in Figures 12 and 13 how our implementation is able to reach results very similar to the optimal performance up to 16 threads (V_1) and 24 (V_2) when solving large instances ($n = 50000$), with an efficiency around 82% for the V_1 implementation and 92% for the second version V_2 of our code. Interesting is the fact that the final

optimized version is able to get almost a perfect performance up to 24 cores, however, there is a significant performance loss when using the maximum amount of cores 32.

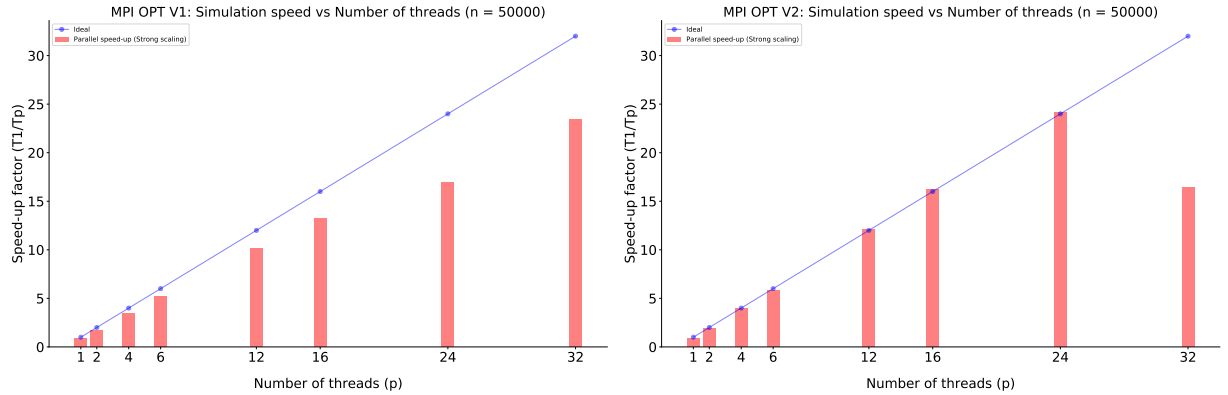
In contrast to our OpenMP implementation, after this threshold, adding more threads is still impacting the performance of the code, significantly decreasing the simulation times when 24 and 32 threads are utilized. Therefore, our optimized MPI versions are able to obtain better parallel performance factors than our previous OpenMP shared memory optimized code (0.79 and 0.68: strong and weak).

On the other hand, we can clearly see that our optimized code is not useful for instances as small as 500 particles: adding more processes/workers is not improving the performance of the code because we are adding more communication overhead and the instance is not large enough to take advantage of the parallel approach. Even the Naive and Vanilla implementations obtain better strong scaling factors with this reduced size instance (but not running times).



(a) MPI OPT V1 results (b) MPI OPT V2 results

Figure 12: Optimized MPI multi-threaded speed-up factor versus number of threads for $n = 500$



(a) MPI OPT V1 results (b) MPI OPT V2 results

Figure 13: Optimized MPI multi-threaded speed-up factor versus number of threads for $n = 50000$

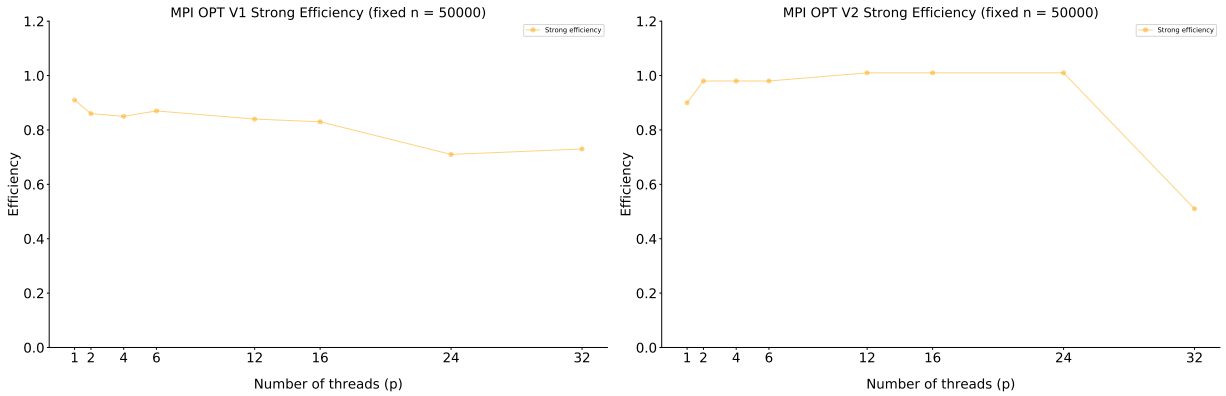
For comparison purposes between our both optimized versions, we include the weak and strong efficiency plots in Figures 14 and 15 that allow us to understand the performance pattern of our implementation. As expected from our previous analysis, results are very good in terms of weak and strong scaling.

From the efficiency plots, it can be concluded that our implementations present both good strong and weak scaling performance when using up to 32 cores (0.82 and 0.92 respectively), however, there is a significant negative impact in this metric when working with 32 cores: we are not taking full advantage of the processors in Cori due to a not completely optimized MPI implementation. On the other hand, the weak efficiency performance is not as good as the strong one, obtaining slightly lower values around 0.77 and 0.88 for each version, respectively.

From the plots, it can be seen that the weak efficiency tends to be very stable across all number of threads around 0.8 for the V_1 implementation and around 0.9 for the V_2 version. On the other hand, the strong

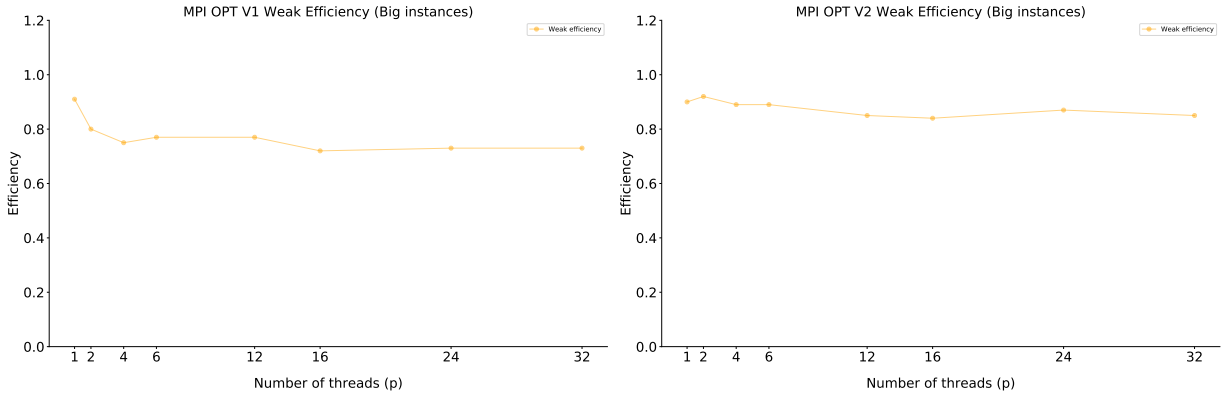
efficiency presents a decreasing trend from values near 0.9 (1-6 cores) up to 0.75 (24-32 cores).

Therefore, based on the results we can conclude that our final implementations have the following characteristics: (1) the parallel overhead tends to vary faster than the amount of work (weak scaling) and (2) the parallel overhead tends to be more stable as we add more and more processors (strong scaling). In addition, this implementation is able to beat our previously coded shared memory approach based on OpenMP directions. For completeness, we include the results of the speed-up obtained with $n = 100000$ in Figure 16, where we can clearly see that the strong scaling factors are consistent with the ones obtained with $n = 50000$.



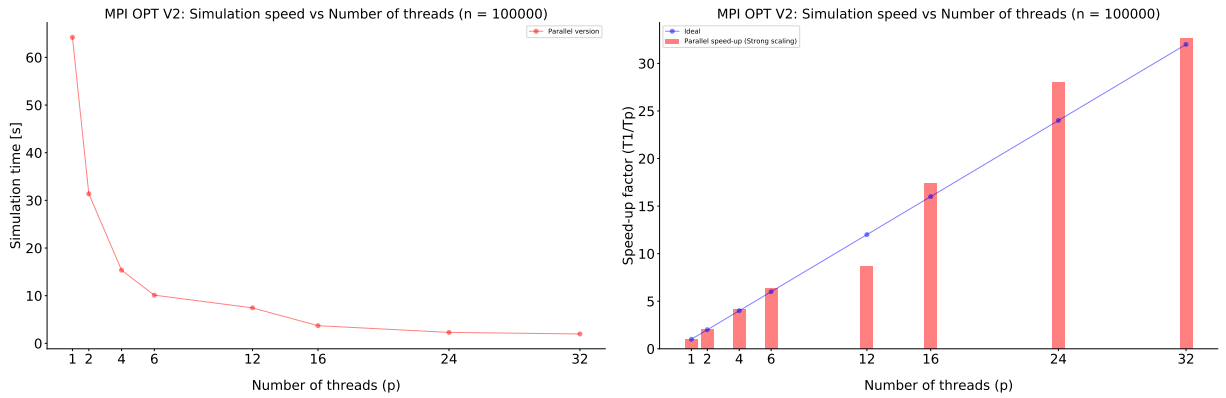
(a) MPI OPT V1 results (b) MPI OPT V2 results

Figure 14: Strong efficiency values: different number of threads and fixed $n = 50000$.



(a) MPI OPT V1 results (b) MPI OPT V2 results

Figure 15: Weak efficiency values: different number of threads and proportional n values.



(a) Solving time evolution plot (b) Speed-up plot

Figure 16: Optimized MPI multi-threaded speed-up factor versus number of threads for $n = 100000$

4.2.3 Time analysis

In order to deeply understand the complexity and behavior of our implementation as well as its bottlenecks and potential improvements, we benchmark the overhead in synchronization when more threads are added for solving different instance sizes n.

In section 3, we briefly described and give an overview of the synchronization and communication trade-offs that we made in order to develop our final optimized code. The numbers in the plot below (Figure 17) represent the proportion of the total simulation time that is spent in synchronization operations. On the x-axis, we have the size of the instance n and the number of threads p used in the experiment. Due to our Cori's quota limitation, the analysis is developed with medium size instances from $n = 5000$ up to $n = 160000$. Similar (proportional) results are obtained with different instances' magnitude.

Each point is an average of a series of runs in order to avoid variance bias. In an ideal world, we expect the sync time to be the same. The increase in time below we attribute to synchronization overhead.

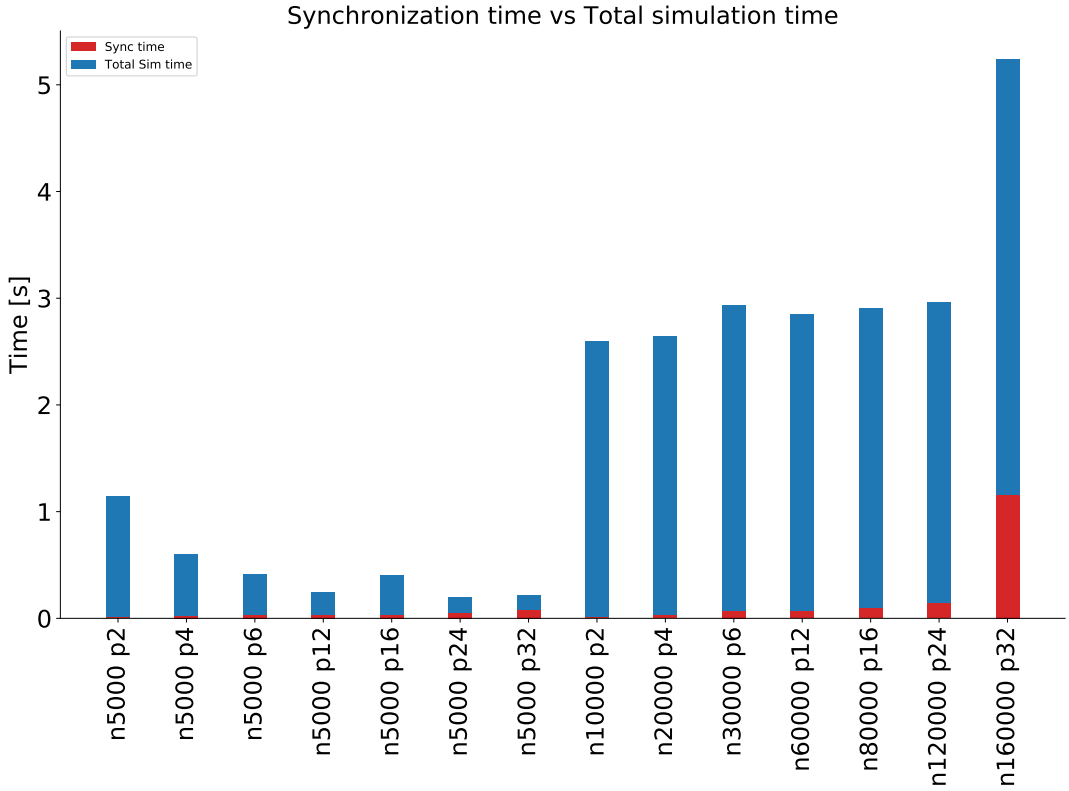


Figure 17: Synchronization time as a proportion of the total simulation time for instances with $n \geq 5000$.

Based on the results above, we can see that our final implementation is able to keep the synchronization time pretty stable and low across different instance sizes and the number of threads included. However, as with the speed-up plot analysis, we observe a different (worse) performance pattern when using 32 threads: our code is not able to take full advantage of the parallel implementation when including the maximum amount of threads to be tested under the distributed memory approach in Cori.

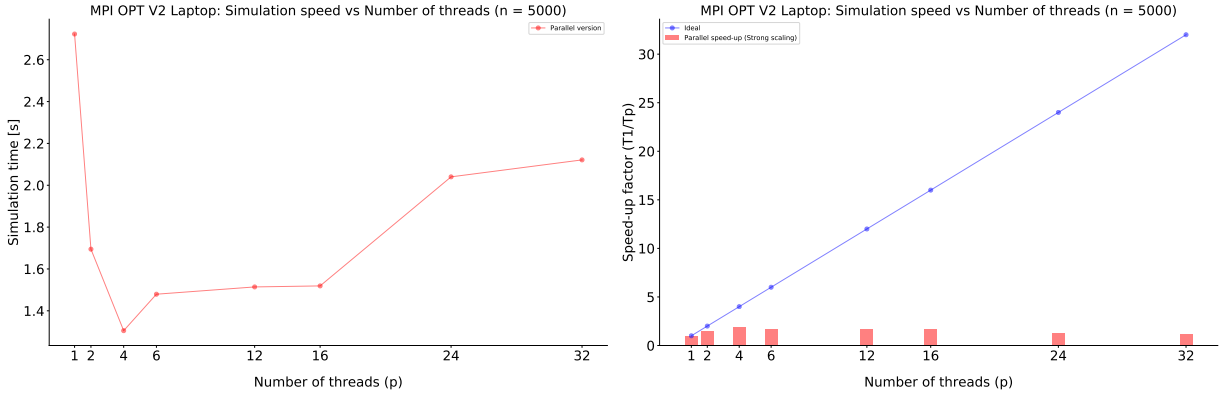
However, when computing the average synchronization time across different values of n and p for $n \in [5000, 1600000]$ and $p \leq 32$, we obtain an average value around 13%, pretty good for our first experience optimizing MPI code.

4.2.4 Machines comparison

Testing the code on a daily use laptop which has only 2 cores with a total of 4 threads (without a distributed memory architecture) gave us a very poor performance in comparison to the results obtained by Cori. From the plots in Figures 18 and 19 we can see that at a certain point, adding more threads is not useful: in fact, the communication overhead is negatively impacting the performance of the algorithm.

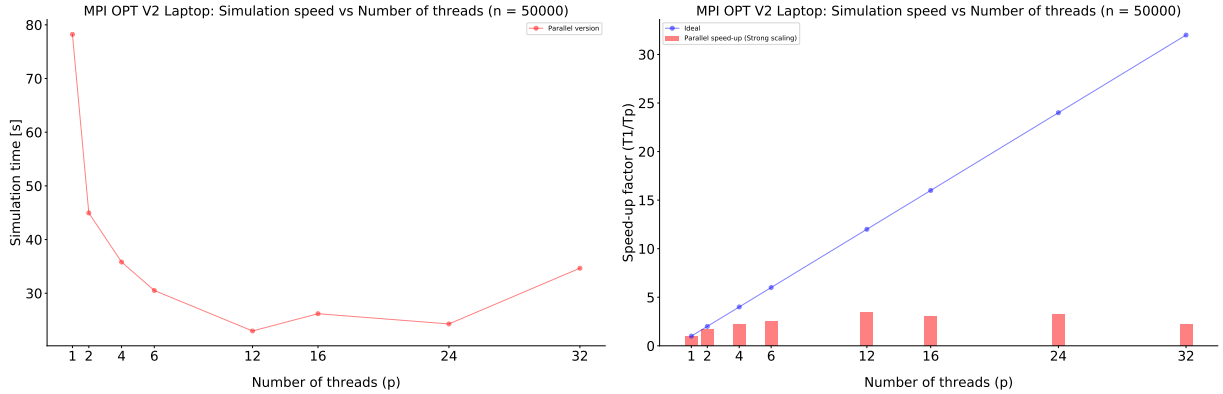
This pattern was expected since our laptop's hardware is adapted to up to 4 threads as well as not being a distributed memory machine and thus, our MPI implementation does not seem as the most suitable one for this hardware. In addition, we can see the large gap between the optimal speed-up factors and the current ones obtained for our laptop computer, telling us that it is not able to exploit the parallelism of the code.

Clearly, our parallel code reached a linear complexity based on the underlying grid data structure. On the other hand, strong and weak scaling values are very low, usually inside the $[0.2, 0.3]$ interval depending on the size of the instances n tested.



(a) Simulation time evolution (Laptop). (b) Speed-up plot (Laptop).

Figure 18: Optimized MPI multi-threaded speed-up factor versus number of threads for $n = 5000$, tested on Laptop.



(a) Simulation time evolution plot (Laptop). (b) Speed-up plot (Laptop).

Figure 19: Optimized MPI multi-threaded speed-up factor versus number of threads for $n = 50000$, tested on Laptop.

Therefore, our MPI optimized code does not present a similar performance to the one observed in Cori when testing it on our laptop: the shared memory approach using OpenMP directions arises as the most suitable one (based on our experiments) for our daily use laptop, as we expected from the discussion performed in the previous part of this project.

5 Conclusions & Future Work

1. $O(n)$ complexity codes for the single and multi-threaded versions of the two-dimensional particle simulator were successfully implemented based on a different and more efficient underlying data structure than the vanilla version.
2. The main strategy applied for improving the Vanilla implementation consists of noticing that the original problem can be modeled in a different way without impacting its expected behavior. Eliminating and replacing extra computations like checking all the interactions between each pair of particles inside the simulation region, by the notion of particle's neighborhood - for interacting purposes - improves the performance while not losing the underlying logic of the simulator.
3. A shared-memory approach using MPI's one-sided routines has been tested. Although its performance was not as expected (no improvement with respect to the Naive version) based on the results obtained using the OpenMP version of the first part of this project, the group's members were able to learn new commands and potentially useful applications in MPI - like creating different communication groups with different shared memory objects - as well as understanding the limitations and difficulties of replicating a shared-memory environment when using MPI.
4. The main strategy for improving MPI is minimizing the amount of message passing between each thread so that the time spent on computation is maximized. We did this by dividing our particle space into columns, and communicating only the necessary neighbors to fill the halo zones of each node. We further improved upon this by only communicating the necessary columns (skinnier than the whole neighbor). For further work, the AllGatherV done at the end can be split into buffers so that each particle is only sent to its corresponding node. We didn't find this optimization to be useful at the scale we did, but for larger simulations with more nodes, this would be useful.
5. Weak and strong scaling factors are significantly impacted (boosted) from their original values around 0.37%-0.3% up to 0.92%-0.88% thanks to a series of different optimization techniques such as: coarse and fine-grained locking, re-formulation of the grid class and updating rules, job partitioning rules, use of efficient MPI functions, etc. Following these techniques, our final MPI implementation obtains better performance than our previous OpenMP shared memory code in both strong and weak scaling factors, as well as simulation times achieved.
6. Different and potentially better data structures can be tested and implemented in order to reach better performance in both the serial and parallel versions: linked lists and/or static arrays would be useful options for developing an alternative approach.
7. A GPU programming view of the problem, could be useful for obtaining better performance due to the structure of the problem. This approach will be tested in the next part of the project, using CUDA as the main tools for programming these extensions.
8. An alternative to OpenMP and MPI is the Pthread API (shared memory like) that allows more control of the parallel execution in comparison to OpenMP while adding more complexity to the code.

6 References

1. Lecture notes and slides from course CS267, Computer Science Department, University of California Berkeley.
2. C++ benchmark – std::vector VS std::list VS std::deque, https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html
3. University of Texas MPI tutorial, http://pages.tacc.utexas.edu/ eijkhout/pcse/html/mpi-onesided.html