University of California Berkeley

Computer Science Department

CS 267

Assignment 2.1: Serial and Parallel Particle Simulation

Submitted by
Xiang Li
Cristobal Pais
Alexander Wu
CS 267
February 16th
Spring 2018

Contents

Assignment 2.1: Serial and Parallel Particle Simulation

X.Li, C.Pais, A.Wu

February 16th 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 OpenMP directions. 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 multi-threaded OpenMP approach reaches factors of 0.79 and 0.68 for strong and weak scaling respectively. 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 powerful OpenMP library.

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 serial and multi-threaded versions of the particle simulation system by using optimal data structures, eliminating unnecessary steps/computations, and parallelization tricks reaching a linear complexity performance $O(n)$ for the serial version which is the best possible asymptotic complexity for this known problem. 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.

In order to obtain a linear performance, two C programs containing a series of optimization techniques will be developed by the team (serial and OpenMP versions), comparing it with the Vanilla particle simulator provided with the statement of the assignment. Concepts such as efficient data structures as well as multi-thread scaling will be studied, described, and applied in order to obtain a well tested and optimized code.

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 serial version of the code is optimized and the data structure parallelization is studied, a simple and straightforward use of OpenMP directives will be able to obtain a significant performance boost as well as good scaling factors.

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 in 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.

Initially, we analyze the "AS-IS" performance of the simulator original files provided with the statement of the assignment. Once the initial results are obtained, bottlenecks of these implementations are analyzed in order to focus our optimizations on the critical elements of the vanilla algorithm. Different data structures will be tested in order to achieve the $O(n)$ linear running time.

Several experimental instances differing in the n value are tested, with $n \in [500, 200000]$ 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:

- **Xiang Li**
  - Offered strategies and implementation choices for the serial version
  - Contributed to writing the report
- **Cristobal Pais**
  - Initial writer of $O(n)$ serial version

  - Wrote library of grid data structures
  - Editor and main writer of the current report.

- **Alexander Wu**
  - Wrote and optimized OpenMP parallel version
  - Cleaned up data structures for optimal parallel performance
  - Changed grid updating strategy

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^{TM} 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 -openmp
2. **Personal computer: Laptop**
   - Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz
     - 4 cores, 8 threads
     - Supports SSE, SSE2, SSE3, SSE4, AVX, AVX2 (among others).
   - Memory
     - 256 KB Level 1 cache
     - 256 KB Level 2 cache
     - 40 MB Level 3 cache (shared per socket)
   - Software
     - Ubuntu 16.04.2 LTS
     - gcc version 4.8.5

  – Default Compiler flags: -O3 -openmp

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, a serial version of the code with an asymptotic complexity of $O(n)$ is developed exploiting 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.

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 decided to 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 will see in section 4.

3.2 Multi-threaded (OpenMP)

Following the successful serial implementation, we use it as the basic structure for our multi-threaded code, exploiting the fact that the structure of the simulator allows us to follow a simple but efficient embarrassingly parallel approach. The analysis is the following:

3.2.1 Return to Naive Grid Updating

The substructure of our parallel code was pretty simple: we would start off by doing a single pass through of $apply\_force()$, which is embarrassingly parallel because all the reads are on locations, and writes to velocity and acceleration. The second pass through is also embarrassingly parallel, because the reads and writes are both within a single particle. However, updating our grid data structure would turn out to be racy because of concurrent writes to vectors.

Our first attempt was to revert to an old strategy: use a single thread to clear and repopulate the grid at each time step. This, in turn, removed our data races, but at a scaling cost. Hence, we then focused our efforts on trying to improve this counter effect.

3.2.2 Coarse and Fine Grained Locking

Since a lot of our parallel code was actually serialized through the single-threaded grid clear and update, we got poor performance when scaling up (both factors around 0.39). Therefore, we decided to to return to lazily updating the grid, but taking a global lock on any update to the vectors. Thanks to this simple modification, we were able to increase the performance factors up to values around 0.49 when we did so.

Upon further analysis, we realized that multiple writes to the grid would be acceptable, as long as they weren't multiple writes to the same grid entry. We implemented finer grained locking by assigning a lock to every "region" of the grid. We kept the number of lock regions proportional to the number of threads to limit the overhead of lock initialization but kept the constant factor (20 locks per thread) high to try and avoid lock contention whenever possible. Using this fine-grained locking strategy, our scaling shot up (0.64 for weak scaling and 0.79 for strong scaling).

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 Serial version

From the results obtained, we can see in Figure 1 that our implementation is able to obtain an $O(n)$ complexity, also when dealing with large values of n. These results are obtained using the already mentioned grid structure alongside **without** its optimized updating rule as described in sections 3.1.1 and 3.1.2. In this case, $n \in \{500, 1000, 2000, 3000, 4000, 5000, 10000, 15000, 20000, 25000, 50000, 100000, 125000, 150000, 200000, 225000, 250000, 500000\}$.

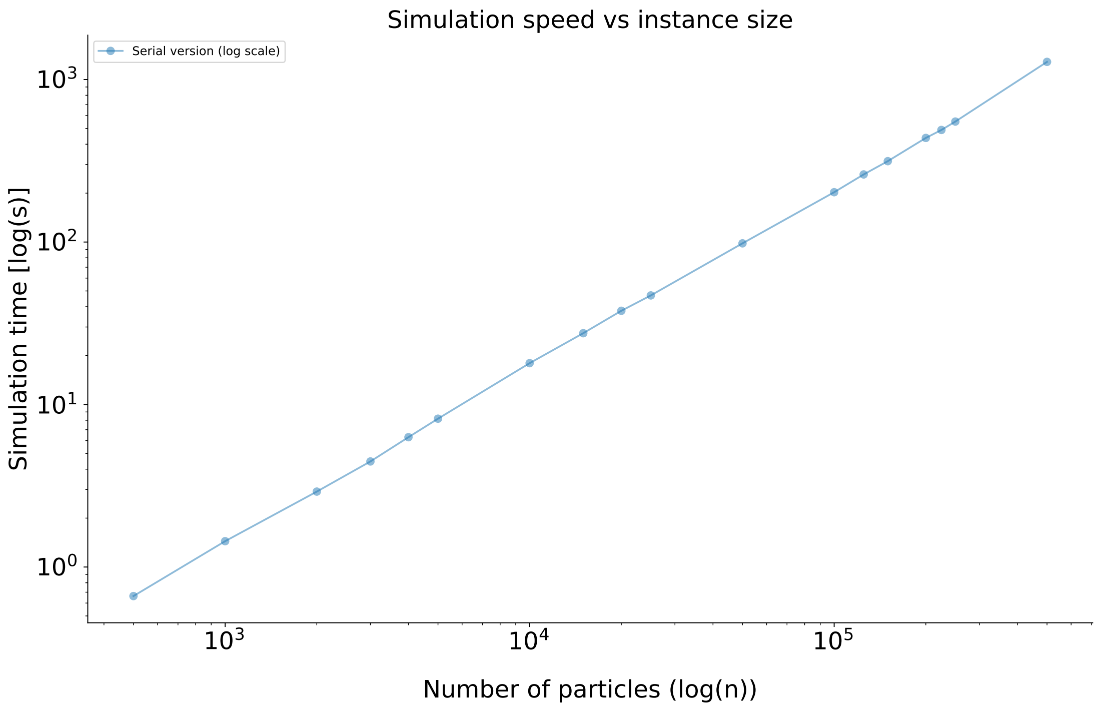

Figure 1: Serial implementation log-log scale plot for instances $n \in [500, 500000]$.

When the updating rule described in section 3.1.2. is included, far better solving times - around 3x - are obtained while clearly keeping the asymptotic complexity of $O(n)$. For completeness, we include the detail of some of the experiments obtained in Cori. From the results in Table 1, we can see that the estimated slope is very close to 1 (1.07) for the preliminary version of our code and slightly above 1 (1.13) for the final serial version, corroborating the visual intuition. In addition, in Figure 2 we include a sample of the results obtained in Cori by the vanilla implementation provided with the assignment: we can clearly identify a second order polynomial complexity (instances with $n > 25000$ are not included in order to not collapse our NERSC's quota with these experiments).

In comparison, we have that the Vanilla serial implementation provided with the assignment is obtaining an estimated slope, as expected, around 2 when using it with the provided *autograder* utility. As mentioned in section 3, this performance is explained by the fact that the logic behind the Vanilla implementation is checking the interaction between each pair of particles that leads to the $O(n^2)$ asymptotic complexity, incurring into an intense computational effort.

Hence, the data structure based on dynamic vectors simulating a grid/matrix that contains the particles inside each of its cells, checking the interactions within the particles' spatial neighborhood (adjacent cells) is able to improve the original $O(n^2)$ implementation where each particle interaction was tested no matter how far or close the other particle was.

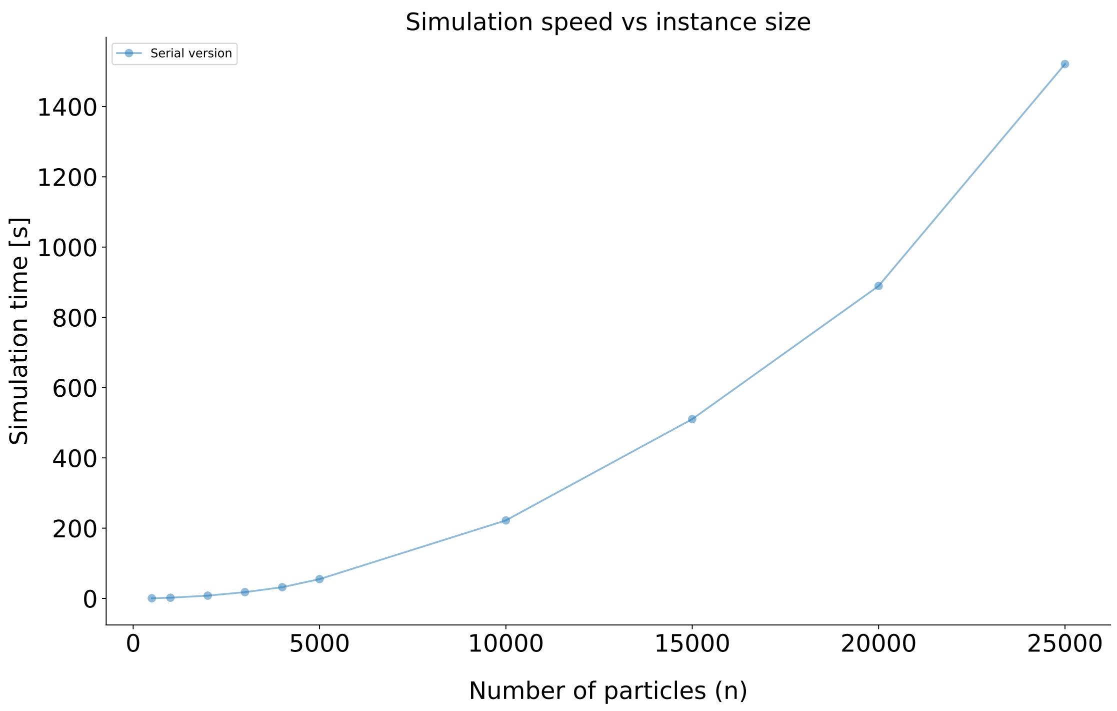

Figure 2: Vanilla serial implementation sample results: a second order trend is already visible among small instances.

Table 1: Summary results from optimized serial implementation $n \in [500, 500000]$

Size $[n]$	**Time preliminary** [s]	**Time final version** [s]
500	0.459	0.109
1000	0.918	0.232
2000	1.900	0.497
3000	2.946	0.842
4000	4.047	1.279
5000	5.161	1.545
10000	10.888	3.622
15000	16.569	5.266
20000	22.362	7.685
25000	28.110	9.436
50000	58.169	20.554
100000	119.722	41.236
125000	153.117	54.758
150000	183.293	67.827
200000	246.894	92.731
225000	295.802	104.014
250000	328.120	123.786
500000	974.418	340.213
Estimated Slope	1.073064	1.133384

4.2 Parallel version: Performance & Processors

Based on the single-threaded version of the previous section, an OpenMP multi-threaded version is developed, including the classic pragma instructions for easily parallelizing the for loops (embarrassingly parallel approach). Two main performance matrix are analyzed: (1) complexity $O(n)$ and (2) weak and strong scaling factors.

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 3, 4, and 5 we can clearly see that our OpenMP implementation is able to reach the desired $O(n)$ complexity. Looking at the slope of both 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 for both cases the estimated asymptotic complexity (slope) is very close to 1 (Table 2). Furthermore, in the case of 16 and 32 threads, we can see that the estimated slope value is below the unit, allowing us to conclude that the OpenMP implementation is satisfying the complexity $O(n)$ requirement.

Table 2: Summary results from parallel implementation using 4, 16, and 32 threads with $n \in [500, 500000]$

Size $[n]$	**Time 4C** [s]	**Time 16C** [s]	**Time 32C** [s]
500	0.189	0.086	0.097
1000	0.296	0.139	0.119
2000	0.609	0.237	0.155
3000	0.960	0.359	0.258
4000	1.291	0.420	0.323
5000	1.643	0.529	0.383
10000	3.483	1.067	0.698
20000	6.992	2.073	1.386
25000	8.763	2.617	1.709
50000	18.012	5.295	3.508
100000	37.178	10.939	7.200
125000	46.995	13.857	9.134
150000	56.708	16.745	10.962
200000	77.157	22.219	14.906
225000	88.162	25.219	17.003
250000	97.676	28.203	19.061
500000	269.298	73.499	47.586
Estimated slope	1.045478	0.978134	0.932867

Therefore, we are able to see the benefits of our serial implementation's data structure in the parallel version, without needing to reformulate its basic components/methods or include a significant amount of new code: just adding the correct and pertinent OpenMP directions is enough for obtaining a well-behaved parallel approach.

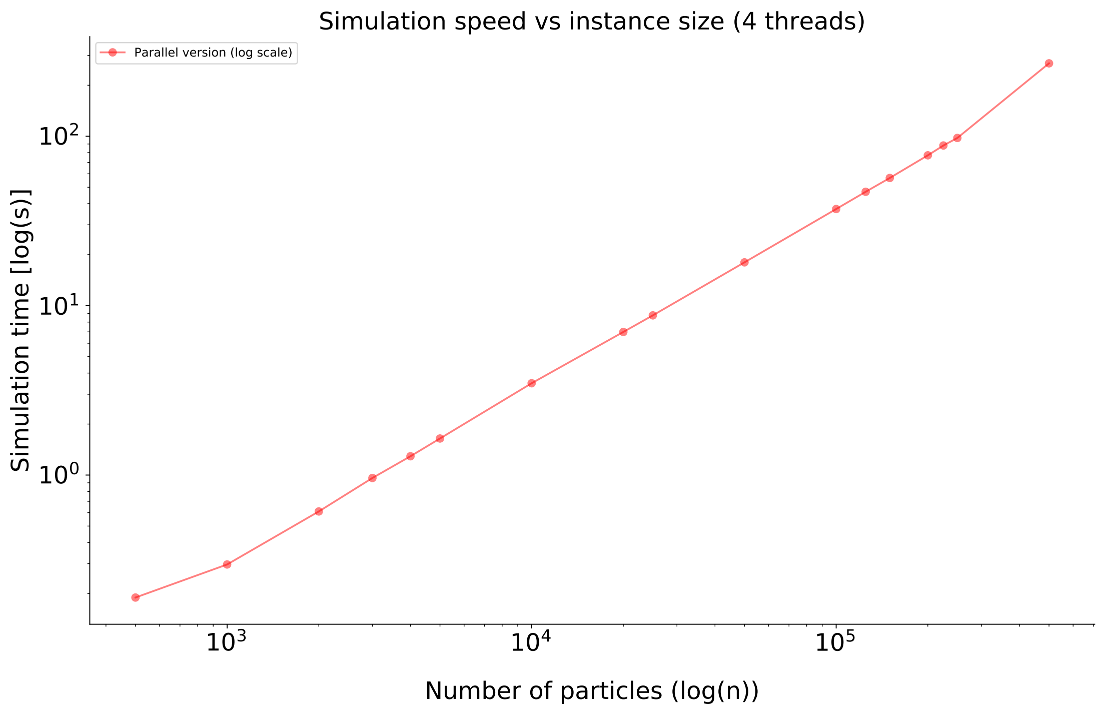

Figure 3: OpenMP multi-threaded (4 threads) implementation log-log scale plot for instances $n \in [500, 500000]$.

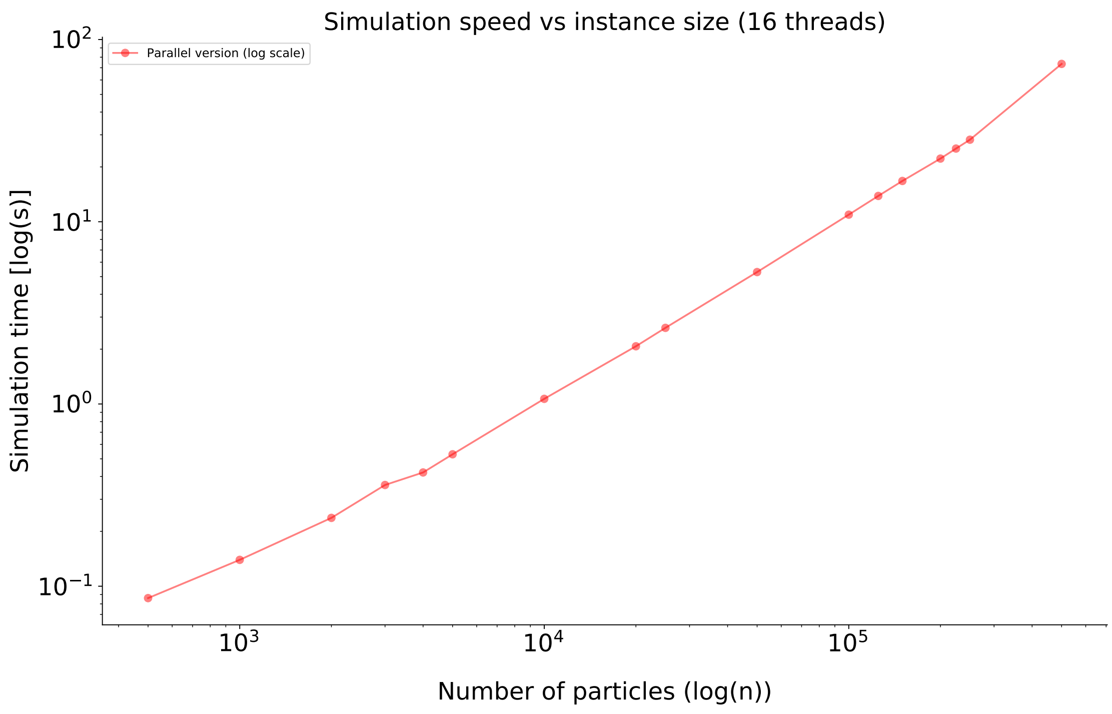

Figure 4: OpenMP multi-threaded (16 threads) implementation log-log scale plot for instances $n \in [500, 500000]$.

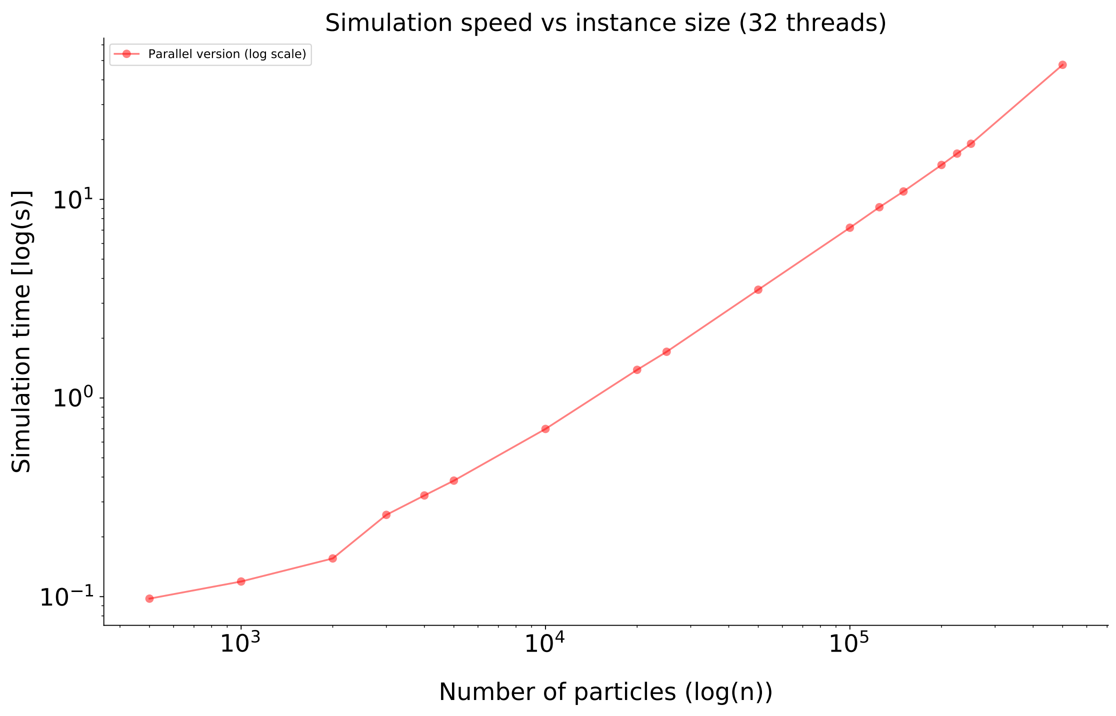

Figure 5: OpenMP multi-threaded (32 threads) implementation log-log scale plot for instances $n \in [500, 500000]$.

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 the Vanilla code provided, the OpenMP implementation obtains very poor results for both scaling factors: 0.3 of strong scalability and 0.18 of weak scalability. Therefore, the naive parallel implementation is not able to obtain a significant performance boost when adding more threads and thus, it is not really useful for large-scale problems due to its lack of ability to take advantage of the underlying hardware/resources available.

With our final parallel implementation, we are able to obtain significantly better results. As can be seen for an instance with $n = 50000$ in Figure 6, we can see how the performance of the simulator is boosted 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 decreasing performance factor, allowing us to expect that the efficiency is around $60\% - 70\%$ due to the bad scaling after including more than 12 threads.

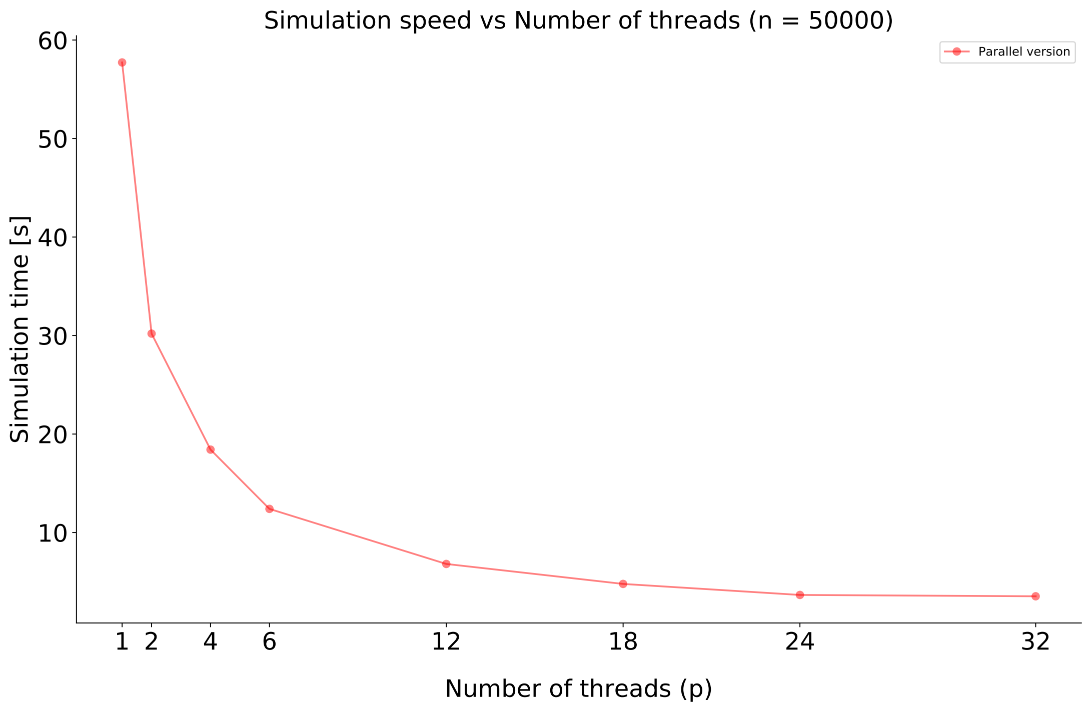

Figure 6: OpenMP multi-threaded performance increase with respect to number of threads ($n = 50000$).

When comparing the attained performance with respect to the ideal benchmark, we can see in Figure 7 how our implementation is able to reach results very similar to the optimal performance up to 12 threads, with an efficiency above 85%. After this threshold, it is clear that adding more threads is not impacting significantly the performance of the code, and also, it is becoming useless at some point: the difference in performance between the 24 and 32 threads version is negligible. One explanation behind this pattern consists of the amount of time spent in communication and synchronization operations, as well as inherent problems to our code's structure: alternative data structures could be better and more efficient for the current problem.

For comparison purposes, we include the speedup plot obtained using the Vanilla OpenMP implementation in Figure 8 as well as the weak and strong efficiency plots in Figures 10 and 9 that allow us to understand the performance pattern of our implementation. As expected from our previous analysis, results are very poor in terms of weak (and strong) scaling, allowing us to notice the great difference in performance between the original code and our final version.

From the efficiency plots, it can be concluded that our code presents a good strong scaling performance

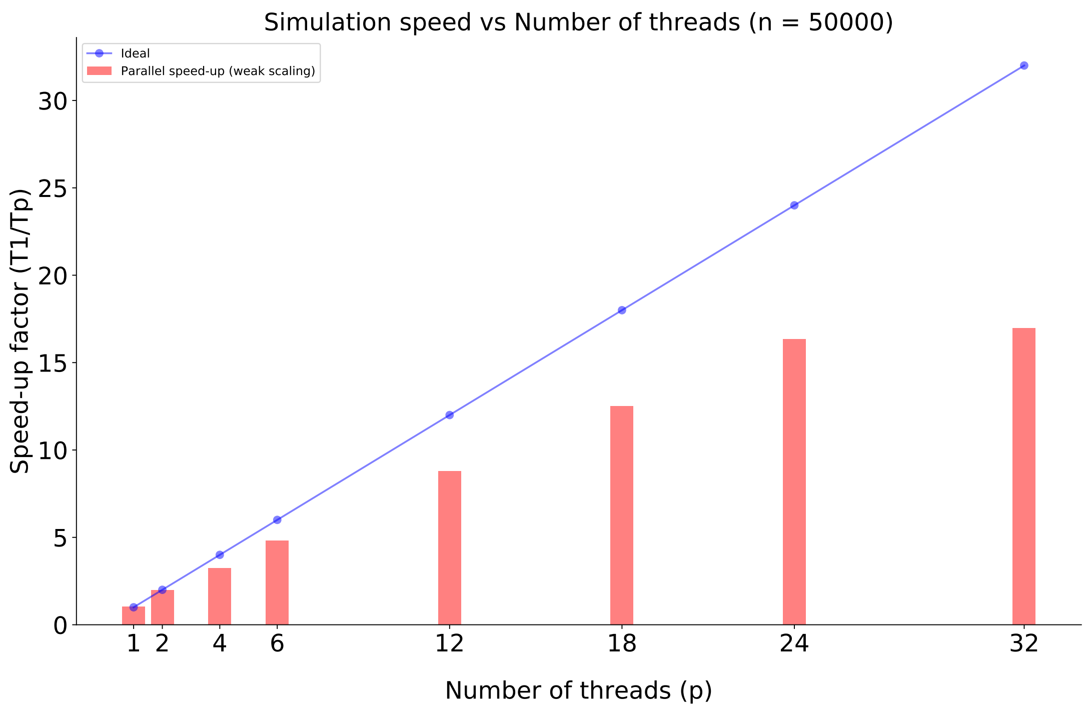

Figure 7: Optimized OpenMP multi-threaded speed-up factor versus number of threads for $n = 50000$.

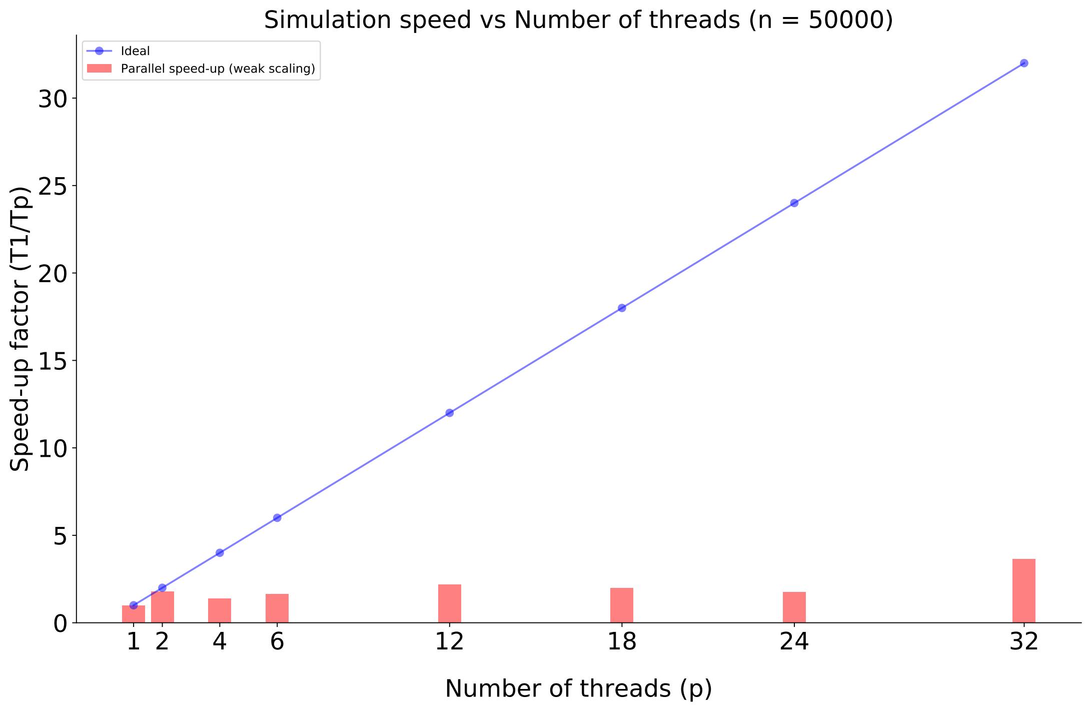

Figure 8: Vanilla OpenMP multi-threaded speed-up factor versus number of threads for $n = 50000$.

(more than 0.7) up to 24 cores, 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 OpenMP implementation. On the other hand, the weak efficiency performance is worse than the strong one, only obtaining values above 0.5 when using no more than 6 cores/threads. After that threshold, a significant decrease in the performance is observed: the average weak efficiency is around 40%.

Therefore, based on the results we can conclude that our final implementation has 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).

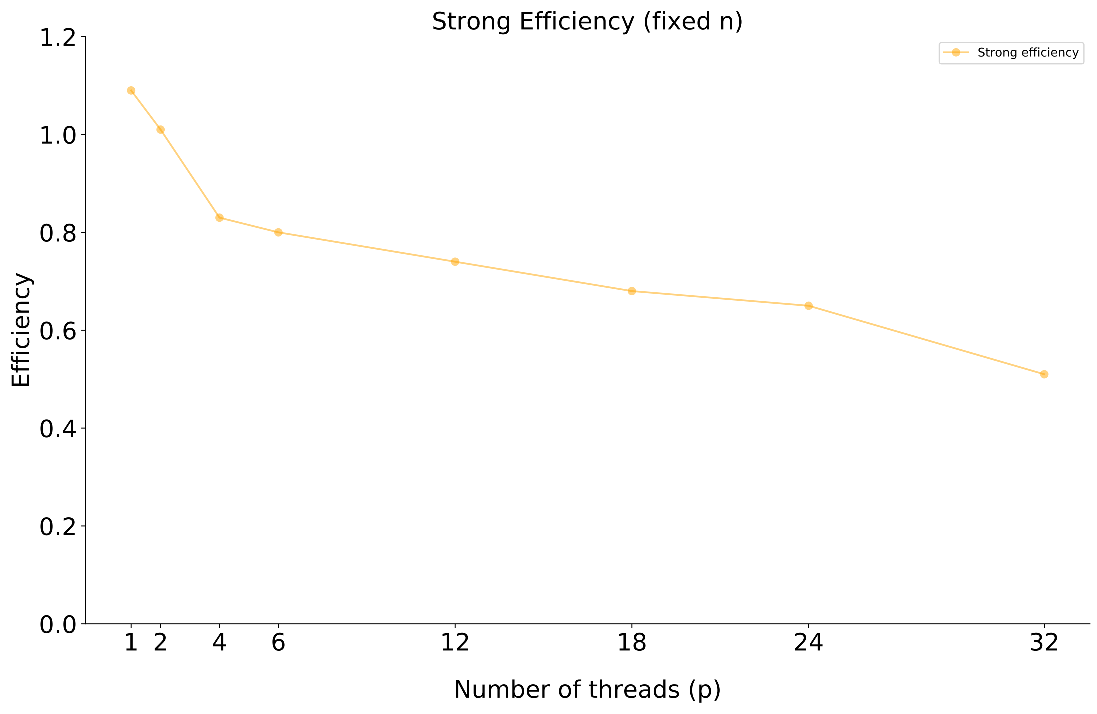

Figure 9: Strong efficiency values: different number of threads and fixed $n = 50000$.

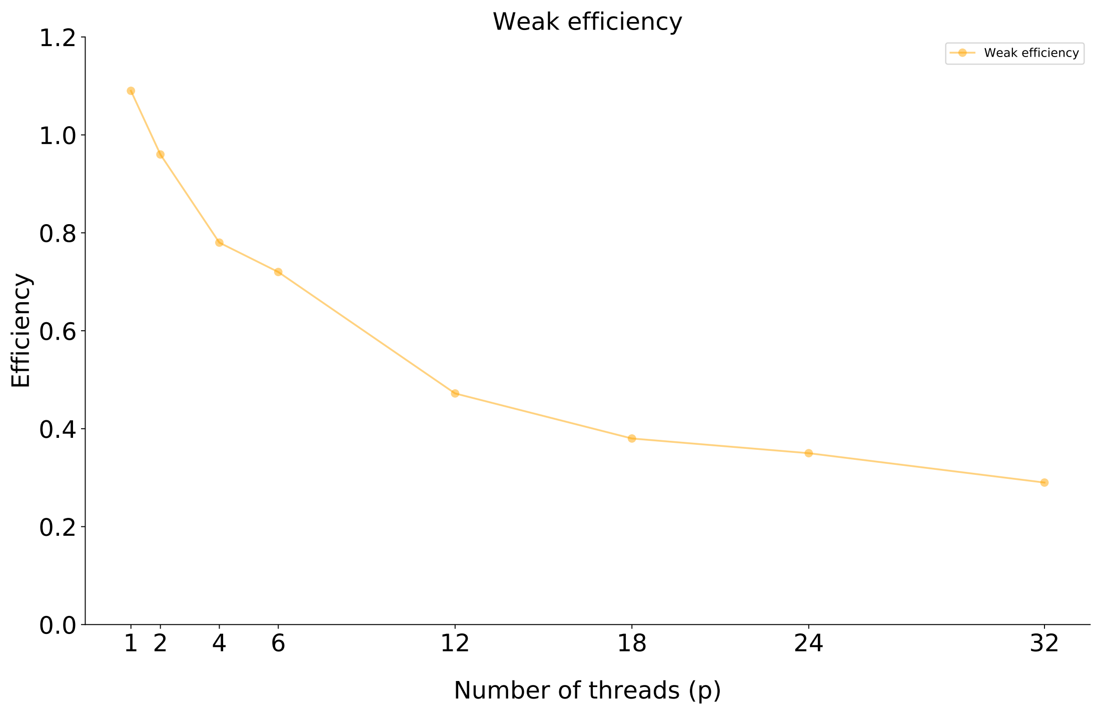

Figure 10: Weak efficiency values: different number of threads and proportional n values.

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.

Sections 3.2.1 and 3.2.2 give a brief overview of the synchronization and communication tradeoffs that we made. The numbers in the below plot (Figure 11) represent the latency of moving each of the particles in the grid for the serial version, as well as the "scaled" latency, which we define to be the sum of the latencies for the particle moving across all threads. Each point is an average of a series of runs in order to avoid variance bias. In an ideal world, we expect the parallel number to be the same. The increase in time below we attribute to synchronization overhead.

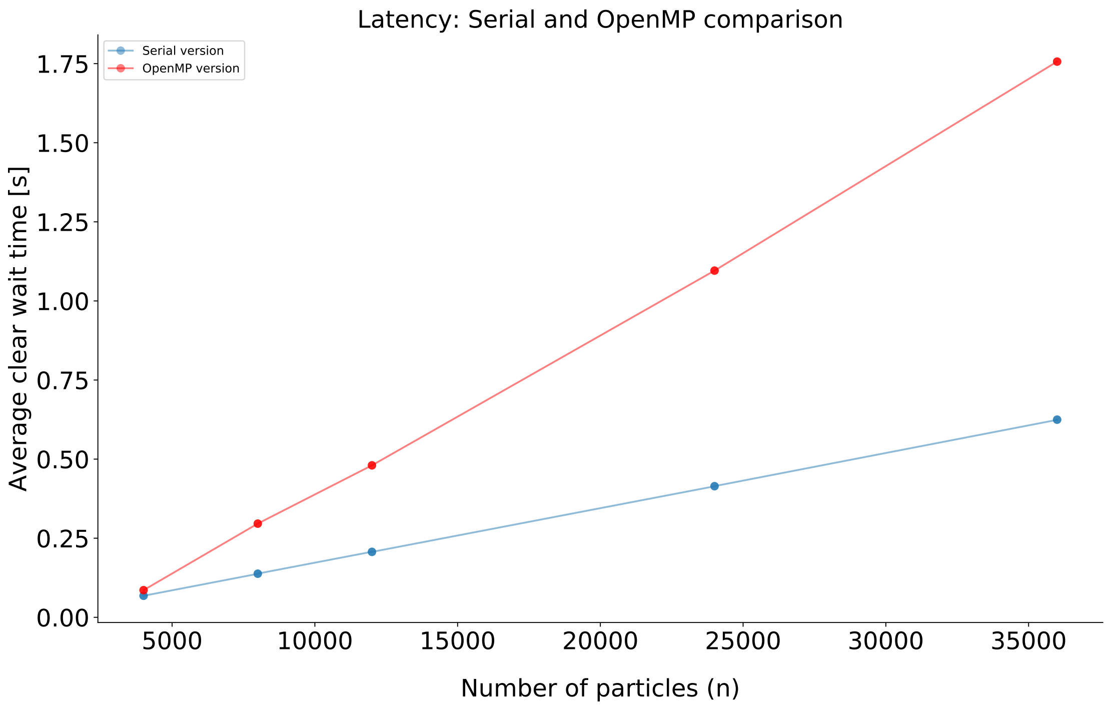

Figure 11: Amount of time spent moving particles and redistributing them across the grid for the serial and parallel versions. The "difference" between curves represents the sync overhead. A clear increasing pattern is identified.

We also measured the latency of initializing 20 locks, but we found it to be consistently below 0.005 seconds, negligible relative to the overall latency.

4.2.4 Machines comparison

Testing the code on a daily use laptop gave us very similar results in comparison to the performance obtained by Cori - keeping in mind that our laptop has only 4 cores. This pattern was expected since we are not exploiting any particular memory hierarchy or hardware architecture for developing our optimized code (in contrast to our previous matrix multiplication project) and thus, the current optimized particle simulation code tends to be more portable.

Clearly, both the serial and parallel code reached a linear complexity. On the other hand, strong and weak scaling values were usually inside the $[0.65, 0.7]$ interval, depending on the values of n tested. Therefore, statistics and plots are very similar to the ones already provided in the previous sections, and thus, are not included for simplicity.

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. Weak and strong scaling factors are significantly impacted (boosted) from their original values around 20%-30% up to 60%-70% thanks to the coarse and fine-grained locking approach implemented in the parallel version after testing the naive grid updating (reset scheme) rule where the scaling performance was significantly affected (decreased).
4. 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.
5. A distributed memory approach, as well as a GPU programming view of the problem, could be useful for obtaining better performance due to the structure of the problem. These two approaches will be tested in the next two parts of the project, using MPI and CUDA as the main tools for programming these extensions.
6. An alternative to OpenMP is the Pthread API 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