Parallelization of Particle Simulation Using Various Environments and Models

ID1217 Concurrent programming, KTH

Ossian Dillner and Christopher Robberts

Software and Computing Platforms

All development and performance evaluation has been performed using our personal laptops and stationary computers. These aforementioned machines are all running Windows and are all equipped with fairly recent models of either Intel i3 or i5 processors. To avoid any kind of discrepancies all final performance evaluations featured in this report were run on the same machine.

The development of the different implementations of the particle simulation have all been performed using C++ in combination with corresponding libraries and compiler directives.

Design and Implementation

The core design concept was using a matrix of particles, represented by an array of vectors containing references to particles. By using the particles X-values to sort them into the corresponding vectors we can guarantee that a particle only needs to be compared to particles in the current or “neighboring” vectors in order to ensure proper particle behavior while also ensuring an O(n) time complexity. An important note is that the aforementioned matrix only contains references or pointers to the original particles, allowing for the preservation of the original data structure that is used for moving particles and saving simulation history for later viewing.

In essence all the implementations are very similar, consisting of three main steps: Sorting the particles, performing collision checks and moving the particles.

The sorting of the particles is done by fetching the particles X-values and comparing them to the number of vectors as well at the size of the simulation area to figure out which vector to contain the particle within. This can unfortunately only be performed serially by a single thread since we've found that with our current configurations the overhead of splitting up and synchronizing this work between different threads is far too high.

The collision checking is performed by assigning a thread a vector for it to go through and compare all particles contained in the given vector with all other particles contained in either the given vector or neighboring vectors. The nature of these collision checks makes the work very easy to parallelize with different methods since there are no data or order of execution dependencies, eliminating the need for most synchronization and making the implementation of different methods of work sharing very simple.

The moving of the particles is done by simply looping through the array where the particles are stored and moving each particle. Just as with the collision checking there are no data dependencies, allowing for easy parallelization.

For the serial and OpenMP implementation we opted to use a class to contain the aforementioned particle matrix and its various members. This was done in order to make the code cleaner and encapsulated, while also providing the possibility of running several simulations simultaneously within the same program.

However, the usage of a class to contain our data structures and functions proved painful to use in the Pthread and MPI implementations and we opted to use a more simpler approach.

When first implementing our design using the Pthread library we tried having two different types of threads, workers and master, using conditional synchronization to synchronize them. Since the sorting has to happen first and by a single thread, the workers we initiated had to wait on a condition to start checking for collisions. This had a huge impact on the performance since every thread had to wait for a certain amount of time in each loop iteration. This resulted in that the execution time would actually be longer for every additional thread. We then changed the implementation so that all computation is contained in the same function, using thread ID's to distribute work. This allowed us to perform all required synchronization with a single barrier function, eliminating unnecessary overhead and achieving proper speedup.

The implementation of our design in OpenMP proved very easy, requiring only a small restructuring of the serial implementation as well as the insertion of a few pragmas to make the simulation run in parallel. All that was required from that point was a little experimenting with scheduling and variable scopes to make the OpenMP version run with proper speedup.

The MPI version of the program proved to be the most difficult one, requiring heavy amount of code restructuring but ending up surprisingly similar to the other implementations. The one big difference being that the master thread or process does not perform any work moving or performing collision checks, it simply sorts particles and handles necessary communications.

Methods of Synchronization and Communication

Pthreads

To assure that certain sections, such as the sorting, is only performed by a single thread we assign a thread as the master thread and only let that thread into these sections, using a barrier function to assure that the other threads wait for the master thread until proceeding.

The barrier function makes use of locks and condition variables, having a global counter that counts the amount of threads that have encountered the barrier. When this global counter reaches the amount of threads in the thread pool (also a global integer value), the last thread signals (broadcasts) to the rest of the threads waiting on the condition variable that it is fine to resume working. The actual parallelization is performed during the collision checking and moving of particles, where each thread has a specific slice of the matrix to operate on.

OpenMP

Almost all synchronization is handled by OpenMP internally, all we had to do was make sure to enclose each section with relevant pragmas, for example ensuring that the sorting section is only performed by a single thread with the “master” pragma and making all the other threads wait with the usage of the barrier pragma.

MPI

Communication is only performed at two points in the program flow, firstly when the master process is finished sorting the particles at which point it distributes the vectors between processes by first sending the vector length followed by the individual particles one by one until it has distributed the matrix evenly between all the processes. When these processes are finished working on their given vectors they then send them back to the master process for further sorting and possible saving.

Proof of O(n) Time Complexity

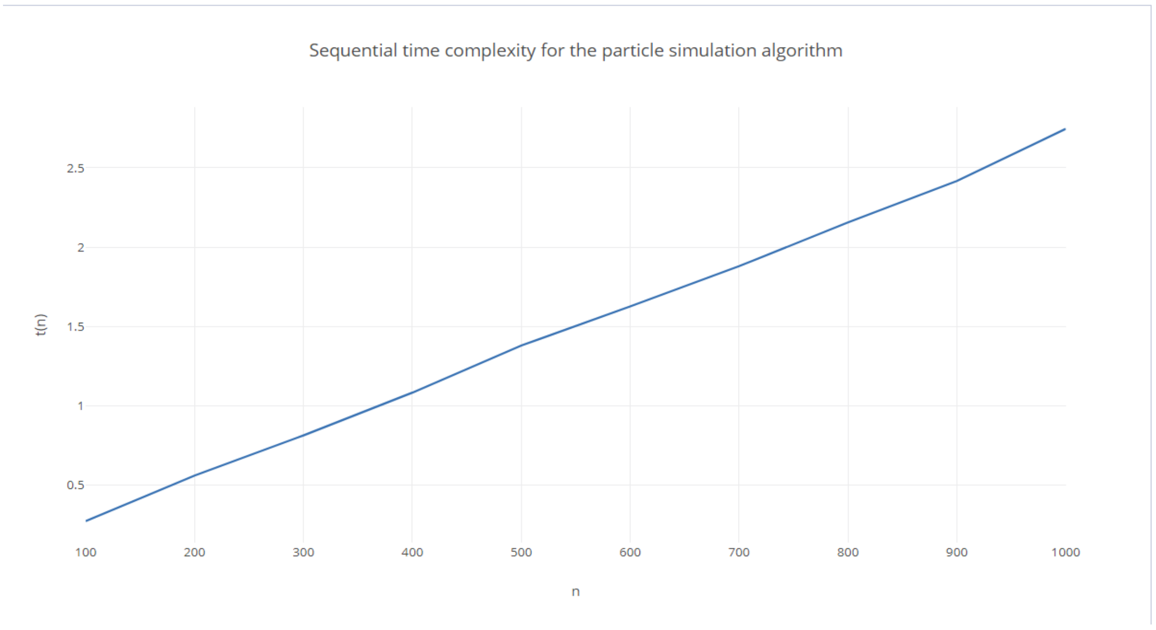

Figure 1: a graph portraying the time growing linearly when work load increased.

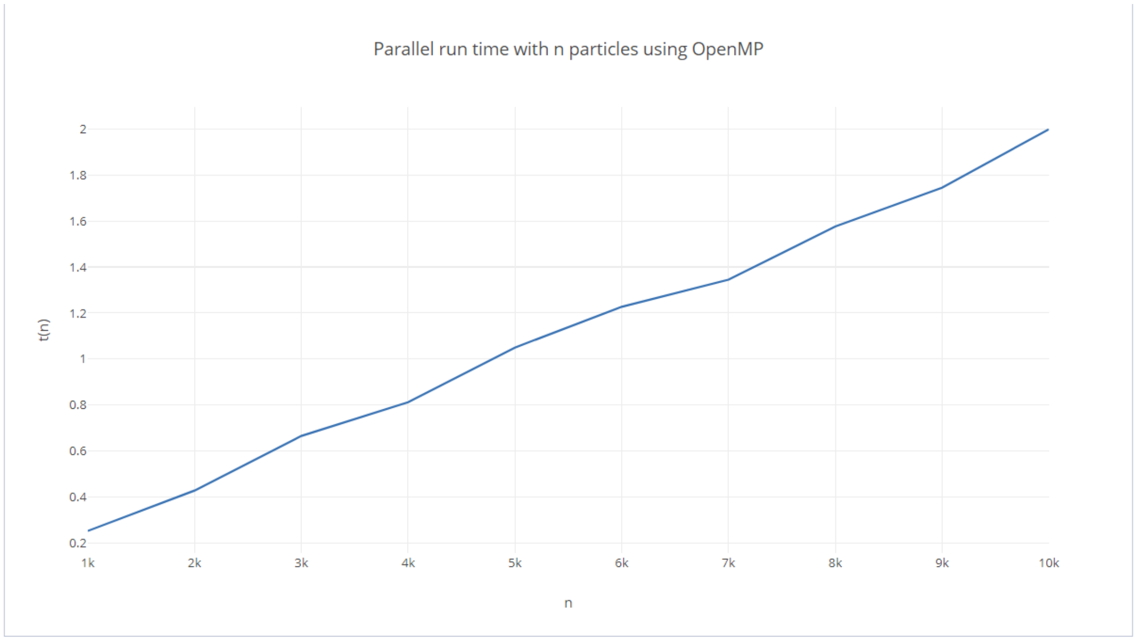

Figure 2: a graph portraying the time growing linearly when work load increased with our OpenMP implementation.

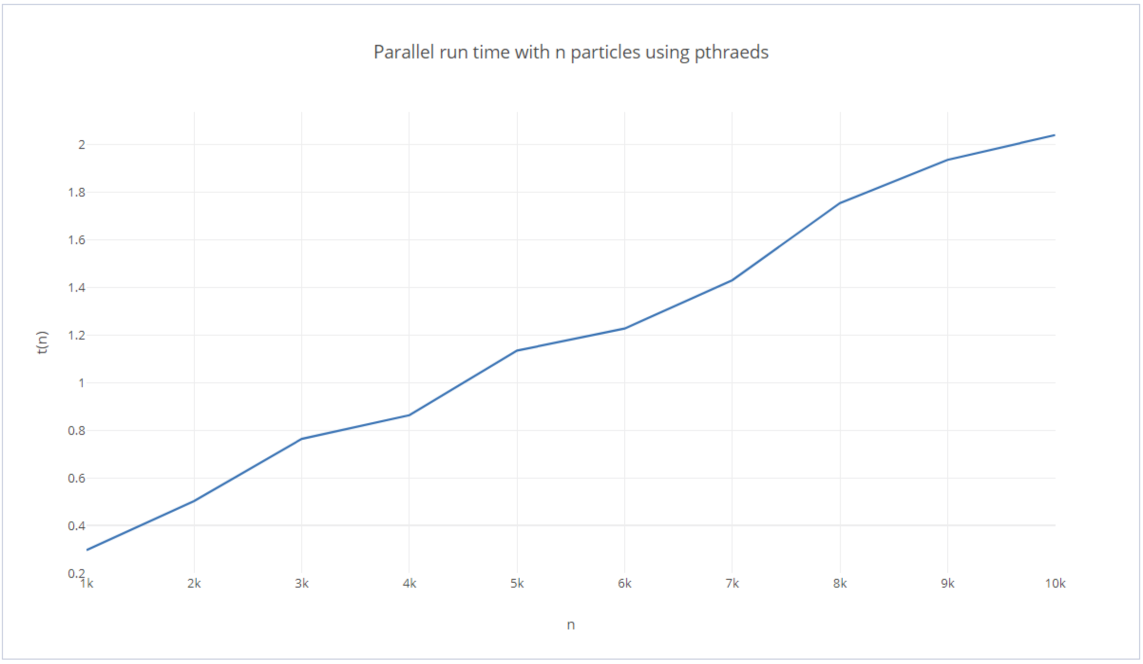

Figure 3: a graph portraying the time growing linearly when work load increased with our Pthread implementation.

Figure 4: a graph portraying the time growing linearly when work load increased with our MPI implementation.

Speedup Measurements

All of these time measurements are the median of five different executions for each amount of processes.

OpenMP

Workload: 10,000 particles.

Sequential time: 6.3891s

Two processes:
Desired: 6.3891/2 = 3.19455s
Result: 4.34237s -> speedup: 6.3891/4.34237 = 1.47133938s

Three processes:
Desired: 6.3891/3 = 2.1297s
Result: 2.3296s -> speedup: 6.3891/2.3296 = 2.74257383s

Four processes:
Desired: 6.3891/4 = 1.597275s
Result: 1.95292s -> speedup: 6.3891/1.95292 = 3.27156258s

Desired time: T/p where p is the number of processors. That would be a little bit more than 3 seconds for two cores where we have around 4 seconds. This goal could probably be reached but we encountered a couple sections that had to run sequentially which made it hard for us to come closer to the desired time. It is possible that we might have missed a section that could be done in parallel.
In conclusion we came really close to the ideal time using parallelization.

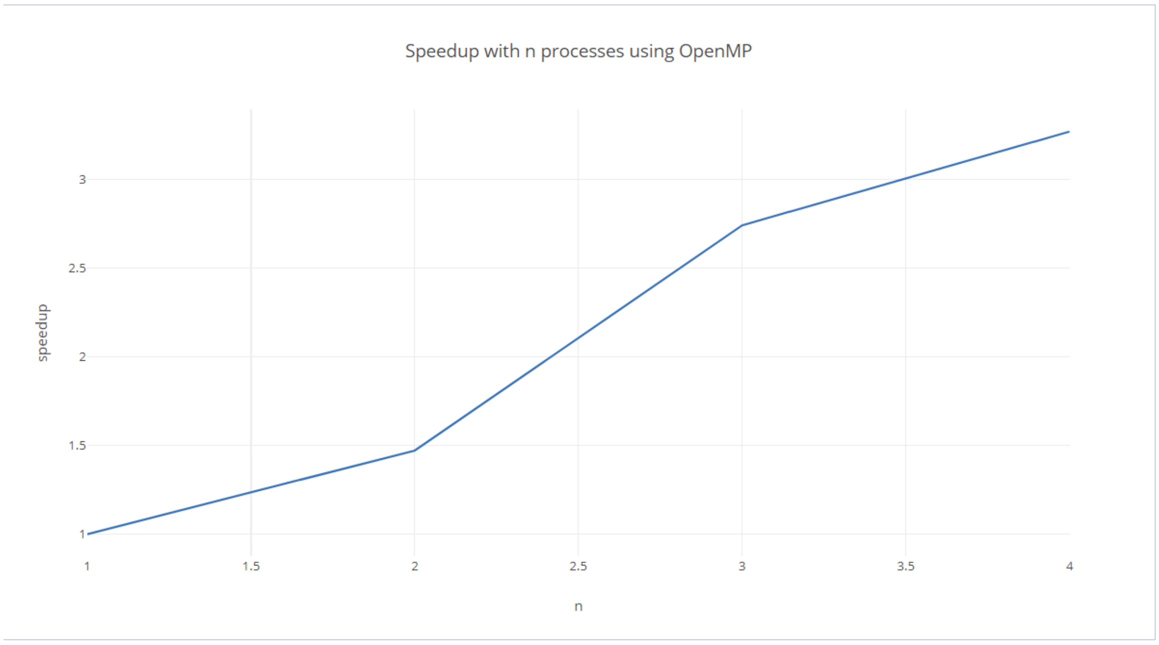

figure 5: Speedup of the program using parallelization with OpenMP

Pthreads

Workload 10,000 particles

Sequential time: 6.3891s

Two processes:
Desired: 6.3891/2 = 3.19455s
Result: 3.78082s -> speedup: 6.3891/3.78082 = 1.68987151s

Three processes:
Desired: 6.3891/3 = 2.1297s
Result: 2.38994s -> speedup: 6.3891/2.38994 = 2.67333071s

Four processes:
Desired: 6.3891/4 = 1.597275s
Result: 1.95144s -> speedup: 6.3891/1.95144 = 3.27404378s

The same argument for the OpenMP implementation applies here. Since we have a couple of sections that can't be parallelized in our implementation, the optimization is probably a second more or less from ideal.

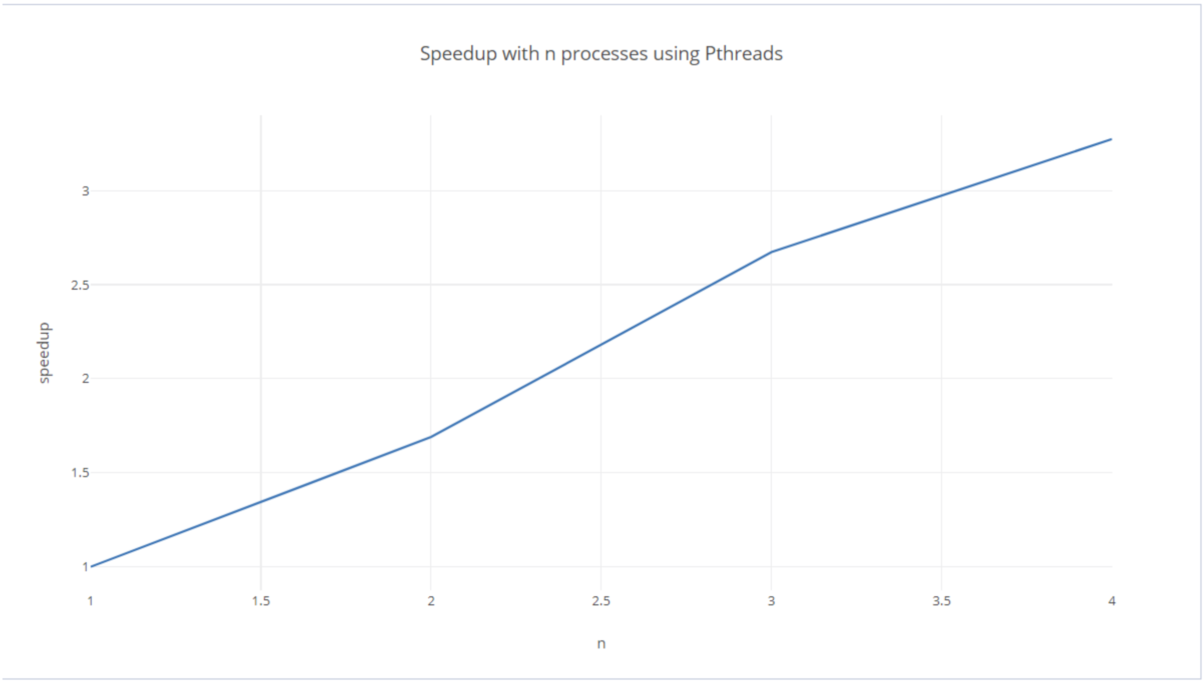

Figure 6: Speedup of the program using parallelization with Pthreads.

MPI

Workload 1000 particles

“Sequential time”: 10.4936s

<u>Two processes:</u>
Desired: 10.4936/2 = 5.2s
Result: 4.87728s -> speedup: 10.4936/4.87728 = 2.15152708s

<u>Three processes:</u>
Desired: 10.4936/3 = 2.1297s
Result: 2.69115s -> speedup: 10.4936/2.69115 = 3.89929956s

<u>Four processes:</u>
Desired: 10.4936/4 = 2.6234s
Result: 1.97011s -> speedup: 10.4936/1.97011 = 5.32640309s

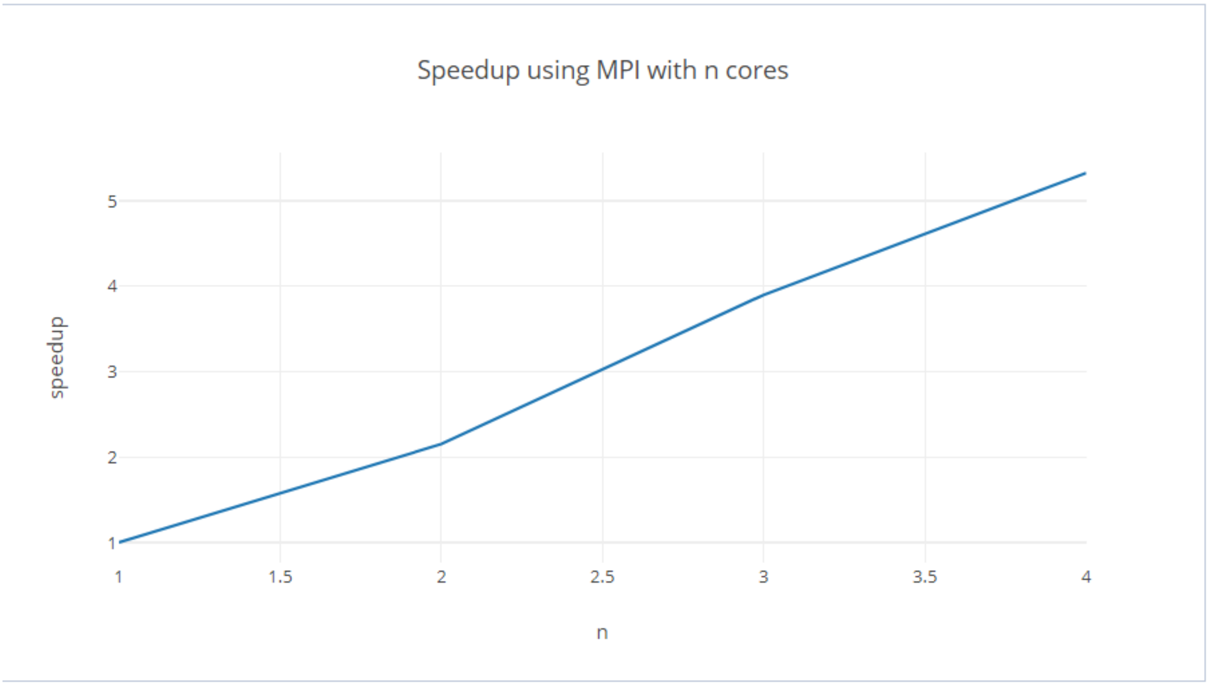

Figure 7: Speedup of the program using parallelization with MPI.

Where does the time go?

When breaking down the runtime into computation time we can see that most time goes in to the collision checking. We can see this from a printout of for example the OpenMP implementation using a workload of 10,000 particles:

number of particles = 10000, simulation time = 1.95292 seconds

indexation time: 0.284999, collision checking time: 1.554001, move time: 0.113920

This is logical since the most amount of work goes in to a particle checking for collisions and this being done for every single particle. Time also gets invested in synchronization time, this was clearly portraited in our first Pthread implementation when we had two functions synchronizing with each other from two different functions in two different loops using condition variables.
The waiting time contributed tremendously and even more so for every additional thread, since that meant yet another thread trying to claim the lock and getting suspended. This is due to each thread having contention for the locks and doing so every loop iteration which effectively makes the program run sequentially.

We still have synchronization methods using condition variables in the barrier function to make sure to wait for the thread taking care of the indexation and the saving before resuming. The latter implementation is faster since we are only waiting for one thread to finish and the thread performing the sequential work does not have to wait for the collision checking to be done, instead it proceeds to help with the collision checking.

Discussion on using Pthreads, OpenMP and MPI

The exercise of implementing the same program using these different frameworks has proven very beneficial in many aspects, especially when it comes to our understanding of when and how to use these different frameworks, libraries and compiler options. This could of course be a report in itself but to briefly summarize what we've learned we can state that OpenMP is a more understandable framework that can be easily explained. The upside of using Pthreads is that it lets you control everything at a much lower level and you gain more control over your program in a shared memory multicore processor and if you are working within a distributed environment with message passing the MPI framework is very effective.