CS3210 Assignment 1 Report

Program Design

Wee first created a sequential C++ program. We then applied OpenMP constructs with the intention of reducing the execution times of the most computationally expensive parts of our sequential program.

To simulate the movement of the particles in each time step, our sequential program executes the following tasks:

1. **Detect Possible Collisions**: For each pair of particles, check whether they collide with each other. Additionally, check whether each of the N particles collide with the walls of the square. All identified collisions are stored in a dynamic array.
2. **Sort + Filter Collisions**: Sort the collisions identified in the previous task by collision time and the indices of the particles involved. Then, remove the collisions that are invalid according to this assignment's simplifying assumptions. For example, particles A and B cannot collide if particles A and C collided half a time step earlier.
3. **Resolve Collisions**: Iterate through all the particles and compute their resultant positions and velocities for this time step, by taking their collisions into account (if any).
4. **Update Particles**: Update all particles with their new positions and velocities (specific to our implementation).
5. **Print movement**: Print the updated positions and velocities of the particles, if required.

These tasks are illustrated in the task dependency graph in Figure 1.

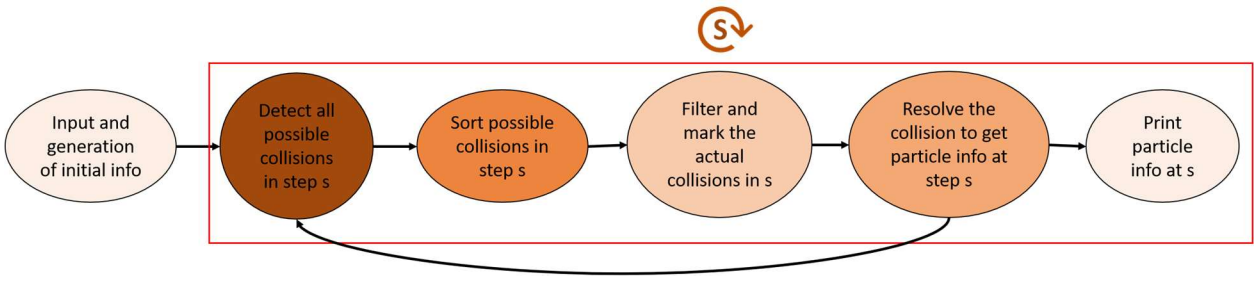

Figure 1. Task dependency graph. Darker shades correspond to higher execution time (estimates obtained via experimentation).

Parallelisation Strategy

The 1st, 3rd and 4th tasks- "Collision Detection", "Collision Resolution" and "Update Particles" - are suitable candidates for loop parallelism. Each of these tasks use for loops to perform the same

operations on independent pieces of data. Hence, we can implement data parallelism. In contrast, task parallelism cannot be used as the tasks are not independent (see Figure 1).

Specifically, in the Collision Detection task (see Figure 4), identifying collisions possible for particle A is independent from the task of identifying collisions possible for particle B. Hence, the collision detection for each particle can be done in parallel.
Similarly, in the Collision Resolution task, the results of a particular collision do not affect the results of other collisions. Hence, collisions can be processed in parallel. The “Update Particles” task can be similarly parallelised.

We also attempted parallelizing the Sorting task, but the three parallel sorting algorithms that we considered did not reduce the execution time of the Sort and Filter task. Hence, these algorithms were not adopted in our final implementation.. Besides, the C++ STL `sort()` implementation is already highly optimised.

We focussed on **parallelising the Collision Detection task**, because this task has a quadratic time complexity with respect to the number of particles, which hinders the program from scaling efficiently with an increasing number of particles. In addition, we found that it accounts for > 90% of the overall execution time of our sequential implementation.

These observations led us to a parallelise the program with 3 different implementations for the collision detection task.

1. Parallel 1 (benchmark)

Figure 2 shows the skeleton code for this implementation. Use multiple threads to parallelise the Collision Detection stage. Each thread has its local list of `currCollisions`. These are concatenated to the global list of `allCollisions` in a critical section (Figure 2, line 16) to prevent race conditions.
Also, note that line 6 in Figure 2 has a `nowait` clause so that threads do not have to synchronize before appending to the global list of collisions.

```
vector<Collision> allCollisions;
#pragma omp parallel num_threads(100) shared(input, N, L, r, particles, allCollisions)
{
    vector<Collision> currCollisions;

    #pragma omp for nowait private(i) schedule(dynamic, 3)
    for(i = 0; i < N; i++)
    {
        for(int j = i + 1; j < N; j++)
        {
           // Append to currCollisions if particles i and j collide
        }
        // ...
    }

    #pragma omp critical
    {
        // Append currCollisions to global allCollisions
    }
}
```

Figure 2. Parallel 1 skeleton code for collision detection (variable names slightly differ from source code).

2. Parallel 2

This approach is similar to the first approach. However, each thread does not keep a private list of collisions. Instead, all threads share a single list, and access to this buffer is controlled using OpenMP's `critical` construct.

```
#pragma omp parallel num_threads(100) shared(N, L, r, allCollisions) private(pColl, wColl)
{
    #pragma omp for schedule(static, 200)
    for(int i = 0; i < N; i++)
    {
        for(int j = i + 1; j < N; j++)
        {
            // continue if no collision

            #pragma omp critical (insert)
            {
                // Append to allCollisions
            }
        }
        //...
    }
}
```

Figure 3. Parallel 2 skeleton code for collision detection (variable names slightly differ from source code).

3. Parallel 3

This implementation differs from the previous approach as it uses a single `for` loop that is equivalent to using two nested loops for collision detection. This is discussed further in the next section ("Using loop fusing to distribute tasks more fairly").

Special Considerations made during parallelisation

Using dynamic scheduling for Parallel 1 & 2

We identified a limitation in the Parallel 1 & 2 approaches. Both approaches are implemented using nested For loops. However, the number of iterations of the inner loop depend on the outer loop's loop variable. This means that the work done in each iteration of the outer loop is not equivalent.

```
1 // Iterate through each combination of particles
2 for(int i = 0; i < N; i++)
3 {
4     for(int j = i + 1; j < N; j++)
5     {
6         // check if particles i and j collide
7     }
8     //...
9 }
```

Figure 4. Skeleton code for Collision detection. In a collision, ***the index of the first particle involved will always be lower than that of the second*** *(since j > i).*

In fact, as can be seen from the above figure, the earlier iterations of the outer loop have a higher workload than subsequent iterations. Hence, **OpenMP's static scheduling option is not optimal** for Parallel 1 & 2, because this form of scheduling allocates iterations to threads in a round robin manner at compile time. The scheduling does not consider the workload of each iteration. This means there is a chance that some threads will only be allocated iterations with lower-workloads, causing them to complete relatively quickly and then simply be idle while waiting for other threads to complete.

To circumvent this problem for Parallel 1 & 2, we chose to use **dynamic scheduling with a small chunk size**. Dynamic scheduling assigns iterations to threads as they complete their work, and hence, helps create a more even distribution of work. This prevents threads from being assigned only lower-workload iterations and then remaining idle. As shown in the figure below, this choice did yield performance benefits in terms of lower execution time.

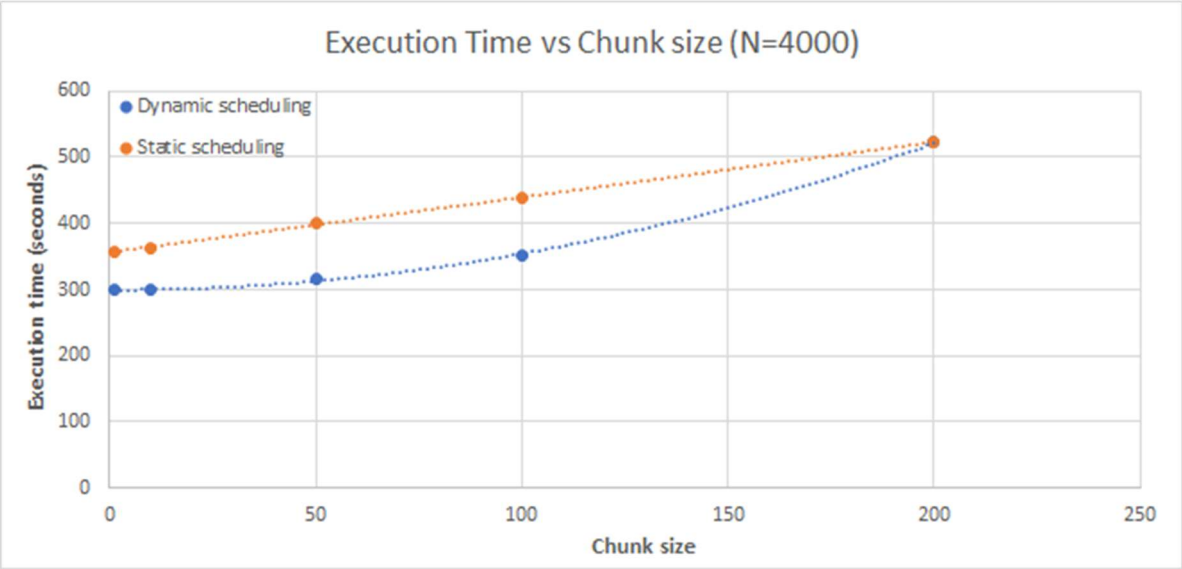

Figure 5. Performance comparison between dynamic scheduling and static scheduling for Parallel 2.

The figure above shows that using dynamic scheduling (for Parallel 2) decreases execution time (and improves performance) by 15%, on average.

Using loop fusing to create evenly sized iterations

As explained above, the loops in Parallel 1 & 2 do not have evenly sized iterations. While dynamic scheduling is one possible method of fairly distributing the iterations, we also tried using loop fusing to ensure that the iterations are evenly sized. As shown in the code snippet below, we transformed the nested loops in Parallel 1 & 2, into a single loop that iterates through all the pairs of particles.

```
#pragma omp parallel num_threads(100) shared(N, L, r, allCollisions) private(pColl, wColl)
{
    #pragma omp for schedule(dynamic)
    for(int k=0; k < N*(N+1)/2; k++)
    {
        int i = k / (N+1);
        int j = k % (N+1);
        // Check if particles i and j collide
    }
}
```

Figure 6. Parallel 3 skeleton code using loop fusing (variable names slightly differ from source code).

Each iteration of this loop corresponds to a distinct pair of particles. Hence, each iteration of this loop is much more likely to have a similar workload, compared with the loops in Parallel 1 & 2.

Results

We evaluated our parallelised programs by measuring their execution time in relation to four parameters:

- Number of threads
- Number of Particles => requires **more processing**
- Particle Density (number of particles per unit area) => **higher chances for collisions**
- Correctness or Performance mode

Our test cases for *Number of Particles* are designed such that each test case has the same particle density (obtained by increasing the length of the square), but a different number of particles.

In contrast, the *Particle Density* test cases all have the same square size, but an increasing number of particles.

1. Execution time vs Number of threads

In general, the execution time of all 3 parallel approaches decreased, up to a certain limit, as the number of threads increased. On the Intel Xeon machine, 60-100 threads were found to give best performance while 20-40 threads gave best performance on the Intel Core i7 machine (see Figure 7).

The number of logical cores on a machine provides a baseline for the number of threads that can be used to run our program optimally. The Intel Xeon machine, for example, has 10 physical cores that each support two threads via hyperthreading. This would mean that the machine can support 20 parallel threads without running into excessive overhead from context switching.

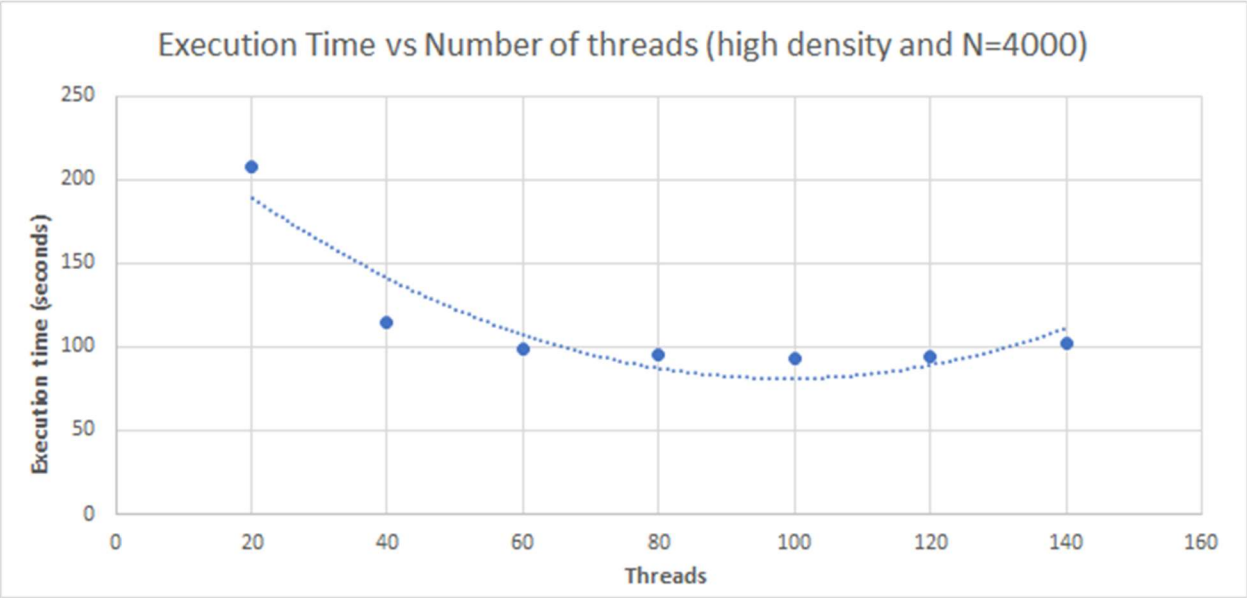

Figure 7. Performance comparison for different number of threads.

The nature of thread scheduling may explain why we continue to see performance increases when the number of threads is set greater than then number of logical cores. The executed process (the simulator) is shared with other processes on the system. Hence, a thread running the simulator process can be switched with another running process. If the simulator's threads are suspended, the execution time of the simulator gets lengthened. However, if we have more threads sharing the work, the impact of a single thread being switched out would be much lower.

Nevertheless, when the overhead of thread management and context-switching outweighs the aforementioned processing benefits, we reach the limit for the number of threads.

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                20
On-line CPU(s) list:   0-19
Thread(s) per core:    2
Core(s) per socket:    10
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz
```

Figure 8: Output of "lscpu", showing presence of 10 physical cores on Intel Xeon, with each core supporting 2 threads via hyperthreading.

Implications:

We ran experiments to decide the optimal number of threads for each of our approaches. These values were used for the other test cases. The results were -

- For Parallel 1, 100 threads on Intel Xeon.
- For Parallel 1, 32 threads on Intel Core i7.
- For Parallel 2, 60 threads on Intel Xeon.
- For Parallel 3, 80 threads on Intel Xeon.

2. Execution time vs Number of particles

The execution times of both the parallel and sequential programs grow quadratically with the number of particles (if number of threads is constant). This is because the Collision Detection stage has a quadratic time complexity and accounts for most of the execution time. The parallelized programs are, nevertheless, about 4 times faster than the sequential ones.

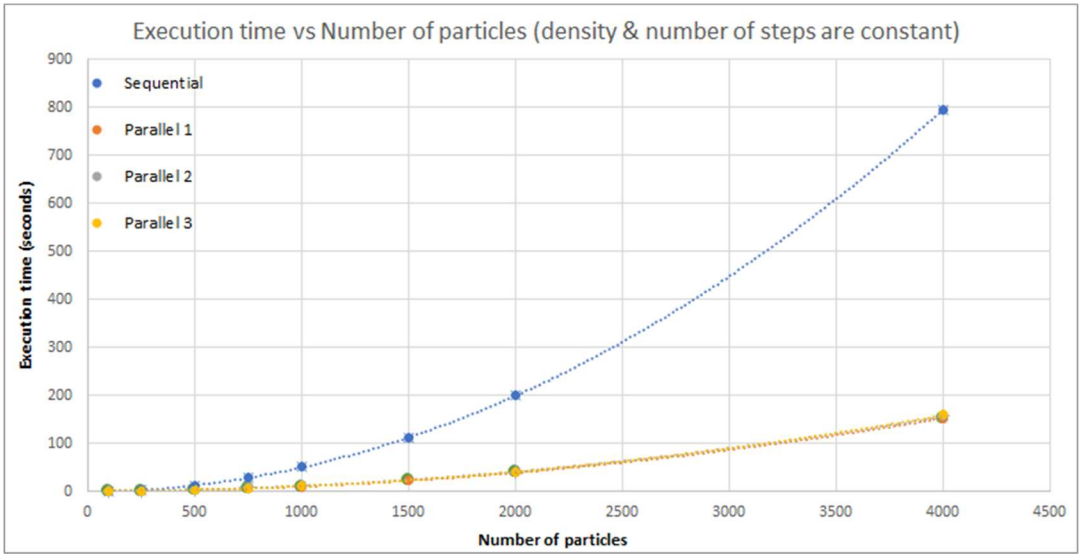

Figure 9. Performance comparison between sequential and parallel programs for different number of particles.

There is not much there to distinguish between the performances of the parallelized programs, but Parallel 1 performs the best. This could be because they all manage to achieve the same goal of equal work distribution between threads. Alternatively, some implementations may not have been tuned with the best parameters (number of threads, chunk size) for reducing execution time.

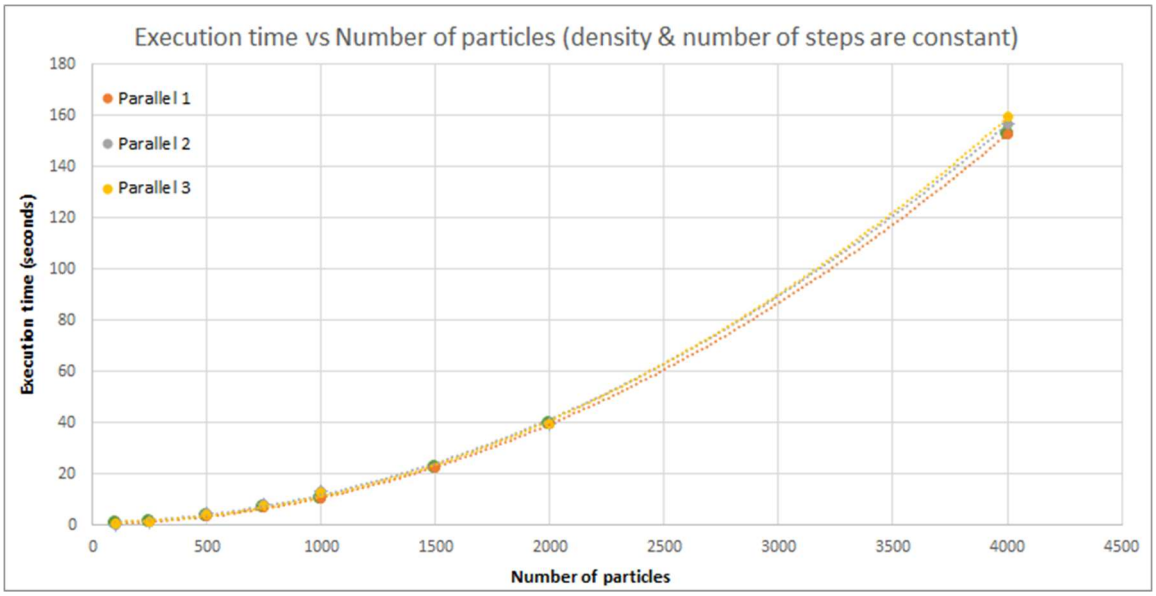

Figure 10. Performance comparison between the parallel programs for different number of particles.

3. Execution time vs Particle density

A higher particle density means that the probability of particles undergoing collisions is higher, since they have less space to move around unobstructed. Comparing Figures 11 and 12 (below) to Figures 9 and 10 (above), we see that the execution times for all the programs are lower for the particle density tests as compared to the number of particles tests (for the same number of particles and steps). That is, they perform better on the Particle density tests.

The main difference between these tests is the number of *possible* collisions (note: not the number of probable collisions!) and the number of collision resolutions. The former is higher for the previous test case (Number of particles) and the latter is higher for this test case (Particle density) as it is likely that there are more collisions.

These performance results provide further evidence that the Collision Detection task is the bottleneck of the simulator program. An increase in the number of collision resolutions does not greatly impact the execution time, whereas an increase in the possible combinations of collisions does negatively impact the performance.

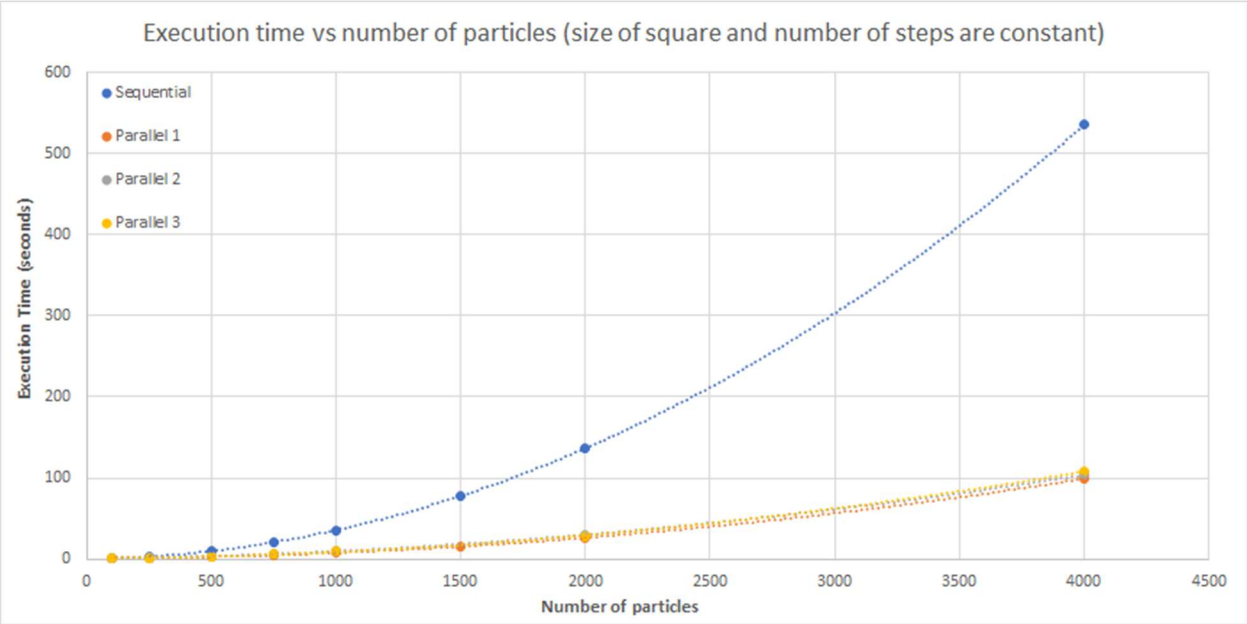

Figure 11. Performance comparison between sequential and parallel programs for increasing density of particles in the square.

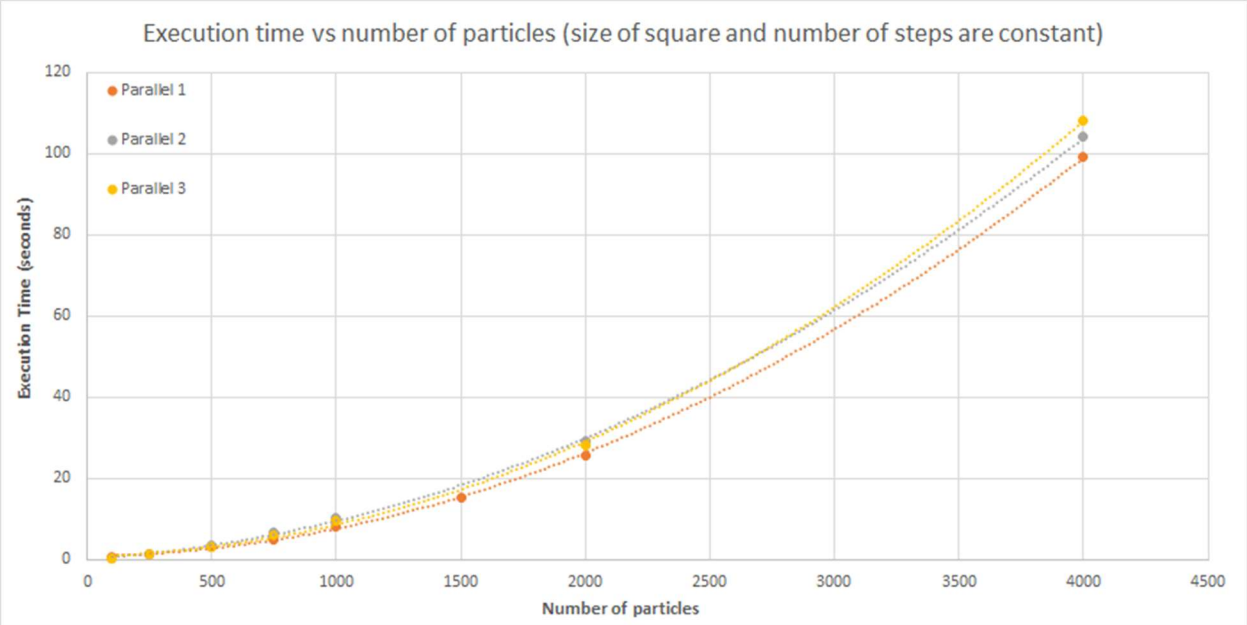

Figure 12. Performance comparison between sequential and parallel programs for increasing density of particles in the square.

Finally, a measure of the benefits of parallelism is shown in the tables below in the form of Speedup and Efficiency. It compares the best sequential time to the time of the **best performing parallelism implementation, Parallel 1**.

The speedup and efficiency results for the previous section (Number of Particles test cases) are similar, albeit slightly lower.

Here, we see that the speedup and efficiency increases as the size of the input increases. This is an indication that **our parallel program is scalable**.

Intel Xeon (10 cores, 20 logical cores):

Num Particles	Sequential Time (in secs)	Parallel 1 Time (best) (in secs)	Speedup	Efficiency	Parallel 1 MIPS
100	0.5714	0.5322	1.07	0.05	11633.81
250	3.1354	1.3	2.41	0.12	21158.92
500	10.0567	3.0399	3.31	0.17	29349.72
750	20.6177	5.0969	4.05	0.2	37093.09
1000	35.7051	8.3131	4.3	0.21	39664.95
1500	78.3212	15.4117	5.08	0.25	47315.36
2000	136.2658	25.6623	5.31	0.27	50126.31
4000	535.462	99.1475	5.4	0.27	51418.26

Intel Core i7 (4 cores, 8 logical cores):

Num Particles	Sequential Time (in secs)	Parallel 1 Time (best) (in secs)	Speedup	Efficiency	Parallel 1 MIPS
100	0.3913	0.2493	1.57	0.2	20424.33
250	2.1747	1.0363	2.099	0.26	25434.12
500	7.095	3.8053	1.865	0.23	23126.69
750	14.6284	7.4067	1.975	0.25	25349.22
1000	25.1915	10.5755	2.382	0.3	31050.41
1500	55.2482	20.9787	2.634	0.33	34687.36
2000	97.2669	37.9503	2.563	0.32	33847.54
4000	383.8827	167.4901	2.292	0.29	30419.07

If we compare the performance of Parallel 1 between the above tables, we see that it has a lower execution time on the Intel Xeon. This is in spite of the Intel Xeon having a lower clock speed of 2.20 GHz, compared to the 3.60 Ghz of the Core i7. However, since the chunks assigned to the threads used for parallelising the tasks are quite small and not computationally intensive, Intel Xeon is able to perform better as it has 2.5 times the number of logical cores in the Core i7.

4. Execution time vs Correctness mode & Performance mode

We tested the execution time for the program under the 2 modes:

- Correctness mode: print the information about the particles at each step
- Perf mode: only print the initial and final information about the particles

We observed a decrease in the execution time by 21% (on average) for Intel Xeon and by 19.4% (on average) for the Intel Core i7.

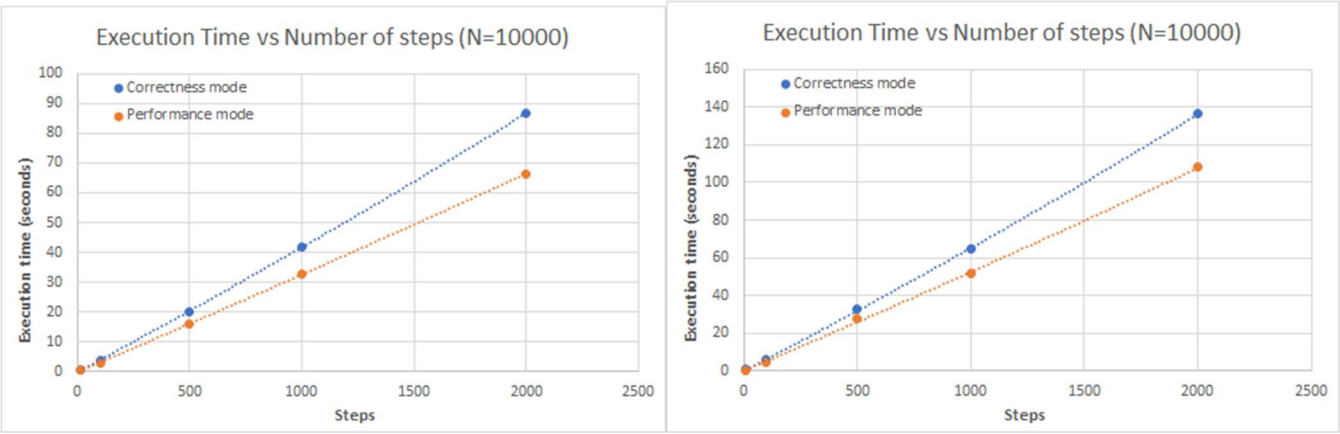

Figure 14. Performance comparison between the correctness and performance modes of the Parallel 1 program. Results are from execution on Intel Xeon(left) and Intel Core i7 (right).

We observed that all variants of the program performed better under the Performance Mode, compared with the Correctness mode. This is almost certainly because Correctness Mode requires the program to make additional system calls to print each time step's results to a terminal or file.

Conclusion

Our parallelised program shows significant performance improvements over the sequential version. Simulation is a convenient alternative to real experimentation as it is easy to get measurements and try different kinds of inputs. Additionally, it can be done in a shorter amount of time.

The current version of our program is suitable for its requirements. However, if the users are trying to trace through the outputs, it can be difficult to do so as the particles' information is only printed per 1 time unit. Finding the particles' movements and printing it at shorter time intervals (like 0.25 or 0.5 time units) can help users trace through and understand the movement of the particles better, but it comes at the cost of execution time. Here, we are effectively adding more "time steps"

and increasing the number of times that collisions have to be detected, which negatively impacts the execution time. Finding the right balance between granularity of the particles' movement vs a shorter execution time depends on the context in which this simulator is used.

Running our program and reproducing the results

Please read the `README` file in our assignment ZIP file, for instructions on compiling our program and running it with sample inputs.

The sample inputs are located in the `input/` directory and the expected outputs are in the `output/` directory.

To test the program with different number of threads, edit the global NUM_THREADS variable in `simulator.cpp`. To test the loop parallelisation with a different scheduling method or chunk size, edit the `pragma` directive in `collisions.cpp`.

Appendix

Assumptions:

Apart from the simplifying assumptions stated in the assignment description, here are some of our assumptions:

- When passing a C++ STL vector between functions, it is not necessary to pass it by reference for best performance. We assume that passing the vector by value is sufficient, and that the compiler will provide optimisations to maintain performance. See references.
- The order of printing the particle information should follow the order of particles in the input.
    - Implication: The print task effectively cannot be parallelised as it may lead to missing output or output the information in the wrong order.

References

Passing vector by value:

- How to return a vector from a function: by value or by reference?
- In C++, what is the best method to return a vector of objects from a function?

When attempting to fuse our nested loops for Parallel 3 approach, we referred to the Stackoverflow forum. In particular, these discussions were useful:

- OpenMP and C++: private variables
- Fusing a triangle loop for parallelization, calculating sub-indices

Explanation for why using more threads than the number of cores performs better:

- Why are 50 threads faster than 4?

Implementation of collision detection:

- Fast, Accurate Collision Detection Between Circles or Spheres