# CS3211 Project 2

Irvin Lim Wei Quan
A0139812A

In this project, we attempt to simulate the movement and behaviour of particles within a closed system, which we fondly refer to as a “galactic pool table”. These particles shall exhibit gravitational pull on other particles, while at the same time exhibiting perfectly elastic collisions.

The program fulfils the entirety of the project requirements, allowing the user to run a simulation of a “galactic pool table” with an input specification file, simulating both the , while producing an output heatmap of the particles’ positions through a Portable Pixmap (PPM) format file.

## Project Overview

A brief description of the program features, as well as a rundown of how the various components work are detailed in the following sections.

### Language/Compilation

The entirety of the program was developed in C, and compiled with the Open MPI compiler, which allows us to make use of MPI implementations in the Open MPI library `mpi.h`.

The program can be compiled using the bundled `Makefile`, which invokes the necessary linkers and compilation flags using `mpicc`, the MPI equivalent of `gcc`. The complete compilation and execution instructions can be found in more detail in the attached `README.md` file.

### Input Specifications

The program takes in input via the command-line arguments, as well as a specification file (whose path is read via command-line arguments). For details on the complete usage of the command-line arguments, you may refer to the `README.md` file.

Some sample specification files are included in the `samplespec/` directory, which serves as both simple test cases to ascertain the functionality of the program, as well as to provide the user with some idea on what features the program is capable of.

### Parallel/Sequential Variants

The same program was developed to run the same simulation either in parallel, where each process is responsible for each region separately and communicates with all other processes to synchronise their particles, as well as in series, where a single process completes all computation for all regions. These programs are labelled `pool` and `poolseq`, for the parallel and sequential versions respectively, and take in the exact same command-line arguments, as well as producing the exact same output.

The parallel version infers the number of expected regions by the number of processes (through the Open MPI −np argument). To pass the expected number of regions to the sequential variant, use the same −np argument to specify the regions; only one process will be executed for the program, and all other processes will be immediately terminated.

## Simulation of System

The program simulation of the "galactic pool table" essentially performs a simulation of the movement of many particles in a closed system. These particles can be either large or small, which correspond to the colours used to render them.

This system can be split into various regions in the form of a square grid, and particles are not allowed to escape outside of the system, which mimics the form of a real-life pool table, which happens to be square. As such, the number of regions must be a perfect square (i.e. 1, 4, 9, 16, 25…). The total number of particles as defined in the specification file (both large and small) will scale with the number of regions, hence the amount of work is not kept constant over $P$ processors, but rather increases exactly with $P$.

## N-body Simulation

In the simulation of the system, the particles possess a finite, positive mass, which causes them to exhibit gravitational forces against each other. The program attempts to simulate this by modelling the problem as an N-body simulation, where each particle exhibits Newton's law of universal gravitation, which states that a particle attracts every other particle in the universe with a force directly proportional to the product of their masses, and inversely proportional to the squared distance between their centres. In other words, the following formula holds:

$$F = G\frac{m_1 m_2}{r^2}$$

where $m_1, m_2$ are the masses of the particles, and $r$ is the distance between the centres of the two particles.

For our purposes of computation, we can compute the proportion of forces without factoring in the gravitational constant $G$, which is approximately equal to $6.674 \times 10^{-11}\mathrm{N} \cdot (\mathrm{m/kg})^2$ . Since the units used in the simulation are not actual SI units but are instead more arbitrary, removing the $G$ term allows us to use numbers in the specifications that are closer to $1$.

In addition, to prevent numerical divergences when particles get too close to each other which cause $F$ to approach infinity, we can make use of a softening parameter, which modifies the gravitational potential for each particle by a small amount, by modifying the above formula slightly (with $G$ omitted as described above):

$$F = \frac{m_1 m_2}{r^2 + \epsilon^2}$$

where $\epsilon$ is the softening parameter, selected as $0.0001$.

## "Bounce-Off" Frobisher Effect

The particles also possess a finite, positive radius, which causes the particles in the system to exhibit elastic collision upon contact with each other. We assume that all collisions are perfectly elastic, i.e. their momentum and kinetic energy is conserved. As such, we can compute the resultant velocity vectors resulting from the collision of two particles in the system as follows:

$$\mathbf{v}_1' = \mathbf{v}_1 - \frac{2m_2}{m_1 + m_2} \times \frac{(\mathbf{v}_1 - \mathbf{v}_2) \cdot (\mathbf{x}_1 - \mathbf{x}_2)}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2} (\mathbf{x}_1 - \mathbf{x}_2),$$

$$\mathbf{v}_2' = \mathbf{v}_2 - \frac{2m_1}{m_1 + m_2} \times \frac{(\mathbf{v}_2 - \mathbf{v}_1) \cdot (\mathbf{x}_2 - \mathbf{x}_1)}{\|\mathbf{x}_2 - \mathbf{x}_1\|^2} (\mathbf{x}_2 - \mathbf{x}_1)$$

where $\mathbf{x}_1, \mathbf{x}_2$ are the position vectors, $\mathbf{v}_1, \mathbf{v}_2$ are the velocity vectors, and $m_1, m_2$ are the masses for the two particles.

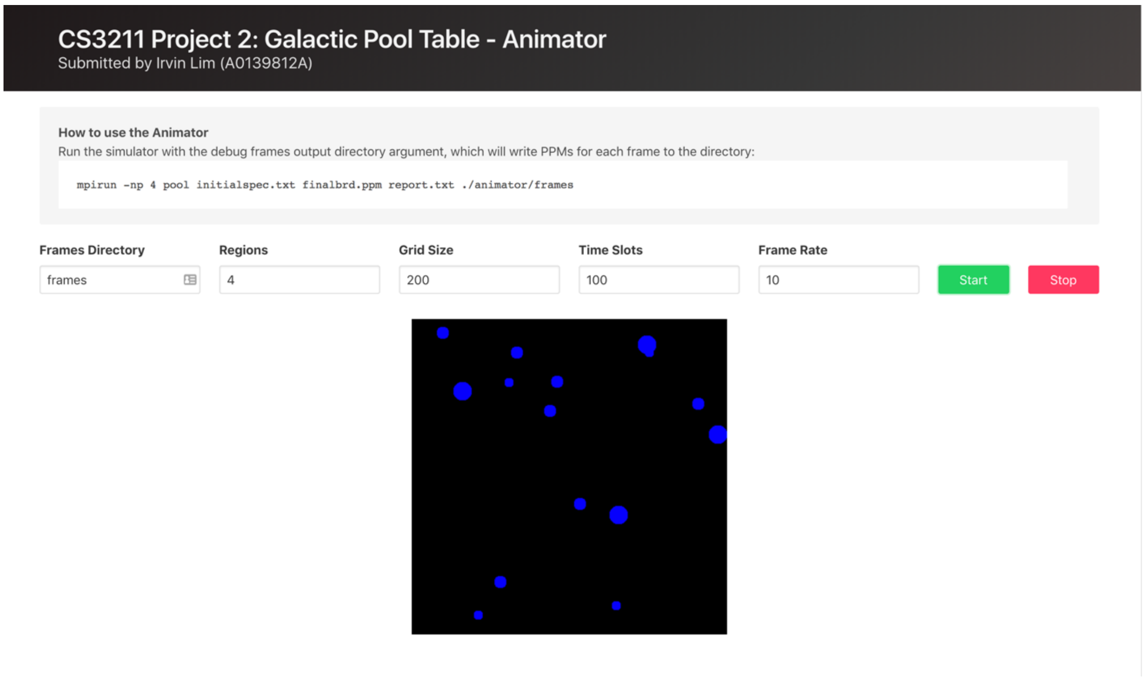
In addition, to simulate the effect of the "galactic pool table", the particles also exhibit the same "bounce-off" effect upon contact with the sides of the "walls" of the system. By assuming that the "walls" have a much, much larger mass than any of the particles in the system, we can assume the particle would simply "reflect" upon collision with the wall, with no change in momentum or magnitude of velocity:

$$\mathbf{v} = \begin{cases} (\mathbf{v}_x, -\mathbf{v}_y), & \mathbf{v}_y < 0 \vee \mathbf{v}_y \geq GridSize \times PoolLength \\ (-\mathbf{v}_x, \mathbf{v}_y), & \mathbf{v}_x < 0 \vee \mathbf{v}_x \geq GridSize \times PoolLength \end{cases}$$

where $\mathbf{v}_x$ and $\mathbf{v}_y$ are the scalar components of the velocity vector $\mathbf{v}$ for each particle, $GridSize$ is the size of the grid in each region (as specified in the specification), and $PoolLength$ is equal to $\sqrt{R}$, where $R$ is the total number of regions.

## Frame-by-Frame Animation

In order to better visualize the simulation, it may be a good idea to display the frame-by-frame movement of particles after each time step, which will help with both debugging the physics simulation, as well as to simply observe the behaviour of particles better. As such, the program accepts an additional command-line argument which specifies the location where output frames should be saved to. These frames can then be piped into a simple HTML animator page I constructed (using GPU.js, no less), which loads and strings all the frames together to produce a smooth animation on a HTML5 canvas:



CS3211 Project 2: Galactic Pool Table - Animator

Submitted by Irvin Lim (A0139812A)

**How to use the Animator**

Run the simulator with the debug frames output directory argument, which will write PPMs for each frame to the directory:

```
mpirun -np 4 pool initialspec.txt finalbrd.ppm report.txt ./animator/frames
```

| Frames Directory | Regions | Grid Size | Time Slots | Frame Rate |
|---|---|---|---|---|
| frames | 4 | 200 | 100 | 10 |

*Figure 1: Screenshot of frame-by-frame animator HTML page using GPU.js*

## Implementation, Challenges and Considerations

This section outlines the implementation details and considerations for both the parallel and sequential variants of the program, while also detailing some challenges and optimisations with regards to parallelization, communication and accuracy.

### Overall Program Pipeline

Although the parallel and sequential variants work differently, the underlying pipeline of tasks that have to be performed is the same. The following table shows a pipeline of tasks that the program has to execute, including remarks on which tasks are parallelizable, or a critical section, etc.:

| | Task Description | Remarks |
|---|---|---|
| **Prelude** | Read input specification file and program arguments | - |
| | Generate particles for each region (based on specification) | Parallelizable |
| **Loop** | Synchronisation of particles between regions | Only in parallel variant |
| | Execution of time step on each region | Parallelizable |
| **Postlude** | Final synchronisation of particles between regions | Only in parallel variant |
| | Generate frame for each region | Parallelizable |
| | Gather all frames into the master process | Only in parallel variant |
| | Output final frame to PPM file | Critical section |

*Figure 2: Overall program pipeline for both the parallel and sequential variants*

## Load Balancing

In the parallel variant of the program, we can shorten the overall runtime of the program by distributing the parallelizable fraction of the workload (i.e. load balancing) across multiple processes.

Each process is inherently responsible for a single region, and by splitting up the computational tasks into regions so that they can be computed independently of other regions, we can parallelize the workload in that manner. From the pipeline above, we can see that the following tasks are parallelizable:

1. Generation of initial particles for each region
2. Execution of time step on each region
3. Generate output frame for each region

While load balancing shortens the overall runtime of the program, its effectiveness is dependent on the slowest process that finishes, since there are some tasks that require all processes to complete before they can all proceed to the next one.

Although splitting workload by region may be a simple option, the workload may not be ideally balanced, since some regions may have substantially more particles than others. However, this trade-off is justified, since the requirements specify that we should be parallelizing the program in this manner, and it is much more efficient to group particles by region since various other requirements (such as horizon regions) would affect the number of particles that have to be communicated and iterated over.

In the sequential version, these tasks would simply be performed $R$ times; once for each of the $R$ regions.

## Data Structures

There are various ways to store the numerous properties of particles in a region within each process. The most straightforward would be to model each particle individually, storing the state of its position and velocity at each time step.

For this program, since particles would have to be sent and received between processes in the parallel version, I chose to store the various properties for a particle within a `struct`, which allows us to store a dynamically sized array of values that can be dereferenced easily and clearly. This also allows us to include additional properties for a particle in the future, such as acceleration for example.

Since we are performing operations on particles in batches by their region, we would also have to store them in a way so that we can index them by their regions (based on their positions). As such, I have chosen to store particles in a 2-dimensional array, which is indexed by regions in the first dimension. This allows us to iterate upon particles which should be computed and sent to other processes easily.

The downside is that particles would need to be reallocated within the 2-D array once their positions are updated at each iteration, which may incur penalties from allocating and copying a large amount of memory. However, this one-time penalty is justified, as compared to having to recompute the region for a particle each time its region needs to be accessed; and also helps with memory access since particles in the same region would be contiguous in memory, which works well with the various operations on an entire region of particles within each iteration.

## Synchronisation of Particles

Since we have decided to split the tasks by region, each process in the parallel version does not actually need to be aware of particles in other regions unless it has to. Out of a total of $NR$ particles, where $N$ is the number of particles per region and $R$ is the total number of regions, a process only needs to have the data for $1/R$ of the particles on average, assuming that we do not factor in horizon regions.

Because a particle's position and velocity updates at each iteration when a process performs the computation for its region, we would either have to perform an all-to-all broadcast across all processes to synchronise all $NR$ particles, or we could only send the particles that are needed by the process to perform computation for its next iteration.

Since sending all $NR$ particles is a waste of communication resources when the process is not going to use it anyway, we would have to select the specific particles to send, and which process to send to. Furthermore, not all particles in the process would have to be sent to other processes, since it is highly likely that most particles would be in the same region in the subsequent iteration. This greatly reduces communication overhead by reducing the number of items that has to be sent for each iteration.

The overall synchronisation strategy is as follows:

1. After updating the positions of particles in the region that the process was responsible for, it now has particles whose new regions could be anywhere on the board.
2. Each process selects the computed particles which are no longer in the current region, and sends them to the process that should be responsible for them next.
3. Each process receives particles from any of the $R$ processes, and merges all of them together into a single array.
4. Each process now has the most updated particles for its region **only**.
5. Each process should duplicate this updated array of particles for its region to other processes which require them for horizon region computation.
6. Each process receives the particles in other regions from other processes, and allocates them to the 2-D array, indexed by their region.

This process ensures that, with the precondition that each process starts with knowing the most updated positions for at least those in the region it is responsible for, that all processes end up with most updated positions of particles that are required for the next iteration of computation, with the smallest amount of data being communicated each time.

Assuming that the `Horizon` parameter is `0`, the total number of particles that have to be sent during each of the above communication procedure in each iteration is at most $NR$ particles, as compared to an all-to-all broadcast which would send exactly $N^2(R-1)$ particles each time, regardless whether other processes needed the data or not.

Because message passing can incur deadlocks if not performed correctly, we also need to ascertain the exact order in which processes should be sending or receiving particles. In other words, when the sender sends a message, the receiver should be expecting to receive it from the sender, and not waiting on another sender who is in turn waiting to receive another message from the receiver. We can solve this deadlock problem by performing a round-robin scheduling approach. An example schedule, assuming 4 processes, is as follows:

| Turn to Send | Process 0 | Process 1 | Process 2 | Process 3 |
|---|---|---|---|---|
| **Process 0's Turn** | $0 \rightarrow 1$ | $1 \leftarrow 0$ | | |
| | $0 \rightarrow 2$ | | $2 \leftarrow 0$ | |
| | $0 \rightarrow 3$ | | | $3 \leftarrow 0$ |
| **Process 1's Turn** | $0 \leftarrow 1$ | $1 \rightarrow 0$ | | |
| | | $1 \rightarrow 2$ | $2 \leftarrow 1$ | |
| | | $1 \rightarrow 3$ | | $3 \leftarrow 1$ |
| **Process 2's Turn** | $0 \leftarrow 2$ | | $2 \rightarrow 0$ | |
| | | $1 \leftarrow 2$ | $2 \rightarrow 1$ | |
| | | | $2 \rightarrow 3$ | $3 \leftarrow 2$ |
| **Process 3's Turn** | $0 \leftarrow 3$ | | | $3 \rightarrow 0$ |
| | | $1 \leftarrow 3$ | | $3 \rightarrow 1$ |
| | | | $2 \leftarrow 3$ | $3 \rightarrow 2$ |

*Figure 3: Round-robin for send/receive schedules for each process. ← denotes receiving from, while → denotes sending to.*

This schedule ensures that the communication process is deadlock-free. Note that the order of processes in which particles are received from is also ensured to be in order, which is an added benefit of using such a schedule. By ensuring that all communications are deadlock free, we can avoid the use of non-blocking calls (i.e. `MPI_Isend` and `MPI_Irecv`), which are harder to work with since we need to check for the completion of the asynchronous calls with `MPI_Wait`, for example.

## Generation of Output

We also notice that the generation of the output frame (via a PPM format file) can be parallelized by regions, as the runtime for this procedure scales with respect to the number (as well as the radius) of the particles, since the procedure iterates on each pixel that the particles occupies within the frame. However, the individual generated frames only constitute regions, and must be stitched together into a single PPM file by a single process, since writing to the filesystem definitely has to be done in series.

This procedure involves both computation as well as communication, which incurs rather significant overhead since it is an all-to-one communication, making the communication time scale with respect to the number of regions (i.e. number of processes). This is considered a bottleneck that cannot be efficiently scaled up even with more processes.

On first glance, it looks like each process could generate a square canvas of length $GridSize$, and have the master process stitch all of the canvases together to give rise to a canvas of length $GridSize \times PoolLength$. For example, with 9 processes and a $GridSize$ of 200, we could try to generate 9 canvases of size $200 \times 200$ each, and produce a complete $600 \times 600$ grid in the master process.

However, this would produce inaccurate outputs when particles are close to the border of a region, since any pixels that are outside of a region would get "cut off", such as the following output frame, generated for 4 regions each with 4 large particles:

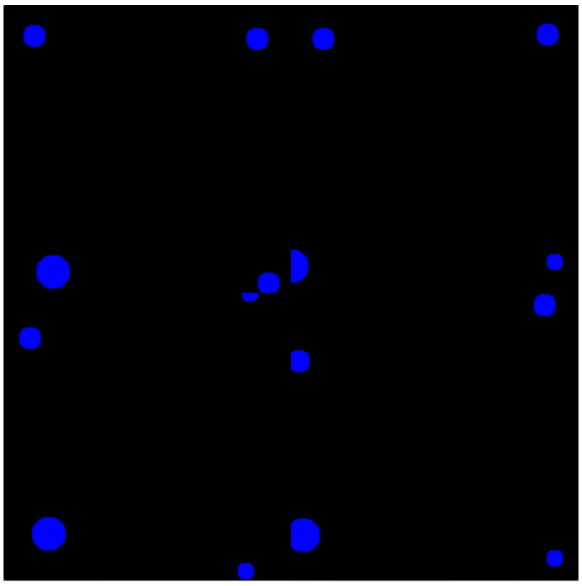


*Figure 4: "Cut off" portions of particles in the rendered output. Notice the incomplete circles at the middle of the image.*

Unfortunately, there is no elegant way to solve this, short of having to either not parallelize the output frame generation, or to increase the size of the frame generated by each process. The latter was chosen and implemented, in which each process would generate a canvas of length $GridSize \times PoolLength$ instead. This incurs heavier communication costs, with a much larger array size that is being sent by all processes to a single master process.

To prevent unnecessarily high communication costs at each iteration, the frame-by-frame output feature should not be turned on except when debugging, or if the frame-by-frame output is so desired.

## Pixel Discretization

In addition to the problems described with generating frames in parallel above, another issue is with the accuracy of the particles being generated on the pixel map. Also colloquially known as *pixelization*, particles in non-integer positions on a pixel map would tend to have their actual positions rounded off, based on the resolution of the image. Since our output frame to simulation grid resolution has a ratio of 1:1, we can expect that any decimal points would be rounded off to the nearest integer.

Furthermore, each particle is to be rendered as a circle, whose border cannot be represented completely discretely on a pixel map of squares. This discretization problem could result in particles being represented by a different number of pixels each time it is rendered at a different position, making them seem larger than they actually are during its movement.

Currently, the implementation samples each pixel within the particle's bounding box, and colours it with the appropriate colour as long as any portion of the particle is within the pixel. This can be done by computing the Pythagorean distance between the particle's origin pixel and each pixel in the box, and comparing that with the radius of the particle.

For particles whose radius $r < 1$, this presents another unique problem since it has to occupy at least one pixel in order to be visible in the resultant image, which means that we would have to round up the particle's radius before the above computation.

## Collision Detection

An overview of how the "bounce-off" Frobisher effect had been achieved has been covered in the previous sections. However, it should be noted that the computation is inherently flawed because of the temporal discretization of time steps, which is much more obvious than the effect of gravitational forces.

The current method of detecting collisions between two particles depends on checking whether the distance between their origins is lesser than or equal to the sum of their radii, which is a straightforward test at each timestep. However, this method to detect collisions is entirely dependent on the `TimeStep` parameter provided to the program, which could result in wildly different simulation results depending on the parameter's value.

Hence, if the time step is too large the particles may seem to pass through each other despite collisions being already implemented. Assuming two particles are approaching each other, and are not yet detected as collided in frame $i$. However due to their high velocity, their new positions in the next frame $i + 1$ could result in both particles having new positions that have gone past each other, making them seemingly have passed through each other.

This is a well-known problem in the business of performing time-wise simulation of objects' movements, since numerical integration always introduces some form of error into the result, with a larger time step introducing a larger error.

Unfortunately, there is also little we can do to prevent this error without shortening the time step and performing additional iterations of computation. One form of optimisation we could choose to adopt would be to only perform smaller timestep computations when necessary, such as choosing to delay the computation for a particle which we are confident would not be involved in collisions or significant gravitational forces, and choose to compute its new position based on its velocity. On the other hand, particles in areas which are more densely populated could choose to have finer timesteps to reduce the error margin. This approach could work if there are wildly varying densities of particles in a simulation.

## Analysis

In this section, we will analyse the program's speedup with respect to changing the problem size in various parameters, using the execution time from both the parallel and the sequential variants of the program.

### Speedup with respect to number of regions

The number of regions is proportional to the total number of particles in all regions, and thus the problem size grows linearly with the total number of regions. By taking the total execution time of the problem, which is the sum of the computation and communication runtime for all time steps, we can compute the speedup of the parallel variant when scaling the problem with respect to the number of regions. All other parameters are kept constant as much as possible between the parallel and sequential program executions.

The $Horizon$ parameter is set to the maximum possible value for each $Regions$ value, since the default behaviour for the sequential program is to take every region into account, so as to keep things constant. In other words, both the parallel and sequential variants will have the same number of computations. The values for the various parameters used are as follows:

| Particles/Region | GridSize | TimeSlots | TimeStep | Horizon |
|---|---|---|---|---|
| 500 | 1000 | 20 | 5 | Maximum |

*Figure 5: Parameters used for comparing speedup with respect to number of regions.*

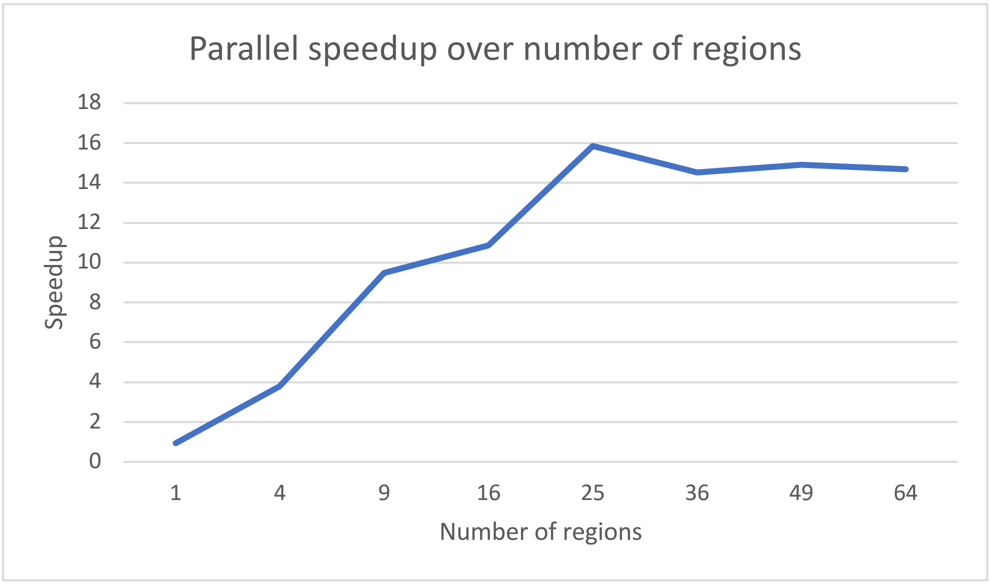


*Figure 6: Graph representing the speedup for parallel over sequential execution by varying the number of regions.*

As the number of regions increases, we can expect that the total number of particles would increase as well, since there are 500 particles per region. Assuming that $N$ is the total number of particles and $P$ is the number of processes, the computation of gravitational forces is a $O(N^2)$ operation since each particle computes its new force based on every other particle in the entire grid, and there is an upper bound of $O(N)$ collision computations. By being able to parallelise the computations over $P$ processes on the parallel variant, it means that the execution runtime of the program would be $O(\frac{N^2}{P})$ instead, but in addition also incurs some communication overhead.

We can also see that the speedup tapers off beyond 25 regions. This is easily explained by the fact that the experiment was performed on the Tembusu machines, which have 2x Intel® Xeon® ES-2620 v2 CPUs, making them have a total of 24 threads. Any parallelization beyond 24 processes would be context switched and not be able to benefit from parallelisation and pipelining via hyperthreading. As such, the speedup remained considerably constant at around 14x to 15x.

## Parallel runtime with respect to number of regions

Interestingly, it was also found that the total runtime for computation and communication in the parallel program execution actually increases when the number of regions are increased. For this experiment, only the parallel variant was used, and the $Horizon$ parameter was set to 0, so that each process would be computing the same number of particles regardless of the number of regions. The exact values of the parameters used are as follows:

| Particles/Region | GridSize | TimeSlots | TimeStep | Horizon |
|---|---|---|---|---|
| 1000 | 200 | 100 | 5 | 0 |

*Figure 7: Parameters used for comparing parallel runtime.*

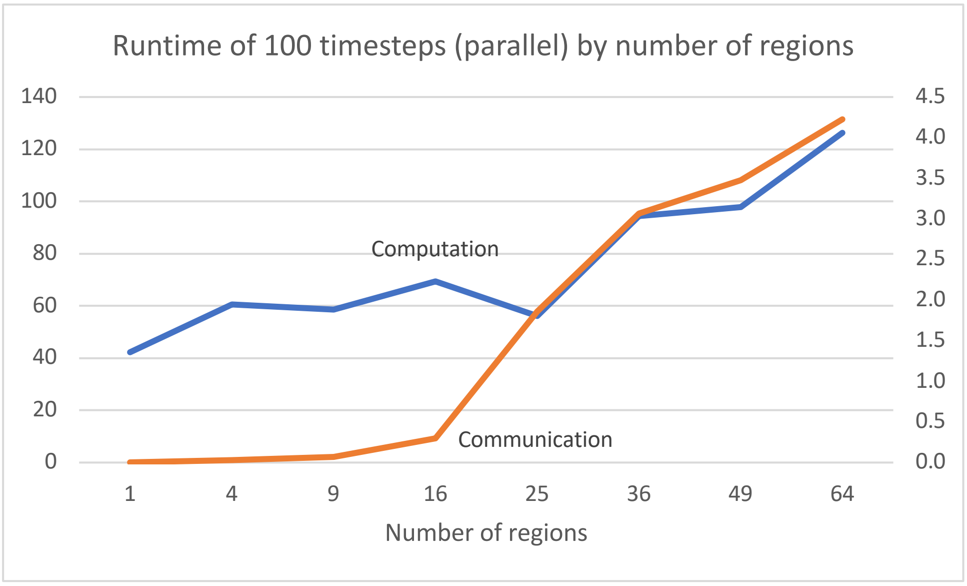


*Figure 8: Graph for both computation and communication runtime of parallel program execution.*

Even though in theory the computational runtime of each process should be constant since the number of particles that are being computed is kept constant, there is a slight increase in computation runtime before 25 regions. One likely reason is that because the overall runtime is dependent on the execution time of the slowest process in each timestep, with an increase in the number of processes there is an increased likelihood of a process being slower than the rest (i.e. law of large numbers). It is possible for that particular process to have significantly more particles in the region, perhaps due to strong gravitational forces attracting many of them there, such that the execution time is no longer as load-balanced as before with an increase in the number of regions.

On the other hand, the communication runtime increases sharply with an increase in the number of regions. This is expected since every single process potentially needs to communicate with every other process to synchronise its particles during each timestep, which makes the communication overhead more apparent as the number of processes increases.

Additionally, beyond 25 regions (i.e. processes), we see the same behaviour of a steep spike of both communication and computation runtime as in the previous section, which is likely due to the context-switching behaviour explained earlier.

## Speedup with respect to number of particles per region

By changing the number of particles per region, we are essentially scaling the problem in the same manner as before, since the total number of particles in all regions would grow linearly with respect to the number of particles per region. The speedup of the execution time was taken when scaling the problem size with respect to the number of particles per region, using the following parameter values:

| Regions | GridSize | TimeSlots | TimeStep | Horizon |
|---|---|---|---|---|
| 4 | 20000 | 20 | 5 | 1 |

*Figure 9: Parameters used for comparing speedup with respect to number of particles per region.*

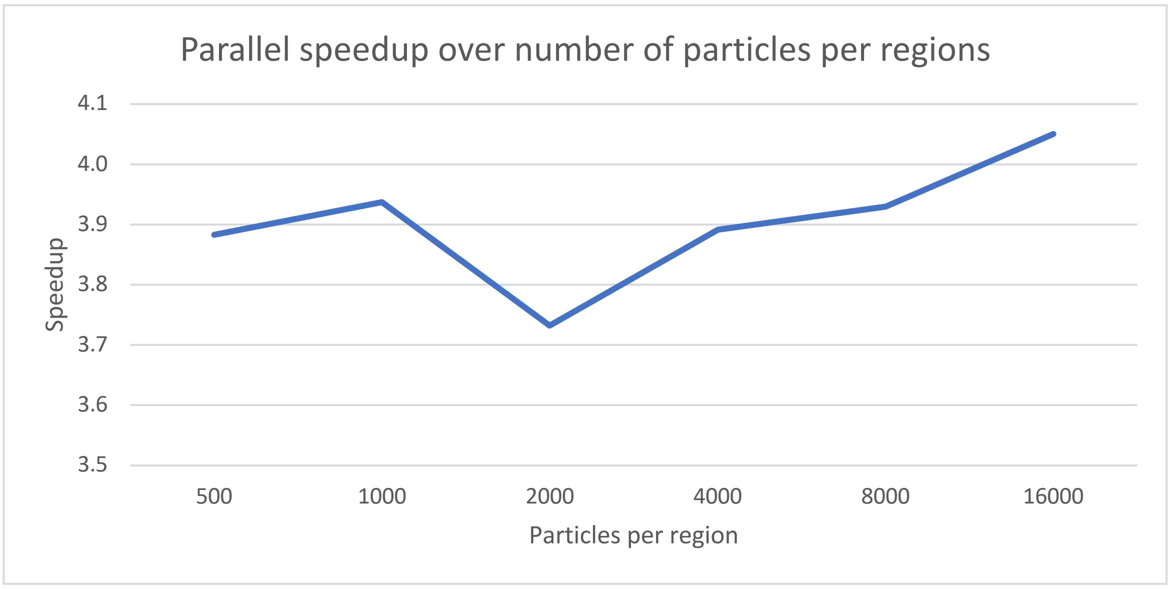


*Figure 10: Speedup for parallel over sequential execution runtime while varying the number of particles per region.*

We see that the speedup is relatively constant at between 3.7x to 4x even when the number of particles per region was increased. Since we had used 4 regions, this meant that the parallel execution runtime was able to load balance the computation rather evenly across 4 processes. Since the workload to process ratio remains constant as we increase the number of particles in each region, it is straightforward to see why the speedup would remain constant. We can also expect that communication costs should increase since there are now more particles' data to be transmitted in every timestep.

## Speedup with respect to grid size

We can also measure the speedup with respect to the size of each region, which can be set via the `GridSize` parameter. This should theoretically have no effect on the computation or communication runtime, and as such we expect that the speedup should be constant at around 4x since the workload is the same, like in the previous section.

| Regions | Particles/Region | TimeSlots | TimeStep | Horizon |
|---|---|---|---|---|
| 4 | 1000 | 20 | 5 | 1 |

*Figure 11: Parameters used for comparing speedup with respect to the grid size.*

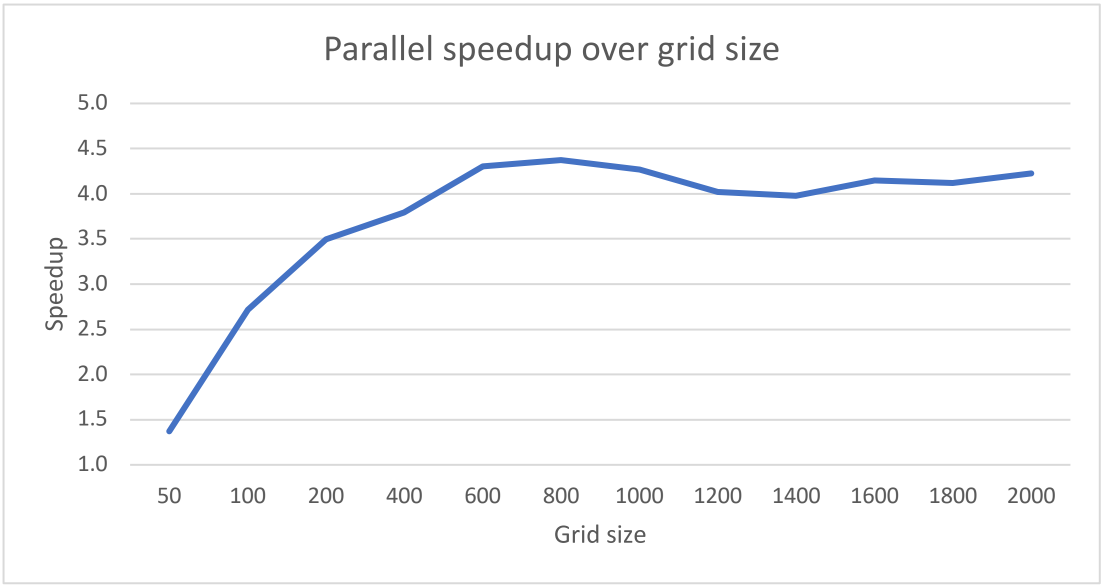


*Figure 12: Speedup for parallel over sequential execution runtime while varying the size of each region.*

We can see that the speedup is similarly constant when the grid size is between 400 to 2000, since the workload is approximately the same regardless of the grid size.

However, while we had previously measured the speedup with respect to both the number of particles in a region, the effect of having more particles in a fixed region was not explained. In other words, the denser a region is filled with particles, it was noted that the runtime significantly increases. The parallel variant of the program seems to scale not as well when the grid is too dense, as can be seen by the smaller speedup when the grid size is 50x50. This is probably due to poor load balancing, as the differences in the computational runtime for each timestep is rather large, with an example shown below:

| Timestep | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|
| **Fastest Time (s)** | 27.216037 | 54.067496 | 94.052588 | 159.15819 | 337.08453 |
| **Fastest Process** | 3 | 3 | 1 | 3 | 3 |
| **Slowest Time (s)** | 33.996779 | 62.501951 | 119.68442 | 240.69688 | 502.45124 |
| **Slowest Process** | 0 | 0 | 2 | 0 | 1 |
| **Range** | 6.780742 | 8.434455 | 25.631832 | 81.53869 | 165.36671 |

*Figure 13: Sample of timestep-wise computational runtime, sampled across 4 processes over 20 timesteps.*

We can see that the range (i.e. the difference between the maximum and minimum times) increases at an increasing rate, which is a telling symptom that the load balancing is poor. The reasons for this could be many – but as explained earlier, a better load balancing heuristic would be to split the number of particles evenly across processes, rather than by region. However, it is also extremely difficult to evenly balance the workload since the number of computations vary wildly depending on other particles as well, which cannot be predicted before actually performing the computation itself.

## Parallel runtime with respect to horizon value

We can also compare the effect of the `Horizon` parameter on the computational and communication runtime for the parallel variant of the program. The following parameters were used:

| Regions | Particles/Region | GridSize | TimeSlots | TimeStep |
|---|---|---|---|---|
| 36 | 1000 | 200 | 20 | 5 |

*Figure 14: Parameters used for comparing parallel runtime with respect to horizon value.*

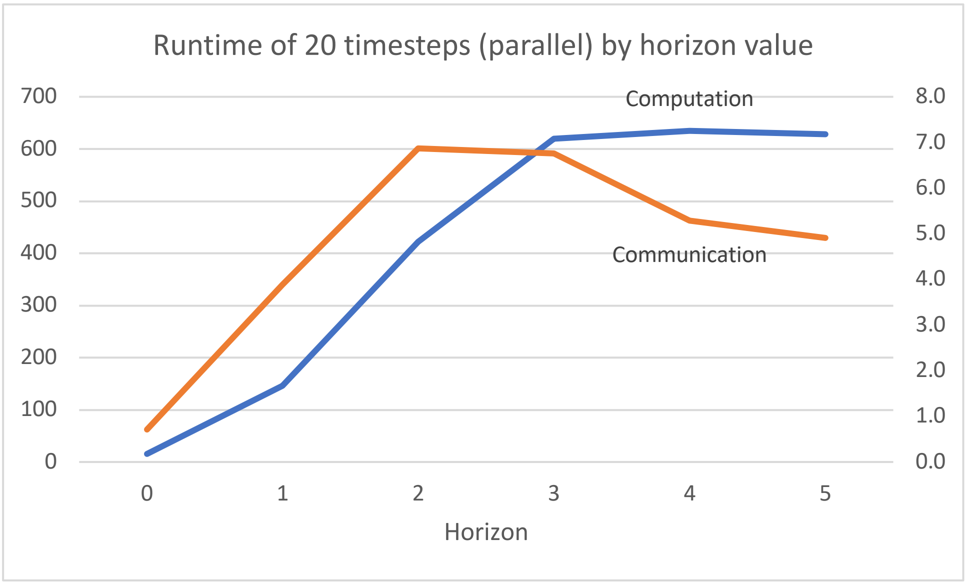
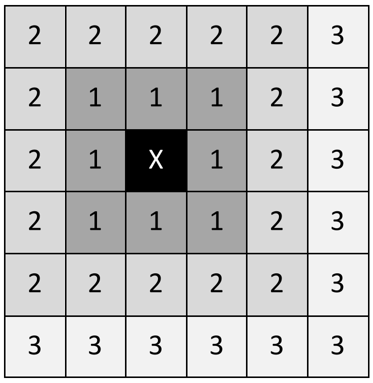


*Figure 15: Graph representing both runtimes for parallel program execution with respect to horizon.*

It is expected that the computational runtime would increase sharply. Numerically, since more particles are now involved in the computation of gravitational forces and elastic collisions, which are $O(N^2)$ with respect to the number of particles. The number of particles is $O((h!)^2)$ with respect to horizon value $h$, which makes the computational runtime $O((h!)^4)$, making a massive difference when increasing the horizon value. This is backed up by the graph above. In addition, the communication runtime would also increase in the same manner, since more particles are needed to be sent to each process for every timestep.

We also notice that the computational runtime tapers off after the horizon value is 3. One likely reason is that, since the runtime is computed based on the slowest process in every timestep, the slowest process when the horizon value is 3 is equally as slow as when the horizon value is 5. For this, we can easily come up with such a case, assuming a 36-region grid:



| | | | | | |
|---|---|---|---|---|---|
| 2 | 2 | 2 | 2 | 2 | 3 |
| 2 | 1 | 1 | 1 | 2 | 3 |
| 2 | 1 | X | 1 | 2 | 3 |
| 2 | 1 | 1 | 1 | 2 | 3 |
| 2 | 2 | 2 | 2 | 2 | 3 |
| 3 | 3 | 3 | 3 | 3 | 3 |

*Figure 16: Diagram representing a 36-region grid with horizon value of 3.*
*Notice that for this region, particles from all regions have to be factored in during computation.*

We can easily see that the region labelled "X" would require the particle data from all regions due to the definition of what makes a horizon region. As such, it is highly likely that the process that is responsible for this region would be the slowest in computation during each iteration. Therefore, increasing the horizon value beyond 3 shows little change in the computational runtime.

# Appendix

| REGIONS | SEQUENTIAL | PARALLEL | SPEEDUP |
|---|---|---|---|
| **1** | 1.300700 | 1.394823 | 0.932520 |
| **4** | 24.600613 | 6.469927 | 3.802301 |
| **9** | 141.116678 | 14.881348 | 9.482789 |
| **16** | 411.938790 | 37.979366 | 10.846384 |
| **25** | 1141.524256 | 72.027094 | 15.848540 |
| **36** | 2182.872090 | 150.442678 | 14.509660 |
| **49** | 4048.972300 | 271.431127 | 14.917126 |
| **64** | 6665.845760 | 453.655345 | 14.693634 |

*Figure 17: Speedup table for program execution with respect to the number of regions.*

| REGIONS | COMPUTATION | COMMUNICATION | TOTAL |
|---|---|---|---|
| **1** | 42.246631 | 0.002410 | 42.249041 |
| **4** | 60.592584 | 0.028906 | 60.621490 |
| **9** | 58.531234 | 0.068104 | 58.599338 |
| **16** | 69.444586 | 0.295231 | 69.739817 |
| **25** | 56.016367 | 1.856829 | 57.873196 |
| **36** | 94.401210 | 3.066557 | 97.467767 |
| **49** | 97.939674 | 3.478400 | 101.418074 |
| **64** | 126.290920 | 4.225729 | 130.516649 |

*Figure 18: Parallel runtime (computation and communication) with respect to the number of regions.*

| PARTICLES/ REGION | SEQUENTIAL | PARALLEL | SPEEDUP |
|---|---|---|---|
| **1000** | 97.607945 | 24.792674 | 3.936967 |
| **2000** | 390.067980 | 104.514777 | 3.732180 |
| **4000** | 1562.647791 | 401.566735 | 3.891378 |
| **8000** | 6253.747280 | 1591.594185 | 3.929235 |
| **16000** | 25493.466900 | 6294.137096 | 4.050351 |

*Figure 19: Speedup table for program execution with respect to the number of particles per regions.*

| GRID SIZE | SEQUENTIAL | PARALLEL | SPEEDUP |
|---|---|---|---|
| **50** | 1425.560413 | 1006.460180 | 1.61147779 |
| **100** | 185.864816 | 68.331287 | 2.72005437 |
| **200** | 109.376935 | 31.302191 | 3.49422617 |
| **400** | 107.510057 | 28.329675 | 3.7949626 |
| **600** | 108.101811 | 25.114922 | 4.30428615 |
| **800** | 109.119215 | 24.951403 | 4.37326971 |
| **1000** | 106.585788 | 24.952133 | 4.27161029 |
| **1200** | 100.763735 | 25.042227 | 4.02375296 |
| **1400** | 100.584189 | 25.294026 | 3.97659862 |
| **1600** | 103.464509 | 24.940966 | 4.14837617 |
| **1800** | 104.034014 | 25.261952 | 4.11820963 |
| **2000** | 105.240439 | 24.904589 | 4.22574486 |

*Figure 20: Speedup table for program execution with respect to the grid size.*

| HORIZON | COMPUTATION | COMMUNICATION | TOTAL |
|---|---|---|---|
| **0** | 0.711483 | 15.488798 | 16.200281 |
| **1** | 3.890306 | 145.723754 | 149.614060 |
| **2** | 6.869859 | 422.479258 | 429.349117 |
| **3** | 6.757798 | 619.851055 | 626.608853 |
| **4** | 5.291969 | 634.728167 | 640.020136 |
| **5** | 4.909457 | 628.143377 | 633.052834 |

*Figure 21: Parallel runtime (computation and communication) with respect to the horizon value.*