CS3210 Assignment 2 Report

Program Design

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

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.

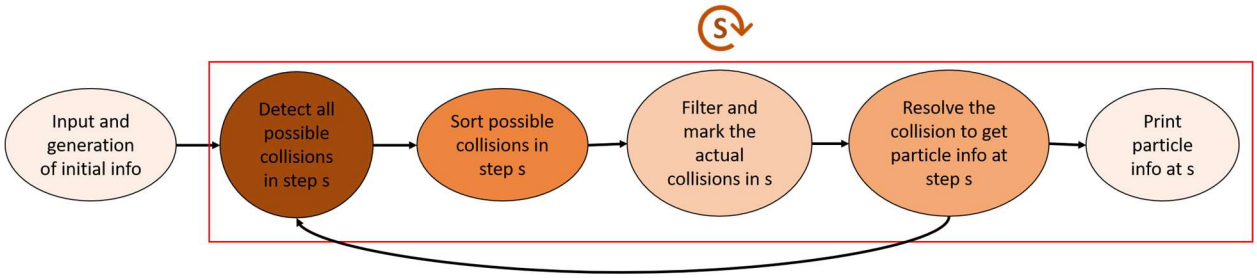


Figure 1. Task dependency graph. Darker shades correspond to higher execution time (estimates obtained via experimentation).

Parallelisation using Open MPI

Across all our parallelisation attempts, the key idea is that we use multiple processes to parallelise the Collision Detection task. The remaining tasks of sorting collisions, resolving collisions and updating particles are done sequentially on a single process.

We considered 3 different approaches to parallelise the Collision Detection task.

1. The “Master-Worker” pattern

On each time step, the master process sends the entire set of particles to every worker process. Each worker uses its rank number to determine the pairs of particles that it needs to process. For each pair of particles in the worker’s “chunk”, the worker performs computations to determine if the pair of particles will collide. Each worker places all its detected collisions into a buffer and sends this buffer back to the master once done processing all pairs of particles in its chunk.

We implemented 2 variations of this approach:

- **Blocking Master:** In this variation, the master uses a single broadcast operation to send the particles to each worker, and blocking receive operations to receive collisions from each worker.
- **Non blocking Master:** In this variation, the master uses multiple non blocking send operations to send the particles to each worker, and non blocking receive operations to obtain collisions from each worker.

2. Ring Pipeline

In this approach, we perform the Collision Detection tasks using a pipeline of M processes, where M is a factor of the number of particles. Process 0 reads the input and initialises a buffer of particles.

On each time step, Process 0 distributes the buffer of particles evenly amongst the M processes. Each process gets a unique partition of the particles, which becomes its “Original Partition”. Each process then performs collision detection for all the pairs of particles in its Original Partition. Detected collisions are stored in a buffer. Next, each process creates a copy of its Original Partition, known as its Working Partition. Each process sends its Working Partition to the process “in front” of it in the pipeline, and then receives a Working Partition from the process “behind” it in the pipeline. After this transfer, each process performs collision detection for all pairs of particles in the cartesian product of the Original and Working Partitions. Once done, the Working Partition is again sent to the process “in front”, and another Working Partition is received from the process “behind”. This continues until the pipeline has run M times. At this point, each process sends its detected collisions back to Process 0, which then proceeds with sorting and resolving collisions.

3. Hierarchical Network

In order to reduce communication overhead of inter-node communication (vs intra-node communication, i.e. within the same node) and to reduce the amount of unnecessary data sent via messages, we also considered a hierarchical Master-slave approach.

Here, we have a top level master, 1 submaster on each node in the cluster, and slave processes on each node that communicate with the submaster of that node. The buffer of particles is broken up into M partitions by the top level master. This master then generates the partition allocations for the slaves. The necessary partitions are sent to submasters of each node. The submaster then sends the required partition(s) to each of its slaves. Each slave then either detects collisions between particles belonging to the same partition (Type 1) or detects collisions between particles belonging to 2 different partitions (Type 2).

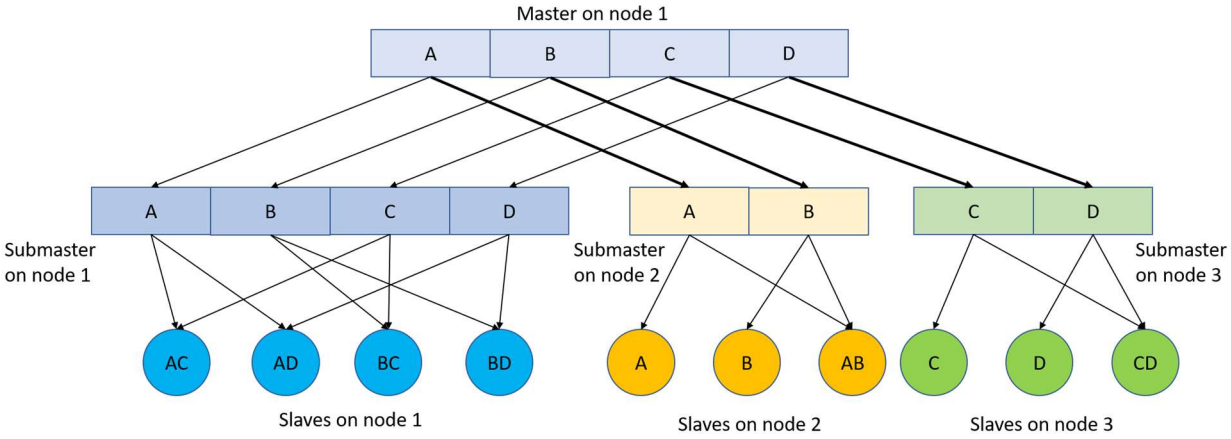


Figure 2. Communication flows for distributing data in the hierarchy pattern. Darker arrows represent inter-node communication, which is more expensive in terms of time. Observe that partitions A and B are only sent once from node 1 to node 2, though they are used in multiple slave processes.

The slaves then send the detected collisions to their submasters, who in turn, forward them to the master. The master then proceeds to sort and filter the collisions, resolve the collisions and updates the velocities and positions of particles. A MPI barrier is placed at the time step so that all processes can begin the next time step together.

If we have n particles and M partitions, MType 1 slaves process possible collisions for $\frac{n\,(n+M)}{2M^2}$ pairs of particles. $\frac{M\,(M-1)}{2}$Type 2 processes do so for $\frac{n^2}{M^2}$pairs. So, Type 2 processes do

roughly twice the amount of work as Type 1, and hence are the bottleneck in terms of execution time of a time step.

In order to get the best possible performance, this paradigm required a highly customised approach and makes a few assumptions that are listed below. Also, the rank file must be is predetermined and should be designed to be the most optimal, in order to achieve the best results (providing a random mapping will not work properly).

Also, this approach is designed to reduce communication time between processes on different nodes. So, the best performance is expected to occur when we have processes spanning across multiple nodes, and not just processes on a single node.

Assumptions for best performance

- Input consists of a sufficiently large number of particles (and it should be at least M).
- Number of processes is $\frac{M\ (M+1)}{2} + K + 1$where M is the number of partitions and K is the number of machines (nodes). This way, we will have 1 master, K submasters and the remaining are slaves.
- Processes are run on **multiple nodes** (as this hierarchy would decreases inter-node communication time).
- Rankfile used for task allocation is optimal in terms of reducing the amount data communication required across nodes and the type of task performed by slaves.

Special Considerations made during Open MPI parallelisation

Binding processes by slot, for the Ring Pipeline

In the ring pipeline approach, data needs to be advanced through the pipeline at the end of each stage. Hence, the performance of each process is closely dependent on how fast it can send data to the process “in front of it” AND how fast it can receive data from the process ‘behind it”.

In order to speed up inter-process communication in the ring, we needed to minimise the number of processes that have to communicate across machines.

Thus, we allocated process by slots, to ensure that most processes send and receive data from processes on the same machine.

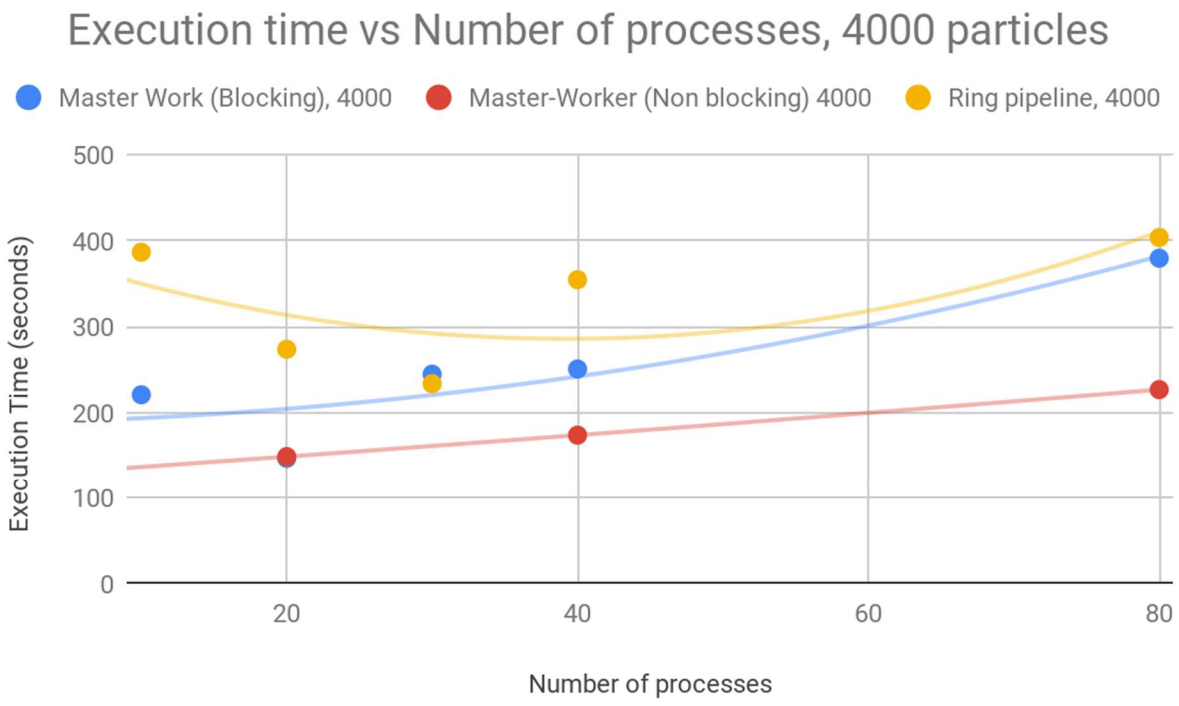
Indeed, this choice yielded performance gains. The table below shows that binding by slots reduced execution time when an input data set of 4000 particles was processed using 40 processes spread evenly across 2 nodes.

Binding processes by node	Binding processes by slots
263s	163s

Results

Execution time versus number of processes on a single machine

For all approaches, the optimal number of processes on a single machine was found to be close to the number of logical cores on the machine, assuming an input data set with at least 1000 particles.



This observation can be justified by the fact that once we use more processes than logical cores on a machine, we have to oversubscribe some of the cores on the machine. This means that

processes must be switched out so that others can make progress. Context switching is accompanied by a variety of performance costs, which leads to performance degradation.

Another observation is that the blocking version of the Master-Worker approach and the ring pipeline suffers more performance degradation as we increase the number of processes, compared with the non blocking version of the Master Worker approach. This is likely because both these approaches depend on all processes to make progress together. In the blocking version of Master - Worker, the Master waits to receive collisions from Worker processes, in order of their rank. This means that time could be wasted if the Master is waiting for collisions from Process 1, which has been suspended as no cores are available. In the non blocking version, the Master is able to accept collisions from **any** process that has completed its work. Similarly, in the ring pipeline, data transmission through the pipeline would be delayed until all members of the pipeline have received CPU time to make progress.

For smaller data sets, the optimal number of processes could be lower than the number of logical cores. This is because using too many processes would cause the task size per process to become highly trivial, which means that the overhead of parallelisation is likely to outweigh any performance gain.

Execution time versus number of logical cores

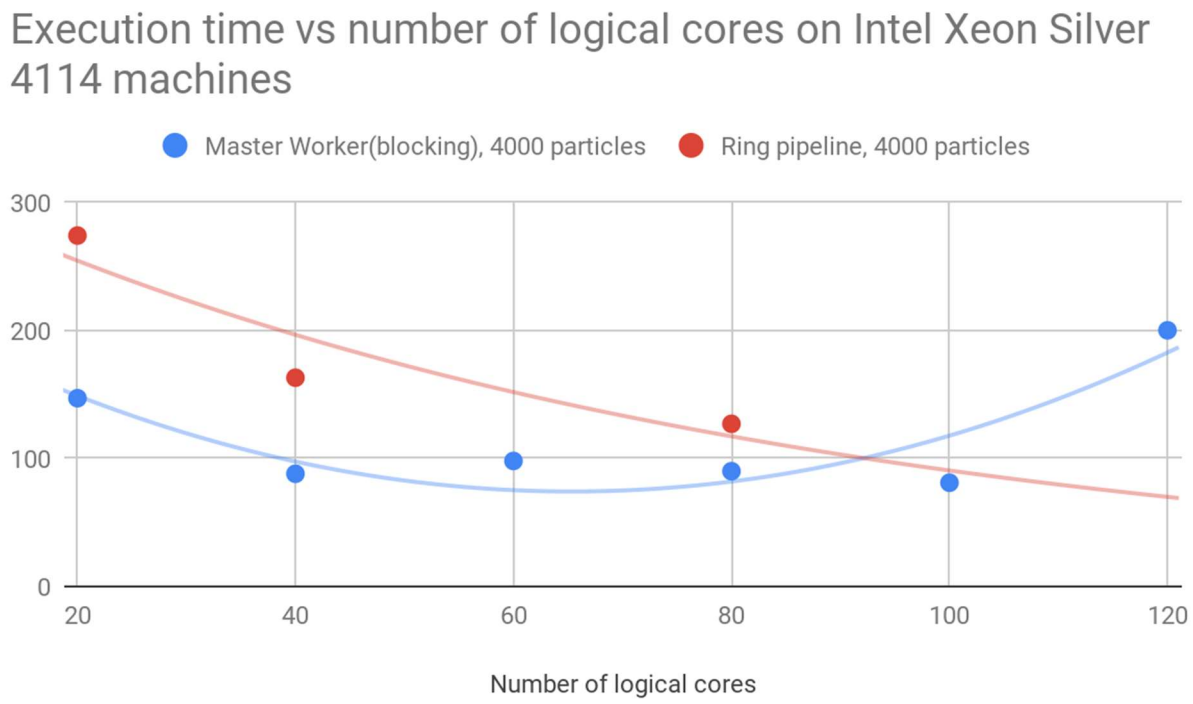
In the previous section, we observed that for a single machine, the optimal number of processes is close to the number of logical cores on the machine (for large datasets).

In this section, we discuss the results of running different parallelisation approaches using several machines, with each machine being allocated one process for each of its logical cores.

In general, using more logical cores increased performance, up till a certain point.

For example, the execution time for the blocking version of Master-Worker was roughly halved when the number of logical cores was increased from 20 to 40. However, adding more logical cores did not further improve performance. This performance “plateau” most likely reflects that the program’s parallel portion was no longer a bottleneck on the overall performance. In other words, any further performance gain would come only from improving the sequential portion of the program. This includes the task of sorting, filtering and resolving collisions.

Performance eventually worsened when 120 logical cores (six Intel Xeon Silver 4114 machines) were used.



Another key observation is that the ring pipeline approach performs worse than the blocking Master - Worker method. The volume of data communicated in each time step is the same in both the ring pipeline and Master - Worker approach. However, the Master Worker approach uses a single large broadcast at the start of the time step while the ring pipeline uses multiple smaller messages at regular intervals of the time step.

This performance difference suggests that Open MPI optimises for larger messages.

Collective communication vs multiple non blocking, point - point operations

When a cluster of machines was used, it was observed that the blocking version of the Master - Worker approach performed much better than the non blocking version.

Blocking version of Master - Worker		Non blocking version of Master - Worker	
Time needed to process 4000 particles, using 2 machines with 20 processes each	2 mins, 27s	Time needed to process 4000 particles, using 2 machines with 20 processes each	> 4 mins

In the blocking version, the master process utilises a broadcast operation to send the buffer of particles to all slaves, on every time step. In the non blocking variant, the master uses multiple point - point non blocking send operations to send this buffer to each process.

The superior performance of the blocking version can likely be explained by the underlying optimisations behind the Open MPI broadcast operation. While the exact behaviour of the MPI broadcast is implementation dependent, it is common for the broadcast to use a virtual tree topology to efficiently transmit data to all processes. On the other hand, because our non blocking version of the Master - Worker pattern sends data directly from the master to each worker process, it uses only the network links available to the master process for each communication.

This implies that the processes in the non blocking Master - Worker implementation face a greater communication overhead. The effect likely becomes more pronounced when the non blocking version is run on a cluster, leading to the poor performance observed.

Indeed, an inspection of the logical cores using `htop` revealed that the processes running under the non blocking approach were spending a significant amount of time in kernel mode. This supports our argument that the non blocking approach causes major communication overhead.

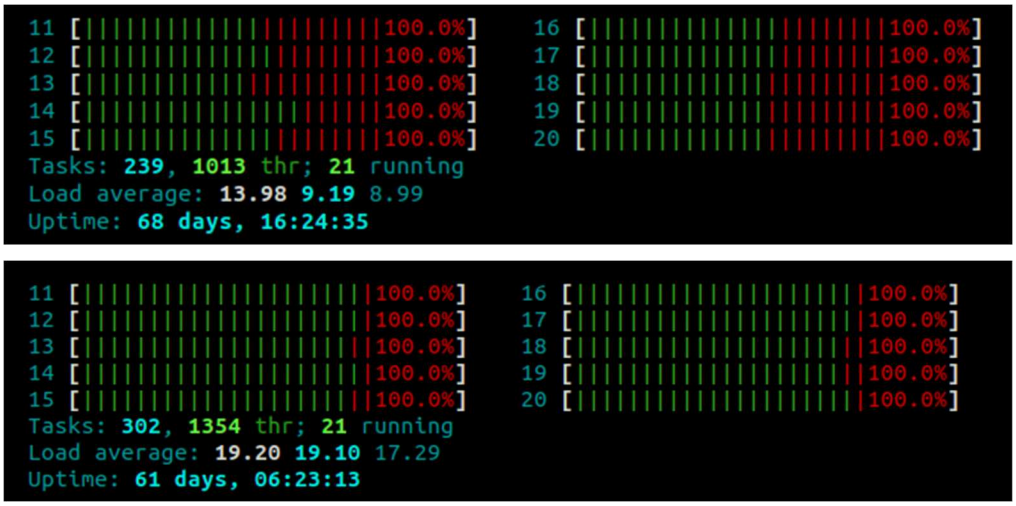
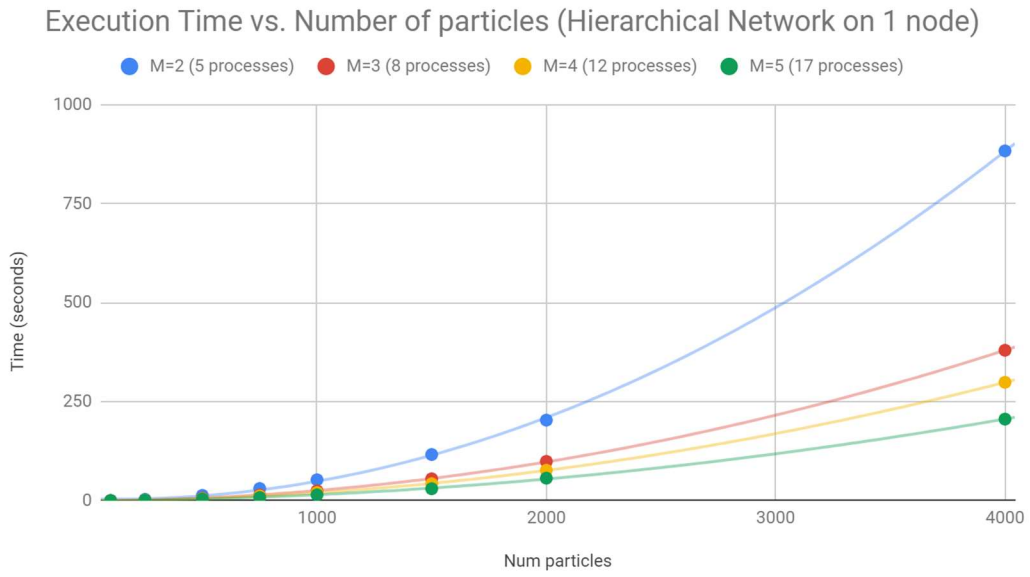


Figure 3: The status of some cores during execution of the non blocking Master Worker variant (top picture) and the blocking Master Worker (bottom picture). The red bars represent time spent in kernel mode, while green bars represent time spent in user mode.

Execution time versus number of partitions

The graph below shows the execution time for different number of partitions, M, on an Intel Xeon machine using the Hierarchical Network implementation. We see that the time decreases

as the number of partitions increases, which consequently decreases the size of the partition (and size of messages). However, this trend may just be due to the fact that more processes are being utilised in parallel.



Due to all the lab machines being occupied for a considerable amount of time, we were unable to collect measurements when this approach was used with multiple nodes. Theoretically, it should perform better than other approaches being used with multiple nodes.

Recap

We fixed a couple of bugs that were present in our earlier implementations:

- Division by 0 error when normalising vectors, which led to `nan` output.
- Earlier, we used the indices of particles in the list of particles to represent the IDs of particles involved in a collision. However, this leads to incorrect output if the input provided consists of particles in a non-sequential order (i.e. when it is not in the order particle 0, particle 1, … , particle N-1).

Parallelisation using OpenMP

We use multiple threads to parallelise the Collision Detection stage. Each thread identifies collisions for a single particle, and keeps this particle's collisions in temporary local list: `currCollisions`. These collisions are concatenated to the global list of `allCollisions` in a critical section to prevent race conditions.

Also, note that line 6 in Figure 3 has a `nowait` clause so that threads do not have to synchronize before appending to the global list of collisions.

```
1  vector<Collision> allCollisions;
2  #pragma omp parallel num_threads(100) shared(input, N, L, r, particles, allCollisions)
3  {
4      vector<Collision> currCollisions;
5
6      #pragma omp for nowait private(i) schedule(dynamic, 3)
7      for(i = 0; i < N; i++)
8      {
9          for(int j = i + 1; j < N; j++)
10         {
11             // Append to currCollisions if particles i and j collide
12         }
13         // ...
14     }
15
16     #pragma omp critical
17     {
18         // Append currCollisions to global allCollisions
19     }
20 }
```

Figure 3. Skeleton code for collision detection- variable names slightly differ from source code (OpenMP).

Parallelisation using CUDA

Create a kernel to execute the Collision Detection task on the GPU. Each thread is assigned a chunk of particles, and checks whether any of these pairs will collide.

On host	On device
- Sorting all possible collisions by time and indices - Assigning zero or one collision to each particle - Resolving collisions of all particles - Updating positions and velocities of all particles - Printing particles' info	- Each thread tries to detect **`chunk_size`** **collisions** - between a pair of particles, OR between a particle and the walls
Memory usage - Unified memory to store particles and collisions - Each thread allocates temporary structures in global memory	

Comparison between OpenMP and Open MPI

Optimal number of processes vs optimal number of threads

As we have shown, the optimal number of processes for a particular machine in a cluster is close to the number of logical cores in the machine. However, the optimal number of OpenMP threads is much larger than the number of logical cores. In our OpenMP implementation, we found the optimal number of threads to be 100, on a machine with 20 logical cores.

This can be explained by the fact that threads are much more lightweight than processes. Notably, threads share the same memory address space. This means that thread context switches have a much lower performance impact on the application (compared with process context switches). This allows us to continue getting performance gains even when the number of threads is a few multiples of the number of logical cores.

Performance on various dataset sizes

Our best Open MPI implementation uses the blocking Master Work approach and it was able to outperform our best Open MP implementation on both small and large datasets.

For larger datasets (4000 particles and more), there is an additional performance gain obtained by running the Open MPI program on a cluster of two nodes.

These results show that Open MPI is quite an optimised implementation of the Message Passing Interface. It is able to minimise the overhead associated with the creation of processes and inter process communication.

However, if we also account for development effort and the infrastructure costs involved in setting up a cluster of machines, then Open MP can be considered more practical. It is able to offer almost as much performance gains as Open MPI with a fraction of the time investment.

Num Particles	Best OpenMP run	Best Open MPI run (20 processes on a single node)	Best Open MPI run (40 processes across two nodes)
100	0.532 secs	0.407s	-
250	1.3 secs	1.181 secs	-
500	4.500 secs	3.146 secs	-
750	9.241 secs	6.371 secs	-
1000	13.099 secs	11.149 secs	-
1500	27.109 secs	22.022 secs	-
2000	47.607 secs	42.259 secs	-
4000	187.514 secs	147.187 secs	88.325 secs

Scalability

Because Open MPI programs can be run on a cluster, they have the potential to scale well with the problem size.

For example, regardless of the number of machines available, the level of parallelism achievable by an Open MP application is limited to the number of logical cores on a single machine. Similarly, an Open MP application is bounded by the memory available on a single machine. This places a limit of the problem size that a particular Open MP application can solve. An Open MPI application, on the other hand, can scale to use the logical cores and memories on multiple machines. This allows Open MPI applications to potentially solve larger problems, by simply making use of more hardware resources.

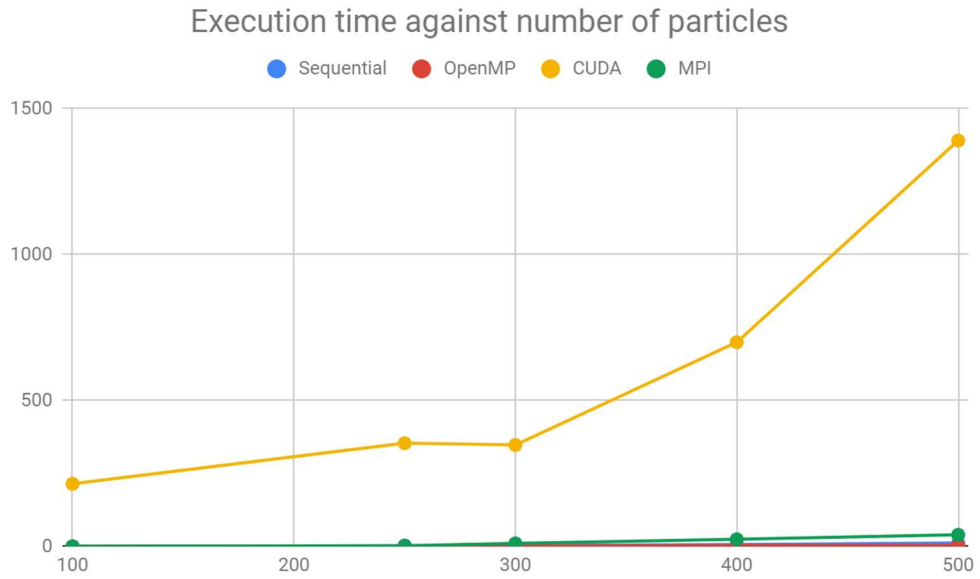
Comparison between CUDA and Open MPI

Flexibility of Parallel Programming Strategy

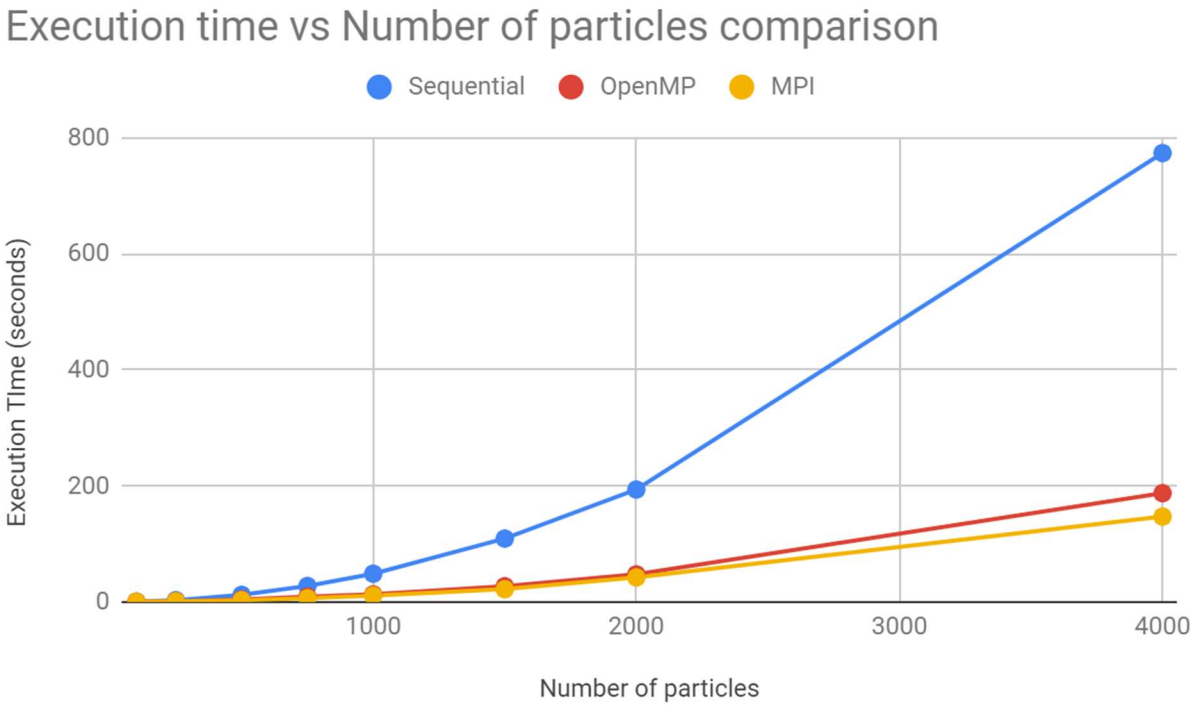
CUDA heavily favours the data parallelism approach for parallel programming, especially the architecture of CUDA enabled GPUs prefers low branch divergence between threads, and sequential accesses to global memory. On the other hand, Open MPI allows more flexibility. As we have demonstrated, it is possible to adopt both data (Master - Worker) and task (ring pipeline, hierarchy network) parallelism in Open MPI applications.

Performance comparison between Sequential and Parallel models

The graph below shows a comparison of sequential and parallel implementations using OpenMP, CUDA and MPI. Due to the nature of GPU execution and the logical complexity of our code, the CUDA implementation suffers greatly (due to divergent threads in blocks, bank conflicts, usage of “slow” unified memory etc.).



The graph below shows the above graph in more detail, zooming in the sequential, OpenMP and MPI implementations.



Overall, although parallelizing a program involves additional complexity and increases code length, the performance gains are worth it in this case.