Parallel Particle Simulation

Au Liang Jun (A0173593W), Yan Hong Yao Alvin (A0159960R)
CS3210 Parallel Computing AY19/20 Sem 1

OpenMPI

15/11/2019

Program Design

Assumptions

Our assumptions remain the same as they were in Assignment 1:

- Each particle only collides once in every step and will phase through subsequent particles.
- After a particle is involved in a collision, if it collides with a wall in the same time step, it will stop with the wall for the remaining duration of the timestep.
- If a particle begins a timestep within another particle, due to only being allowed to collide once in the previous timestep, they will immediately collide at the start of the timestep and follow collision rules as described (even though they are within one another).

Algorithm

The algorithm used in the OpenMP section of the first assignment to perform collisions, as well as the Object-Oriented approach have been retained in our MPI implementation. However, to enable the program to distribute work to MPI processes, implementation changes were made.

Implementation

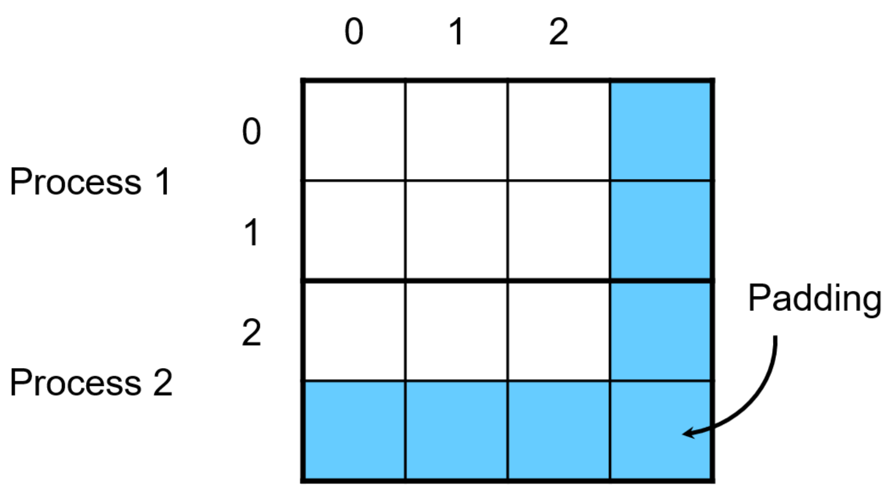
Comparison with OpenMP & CUDA

While we similarly make use of data parallelism on the particles in all of our parallelisation approaches, the granularity of tasks is more coarse in MPI, similar to OpenMP, but different from CUDA. In CUDA, the extremely lightweight nature of threads and large number of available computation cores allows for highly fine grained tasks where each thread only needs to calculate collision times for 1 particle or execute 1 collision event.

However in MPI, creating a separate processes to perform computations for each particle is inefficient as the number of processes needed in communication will increase in order of particle count. Therefore it is beneficial to limit the number of processes and instead increase the amount of work assigned to each process. The granularity of tasks is similar to OpenMP, as in OpenMP the amount of data/work that each thread does is roughly evenly distributed across all threads during runtime. The main differences in the implementation is the need to explicitly specify which data each process performs the task on, and explicit communication to pass data between processes in MPI.

Padding

We make use of data parallelism and ensure scalability with process and particle count by assigning each process an equal number of particles to compute the collision times and execute collision actions for. If the number of particles is not perfectly divisible by the number of processes, we pad additional particles to ensure all processes are assigned an equal number of particles. This is needed to ensure correct communication operations in collective communication operations used as each process must communicate the same amount of information.

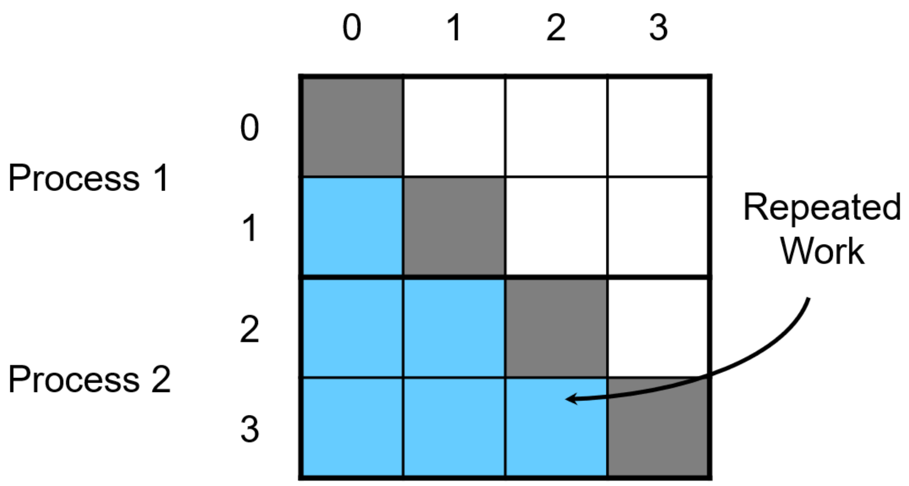


Particle-Particle Collision Times - 3 Particles, 2 Processes

We illustrate our work distribution in the figure above. Given 3 particles and 2 processes, we will attempt to distribute the particles evenly among the processes. As 3 particles is not perfectly divisible with 2, we add padding so that both process 1 and 2 have the same amount of information to communicate. The padded particles are otherwise ignored in our collision detection and execution logic to ensure correctness is not affected.

Repeated Work

We also realised that there exists a tradeoff between reducing the amount of repeated work and communication. For example, consider the calculation of particle-particle collision times.



Particle-Particle Collision Times - 4 Particles, 2 Processes

The time that particle X collides with particle Y is the same as the time at which particle Y collides with particle X. As such, the matrix of particle collision times is symmetric, and only one half needs to be calculated. However, only calculating one half would result in a need to communicate the information to the other half. From the above figure, the collision time at [2][1] is the same as the collision time at [1][2]. If only the white half is calculated, the value at [1][2] would need to be communicated from Process 1 to Process 2.

The same dilemma is presented when executing particle-particle collisions. If the colliding particles belong in different blocks, and the execution of the collision is applied in only one block, the outcome of the collision (resultant positions and velocities) will need to be communicated to the parent process of the partner particle.

To avoid the additional communication overhead, we have elected to simply perform the repeated work in our implementation. We believe that this would not have a large impact on overall run time, given that the upper bound of work that needs to be performed by each process is largely unchanged. In the above figure, we see that each process needs to calculate at most 6 values, up from 5 if repeated work is not performed. This results in a negligible impact to run time as the processes need to synchronise after the calculations are completed anyway.

Communication

Communicator & Topology

In our implementation, we did not see the need to use multiple communicators. The processes in our problem work on data that needs to be shared frequently with all other processes, and as such there is no benefit from the ability to communicate with a subset of processes, which multiple communicators provide.

The all-to-all communication pattern also makes topologies unsuitable for our program. Topology grants communicators a structure for communication. However, since each process communicates with every other process, there is no structure and ordering that can be constructed that would optimise communication.

MPI Calls

As described earlier, each process needs to communicate information with all the other processes. As such, we explored the collective communication functions provided by MPI (operations that involve all processes in a communicator). Collective communication operations are blocking by default. This is suitable for our use case, given that at each stage of communication, all processes will need to receive all the data before it can safely proceed with computation. The following calls were used in our program:

`MPI_Allgather`: `MPI_Allgather` performs a scatter and gather operation that results in every process receiving data from every other process. This is utilised in our program to communicate particle information, which every process requires access to in order to perform particle-particle collision.

`MPI_Bcast`: `MPI_Bcast` broadcasts data from a single process to all processes in the communicator. In our program, the master process reads from the input stream and uses this communication call to distribute the particle data to the other processes.

Updated Pseudocode

The changes performed on our original OpenMP algorithm (`moveParticlesV3()` in the Assignment 1 report) are highlighted in red.

```
moveParticlesMPI():
n: number of particles
k: block size

1. For each particle in the processes's block, compute the time it takes to collide with
   all other particles and the wall. -- $O(kn)$

2. Initialise an empty event found set. Initialise an array called partner, where
   partner[i] = j indicates that particle i found particle j to collide with. Repeat
   while event found set does not contain all particles: -- $O(kn^2)$

    2.1. For each particle in the processes's block without found event, find its
         earliest event: -- $[O(kn)]$

        2.1.1. Check against wall collision times

        2.1.2. For every other particle j without found collision:

            2.1.2.1. Check against particle collision time with j, adding j to the
                     partner array if there is a collision.
```

2.2. Perform an `MPI_Allgather` operation to send the `partner` array for each process to all processes to be used in the next check.

2.3. For each particle in the processes's block, check the earliest event found based off the `partner` array: -- [O(kn)]

2.3.1. If a particle X has partner Y, check that particle Y has partner X. If so, set the particles to collide with each other, and add both particles to the event found set.

2.3.2. Else, particle is set to collide with a wall/not have a collision. Add particle to event found set.

2.4. Perform an `MPI_Allgather` operation to send the found set for each processes'. Each process then independently checks if all particles are in the event found set.

3. For each particle event (collision or no collision) found in the previous step, simulate the movement of the particle for the timestep. Each processes only executes events for particles within its block. -- O(kn)

4. Perform an `MPI_Allgather` operation to update all processes on the new positions and velocity of all particles.

Results

The testing methodology remains the same as Assignment 1, with the same file inputs utilised for the various problem sizes. Our observations are in seconds.

Mode: perf
L: 20000
r: 1
S: 500

Machine Nodes

The experiments were run on `soctf-pdc-001` to `soctf-pdc-008`. Each machine had the following specifications:

Dell Precision 7820
Intel® Xeon® Silver 4114[1]
10 Cores, 20 Threads
2.20GHz Base Frequency, 3.00GHz Turbo

Run Time With Varying Machine and Process Count

Problem Size (N = 5000)				
Machines	**1**	**2**	**3**	**4**
Processes				
4	209.537	192.689	192.296	192.357
8	113.022	105.723	108.105	100.346
16	102.558	63.244	63.941	62.036
32	119.892	53.626	54.007	41.178
64	112.651	84.534	66.353	54.137

Run Time With Varying Machine and Problem Size

20 Processes per Machine				
Machines	**1 (20 processes)**	**2 (40 processes)**	**3 (60 processes)**	**4 (80 processes)**
Problem Size (N)				
1000	**3.927**	4.604	10.691	23.037

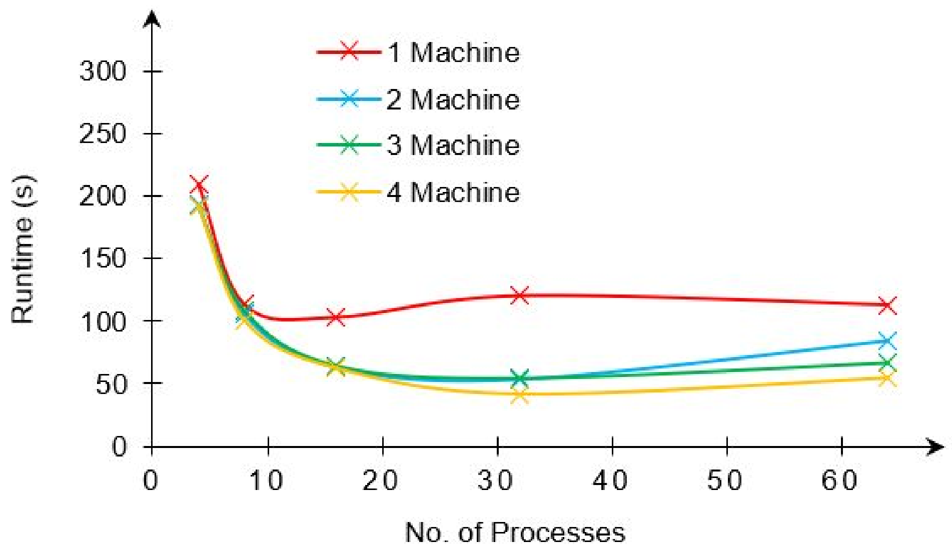
[1] "Intel® Xeon® Silver 4114 Processor (13.75M Cache, 2.20 ...." https://ark.intel.com/content/www/us/en/ark/products/123550/intel-xeon-silver-4114-processor-13-75m-cache-2-20-ghz.html. Accessed 29 Sep. 2019.

2000	12.954	9.883	**9.032**	8.5375
3000	29.086	18.413	15.921	**14.897**
4000	51.932	30.549	25.008	**22.978**
5000	82.706	44.432	39.375	**38.774**
6000	119.078	**62.613**	86.934	86.367
7000	161.378	**85.757**	125.306	123.726
8000	211.586	**112.185**	162.937	184.759
9000	266.251	**140.901**	182.817	183.051
10000	330.176	**172.470**	229.822	220.302

Note that the best run time for each problem size is bolded and underlined, and is later used in our comparison with OpenMP and CUDA. The best time is used as we would like to compare the different parallel frameworks under optimal configurations.

Analysis

Run Time against No. of Processes (N = 5000)



Given a fixed number of machines (a single line), run time starts high before dipping down to a minimum and rising again. This is likely due to the fact that at a low process count, the amount of work completed by each process is relatively high as compared to the communication overhead. As we increase the number of processes, the amount of work performed per processor decreases, reducing the execution time of the program. However, more communication also has to be performed given that there are more nodes in the communicator. This increase in communication overhead eventually outweighs the reduction in per-process workload, and the running time now increases if the number of processes is increased.

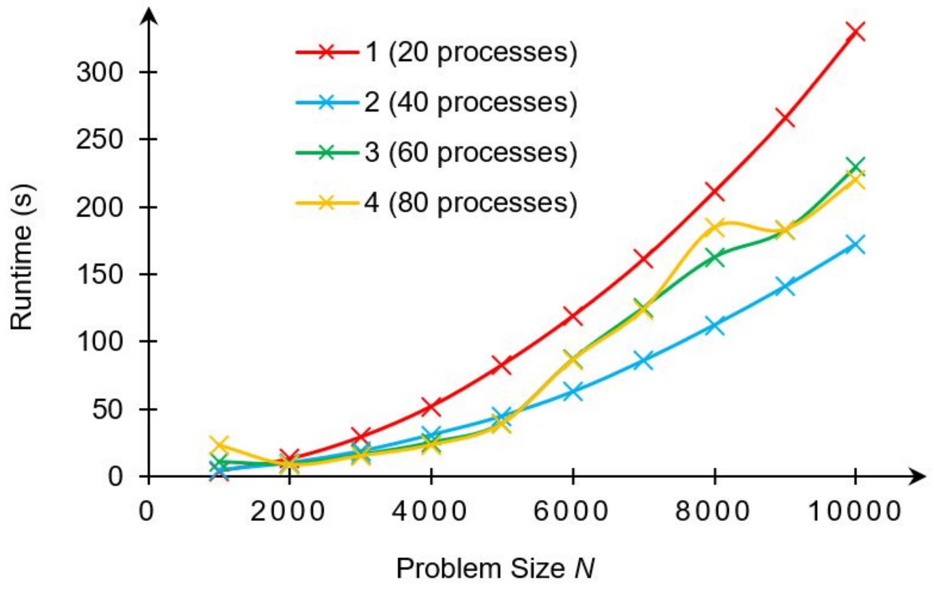
Between the different machine counts, we notice that in general, fewer machines result in greater running time, across all process counts.

- At 8 processes: The running time is similar across machine counts, with more machines taking marginally less time. This is noteworthy, as each individual machine has 10 physical cores and 20 logical cores. As such, it would be reasonable to expect that running 8 processes on a single machine would be faster than running 8 processes on multiple machines, given that the number of cores on a single machine is not exhausted, and communication between processes should be faster within a machine than between machines. However, it appears that the communication overhead between machines is not significant, given that the opposite trend was observed.

- At 64 processes: At a higher process count, the trend observed is expected. On a 20 thread machine, running more than 20 processes would result in the process having to perform time-sliced execution, increasing the overall execution time of each process. Introducing more machines would reduce this effect and allow for faster overall execution. Once again, this also shows that communication overhead is not significant enough to outweigh the benefits of splitting the work across machines.

Although subtle, we also observe that the minimum point of each line gradually moves to the right as we increase the number of machines. This is to be expected, given that with a higher number of machines, the point at which increasing the number of processes ceases to benefit execution time would be higher. More physical cores would mean that more processes can execute in parallel.

Run Time against Problem Size



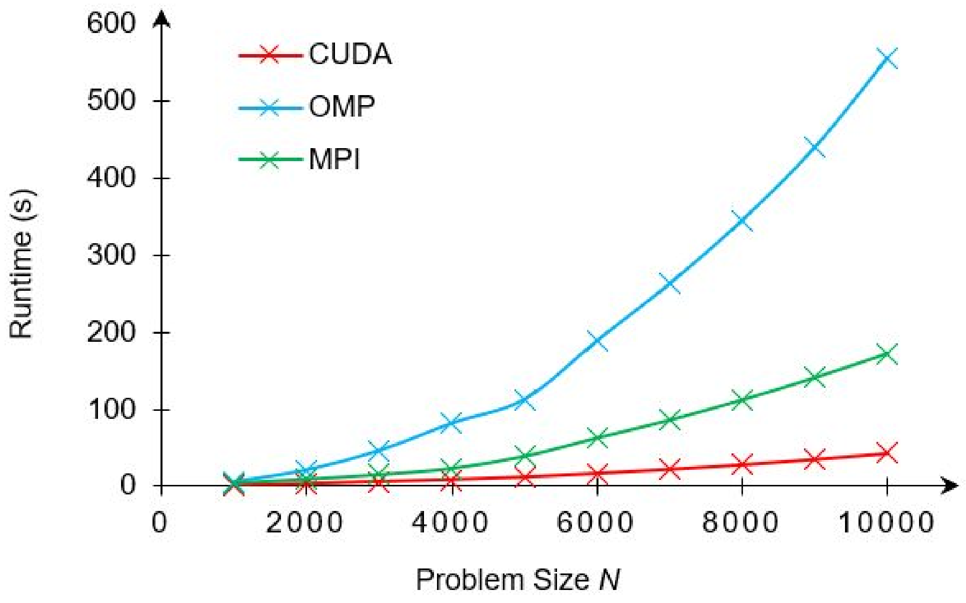
One important observation is that time taken can increase as we increase the process count as seen between 2 and 3 processes in the graph above. This is a phenomenon not observed in the OpenMP or CUDA implementations, which experienced increased speedup as more hardware resources were used. This indicates that the overhead of using more processes outweighs the speedup obtained from reducing the amount of computational work for each process at some breakpoints. As we make use of all-to-all communication in our implementation, the communication required increases proportionally to the process count.

- For N = 1000, the run time increases as processes used increases beyond 20. As the problem size is small, each task only needs to perform computations for a small number of particles. The computation then becomes small relative to the communication and leads to the increased overhead of using more processes to be apparent quickly.

- For N >= 5000, the execution time of the program is longer when using 60 processes (cores) compared to 40 processes. One possible reason why this is observed for larger problem sizes is that more information needs to be communicated. This could exhaust the capacity of the buffers used, which requires the processes to block in the communication step for longer while data is sent and the buffers are available again.

We also notice that between one machine (red line) and multiple machines, the time taken for one machine grows at a faster rate as problem size increases. This can be attributed to the fact that in a single machine, the communication overhead is a small percentage of run time, as compared to the actual program execution. As such, as the amount of work increases, a single machine is affected more than a multi-machine configuration.

Comparison with CUDA and OpenMP



When comparing the results of our simulation implemented on the 3 different parallel frameworks, we find that CUDA has the fastest runtime and OpenMP the slowest. MPI is significantly faster than OpenMP. This shows that there is a benefit to using a larger number of processes across multiple nodes in a distributed memory framework, even with the increased overhead of communication between processes.

This program is more suited to CUDA as compared to MPI as the data parallelism in this problem requires a significant amount of memory to be shared and made consistent across all threads or processes running the program to ensure correctness in simulation. This gives a large overhead to the MPI processes compared to CUDA, and this is reflected as the CUDA implementation is much faster than the MPI implementation.

Possible Enhancements

We used only blocking communication in our program, which can be inefficient as some processes may reach the communication earlier than others and idle there. There exists some room for using non-blocking communication and overlapping communication with computation in the program. One possible location is in computing the wall and particle collision times. We can perform a non-blocking Allgather operation, and have each process compute wall collision times and particle-particle collision times involving only particles within its block. Then, each process could then wait for the updated particle locations and velocities to be received to correctly compute the remaining particle-particle collision times.

Another possibility allowing for further overlapping of communication and computation would be at the end of the `moveParticles()` function, when updated particle information is shared. Instead of the blocking Allgather, each process can perform asynchronous send of its block of particles to all other processes, then perform asynchronous receive of updated particle information from all other processes. Then the process can loop through to check for received messages and begin computation of particle-particle collision timings for the next timestep, with particles which updated positions and velocities have been received. This requires every process to communicate with every other process however, which may be more inefficient than an Allgather communication.

Conclusion

To recap on the conclusion made in the Assignment 1 report, we believe that the particle-collision problem is more suited to CUDA than OpenMP, due to the large number of identical operations to be performed (exploitable by a GPU). We also felt that this observation would extend to other problems, barring scenarios where there are large sections of task parallelism, or if there exists development or hardware limitations such that CUDA programming is not feasible.

On the other hand, the problem is largely unsuitable for MPI. MPI provides a structured and refined way to communicate across independent processes on distributed nodes, a powerful tool wasted on this problem's need to communicate all-to-all at every stage. The problem essentially requires global access to all data, and an MPI implementation would largely be simulating the capabilities of shared memory. As a result, our MPI implementation was no match for CUDA in terms of execution time, which as covered is perfectly suited for such a problem. Between MPI and OpenMP, while MPI yields better results in our experiments, it does so through the sheer effect of increasing available hardware resources. However, this introduces large communication overhead, which is arguably an inefficient use of resources. We thus believe OpenMP is inherently more suitable for this problem, on top of being easier to program for it.

However, there are also situations where a problem like this can only be served by MPI. When the problem size grows beyond the storage and computational ability of a single machine, MPI offers a way to split the work up across independent nodes. A single shared memory unit would also not be able to serve many CPUs without introducing a significant amount of contention. In such situations, distributed memory would prove advantageous, even if the data stored in each memory node is identical.