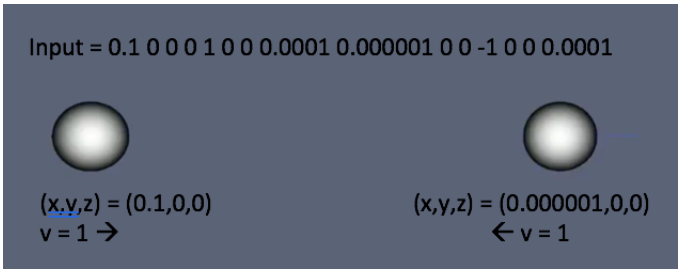
mdsw22

Numerical Algorithms and Parallel Computing Summative

Step 2: Video of 2 body collision: https://youtu.be/uXW8r6olGNo



Final **exact** position of merged particle is calculated to be: x=0.0000005.
Mass of particle 1 and 2 = 0.0001

Non-adaptive time-stepping

Time Step Size	**COLLISION**	**Absolute Error** (exact position – obtained position)	**Relative Error** (Absolute error / exact position)
1e-6	DOES NOT COLLIDE	N/A	N/A
1e-7	DOES NOT COLLIDE	N/A	N/A
1e-8	DOES NOT COLLIDE	N/A	N/A
1e-9	DOES NOT COLLIDE	N/A	N/A
1e-10	0.00000049999999837621142180939	1.2623788351931171e-15	2.5247576703862342e-09
1e-11	0.00000049999999837621353939176	1.2623786234348803e-15	2.5247572468697606e-09
1e-12	0.00000049999999837631094818069	1.2623688825559869e-15	2.5247377651119738e-09
1e-13	0.00000049999999837635848225018	1.2623641179956586e-15	2.524728235991317e-09

As time step size is reduced, accuracy increases (absolute error and relative error decreases) however execution time increases and so there is a trade-off whether you want accuracy (minimising error) or speed of execution.

Adaptive time-stepping
Epsilon = 0.0001 and Initial Time Step Size = 1e-6

Position after collision	**Error**
0.00000049999999837628236081872	1.2623717412921839e-15

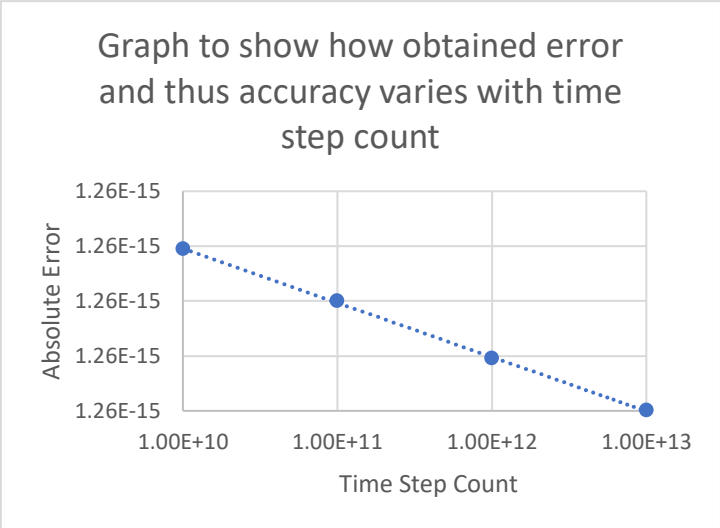
Adaptive time stepping results in a collision despite having an initial time-step size of 1e-6 as well as completing execution in a much faster time than using non-adaptive time-stepping. Gives a position in between that obtained with 1e-10 and 1e-11 fixed time step size however in a much lower execution time.

Convergence Order $|F^{(i+1)} - F^{(i)}| \leq C|F^{(i)} - F^{(i-1)}|^p$

Fixed p for p=1, repeatedly calculated C for various i and this gave me a constant value. Doing it for p=2 did not. (calculations not shown due to lack of space). A sequence converges with order q to L if: $\frac{|x_{k+1}-L|}{|x_k-L|^q} < M$ as k → infinite. Plugging in value for X_k and X_{k+1}, I obtained a value for the convergence order of q=1. Result confirmed.

Compare accuracy to cost in terms of time step count
In numerical analysis, order of accuracy quantifies the rate of convergence of a numerical approximation of a differential equation to the exact solution.

Figure 1



These results say that accuracy increases (error decreases) as time step size decreases (time step count increases) however the rate of accuracy increase is diminishing. Thus, there is a tradeoff between time-step count (computational cost) and accuracy.

Logarithmic x-axis illustrates the fact that the increase in accuracy gets less and less as time-step count is increased.
(see Figure 7 at end of document for linear axis plot)

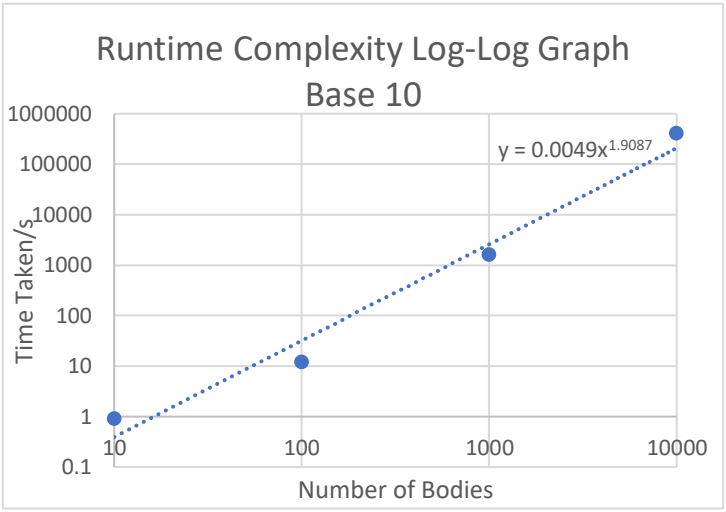
2b) Run Time Complexity Derivation

Due to every particle having to do calculations between themselves and every other particle, the time complexity will be quadratic (O(N2)). This is because, in a nested for loop, the number of steps will be (n-1)+(n-2)+...+0, which rearranges to the sum of 0 to n-1 which is $T(n) = \frac{(n-1)((n-1)+1)}{2}$. Thus, T(n) will always be $\leq 1/2(n^2)$; by the definition, thus T(n) = $O(n^2)$.

Experimental Algorithmic Complexity (averaged over multiple runs)

Number of particles	Time Taken (s)
10	0.887303
100	22.61550
1000	1160.955
10,000	130493.332

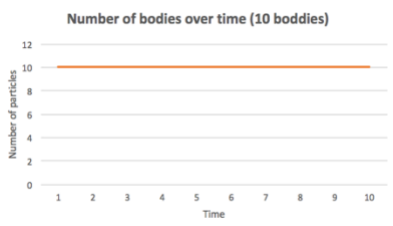
Figure 2

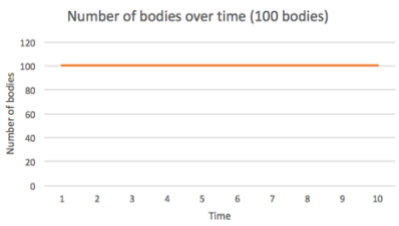


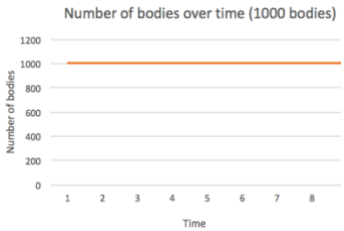
Data is plotted on both a regular scale graph as well as on a log-log graph base 10. This plot illustrates the fact that the complexity is $O(N^2)$ as the number of particles is increased by a factor of 10 in each run. *(See Figure 8 at end of document for linear axis plot)*

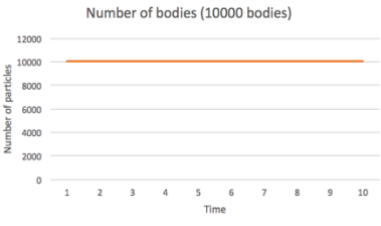
Equation of graph: $y = Cx^n$ where n = 2 would be a perfect $O(n^2)$ relationship. My graph has a gradient of **1.91** which proves the fact that my algorithm is at **worst** $O(n^2)$. Thus, derivation of run-time complexity **proven** with experimental results.

2c) Number of particles over time









Over multiple repeated tests, I did not observe any collisions with experiments involving any number of particles. This is due to the very small distance required for two particles to collide. If I was to repeat this experiment I would increase this distance for more interesting results. (*NB in graphs above I used an x-axis scale of time with no units as an arbitrary measure of time)*

Step 4: Scaling Experiments: Scaling Plots (1,2,3,4 cores) Averaged over multiple runs (x5)

Machine Specification:
MacBook Pro 2016, Processor 2.7GHz Intel Core i5, Memory 8GB 1867 MHz DDR3, 4 Cores
(Long time taken as using very small time-step size)

Number of Particles	1 Core Execution Time /s Average	2 Core Execution Time /s Average	3 Core Execution Time /s Average	4 Core Execution Time /s Average
10	0.041749	0.701793	1.198076	1.221685
100	7.197672	5.564444	4.598785	1.843821
1000	1160.955	720.854967	583.639939	525.778739

Figure 4

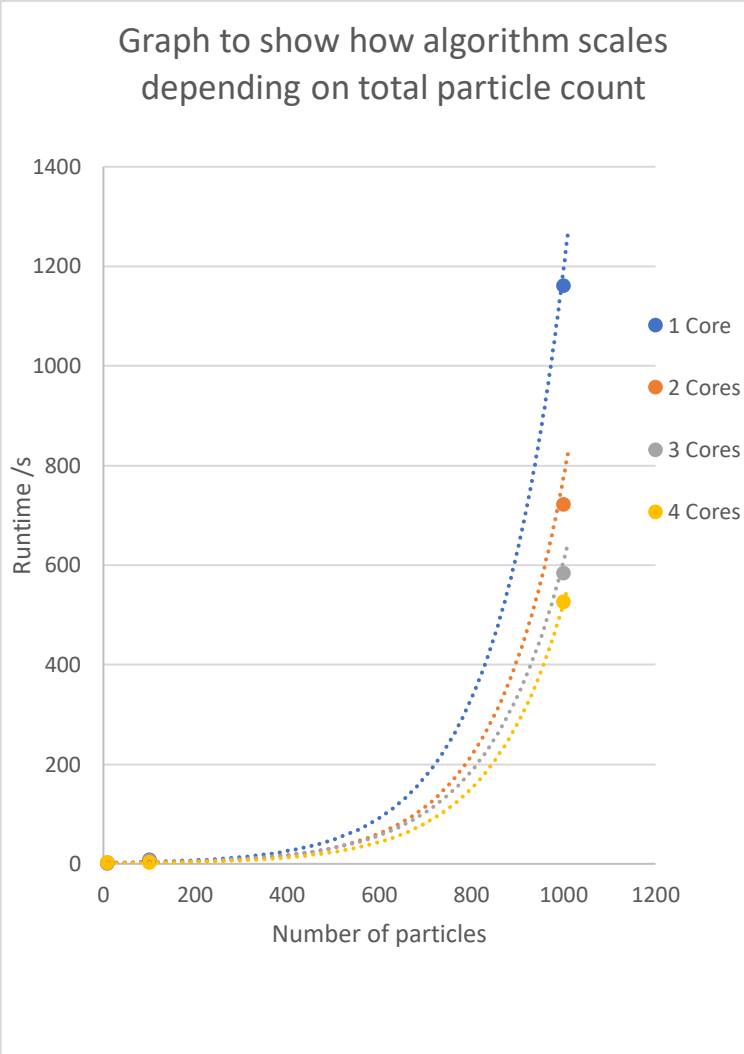


Figure 3

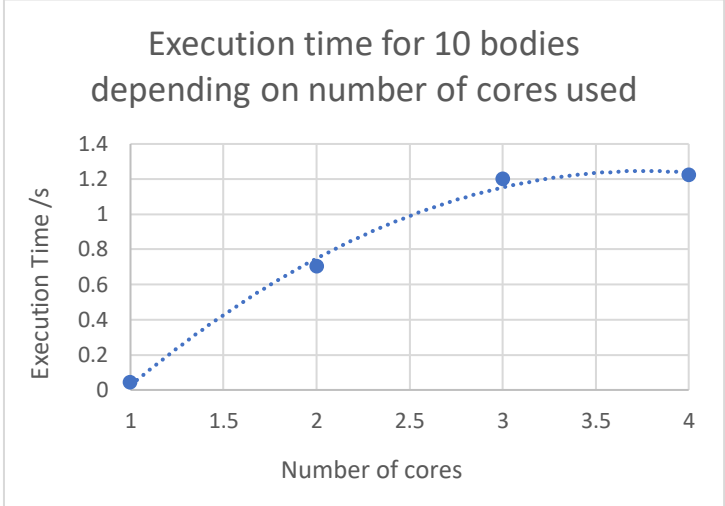
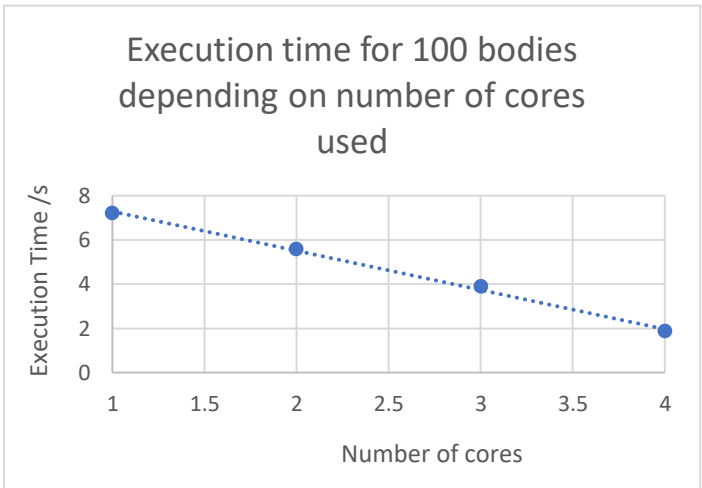


Figure 5



1. *How does the scalability for very brief simulation runs depend on the total particle count?*

For very brief simulation runs (final time set to be 0.1s.), the scalability of the algorithm over multiple cores increases as the total particle count increases. Figure 3 shows that when the number of bodies is small (10), an increase in the number of cores results in an increase in the execution time. This is due to the overhead required in setting up the multithreading which overshadows the benefits of splitting the workload over multiple cores. In contrast to this, when the number of bodies is increased (>100), running on multiple cores reduces the execution time. (See figure 4 and 5).

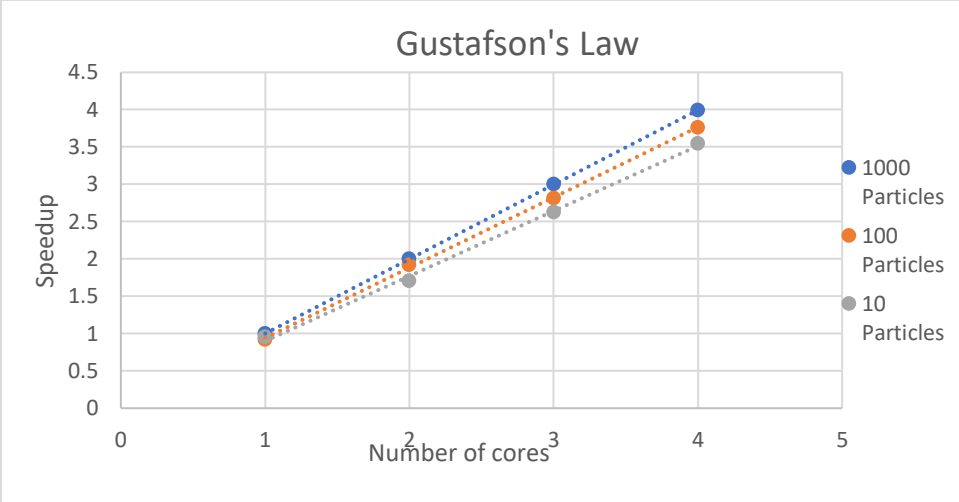
2. *Calibrate Gustafson's law to your setup and discuss the outcome. Take your considerations on the algorithm complexity into account.*

Algorithm complexity $O(n^2)$.

$t(1) = f.t(p) + (1 - f)t(p)$ where f is the part of the code that runs serially, (1-f) is the part that runs in parallel and p is the number of cores/processors. We assume that original problem is chosen such that whole machine is exploited, i.e. problem size is scaled and the computation time is kept constant: weak scaling. Thus, increasing number of CPUs in order to solve bigger problem sets in the same time.

Let t(1) be the time taken by one processor. t(p) is the time required by p processors (i.e. the parallel system), the speedup, $S(p) = f + (1 - f)p$.

Figure 6



Gustafson's law is saying that if the workload is scaled up to maintain a fixed execution time as the number of processors increases, the speedup increases linearly. The graph above illustrates this law well. (table of results not shown due to lack of space)

I set constant execution time = 30s.

3. *How does the parallel efficiency change over time if you study a long-running simulation?*

Number of cores	Small tFinal = 0.1s	Big tFinal = 10.0s	Big tFinal/ small tFinal
1	7.197672	139.426895	19.37110985
2	5.564444	105.434694	18.944833
3	4.598785	83.343343	18.129393

As you can see in the table below (for 100 particles), the ratio in the final column decreases as number of cores increases. This illustrates the fact that, for long running simulations, using multiple cores increases parallel efficiency. When tFinal = 10, having more cores is beneficial to the efficiency of the execution of the program. From this data, we can conclude that the more cores used in parallel programs, the more efficient the program will be, until the point that the execution time is so short that the overhead in setting up the multithreading overcomes the increase in parallel efficiency.

Step 5: Design a strategy how to parallelise your code with MPI.

A distributed memory interface would allow communicating processes to divide the data equally among all processors. The 'central communicator' would be responsible for particle allocation to processes. For each particle 1 to n, the responsible (root) process would broadcast the necessary information (mass, position) to each other process and all would calculate the distance and thus the force imparted between their particles and the particle in question. This can be done with the use of MPI's collective operation **MPI_Bcast** which broadcasts a message from the process to all other processes in the group (one to all). Note that all MPI collective operations are blocking which means that the operation terminates as soon as you can read the buffer. Moreover, blocking collectives always synchronise all ranks, i.e. all ranks have to enter the same collective instruction before any rank proceeds, eliminating the need for a barrier.

Once all processes have finished at this stage, the partial forces would be gathered at the process that previously broadcasted the necessary data and combined (in a sum) to result in the total force on that particle. This can be done with MPI's collective operation **MPI_Reduce** which applies a reduction operation on all tasks in a group and places the result in one task (all to one). The other processes would then wait for this one to update the state of the particle, before continuing on with the next particle (synchronisation).

MPI functions become available through the header <mpi.h>. MPI code requires explicit initialisation and shutdown of the MPI environment. Thus, at the beginning of our main function one must write: MPI Init(&argc, &argv) which initialises argc and argv. And at the end: MPI Finalize().

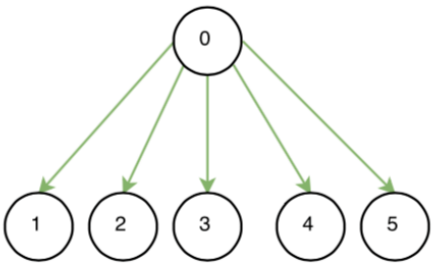
MPI abstracts from processes/threads and calls each SPMD instance a rank (continuous numbering starting from 0). The total number of ranks is given by size.

In the main function:

```
int rank, size;
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
```

MPI_Comm_size determines the total number of involved and MPI_Comm_rank determines the processes own identification, ranges from 0 to N-1. MPI_COMM_WORLD is the predefined communicator that includes all MPI processes. (Communicators and groups are objects that are used to define which collection of processes may communicate with each other).

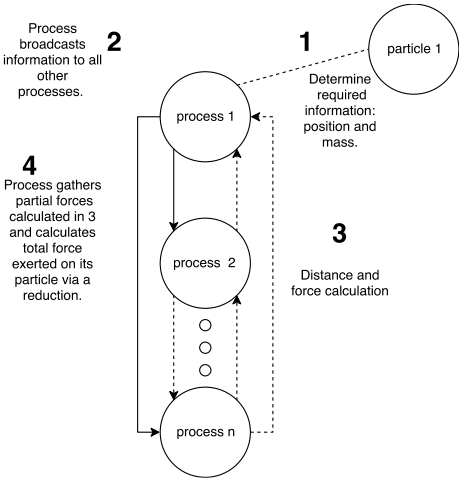
Broadcast




```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator)
```

Data here is a pointer to the position and mass information the root process is to send to all other processes. Count is the number of items to be send. Datatype is MPI_DOUBLE. Root is process 1 and the communicator is MPI COMM WORLD.

This diagram illustrates the process involved in the explicit message passing between processes in MPI.



Here the root process is process 1 which broadcasts its information to all the other processes before gathering all the partial forces and combining them in a reduction to get the total force. In between broadcast and the reduction there is a requirement for all processes to be synchronised (which is achieved with all MPI collective operations).

Other Videos

Video of 10 body simulation: https://www.youtube.com/watch?v=_KPsyCwwUiI

Video of 100 body simulation: https://youtu.be/XFUiEPleQm0

Video of 1000 body simulation: https://youtu.be/ujU34Oxuj4M

Extra Graphs (Sections 2 and 4)

Figure 7

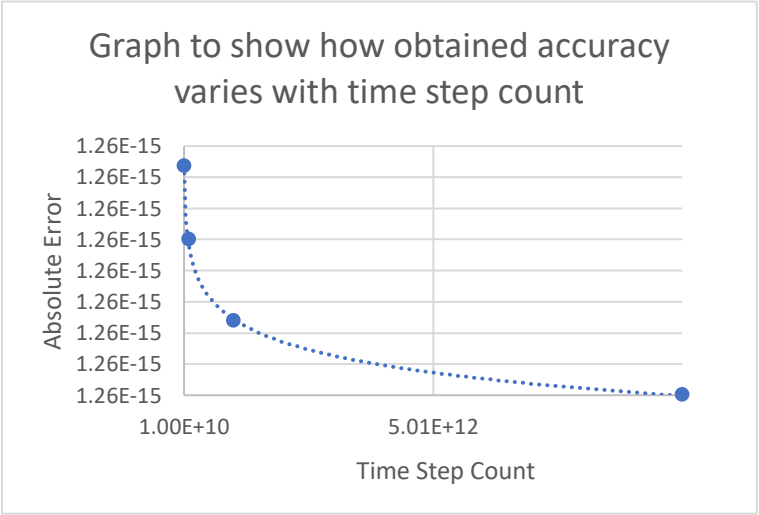


Figure 8