Space Bodies Assignment

cmkv68

February 19, 2018

Numerical Experiments

Consistency & Convergence

The table below describes the information related to the two bodies used for the collision experiment.

Body ID	$s(x)$	$s(y)$	$s(z)$	$v(x)$	$v(y)$	$v(z)$	Mass
1	+0.1	+0.1	+0.1	-2.0	-2.0	-2.0	$1e^{-11}$
2	-1.0	-1.0	-1.0	+2.0	+2.0	+2.0	$1e^{-11}$

Since the mass is negligible, its effect on the force is also negligible. This helps to easily predict where the bodies will collide; it occurs at $(x, y, z) = (-0.45 - 0.45 - 0.45)$ at time $t = 0.2$. The error is calculated through calculating the difference between the position of one body and the intended position upon collision. The following table shows the timestep used, and the error value left over. We calculate the error of only one dimension; x, as the other dimensions (y, z) would follow the same values. The adaptive timestep has a baseline value of 10^{-4} and reduces contingent to how close two bodies are close to colliding towards each other. Adaptive holds a hard limit of 10^{-10}.

Timestep i	Error F	x_a	x_b	Steps	Error Range	C
Adaptive	0.000001997990	-0.449998	0.500503	1716457	0.000003994500	N/A
$10^{-6}/2^1$	0.000001999990	-0.449998	0.500003	275000	0.000003998500	N/A
$10^{-6}/2^2$	0.000001000020	-0.449999	0.499992	550000	0.000001998520	1.9997
$10^{-6}/2^3$	0.000000499961	-0.45	-0.45	1100000	0.000000998475	2.0010
$10^{-6}/2^4$	0.000000250068	-0.45	-0.45	2200000	0.000000498564	1.9993
$10^{-6}/2^5$	0.000000125078	-0.45	-0.45	4400000	0.000000248608	1.9857
$10^{-6}/2^6$	0.000000062134	-0.45	-0.45	8800000	0.000000123183	2.0701

A numerical approximation is used by determining the order experimentally. We fix $p = 1$ i.e. we have a linear convergence. We use the formula $|F^{(i+1)} - F^{(i)}| \leq C|F^{(i)} - F^{(i-1)}|^p$ to compute multiple constants C_i. The objective is to show that C_i is roughly equal to C.

The different C values ($C = \frac{u_h - u_{h/10}}{u_{h/10} - u_{h/100}}$) computed shows a approximation around $C \approx 2$. This allows us to confirm that the code converges in a linear fashion.

The adaptive timestep uses the base timestep of 10^{-6} to a possible maximum limit of 10^{-9}. We can see that the adaptive timestep uses more iterations than $h = 10^{-6}/2^1$ to reach the collision but produces a similar error, but this is due to the initial positions of the bodies. Choosing a larger range will help.

However, choosing a larger initial distance between the bodies would take many more iterations to produce results for the static timesteps.

Complexity

Note: *A seed for the Random Number Generator is used to ensure that the sequences of non-repeating numbers used to generate the random bodies are consistent and repeatable regardless of the body size. This is used to generate random but consistent bodies.*

Under the assumption that the timestep and time limit is fixed, the most dominant function `updateBodies()` which utilises a nested loop that iterates through the number of bodies initiated. For each iteration, a force for a given body is calculated by comparing its position against every other body in space. This results in `updateBodies()` to run in $O(n^2)$.

Procedures have been taken to reduce the constant; Each body only calculates its force against bodies that precede them in the order of initiation i.e. Body 2 calculates force from Body 1 and Body 0 whereas Body 3 calculates from $0, 1$ and 2. Various other improvements tend to consist of trading computation time for storage.

Whilst `updateBodies()` would continue to run in $O(n^2)$, the hidden constant would be drastically reduced to a factor of $\frac{1}{2}$ of the original number of calculations needed. There are also various optimisations are the compiler level that are not considered during the calculation, making it more difficult to create a fair comparison between real data and theoretical outcomes.

Statistics

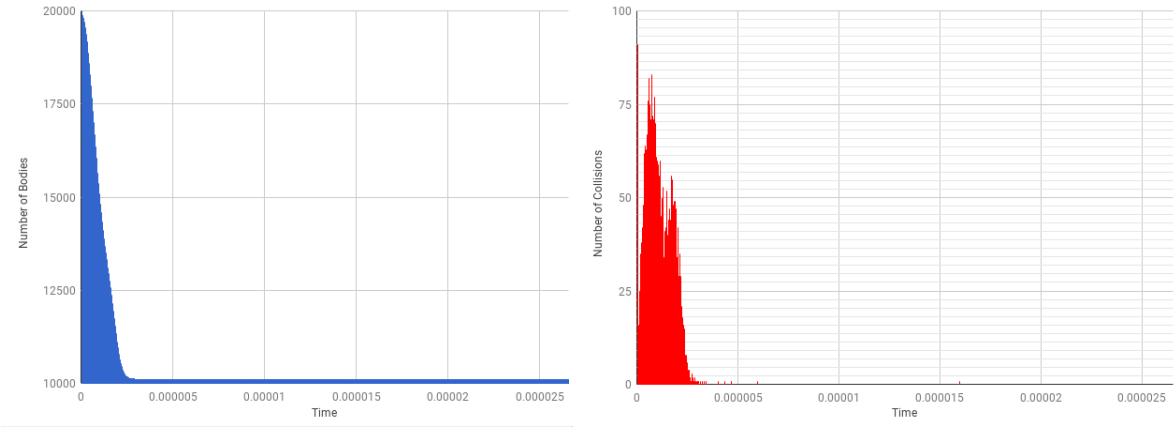

Figure 1: A table showing bodies and collisions over time.

Due to the nature of the bodies interacting with a force, collisions are unpredictable and are likely to fly away as they progress over time. For the sake of the simulation, each randomly generated body (using the seed mentioned prior) has a value ranging from -0.000001 to 0.000001 for all of its displacement attributes (s_x, s_y, s_z) and its velocities (v_x, v_y, v_z). The mass for each body would be infinitesimally small (10^{-11}) to reduce its effect upon force generation. We use a 20,000 body simulation in order to increase the chances of collisions. Adaptive time stepping is enabled. The base timestep is 10^{-7}.

A good portion of bodies are merged upon spawning; the first time step shows a considerable number of collisions due to the high probabilistic outcomes as the density of bodies in a small area affords.

Simulating the outcome using a larger range makes it unlikely for a collision to occur.

Issues were faced in terms of running the simulation on Paraview due to the wide spectrum of values produced; i.e values are range from -10^{-8} to 10^{21}. It is also noted that in the event of collisions during collisions, it reduces the number of operations as bodies are fused, improving the running time.

Scaling Experiments

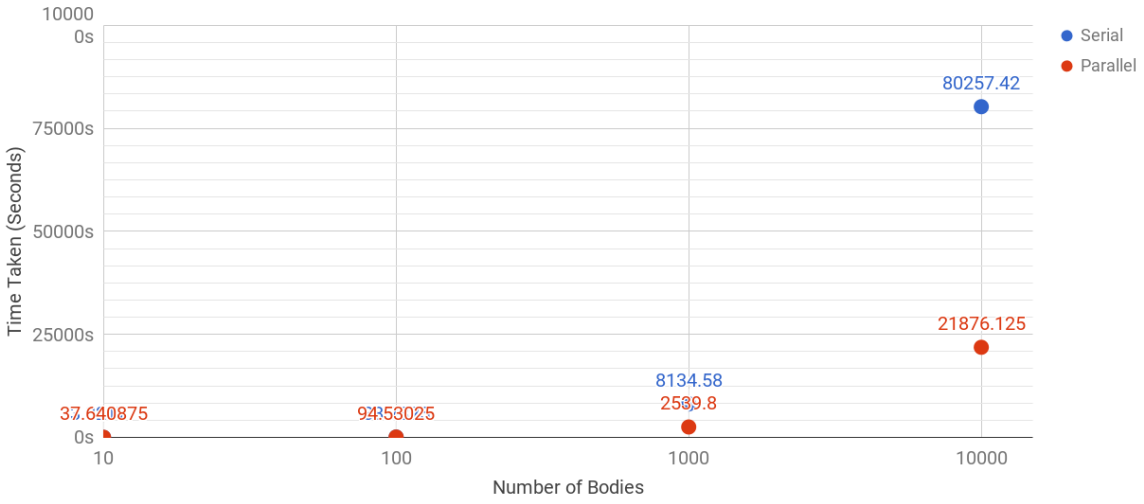

Figure 2: A scaling plot of serial vs parallel runtime for various body sizes.

To ensure that parallel modifications did not break the code, an MD5 sum of the Paraview files computed from both parallel and serial simulations are used to verify any difference in results. The personal computer used consists of a `Intel i7 3770k` processor at a 3.7Ghz clock speed, powering 4 cores (and 8 threads). It utilises 32GB of memory and the storage consists of a SSD hooked up via SATA3. It is using a fresh installation of Ubuntu 16.04 LTS and has no other additional programs running. Adaptive timestepping is not utilised as the serial simulation would take too much time, especially in the case for 10,000 bodies. The results are shown in the table below. Parallel programs will utilise the full 4 cores/8 threads.

Type	CPU Time	Real Time (ms)	Real Time
Parallel	10	301.127	37.640875
Serial	10	3.32177	3.32177
Parallel	100	756.242	94.53025
Serial	100	88.7825	88.7825
Parallel	1000	20318.4	2539.8
Serial	1000	8134.58	8134.58
Parallel	10000	175009.01	21876.125
Serial	10000	81058.82	81058.82

Body Count	10	100	1000	10000
Performance Increase	0.08824901121	0.9391967122	3.202842744	3.705355496

It should be mentioned that the performance increase is measured by looking at the time taken to run the whole simulation. I talk about this in detail in Question 2.

Scaling Plot

For larger sets of bodies, parallel programming shows a considerable improvement against serial. The issue where the smaller set of bodies is answered in detail in Question 1. All simulations here have been recorded and displayed on Paraview. The video is here (https://www.youtube.com/watch?v=JAh_YskmOXc) .

Questions

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

There is a lot of overhead involved in initiating a set of threads for it to be utilised for parallel operations, to the extent that *may take more time than the actual simulation itself.* This may include each thread initiating their own set of variables, and initiation of a shared set of memory. This was the case for 10 and 100 bodies simulation, where it is evident that parallelisation affected the timing of the results in a negative manner. For a small operation, it is often better to compute the operation serially.

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

Gustafson estimated the speed-up S gained by using N processors (instead of just one) for a task with a serial fraction(which does not benefit from parallelism) K as $S = N + (1 - N)K$. The table below shows the time measurements between serial and parallel times, measured via the CPU time. From here, we can deduce K by looking at the time spent on serial operations as a fraction of the overall time.

Number of Bodies	Threads	Serial Time	Parallel Time	Total Time	K
20000	1	0.026708	35.1801	35.206808	0.0007044473923
20000	2	0.028066	17.64145	17.669516	0.0007420183465
20000	3	0.026398	12.09363333	12.12003133	0.0006748307414
20000	4	0.026561	9.3316	9.358161	0.0006612924385
20000	5	0.023714	9.29126	9.314974	0.0004690381061
20000	6	0.020047	9.262066667	9.282113667	0.0003273934802
20000	7	0.026727	9.008214286	9.034941286	0.0003931372192
20000	8	0.028578	8.795575	8.824153	0.0003793082384
50000	1	0.055767	219.885	219.940767	0.0002255879951
50000	2	0.054913	110.6975	110.752413	0.0002318856395
50000	3	0.057161	75.45833333	75.51549433	0.0002303044535
50000	4	0.055998	58.141	58.196998	0.0002340699913
50000	5	0.04972	57.4432	57.49292	0.0002192463943
50000	6	0.054618	56.70416667	56.75878467	0.0001566519646
50000	7	0.049742	55.82085714	55.87059914	0.000146939975
50000	8	0.056105	54.79475	54.850855	0.0001151510654

As the processor in question includes hyper-threading, it may obfuscate the results in some manner. The program depends on its floating point operations. The use of hyper-threading provides the illusion

of 8 threads, whereas in reality, floating point registers are shared between a virtual thread and a physical core, reducing the effectiveness of the extra threads. This is shown in the graph below, where diminishing returns can be seen from 4 threads onwards. We can see this in Figure 3, where the results reach diminishing returns after 4 threads. Therefore, we treat the rest of Gustafson's formula using 4 threads; representing the physical cores.

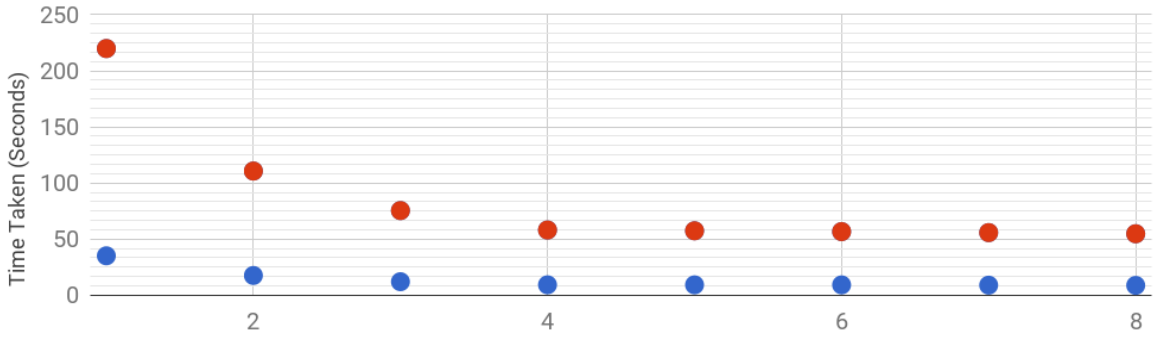

Figure 3: Red represents 50,000 body operations, and blue represents 20,000 bodies.

For 50,000 bodies $K = 0.0002255879951$ is chosen from the 1 thread operation. This Results in S being $4 + (1 - 4) \times 0.000225.. = 4.000675$. This law is respected when we compare the speed-up from 4 threads against 1 thread, where the speed-up is 3.77x. ('8 Threads' provides a speed-up of 4.0097x which is within margin of error). This goes in hand with our scaling plot where we can see a 3.7x increase between parallel and serial simulations in the scaling experiments.

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

As mentioned in Question 1, initiating a set of threads for parallel operations is very expensive. However, it is also the case that initiation is done once at the beginning of the program. Over time, the serial operation becomes a smaller fraction of the overall running time. This makes parallel operations the dominant factor in the runtime. This consequently makes the parallel efficiency increase as the threads are more utilised for a longer duration of the simulation.

This is however, assuming that the simulation is running for an extended period of time. If, like question 1 the simulation is run short where the serial operations dominate the running time, the parallel efficiency is considered very low.

Distributed Memory Simulation

Assumptions and Setup

For the sake of simplicity:

- We assume near zero latency for transmission.
- The master-slave model is adopted; the master rank does not perform any major computation; as this is distributed to the slaves

MPI is SPMD (Single program; multiple data), so every CPU in the MPI network will have a copy of the same code, but handles different sets of instructions and data based on their rank. We check the

ID of their rank to determine whether it is a master or a slave, we call master rank 0. The master has its own set of functions, and so does the slaves.

Operation

The master rank does compute heavy operations; this is divided and split between the slaves. The master rank does, however, initiate the data structures that the slaves will utilise, including the bodies, timesteps and other helper variables that are also used in the OpenMP program. The contents of the bodies are broadcast to the slaves (via `MPI_Bcast`) at every iteration which reduces the number of redundant transmissions during the slaves split-computation of the force loop in `updateBodies()`.

The primary use for parallelization during our OpenMP program was to handle the floating point calculation of forces for each body. The master rank splits segments of the loop (which could be via `MPI_Scatter`, but this is contingent on the number of bodies. It may be better to propagate them through other means given a large enough set of bodies, as the MPI buffers could face heavy stress.)

The master rank uses an `MPI_send()` to send extra but relevant information to a slave, and the slave receives data using `MPI_recv()`. Received data needs to be verified by checking the status of the received message. (This applies to all messages transmitted; if it isn't a valid response then we should request the message again or have error handlers in place.)

Slaves also have their own local variables that are not necessarily needed to be stored in the shared pool of memory; for instance, helper variables for inner loops and variables that help make code more human-interpretable. Otherwise, `gets` will also suffice for a slave to retrieve information from the global space, for instance, the positions and velocities of bodies.

The distance calculation would be its own function for the MPI program, and is called by slaves to increment the overall force for a given body. It is also used to determine whether two bodies have collided, where a non-blocking `immediate_send` could be initiated from a slave to the master rank, indicating a collision between two bodies.

Once a slave rank has finished computation of a force increment, they are sent back to the master rank via `MPI_reduce`. The master rank will need to verify that all the results have been received prior to continuing with the rest of the operations needed to update the bodies.

The adaptive timestep can be computed by having slave nodes calculate potential collisions using various time step sizes. The results are collated back to the master rank to determine the best timestep to utilise.

In terms of updating the bodies, this can again, be distributed across the network using a fairly simple `MPI_Scatter` and `MPI_Reduce`. Like the OpenMP program, the iteration is repeated for each step until the simulation is finished.