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 #	$s(x)$	$s(y)$	$s(z)$	$v(x)$	$v(y)$	$v(z)$	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	$1e^{-11}$

The bodies collide at $(x, y, z) = (0.155, 0.155, 0.155)$ at time $t = 0.275$. The error is calculated through calculating the difference between the position of one body and the other body upon collision. The following table shows the timestep used to calculate the collision, 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 h	Error $\overline{u}_h$	x_a	x_b	Ratio	Steps	Range
Adaptive	0.000001997990	-0.449998	-0.450002	0.500503	1716457	0.000003994500
$10^{-6}/2^1$	0.000001999990	-0.449998	-0.450002	0.500003	275000	0.000003998500
$10^{-6}/2^2$	0.000001000020	-0.449999	-0.450001	0.499992	550000	0.000001998520
$10^{-6}/2^3$	0.000000499961	-0.45	-0.45	0.500039	1100000	0.000000998475
$10^{-6}/2^4$	0.000000250068	-0.45	-0.45	0.499865	2200000	0.000000498564
$10^{-6}/2^5$	0.000000125078	-0.45	-0.45	0.499688	4400000	0.000000248608
$10^{-6}/2^6$	0.000000062134	-0.45	-0.45	0.502946	8800000	0.000000123183
$10^{-6}/2^7$	0.000000031729	-0.45	-0.45	0.492452	17600000	0.000000061368
$10^{-6}/2^8$	0.000000014415	-0.45	-0.45	0.541966	35200000	0.000000028684
$10^{-6}/2^9$	0.000000006426	-0.45	-0.45	0.607915	70400000	0.000000012311
$10^{-6}/2^{10}$	0.000000002519	-0.45	-0.45	0.775391	140800000	0.000000004178

A numerical approximation is used

The order of accuracy p can be obtained by taking the third last error u_h to the expression $p = \log \frac{u_h - u_{h/10}}{u_{h/10} - u_{h/100}}$. Using the table, $p = 0.7152957794...$, making our convergence rate $h^p = (10^{-6}/2^8)^{0.7152957794...} = 9.67434210 \times 10^{-7}$. Implications show that the consistency is sub-linear as our $p < 1$.

We can see that the adaptive timestepping 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. However, choosing a larger initial distance between the bodies would take a long period of time to produce results.

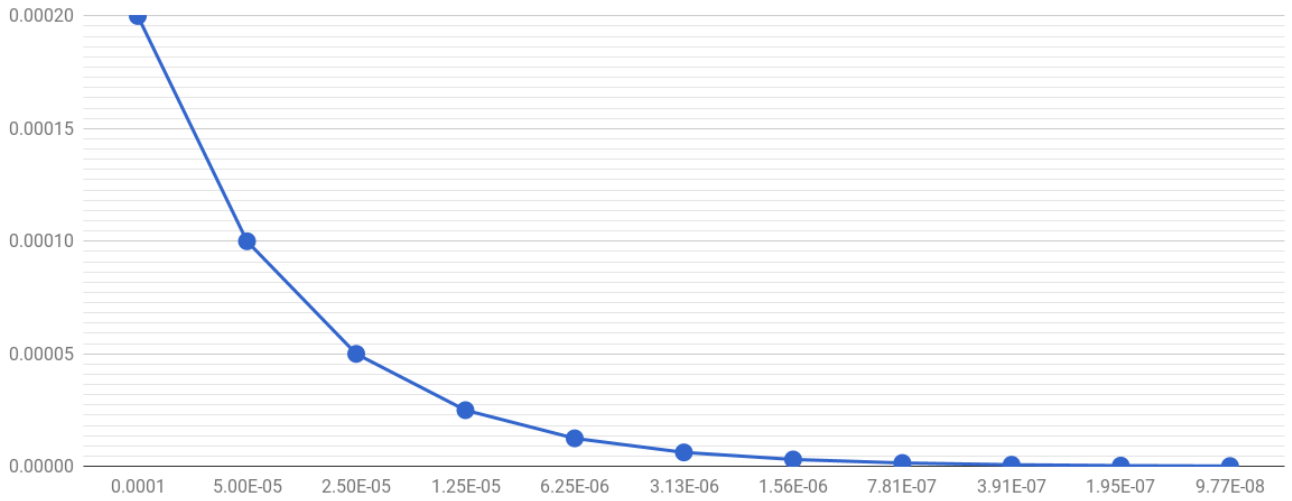

Figure 1: A chart showing the the timestep used against the error produced. The chart shows a clear convergence towards zero.

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.

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.

Statistics

Each randomly generated body (using the seed mentioned prior,) has a value ranging from -1 to 1 for all of its attributes $(s_x, s_y, s_z, v_x, v_y, v_z)$. The mass for each body would be infinitesimally small to reduce its effect upon force generation. We use a 10000 body simulation in order to increase the chances of collisions.

Scaling Experiments

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

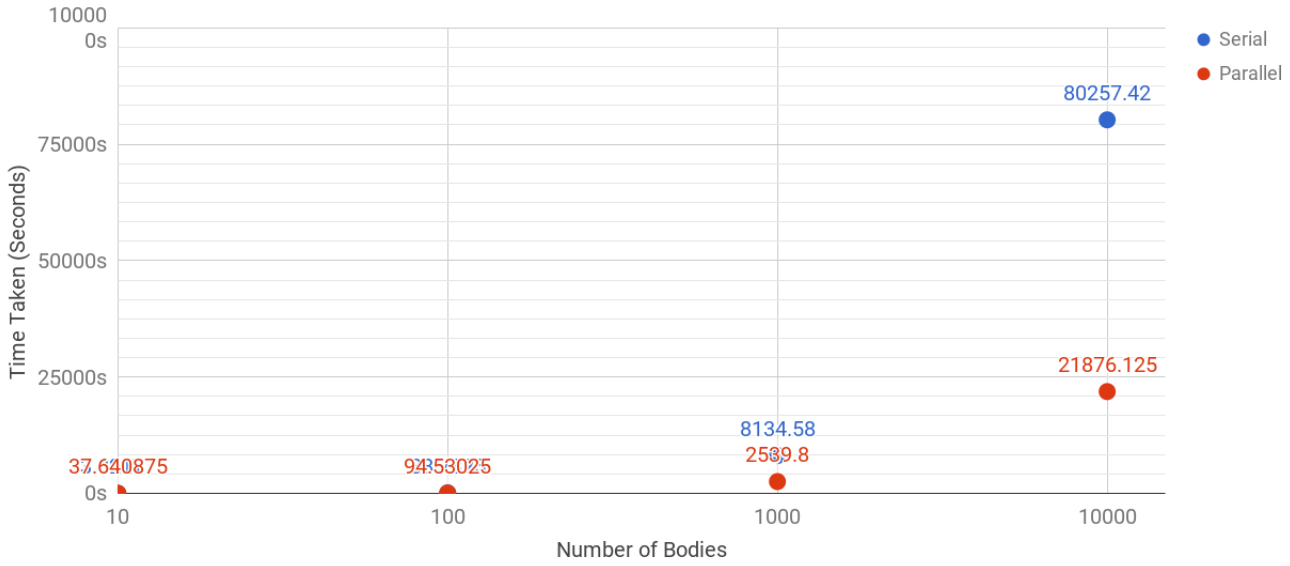

Figure 2: A scaling plot of serial vs parallel runtime for various body sizes.

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.

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 loop for a set of parallel processors, to the extent that *may take more time than the actual simulation itself.* This may include each processor initiating their own set of variables. This was the case for 10 and 100 bodies, where it is evident that the initialisation of multiple threads affected the timing of the results in a negative manner.

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

Gustafson estimated the speedup 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. 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 (Per Thread)	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 hyperthreading, it may obfuscate the results in some manner. The program depends on its floating point operations. The use of hyperthreading 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. Therefore, we treat the rest of Gustafson's formula using 4 threads; representing the physical cores.

For 50,000 bodies $K = 0.0002255879951$ is chosen from the 1 thread operation. This Results in S being $4 \times (1-3) \times 0.000225.. = 4.000675$. This law is respected when we compare the speedup from 4 threads against 1 thread, where the speedup is 3.77x. ('8 Threads' provides a speedup of 4.0097x which is within margin of error).

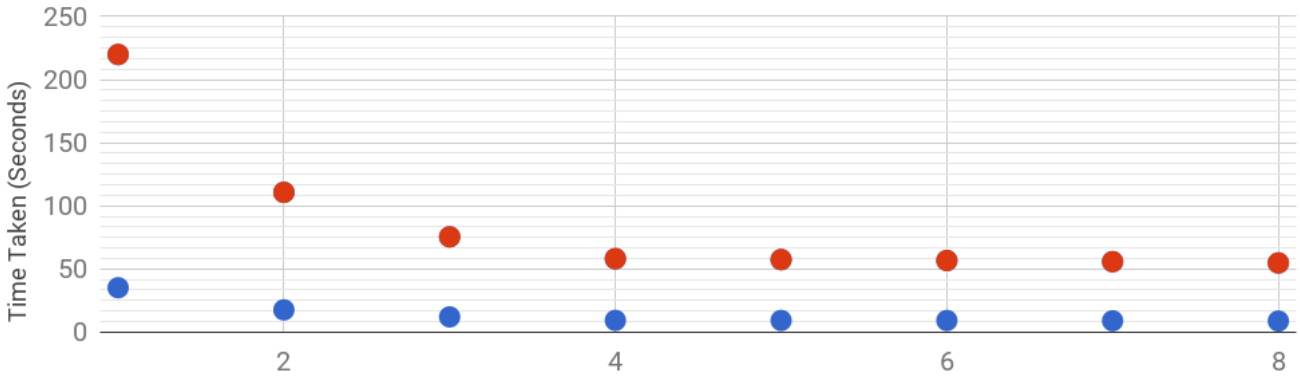

Figure 3: Red represents 50,000 body operations, and blue represents 20,000 bodies.

It depends on whether you fix the problem size. It hence depends on your purpose. It is crucial to clarify assumptions a priori. It is important to be aware of shortcomings.

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

- Parallelism on a single machine now depends on other factors

  - makes more heat
  - poor code
  - transistor noise
  - contingent on other factors on the machine

Distributed Memory Simulation

Assumptions and Setup

- MPI is SPMD (Single program; multiple data)
- we check the ID 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 master
- Assume perfect zero latency theoretical model between ranks
- For sake of simplicity the master-slave model is adopted; the master rank does not perform any major computation; this is distributed to the slaves
- Every CPU in the MPI network will have a copy of the same code, but is allocated different sets of data in regards to the loop position they are to compute

Operation

- master rank initiates a large array of bodies and `MPI_Bcast` to slaves
- whilst this utilises a lot of data transmission, it is necessary as when calculating the force for a given body, it compares its position against every other body in the space.
- for calculating the force
- Each rank will perform segments of the loop via `MPI_Scatter` (contingent on the number of bodies; it may be better given a large enough set (talking 100000+) to propagate them manually as the MPI buffers could heavily stress)
  - they're passed the relevant information;
  - master rank uses `MPI_send()` to a slave; slave receives using `MPI_recv()`
  - verify data is received properly through 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 have their own local variables that they can use to compute
  - also `gets` relevant information from the global space such as the positions and velocities of bodies
  - once the slave rank has finished computing the result, being the force increment for a body, are sent back to the master via `MPI_Reduce`
- will need to check that we have all the results!
- they also calculate
  - non blocking communication between the slave and master to send/receive information about short distances between two bodies as we calculate the potential collision afterwards; we use an `immediate_send`.
- for collisions
- contingent on the number of collisions occurred
- this can be distributed to nodes but depends on how fast data can be transmitted to
- for adaptive timestepping
- we can distribute the potential solution between nodes

- for updating the bodies
- given enough bodies this can be distributed across the network too using a `MPI_Scatter` and `MPI_Reduce`
- Process is repeated accordingly

Issues