Project: The N-Body Problem

Concurrent Programming (ID1217), KTH

David Lindström, 2014-05-19

Concurrent Programming, ID1217
KTH, 2014-05-19

David Lindström
davli@kth.se

Introduction

This report is about different approaches to simulating the gravitational N-body problem, using both parallel and sequential algorithms. The project is a part of the course Concurrent Programming (ID1217), which was taken at KTH in Stockholm during spring 2014.

The gravitational N-body problem is the problem of simulating a large quantity of astronomical bodies in space. Each body has a mass, an initial position and an initial velocity, and gravity causes the bodies to accelerate and to move. The motion of an N-body system is simulated by stepping through discrete instants of time, and at each timestep we calculate the forces on every body and update their positions and velocities [1].

Four different solutions to this problem was implemented in Java, using a shared memory programming model. A sequential "brute-force"-approach, in which all forces between all bodies were calculated, and also a parallelized version of it. Also, the Barnes-Hut method was implemented in both a sequential and parallelized version. These implementations will be explained and discussed more later in this report. Tests were performed in Windows 8.1 on a Lenovo IdeaPad S510p, on an Intel Core i5-4200U processor [2].

Tests with 120, 180 and 240 bodies where performed, which showed that the sequential $O(N^2)$ implementation was the slowest (with the exception of the parallel $O(N^2)$ ran with only one worker). The parallel version of it reached better and better results for larger input, and for 240 bodies and 4 workers it was about three times faster than the sequential one. The sequential Barnes-Hut method was faster than the sequential $O(N^2)$ program for all input sizes, but only faster than the parallel brute-force approach for input size 240. The parallel Barnes-Hut was as expected the fastest of all four implementations, and for an input of 240 bodies it ran for about 8.5 seconds. This is a major speedup compared to 60 seconds for the sequential $O(N^2)$ program on the same problem size.

David Lindström
davli@kth.se

Programs

Below is a summary of the four different programs implemented. I will refer to them as *program 1-4* when discussing them later. In addition to the methods, an input randomizer was implemented, both for reading directly from and to write to a file to be able to test the four programs on the same input. To make debugging easier, a simple graphical program was also made to be able to see that the bodies were moving in a way that seemed correct. A minimum distance was also set for the force-calculations to prevent bodies from getting to close to each other. This prevents them from being hurled away from the force getting too big when distance gets too small between two bodies.

1. A sequential O(N^2) program

This implementation was a straight off implementation of the sequential algorithm described on page 556 in the coursebook [1]. It sequentially calculates the forces working on each body, from each body, which is $O(N^2)$. After doing this, it calculates the new velocities and positions for the bodies, using the forces calculated. This is $O(N)$. I will not discuss the implementation of this algorithm any further, since no particular optimizations were made to algorithm given in the book.

2. A parallel O(N^2) program

A parallel version of program 1 was also implemented, and tested with 1 to 4 workers. This was also a fairly straight off implementation of the pseudocode on page 560-561 in the coursebook [1]. A threadpool with size equal to the number of workers were used, and both the force-calculations and the moving of bodies were parallelized. A counter-barrier was implemented, using *java.util.concurrent.atomic.AtomicInteger*, to make the program wait until all force-calculations have been made before starting to move bodies. The following operations were made each timestep of the simulation:

1. **Main-thread:** Create a set of force-calculating threads equal to the number of workers used. Add them to the threadpool for them to start executing (described at point *2* below). When a force-calculating thread is ready it increments the counter-barrier, and the main-thread waits for all threads to have done this before moving on.
2. **ForceCalcThread:** Instead of having a single point representing the force on a body, the force on a body is represented by an array of points, in which each thread adds the force of the calculations that thread has made. Each thread only calculates the force from every *numWorkers* body to all the bodies after that one in the body-array. This approach means that we can avoid using locks at the expense of having to use more memory to store the forces calculated in each thread. When the thread is done calculating forces, it increments the counter-barrier.
3. **Main-thread:** When all force-calculations are done, create a set of *MoverThreads* equal to the amount of workers we are using. Also create a new barrier to tell when they are done. Add the threads to the threadpool and wait for them to finish.
4. **MoverThread**: The updating of positions and velocities can be done independently for each body, which means that we can distribute the bodies among the *MoverThreads* and let each thread take care of a set of bodies. Since the force working on a body is split up in an array of different forces, we add them up before calculating differences in velocity and position. When a thread is done with its share of bodies, increment counter-barrier to signal that it is done.

3. A sequential O(N*log N) Barnes-Hut program

A sequential implementation of Barnes-Hut method was also implemented. This algorithm creates a quadtree from the simulation space by dividing the space into quadrants. The quadtree is created by inserting one body at a time, and splitting a region into four equally sized child-regions if it already contains a body. This is illustrated by *figure 1* below.

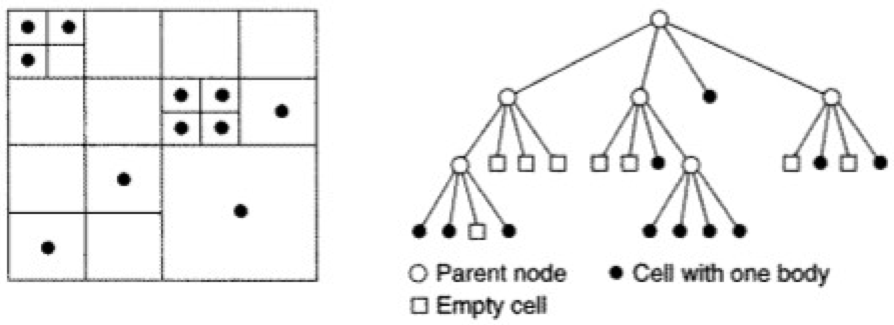

Figure 1: Bodies inserted into a quadtree. Each cell/quadrant contains at max one body, which is done by splitting a region as many times as needed at construction.Image source [1].

When calculating forces for a body *b1*, forces from bodies in quadrants far away can be approximated by the force from the center of mass in that quadrant. This instead of calculating the force working from each the individual bodies in it. This makes force-calculating $O(N * \log N)$ instead of $O(N^2)$, and we are able to reduce the most time-consuming part of the previous programs at the expense of having to create a quadtree each timestep.

This makes the algorithm work as follows for each timestep:

1. Create a quadtree from the simulation space. If this is not the first timestep, use the center of mass from the previous quadtree as middle point for the new one.
2. Recursively calculate masscenter for each quadrant in the new quadtree by starting at the leaf-nodes and pass masscenter upwards to parents.
3. For each body, calculate the force working on it by querying the quadtree. Start at the root, and perform the following test for each node visited: If the center of mass for the quadrant is far enough away (specified by a user parameter), approximate the entire subtree by the masscenter of that cell. If not, continue downwards to the four children.
4. Calculate new velocities and positions in the same way as in program 1.

David Lindström
davli@kth.se

The reason for using the center of mass of the previous quadtree as the center of the new one is to create a more balanced tree. This was done assuming that the center of mass would have a somewhat equal distribution of bodies at each side of it. If we would have a large spread of possible masses, this would not necessarily make the tree more balanced. We could for example have a giant planet on one side of the masscenter and several small ones on the other, resulting in an unequal distribution of bodies on level one. This could also lead to a problem if one body would be hurled away very far, since the implementation was done using a fixed size for the root-quadrant of the quadtree. A fixed size could potentially mean that not all bodies will fit in the root-node, which makes the algorithm fail. This could have been solved by dynamically calculate the needed size of the root-quadrant each timestep. However, since it was not needed to compare the algorithms in the assignment, this was never implemented.

David Lindström
davli@kth.se

4. A parallel O(N*log N) Barnes-Hut program

A parallel version of the Barnes-Hut method was also implemented and tested using 1-4 workers. All four stages described in the sequential Barnes-Hut were parallelized. For the different threads, a threadpool was used. To synchronize quadtree-construction, force-calculation and moving, the same implementation of a counter-barrier as in program 2 was used, using *AtomicInteger*. Below is a description on how each of the four parts were parallelized:

1. To parallelize the quadtree construction was the hardest of all steps, and also the one which in the end weren't fully parallelized. To split the work, the root of the quadtree was split up as many times as needed for each *InsertionThread* to get a subtree each to insert bodies into. Each *InsertionThread* then goes through the full array of bodies to determine which ones that belongs to their subtree. If a body is within the region belonging to the thread, insert it. This means that the only part being done in parallel is the log N operations for inserting a node, and not the N operations for deciding which body that belongs to which thread/region. Further discussion about the reason for this below.
2. The fact that each *InsertionThread* handles independent regions of the space means that masscenters can be initialized for the nodes in a subtree as soon as the construction of it is finished. This means that we don't need any barrier between constructing a subtree and calculating the center of mass for the nodes in it. Once all *InsertionThread*'s have finished, the main-thread sequentially updates the masscenter of the levels above the subtrees that were split up among the threads. This will generally be very few operations (in the case of 4 workers, only one new masscenter to calculate for the top quadrant) so it was left sequential.
3. Once the quadtree is constructed, calculating the forces working on each body is read only. This means that the bodies can be equally distributed to force-calculating threads, which can execute independently. A counter-barrier is however needed to prevent the next step to start before all force-calculations are done.
4. Analogue to the parallel $O(N^2)$ program, the updating of velocities and positions for each body is independent. This means that we can distribute the bodies among *MoverThread*'s, which can execute independently. Similar to step 3, we will however need a counter-barrier to halt the main-thread until all threads are done.

As mentioned when discussing the insertion of bodies into the tree (step 1), full parallelization of that step was not achieved. The first approach to the insertion was to use a list of *java.util.concurrent.ConcurrentLinkedQueue<E>*, to enable concurrent sorting of the bodies into child-regions before inserting them to the tree. For the test cases 120, 180 and 240 bodies, this implementation was however much slower than to not perform this sorting concurrently, and was therefore left outside the final program. When running on larger input sizes one might however consider to implement a similar solution, to reduce the *N* operations in this case needed to sort the bodies among the insertion-threads.

David Lindström
davli@kth.se

Evaluation

In this part the results from the tests on the four programs will be presented. The programs were evaluated using the same input and the same parameters. When evaluating the number of timesteps needed, five tests showed that program 1 needed about 17000 timesteps to run for 15 seconds when using 120 bodies. This number of steps were then used on all four programs.

Table 1 shows a summary of the runtimes for the four programs, and *Figure 2* shows an illustration of it. When tests were performed, each program was ran five times. The runtime below is the median runtime (in seconds) for each program and input size. These will be discussed individually in the sections below.

	120 bodies	**180 bodies**	**240 bodies**
Program 1	15,01	33,75	60,06
Program 2, 1 worker	17,55	38,45	67,56
Program 2, 2 workers	10,43	22,84	39,26
Program 2, 3 workers	7,92	16,32	28,01
Program 2, 4 workers	6,54	12,84	21,70
Program 3	7,34	12,56	15,55
Program 4, 1 worker	9,94	15,33	18,53
Program 4, 2 workers	6,90	10,31	12,61
Program 4, 3 workers	5,28	7,81	9,42
Program 4, 4 workers	5,33	7,25	8,47

Table 1: Runtime [s] for the different programs, run with input sizes 120, 180 and 240 bodies.

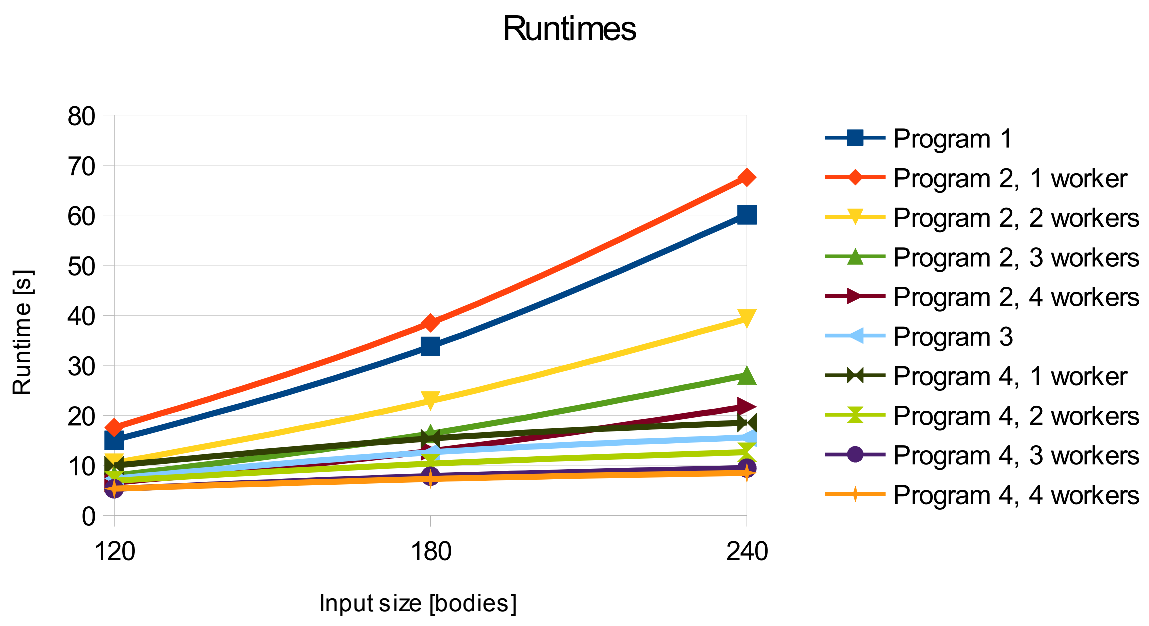

Figure. 2: Running time [s] as a function of input size for the different programs.

David Lindström
davli@kth.se

1. A sequential O(N²) program

The first program is quite expected the slowest one (not counting program 2 with one worker). The time grows fast with the number of bodies, and when input is doubled from 120 to 240 bodies, the runtime is about four times slower. Roughly looking at the three measurements made, it seems like the runtime doubles for every 60 bodies added.

2. A parallel O(N²) program

As described in the assignment, tests were made using 1, 2, 3 and 4 workers. For the first case, when only using one worker, the overhead from threads makes it run slower than program 1. The overhead also seems to grow with the number of bodies, which could partially be because of the inefficiency of having a one-point array for the forces when having only one worker. For the tests where several workers were used, the benefit of parallelization increases a little bit more when the input size grows. This is probably due to the smaller impact of thread-overhead as the number of bodies grow. In other words, if we have more bodies, thread creation and synchronization will be a smaller percentage of the runtime. However, as input is increased by 60 bodies, the runtime still almost doubles, which still makes it quite slow if input would increase even more.

3. A sequential O(N*log N) Barnes-Hut program

The sequential Barnes-Hut method makes runtime substantially faster compared to the previous two programs. The only case where it is slower is for 120 bodies and compared to program 2. The gain is however the most when the number of bodies increase, due to the $O(N*\log N)$ upper bound instead of $O(N^2)$. The expense of this, which is most notable for fewer bodies, is the additional steps added when calculating forces. Instead of doing a $O(N^2)$ force-calculation and a $O(N)$ moving operation as in the previous programs, we have tree-construction, calculation of masscenter and force-calculations, which are all $O(N*\log N)$, and then the same move-operation after this. These additional steps is what makes program 2 run faster on small input with 4 workers.

The parameter *far,* which controls the distance of when to approximate or not, was set to 2.1. This lead to the percentage of approximations in the force-calculation-phase being 82,0%, 87,8% and 84,4% for 120, 180 and 240 bodies. This fulfills the project description demand that far should be set so that 80-90% of the bodies should be considered far enough away to be approximated.

David Lindström
davli@kth.se

4. A parallel O(N*log N) Barnes-Hut program

The last of the tested programs was the parallel Barnes-Hut method, which was also tested with 1, 2, 3 and 4 workers. The same value for the parameter *far* was used as in the sequential Barnes-Hut, which means an equal amount of approximations. For the case of one worker, the program was as expected slower than the sequential Barnes-Hut due to parallelization overhead. The tests where more than one worker was used was however all faster than the sequential Barnes-Hut, as well as all the other programs. The program tested with 4 workers was the fastest of all programs, except for input 120 bodies, in which the same program with 3 workers was a little bit faster. When comparing to the sequential Barnes-Hut on the input of 240 bodies, the parallel Barnes-Hut with 4 workers was almost twice as fast. This is still substantially faster, but the speedup wasn't quite as good as when comparing programs 1 and 2. To achieve even more speedup, the construction of the quadtree should be parallelized to a larger extent than it was, at least if one wants better performance on even larger input than was used in this project.

Learnings

For me, most of the time was spent trying to parallelize Barnes-Hut method. Besides more knowledge of the N-body problem, the process of doing so was a generally good training in finding concurrency in problems. Also, it was a good training in how to try to avoid using synchronization where there is no need for it, in order to increase performance. In the homeworks in the course, performance hasn't been the focus. In my case it lead to a use of synchronization-methods where there sometimes were possible work-arounds available, which is something I have realized when getting more knowledge.

Conclusions

As expected, programs 1 and 2 were substantially slower than the two Barnes-Hut implementations, at least as the input size grew. The conclusion to draw from that is that even though program 2 made use of more concurrency than the others, a slow algorithm is always a slow algorithm. However, if one really needs to be precise about the calculations, it is a good improvement to the sequential version of it, if approximations aren't an option.

Which algorithm to pick also depends on which input size to use. If calculations are never bigger than, for example, 120 bodies, one might consider implementing the parallel $O(N^2)$ algorithm instead of Barnes-Hut, since it isn't that much slower with 4 workers. The benefit of this would be that the calculations are approximation-free at the expense of a marginally higher running time. The parallel $O(N^2)$ algorithm is also a more simple one to implement, which might be a plus as well. However, since the parallel Barnes-Hut method was so much faster on larger input, there really is no discussion about it being the best one to pick if input is expected to be large.

David Lindström
davli@kth.se

References

[1] Gregory R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*,Addison-Wesley; (fifth printing 2006), ISBN 0-201-35752-6

[2] http://www.webhallen.com/se-sv/datorer_och_tillbehor/193321-lenovo_ideapad_s510p_156-i5-4200u-8gb-1tb__8gb_ssd-gt720-win_8 (Retrieved 2014-05-19)