Parallelization of the N-body problem using OpenMPI

Ben Witzen

December 5th, 2015

---

1. Introduction

The assignment requests of me to implement a parallel program that allows to run a simplified N-body simulation. An N-body simulation aims to simulate the trajectory of objects in space, which are affected by each other's gravitational pull. Because each object affects each other object, parallelization of this simulation is nontrivial.

Parallelization will be achieved by building an application that makes use of the MPI (Message-Passing Interface) library. For this assignment, I have elected to use the OpenMPI implementation [1]. In addition, I will be implementing local parallelization to further improve the performance of the application.

2 Implementation

2.1 Overview

In order to effectively parallelize the problem, we first need to analyze the sequential implementation and locate opportunities for parallelization. Due to the nature of the OpenMPI library, which is designed to compile one program that runs simultaneously over multiple machines, parallelization of (operations on) arrays is the most obvious place to start. The array of body structures lends itself well to this, as it allows for a conceptually easy split of work between the nodes. Furthermore, the computation of force on a body only requires limited information on the other bodies, which means such a split will incur little communication. This approach will be the focus point of my parallelization efforts.

It should be noted that the iteration loop for time steps is obviously not parallelizable, as this loop carries a dependency on previous iterations of that loop.

2.2 Differences between my parallel version and the sequential version

A sequential program was provided as an example. The proposed parallel code has been written from scratch, but uses the general ideas put forth by the sequential version. To improve readability, the parallel version eliminates the use of global variables and was stripped down of most code relating to the drawing and updating of PPM files. Furthermore, several loops in the sequential version were fused together, reducing the amount of function calls and loop overhead taking place. The fusing of the loops is unlikely to highly impact execution times by itself, but was needed for an optimization discussed later in this report.

2.3 Implementation details

An OpenMPI executable is compiled once, distributed over a number of nodes, and started on each of those nodes at the same time. This means that all nodes fundamentally run the same code.

A typical run of the proposed application can be seen in table I. Each node processes the arguments independently, but only the master node initializes all bodies. The initialized

bodies are then distributed among itself and the other nodes, using the `MPI_Send` and `MPI_Recv` functions. The other nodes can not initialize their own bodies because their initial state depends on a random seed. Having those nodes initialize their own bodies would lead to inconsistencies when compared to the sequential version. Furthermore, it should be noted that it is quite unusual for an OpenMPI application to have all nodes to perform the argument reading independently, but in the case of the DAS-4 it does not impose any problems. As such, there is no merit in complicating the code by having only the master do this and having it send around the arguments to the other nodes.

Bodies are distributed over the nodes as evenly as possible. If an even division is not possible (amount of bodies not divisible by amount of nodes), the nodes are distributed in such a way that the maximum difference between the most heavily loaded node and least heavily loaded node is one body.

	Master node	Other node(s)
1	Read arguments. Prepare local state.	Read arguments. Prepare local state.
2	Initialize all bodies.	
3	Send entire dataset to all nodes.	Receive entire dataset.
4	Start local timer.	Start local timer.
5	Perform computation on private subset of bodies.	Perform computation on private subset of bodies.
6	Stop local timer.	Stop local timer.
7	Receive private subsets from the others.	Send private subset to master.
8	Report back final state and local time to user.	Report back local time to user.
9	Clean-up.	Clean-up.

Table I: Global overview of the steps performed by each node.

At the end of a successful simulation, all the other nodes send the results of their final iterations back to the master node, who will ultimately report on the results of the of the simulation.

Step 5 in table I is of particular interest, as this is the timed segment of the code and the backbone of the simulation. Each body carries the state information as listed in table II. In the program, two additional variables for temporary force accumulation are used. These are reset between every iteration, and are not actually part of a body's properties.

name	description
position	X and Y coordinates of this body.
velocity	Speed of the body in the X and Y directions.
mass	Weight of this body.
radius	Size of this body.

Table II: Relevant contents of a body structure.

In order to calculate the position and velocity of a body in the next time step, its own position, velocity, mass, and radius for this time step need to be available. These are obviously already present at the node. In addition, the position, mass, and radius of *all other bodies* for this time step need to be available.

Since the mass and radius of all bodies are constant, it suffices for the master node to send these values around to all other nodes at the initialization step. Each node can then store and retrieve these values as needed, without the need for further communication with the other nodes during the time step iterations. However, as the position of all

bodies change between each time step, it is necessary for a node to receive this information for all bodies in the simulation before it can continue calculating the new values for the next time step.

Position is stored in two `doubles`, one for the X position and one for the Y position. This means that during each time step, if the simulation contains M bodies of which N are assigned to a particular node, that this particular node needs to send away $N * 2$ `doubles` to all other nodes and receive in total $(M - N) * 2$ `doubles`. The information to receive is likely scattered between many different nodes.

This particular situation lends itself well for implementation using the `MPI_Bcast` function, which can be used to send and receive broadcasted information. Because this communication step is executed during every iteration, it deserves extra attention in terms of optimization.

```
1  void broadcast_positions(…) {
2    for (int i = 0, offset = 0; i < mpi_size; i++) {
3      double data[count[i] * 2];
4
5      // loop to "package" the data
6      if (i == mpi_rank) {
7        for (int b = offset, d = 0; b < offset + count[i]; b++){
8          data[d++] = X(b);
9          data[d++] = Y(b);
10       }
11     }
12
13     MPI_Bcast(data);
14
15     // loop to "unpackage" the data
16     if (i != mpi_rank) {
17       for (int b = offset, d = 0; b < offset + count[i]; b++){
18         X(b) = data[d++];
19         Y(b) = data[d++];
20       }
21     }
22
23     offset += count;
24   }
25 }
```

Listing I: Simplified version of the broadcasting function.

The broadcast function is displayed in simplified form in listing I. Remember that this function is executed on each node separately. The outer loop takes one iteration for each node, including the node itself. In the case the loop is currently targeting a node which is not self, the node "packs" the data and sends it over the broadcast. In the other cases, it is receiving a broadcast from another node, which it will then unpack and store in its local body array. In short, this means that if there are N nodes, each node will send one broadcast and receive $N - 1$ broadcasts. Because all nodes perform the loop in the same order, and as such work on the broadcasts in the same order, the blocking time is minimized.

The remainder of the code is very reminiscent of the sequential version.

2.4 Compilation

Unless otherwise noted, all applications that are tested are compiled using the `-O3 -ffast-math -march=native` compiler flags. This includes the sequential version.

3 Further optimization using OpenMP *(Bonus assignment)*

3.1 Overview

OpenMPI allows for the parallelization of an application by running it on multiple nodes. However, those nodes themselves typically are also able to run programs parallel by multithreading. Currently, we are not utilizing this opportunity for parallelization, which seems suboptimal.

OpenMP (not to be confused with OpenMPI) allows for local parallelization using a shared memory model by the use of relatively simple compiler directives called pragmas. An in-depth explanation of the standard is out of scope for this report and can be found at [2] or [3]. We will be looking at OpenMP's ability to parallelize `for` loops, which only requires that the iterations of such loops are independent of each other.

3.2 Implementation

The time step iterations are not valid targets, as they are dependent of each other and thus cannot be parallelized. However, the various loops in our algorithm that constitute the update of one time step, namely `clear_forces`, `compute_forces`, `compute_velocities`, and `compute_positions`, look promising. Indeed, their iterations are independent of each other, and thus can be calculated in parallel. Note that this already happens in the OpenMPI implementation, so we will only be increasing the level of parallelization here.

It should be noted that inserting pragmas to initiate parallelization is not free. Setup is required to initialize a multithreaded block, which has some cost. Since the four loops in our main computation are split up, this would require four of such initializations per iteration. Luckily, it is possible to fuse the four loops into one, which then allows us to parallelize most of the computation using just one pragma.

```
 1 #pragma omp parallel for schedule(static) if(my_len > 8)
 2 for (int b = offset; b < my_len + offset; b++) {
 3   clear_forces;
 4
 5   for (int c = 0; c < len; c++) {
 6     if (b != c) {
 7       compute_forces;
 8     }
 9   }
10
11   compute_velocities;
12   compute_positions;
13 }
```

Listing II: Simplified display of fused computation loop with an OpenMP compiler directive (line 1).

The first pragma parallelizes the private workload of a node. Because the body of this higher-scope `for` loop almost always encompasses the same amount of computation regardless of iteration, I have chosen for a static schedule, which incurs the least

overhead. As soon as the for loop is encountered, the iteration space is split in equal chunks and each thread then computes its own portion of this iteration space.

The `if` clause states that if this node has eight or fewer bodies to calculate, it should just do so sequentially. The calculations for one body are not particularly expensive and the overhead caused by initializing parallelization in such cases is probably not worth it.

One might wonder why the second `for` loop (on line 5 in listing II) is not annotated with a pragma. While it is indeed also a loop with independent iterations, it is nested in a parallelized loop. In the ideal case with the correct settings, all threads are already working on the outer loop. In this case, nested parallelization will not do much as there are no more idle threads available.

The flag `-fopenmp` is added to applications compiled with the OpenMP pragmas active.

4 Experimentation

4.1 Setup

Experimentation will take place on the DAS-4 supercomputer. It offers nodes with dual-quad-core Intel CPUs running at 2.4 GHz with 24 GB memory available. All cores have hyperthreading enabled, so they locally support up to sixteen threads (this is only relevant for the OpenMP tests).

Set-up, data initialization, data distribution at the beginning and data collection at the end and clean up are not timed. Only the kernel computation is timed. In case of the OpenMPI tests, the timing is done independently on each node, with the slowest node dictating the time registered for that simulation.

Each tests was repeated three times, with their average running time being reported.

4.2 Potential and expected speedup

For the OpenMPI-only application, a speedup of N is theoretically expected against the sequential application, where N is the amount of nodes available. However, it is unlikely that this speedup will be reached as parallelization incurs overhead. In addition, the speedup will depend on the amount of bodies in the simulation, with higher amounts expecting a higher speedup as the overhead cost's impact decreases. However, lower counts may potentially improve cache usage.

For the bonus assignment using both OpenMPI and OpenMP, a speedup of $N * M$ would be expected, with N the amount of nodes and M the amount of threads available on a single node. Again, the actually expected speedup will be lower. This version also benefits from a higher body count.

Because the time steps are not parallelized between each other, it is expected that the relationship between iteration count and runtime is linear; increasing the amount of iterations tenfold should also increase the runtime tenfold, regardless of whether the test is performed on the sequential or parallel version.

4.3 Runtime tests

Various tests were run, with differing step counts, iteration counts, and different amount of nodes. The results of these tests are listed in table III.

Absolute runtimes, *in seconds*									
test		**seq**	**OpenMPI**, *amount of nodes*						
bodies	**steps**		**1**	**2**	**4**	**6**	**8**	**12**	**16**
25	10^4	0.5	0.9	0.5	0.5	0.5	0.6	0.9	1.2
50	10^4	1.9	3.6	1.9	1.2	1.1	1.0	1.2	1.4
75	10^4	4.1	8.0	4.1	2.3	1.8	1.7	1.7	1.7
100	10^4	7.3	14.1	7.2	3.8	2.9	2.5	2.3	2.2
200	10^4	29.1	56.4	28.3	14.4	10.0	7.7	5.9	5.2
400	10^4	115.9	225.6	113.1	56.8	38.5	29.0	20.4	15.7
600	10^4	260.8	507.3	253.9	127.7	85.4	64.6	43.7	33.9
800	10^4	465.4	x	451.1	226.3	152.4	113.9	77.3	58.3
25	10^5	4.7	8.9	5.3	3.9	3.8	4.1	5.7	6.9
25	10^6	46.9	88.9	53.2	37.8	37.1	39.5	52.6	63.6
100	10^5	73.0	141.2	71.5	37.2	27.2	22.5	19.5	17.2
100	10^6	729.6	x	715.2	372.3	270.5	223.9	192.0	167.7

Table III: Absolute runtimes under various settings. Lower is better. Fields marked with x took too long to run on the DAS-4. The optimizations for the bonus assignment were disabled in all these tests.

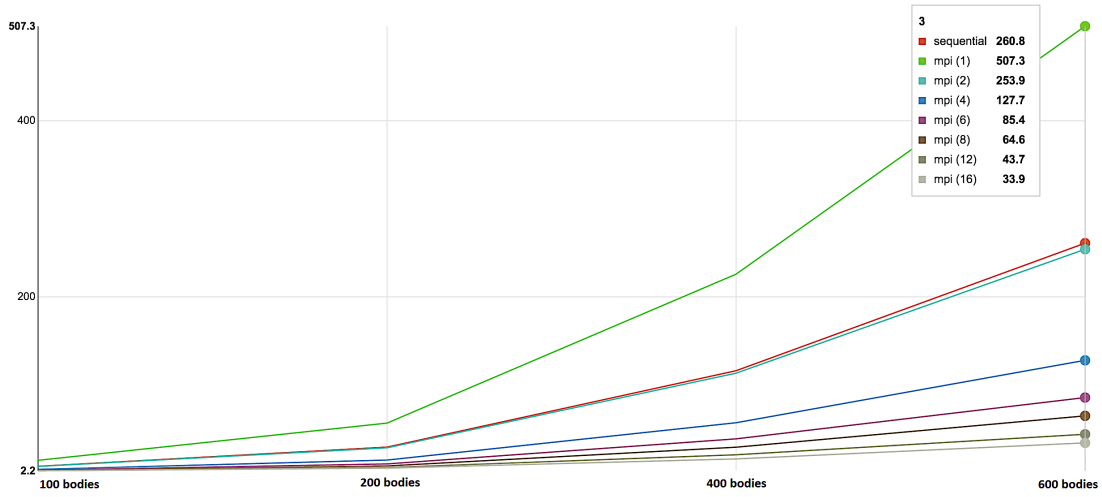

Figure I: Several of the runtimes in table I visualized. The runtimes for the 100-, 200-, 400-, and 600-body runs are plotted.

From these results it becomes clear that the cost of parallelization is fairly significant; two nodes are required to match the performance of the sequential version. As was expected, a higher amount of iterations does not improve the performance of the parallel application. However, increasing the amount of bodies does significantly improve the performance when compared to the sequential version.

For low body counts, it seems unbeneficial to use the maximum amount of nodes. As we increase the amount of bodies, it becomes increasingly rewarding to also increase the amount of computing nodes involved. For the largest problem size tested, a speedup of over 8 times is reached, as can be seen in table IV and figure II.

Speedup, *against sequential*				
problem size	**4 nodes**	**8 nodes**	**12 nodes**	**16 nodes**
100 bodies	1.9	2.9	3.2	3.3
200 bodies	2.1	3.8	4.9	5.6
400 bodies	2.0	4.0	5.7	7.4
600 bodies	2.0	4.0	6.0	7.7
800 bodies	2.1	4.1	6.0	8.0

Table IV: Speedups against sequential program for some of the tests performed in table I. Higher is better.

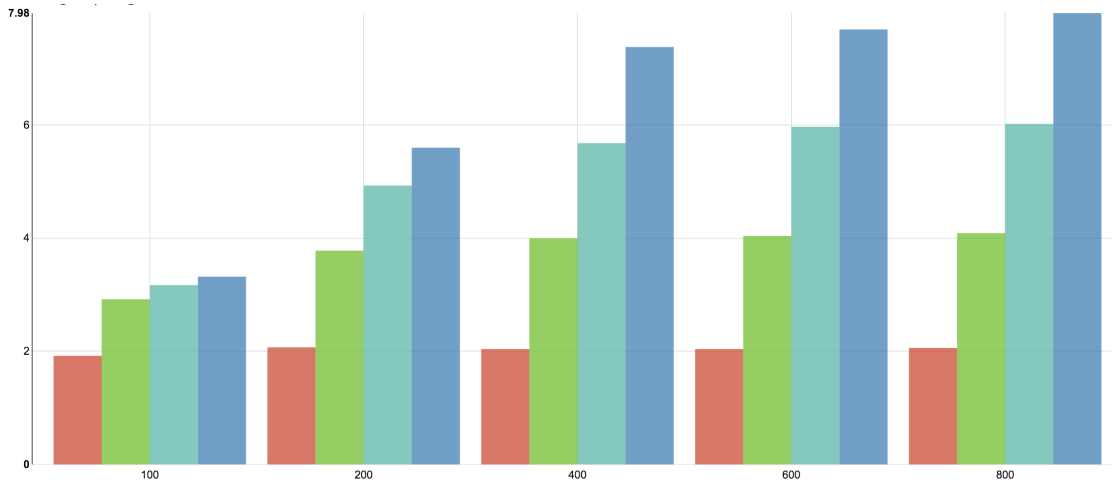

Figure II: Speedups reached against sequential program.
The bars (in order) represent the 4-node, 8-node, 12-node, and 16-node runs.

4.4 In-depth analysis

From the tests in section 4.3, we can conclude that doubling the problem size (in bodies) quadruples the execution time for the sequential version. This also holds for the parallel version, but only on sufficiently large problem sizes, starting from 400 bodies. This means the relationship between the sequential and parallel version is linear; for any sufficiently large problem size (in bodies), if the sequential version takes M seconds to compute, the parallel version will take $M/(S * N)$ seconds, with S the factor of speed-up a single node accomplishes and N the amount of nodes. Concluding on the tests ran, S appears to be around 0.5.

Why does the parallel version of the program, when ran on one node, perform so poorly compared to the sequential version? The answer lies in the way an iteration is handled by the parallel version. Due to the structure of the code, which was optimized for multi-node execution, even when no other nodes are available, the one computing node will still make a call to `broadcast_positions` every iteration, which in turn contains the logic for packing and unpacking data, as well as a call to `MPI_Bcast`. Such logic is of course absent on the sequential version. While this could be fixed by checking on total node count before calling the function, this is fairly useless; why would one use the parallel version if they intend to use one node only? Furthermore, checking on the total amount of nodes would slightly slow down the application.

This overhead of the `broadcast_positions` function, which is required for the parallel application but does not contribute to the computations, is also the cause of the application not reaching the ideal speedup of N when given N nodes.

4.5 Tests with OpenMP enabled *(Bonus assignment)*

We also study the effect of the OpenMP addition enabled by running a few additional tests. Since we effectively increase the amount of threads used from 16 to 256, we are expecting a huge speedup here.

Absolute runtimes, *in seconds*					
test		**sequential**	**16 nodes, no OpenMP**	**16 nodes, with OpenMP**	**Total speedup**
bodies	**steps**				
50	10^4	1.9 s	1.4 s	1.5 s	1.3
100	10^4	7.3 s	2.2 s	2.2 s	3.3
200	10^4	29.1 s	5.2 s	3.0 s	9.7
400	10^4	115.9 s	15.7 s	4.0 s	29.0
600	10^4	260.8 s	33.9 s	5.3 s	49.2
800	10^4	465.4 s	58.3 s	7.7 s	60.4
1600	10^4	x	x	23.9 s	N/A
3200	10^4	x	x	79.3 s	N/A

Table III: Performance measurement of the OpenMP addition to the OpenMPI implementation. For times, lower is better. For speedups, higher is better. Total speedup denotes OpenMP-enabled versus sequential. Fields marked with x were not available from the previous tests, and take too long to be tested on the DAS-4.

As can be seen in table III, the OpenMP addition only performs well when the problem size (body count) is sufficiently big. This can be explained by the if clause discussed earlier, which causes multithreading to not take place if the problem size is small. Recall that first MPI splits the problem size in 16, and afterwards OpenMP will do the same.

However, once the point of a sufficiently large problem size is reached, the gains are huge, with a speedup of over 60 for the 800-body problem. In addition, the combined OpenMPI/OpenMP application is easily capable of running extremely large problem sizes that were not feasible to be ran sequentially or only with OpenMPI enabled.

If we assume the sequential runtime to quadruple for each doubling of the problem size, we can assume a sequential run time of approximately 5580 seconds for the 3200-body problem. In this case, the theoretical speedup of the OpenMPI/OpenMP application would be slightly over 70.

5 Conclusions

Message-passing is a powerful tool to spread the computation of complex simulations over multiple computing nodes. However, careful consideration needs to be taken to reduce the amount of data transfers, which are very expensive. Furthermore, using only message-passing is not enough to achieve maximum performance—local parallelization requires equal care. Furthermore, not each problem lends itself well to parallelization. Even algorithms that lend themselves well for parallel implementation may run better locally, especially if the problem size is small.

The proposed parallel version makes use of both local and cross-node parallelization to achieve maximum performance. It has been proven to provide a speedup of over 60, but it is very likely to perform even better if the problem size is further increased.

Higher speedups could probably be reached by offloading the computation to a GPU if possible. Another way to increase performance would be to reduce the frequency of broadcasting body positions around, since broadcasting is the most expensive overhead

in this application. However, doing so will affect the correctness of the application, so when strict correctness is required, this second option will not do.

6 References

[1] http://www.open-mpi.org/
[2] http://openmp.org/wp/
[3] https://en.wikipedia.org/wiki/OpenMP