A Multithreaded N-Body Simulation

Sam Moore
20503628@student.uwa.edu.au

Contents

1 Introduction

An N-Body simulation's purpose is to solve for the positions and velocities of an arbitrary number of interacting bodies using numerical methods.

Complex systems involving large numbers of bodies, interacting over long periods of time, are often of interest. Hence there is considerable motivation to improve the speed (amount of time simulated over real time required) of N-Body simulations. This report focuses on increasing the speed of an N-Body simulation by introducing parallelism using threads.

The bodies in this simulation interact according to Newtonian laws of gravity. The force acting on a body A due to body B is given by:

$$\boldsymbol{F_{AB}} = G\frac{m_A m_B}{|\boldsymbol{r_B} - \boldsymbol{r_A}|^3}(\boldsymbol{r_B} - \boldsymbol{r_A})$$

Where m_A and m_B are the masses of the two bodies; $\boldsymbol{r_A}$ and $\boldsymbol{r_B}$ are the position vectors of the bodies, and G is the universal gravitational constant.

A template single threaded program was already provided by UWA. Some modifications were made in order to simplify the process of introducing threading to the program, and in one case to correct an error in the program.

2 Euler's Method

Euler's method has been used. A continuous time interval is approximated by a series of steps of length Δt, during which each body undergoes motion with constant acceleration.

The algorithm in pseudo-code is:

```
FOR each time interval of duration dt LOOP
        FOR each body A // can be multithreaded
                COMPUTE the force acting on A due to all other bodies B
        DONE

        FOR each body A // can be multithreaded
                COMPUTE the velocity of A as:
                        V += (Force on A / mass of A) * dt
                COMPUTE the position of A as:
                        X += V * dt
        DONE
DONE
```

The pseudo-code is misleading; it may appear that the two loops could be combined into a single loop. However, in the case where bodies are interacting, the force on any given body will depend upon the positions of other bodies. Consider two bodies A and B interacting within a single time step. If the force on A is computed, and A's position is updated before the force on B is computed, then the force experienced by B will not be equal and opposite to that experienced by A - violating Newton's third law.

The template program provided by UWA places force and position updates for each body in the same loop. This can be fixed by moving the position updates to a separate loop. In the multithreaded implementation, it is also important to ensure that *all* forces are computed before any positions are changed, and vice versa before force computations may continue.

3 Parallelisation

The original algorithm can be parallelised by dividing each of the two loops indicated between a number of concurrently running threads. Forces may be updated for separate bodies simultaneously with no risk of conflict, since the force of one body is independent of all others. Similarly, positions may be updated simultaneously. However, as discussed above, force and position updates must be done separately.

The most effective way of parallelising the computations is as follows (not showing graphics related code):

```
CREATE n threads
FOR each thread T assign a subset of bodies

FOR each thread T
        FOR each time interval of duration dt LOOP
                FOR each body A in T's subset
                        COMPUTE the force acting on A due to all other bodies
                DONE

                BARRIER Wait for all threads to finish computing forces

                FOR each body A in T's subset
                        COMPUTE the position of A
                DONE

                BARRIER Wait for all threads to finish computing positions

        DONE
DONE
```

This algorithm avoids the cost of repeated thread creation and deletion; threads are only created once. The barriers between force and position updates ensure that forces and positions are never updated at the same time in separate threads.

3.1 Graphics

The template program provides graphics using OpenGL and freeglut. In the multithreaded implementations, the graphics can be run in the main thread independent of computations with no effect on the final output of the simulation. However, the following problems may arise if the graphics thread is not synchronised with position updates:

1. If a body's position is being altered whilst the graphics thread is drawing that body, the body may be drawn partway through the update of the three position coordinates. This will cause the body to be drawn displaced from its actual position by a small amount.
2. Because position updates are finished in any order, with the barriers correctly implemented, the display may show some bodies remain stationary whilst the thread responsible for those bodies waits for other threads to reach the barrier.

To avoid these problems, the graphics thread must not commence drawing if position updates are in progress. Similarly, position updates must not commence whilst the graphics thread is still in the process of drawing.

4 POSIX Threads (pthreads)

4.1 Barriers

Because `pthread_barrier` is unavailable on many UNIX systems, it was necessary to implement a barrier.

The barrier functions as follows:

1. Initialised as "inactive" and with the total number of participating threads specified
2. When a thread calls `Barrier_Join` the barrier becomes "active" (if it wasn't already)
3. Each thread entering the barrier by calling `Barrier_Join` increments a counter.
4. Threads calling `Barrier_Join` must sleep using `pthread_cond_wait` until the counter reaches the total number of participating threads
5. The final thread to enter the barrier awakens the sleeping threads using `pthread_cond_broadcast`, and sets the barrier as "inactive".

The concept of "inactive" and "active" barriers was useful for synchronising the graphics thread with the position computations. The function `Barrier_Wait` allows a thread to pass the barrier if it is inactive, but forces the thread to wait if the barrier is active. The calling thread does not affect the barrier's counter of finished threads.

The function `Barrier_Enter` can be called in a thread to set the barrier as "active", and then continue to perform tasks *before* joining the barrier. ie: The barrier may be considered "active" by calls to `Barrier_Wait` before any threads have called `Barrier_Join`.

By calling `Barrier_Enter` in the position threads before the position updates begin, and `Barrier_Wait` in the graphics thread before the drawing begins, the graphics thread cannot commence drawing whilst positions are being updated. Similarly, calling `Barrier_Enter` in the graphics thread before drawing, and `Barrier_Wait` in all position threads, ensures that the position updates cannot be started whilst drawing is in progress.

To avoid deadlocks, three separate barriers have been used; one for each of the force, position and drawing updates.

4.2 Division of Labour

In the pthread implementation, some mechanism is needed to divide the bodies between threads. This has been done by creating a data structure to describe an array of bodies, with the length of the array. Division of the array into subsections can be accomplished in the C programming language using pointer arithmetic.

A global array of these data structures (`System`) is constructed before thread creation, with each `System` containing a subsection of the original bodies array. Each thread is passed a pointer to one of the `System`s through `pthread_create`

Some helper functions have been implemented to perform tasks for a given `System` of bodies, and these are called where needed in the threads.

4.3 Single Threaded Section

One disadvantage of having all computation threads continually running is that it is difficult to execute serial code in only *one* thread. For example, a periodic printing of performance metrics and the updating of the number of steps computed must both be performed at the end of every time step by only one thread. To assist in executing serial code in one thread out of a "team", a function has been added to the `Barrier` implementation; `Barrier_JoinCall` will cause the last thread entering the barrier to execute a specified function containing serial code. This takes advantage of the fact that serial code can often occur immediately after a barrier.

5 OpenMP

The implementation in OpenMP is significantly simpler than pthreads. OpenMP automatically divides labour for threads encountering `omp for`, which eliminates the need for constructing the `System` array. OpenMP also incorporates implicit barriers after most code sections. The use of `omp single` directives greatly simplifies the assignment of serial code to one thread in a team.

Because OpenMP does not offer as much low level control over threads, the integration of graphics with the simulation differs. Specifically, because OpenMP does not have a barrier structure (instead using `omp barrier` directives), all threads involved in a barrier must occur within the same `omp parallel` section.

Instead of creating a separate thread to run graphics, the OpenMP solution uses an `omp master` directive to perform graphics related code within the main computation loop. This occurs after the position computation barrier, and is followed by an explicit `omp barrier`. It is important that the graphics related functions be run in the main thread; using `omp single` will often result in unusual behaviour, such as flickering of the display or the display not being drawn at all.

6 Barnes Hut Algorithm

The Barnes Hut algorithm approximates a distant group of bodies by the group's centre of mass. This is done by the construction of a tree representing regions of space containing groups of bodies. Nodes at each level of the tree represent a volume obtained by dividing the volume of the parent. The force on a body can be approximated by a depth first search on the tree, stopping at nodes for which the centre of mass is "distant" to the body, otherwise continuing to the nodes children until nodes containing only one body are reached.

Figure 1 shows a graphical representation of the Barnes Hut algorithm. In this image, only Nodes (depicted as cubes representing the space within the node) containing exactly one body are shown.

It is possible to parallelise the recursive algorithms used in constructing the tree and calculating forces. However, we have only implemented a serial version of the tree. Despite this, the calculation of forces can still be parallelised external to the tree.

I have reached the page limit... but I have lots of graphs to talk about still. Feel free to stop reading here.

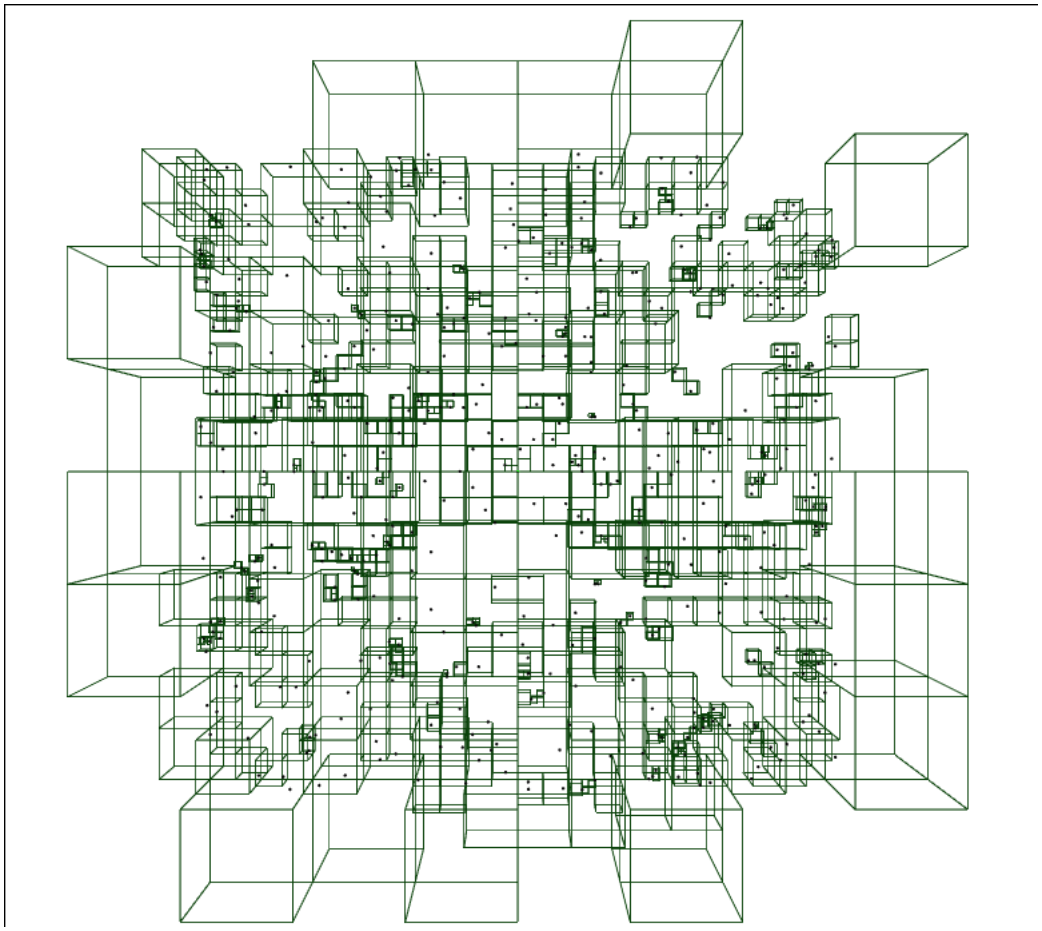

Figure 1: Barnes Hut - One body per cube

6.1 Errors

Figure 2 shows the mean "error" between the Barnes Hut Algorithm and more exact brute force algorithm. This error is calculated by summing the absolute difference of each body's position components, and averaging over all position components. Note the logarithmic y axis. The error increases roughly linearly. There is little change in error for different values of θ between 0 and 10.

Note: The same error analysis script confirmed that the multithreaded versions of the brute force algorithm gave the same outputs.

6.2 Performance for different θ values

Figure 3 shows the time taken for the (single threaded) Barnes Hut Algorithm to run a specified number of steps, for different values of θ. The brute force algorithm is shown for comparison; for $\theta < 1$, the brute force algorithm is generally faster. Taking into account both figures 3 and 2, it would appear that $\theta = 10$ is a good balance between accuracy and speed, for this particular field of bodies.

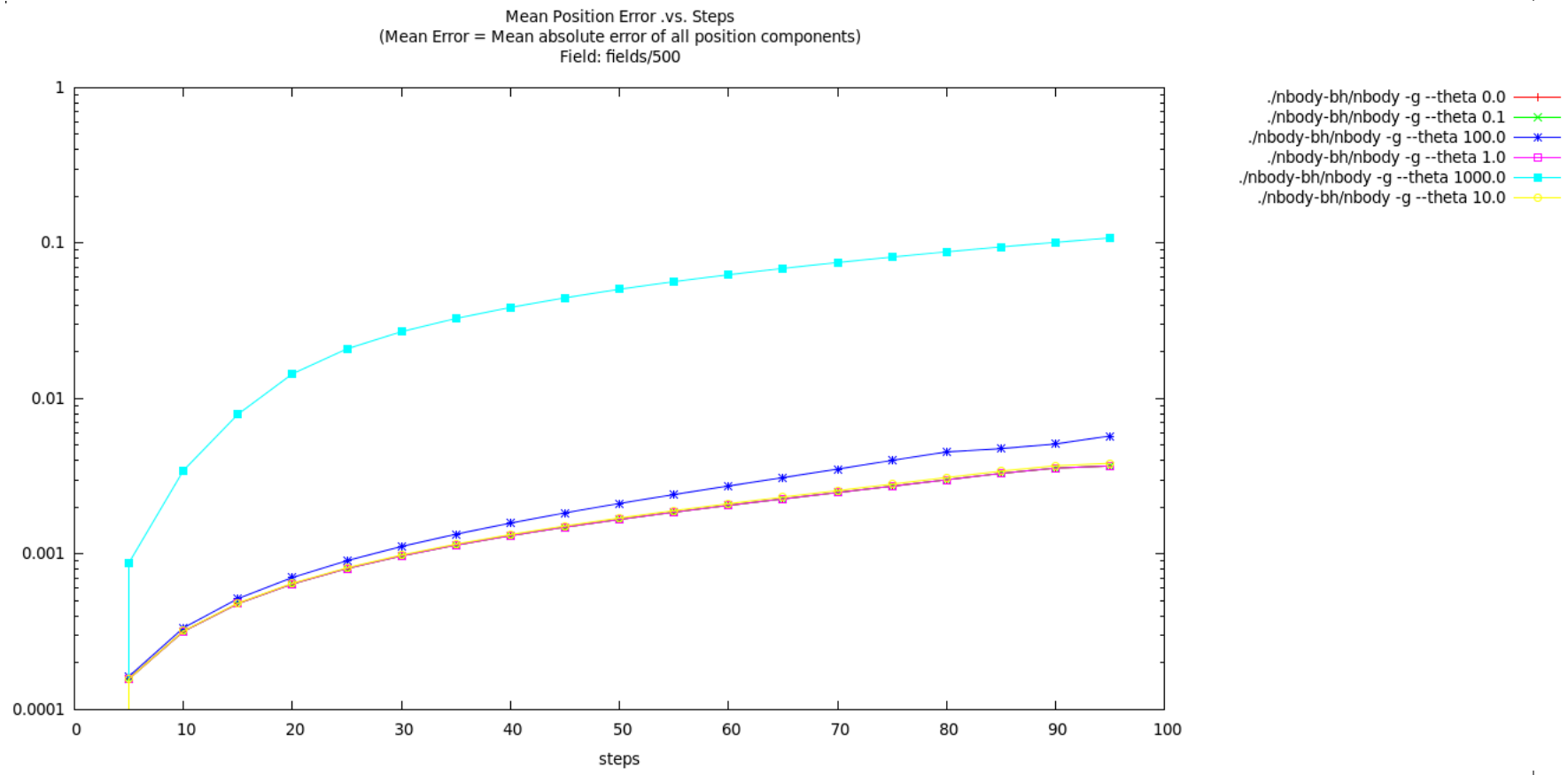

Figure 2: Barnes Hut mean error

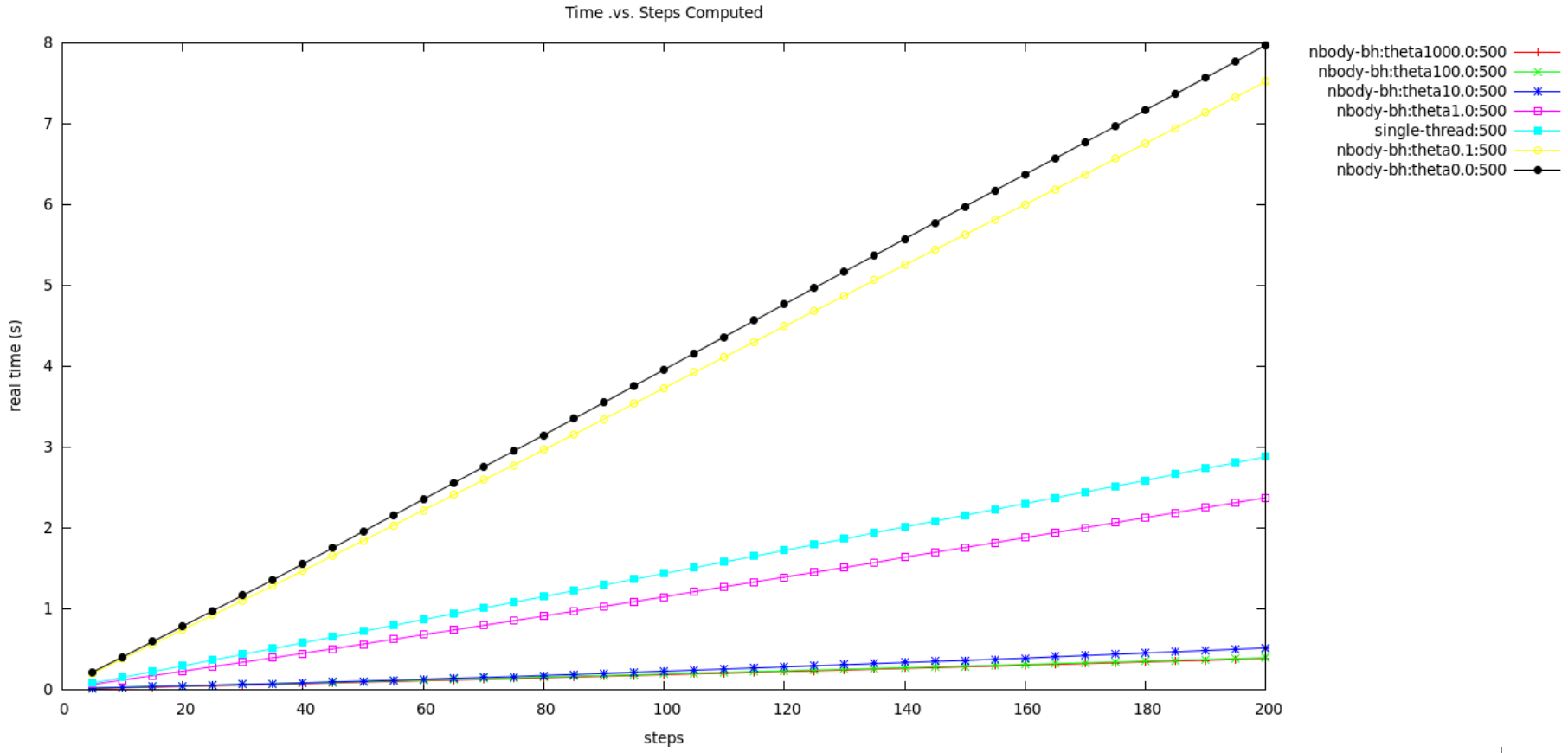

Figure 3: Barnes Hut single threaded performance

7 Performance Analysis

7.1 Time vs Steps

Figure 4 shows the time taken to compute a varying number of steps for different programs. The field provided with 500 bodies was used.

Several things can be noted from this graph:

- With $\theta = 10$, the Barnes Hut algorithm is significantly faster than brute force algorithms
- Altering the number of threads has a significant effect on the brute force algorithm performance, but little effect on the Barnes Hut algorithm performance. This is because the vast majority of the Barnes Hut algorithm has not yet been multithreaded.
- The optimum number of threads is equal to the number of cores available on the host machine (Intel Core i5). (Pleiades was not available at the time of testing).
- The pthreads and openmp implementations perform with similar speeds
- The single threaded, brute force algorithms perform the worst

The time taken to run varies with CPU load due to other processes, and so these results should be considered as qualitative. Measurement of CPU cycles (as opposed to real time) is not a suitable metric, since on many UNIX systems, this measurement includes cycles run by all threads in a process.

7.2 Time to compute vs Number of Bodies

Figure 5 shows the time taken to compute 1000 steps against the number of bodies in the initial field. The brute force algorithms appear $O(n^2)$. Barnes Hut appears to be $O(nlog(n))$. The fastest algorithm is Barnes Hut with $\theta = 10$.

This experiment revealed (and was limited by) a flaw in (my implementation of) the Barnes Hut algorithm. When the body fields are randomly generated, the result is usually unbound orbits. As the size of the volume occupied by bodies is increasing, the algorithm must continually resize the root node of the tree, and then all children nodes, and ultimately create additional child nodes to contain single particles. Although the implementation attempts to minimise allocations of memory, it was found that for large numbers of bodies the program often became unresponsive. This may be due to the more closely approaching bodies experiencing extremely large forces, and moving away from the centre of the universe at high velocities. .

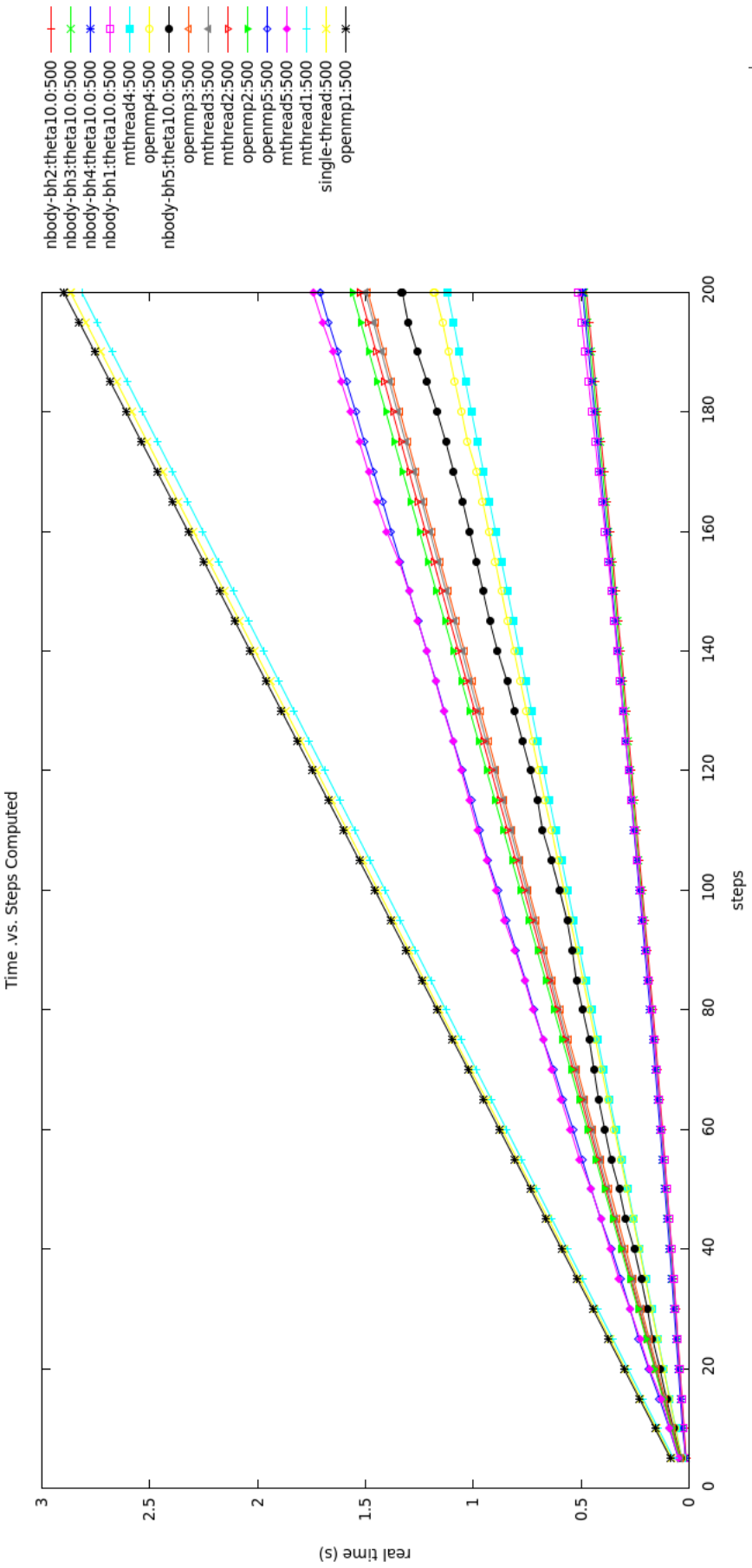

Figure 4: Implementation Performance

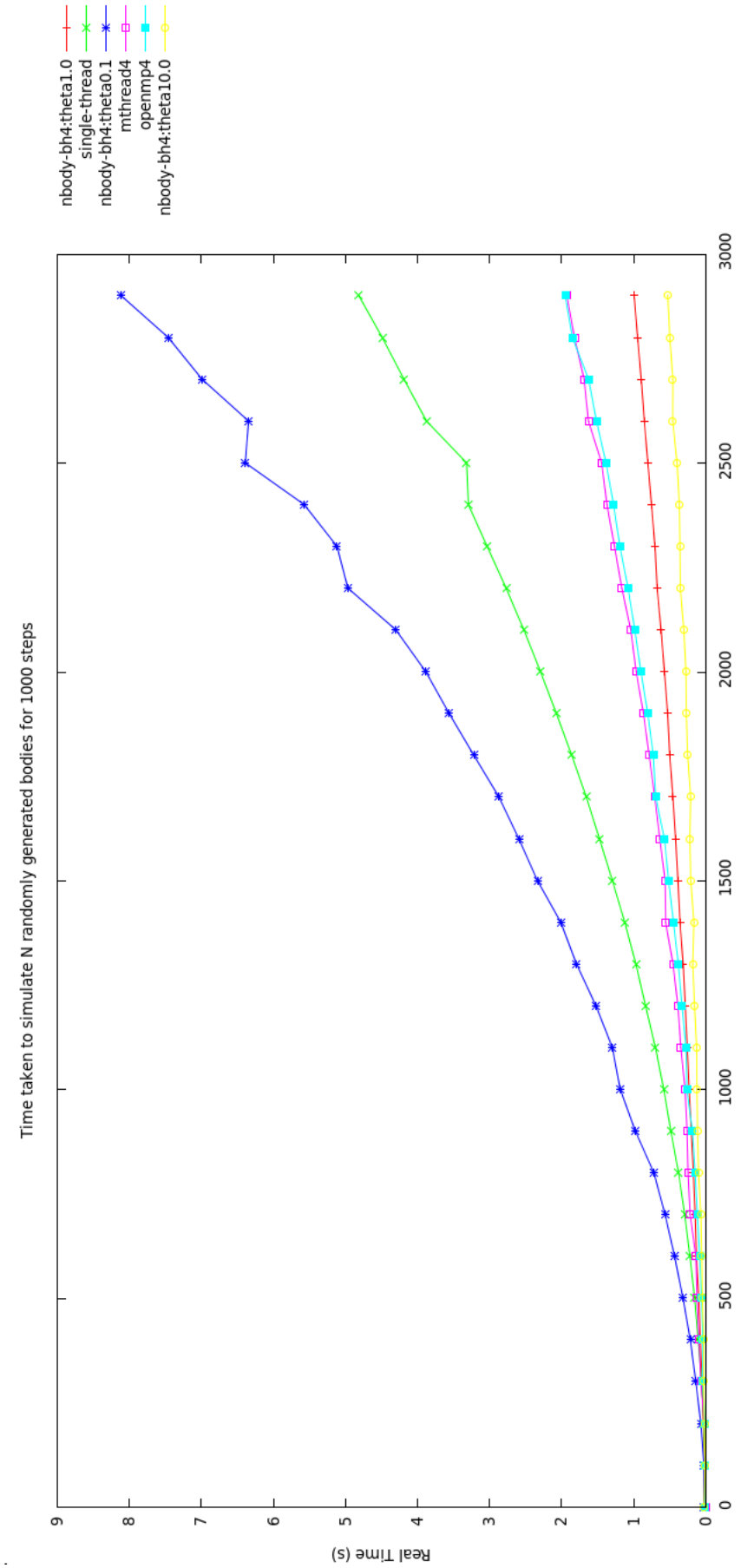

Figure 5: Efficiency vs n

7.3 Analysis with Callgrind

The tool *callgrind* (part of *valgrind*) can be used to generate information on the performance of a program, and to identify expensive lines of code or the cost of different function calls. Figure 6 shows a screenshot of *kcachegrind*, a graphical tool to display the results obtained by callgrind. The program analysed is the Barnes Hut N-Body simulator, running for 10 steps, without graphics enabled.

A larger copy of the image is attached with this report, file name "nbody-bh/kcachegrind.png". Several things are clear from this display:

1. The most expensive computations are due to the force computation function `Node_Forces`. Close to 93% of the program's time is spent in this function.

2. Although there are only 500 bodies, the force calculation algorithm was called 15200000 times. This is significantly greater than the approximate 2500000 force computations for the brute force algorithm.

   For this particular initial field and choice of value for θ (0.5), the single threaded Barnes Hut algorithm performed worse than the single threaded brute force algorithm. As shown in Figure 5 however, the Barnes Hut algorithm performs extremely well for larger numbers of bodies. The "better" algorithm to use is determined by the problem.

3. Most of the cost of a force calculation is due to two calculations (distance calculation and the actual force update) The calls to `sqrt`, a normally expensive function have a much lower total cost, but are called the same number of times. The extra cost of the two calculations might be due to the costs of following the pointers (the code has not been optimised),

4. The `for` loops, which were used after replacing the original 9 body variables (x, y, z, Vx, ..., Fx, ...) with three arrays, to condense the code, have a small but significant cost associated with them.

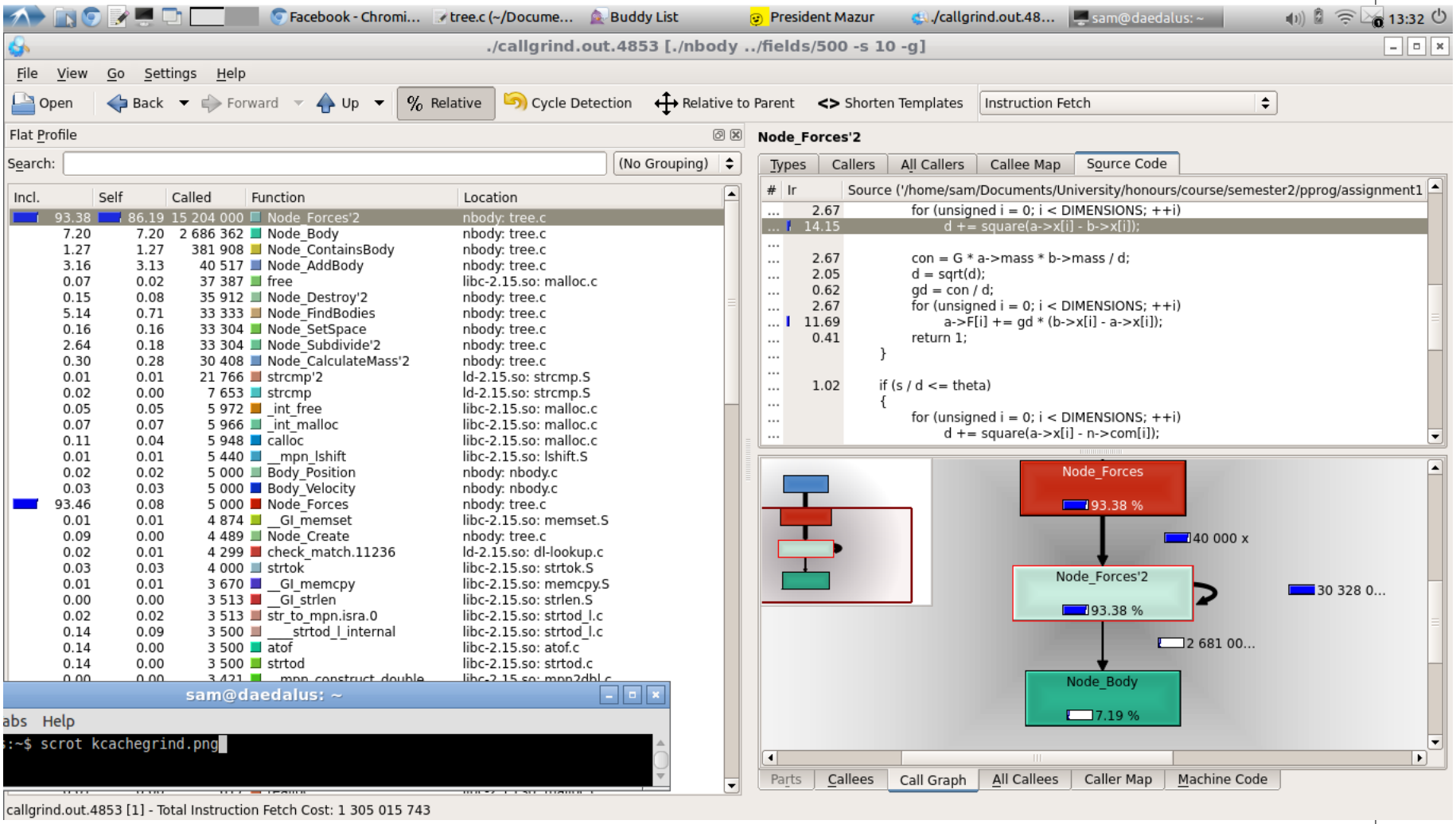

Figure 6: Kcachegrind reveals why my program is slow