N-Body Simulations

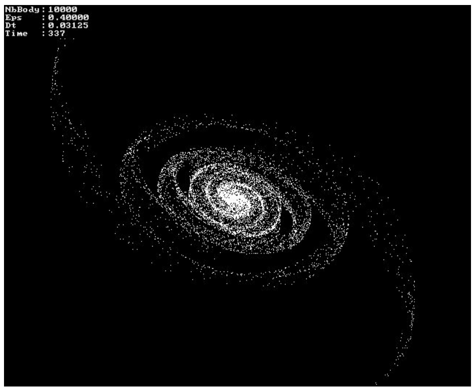
EC527
Boston University
Spring 2013

Project Report

Kushal Prasad
Baris Tevfik

1.Introduction

An N-Body is a simulation of a dynamical system of particles, usually under the influences of forces such as gravity. The classical N-Body problem simulates the evolution of a system of N bodies, where the force exerted on each body arises due to its interaction with all the other bodies in the system. The simulation proceeds over timesteps, each time computing the net force on every body and thereby updating its position and other attributes. If all pairwise forces are computed directly, this requires $O(N^2)$ operations at each timestep. Hierarchical tree-based methods have been developed to reduce the complexity. N-body algorithms have numerous applications in areas such as astrophysics, molecular dynamics and plasma physics. Below figures are examples of N-Body simulations in the area of astrophysics. Figure1 shows a screenshot of simulated galaxy formation where as figure2 shows a simulation of two galaxies colliding into each other.



```
NbBody : 10000
Eps    : 0.40000
Dt     : 0.03125
Time   : 337
```

Fig1. N-body Cosmos
Source: http://code.google.com/p/nbody-cosmos/

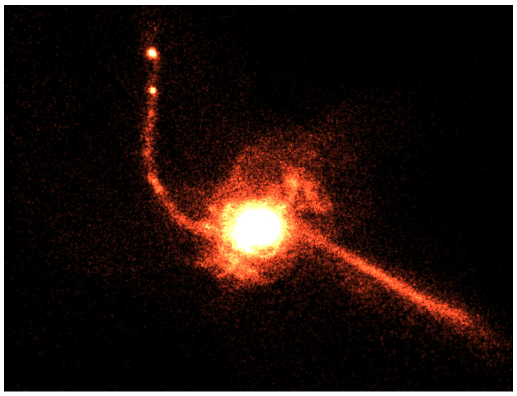


Fig2. Galaxies Colliding
Source: http://www.projet-horizon.fr

Although N-Body simulations have many applications, in our project we focused on simulating the behavior of particles under each others influence taking into account only their gravitational force. We have developed several versions of this simulation ranging from a naive, $O(N^2)$ complexity serial version to a Cuda implementation and an implementation using Barnes-Hut algorithm. This algorithm is a tree-based method that reduces the complexity of the problem to O(N Log N).

In this report we will discuss each version of the implementation of N-Body gravity simulation along with our experiments and results.

2.Serial

Newton's law of universal gravitation only accounts for two bodies, m1 and m2. However, in space there are more bodies to be accounted for. We can generalize this problem to have a number of N bodies to deal with. Thus, we can create an equation that can deal with an arbitrary number of bodies. It is assumed that no external forces are acting on the system. When studying the N-body problem, we need to focus on a single body, whose motion is of our primary interest. We assume that this body, known as the ith body is able to move freely, but the other N - 1 bodies are stationary. The total force of gravity on the ith object is given as:

$$\mathbf{F}_g = -\sum_{k=1}^{N-1} \frac{G m_i m_k}{r_{ki}^3} \mathbf{r}_{ki}$$

r is the vector distance between object k and object i.

In the next section we will discuss our first implementation of this equation where each pairwise forces are calculated directly.

a. Naive Implementation

The implementation firstly initializes each particle with a random position, random velocity and a random mass in a range. There are three main loops that iterate through each element in the array. Each particle is picked up and the forces exerted by all the other (N-1) particles on it are analyzed. The cumulative acceleration and particle mass are used to determine it's velocities and position after a time-step of 'dt' units and stored in temp array. Once all the particles next positions are determined, they are updated into the original array. Below code shows what each of the three loops do in the simulation of N-Body gravity. Please refer to *serial1.c* for the complete code.

```
for(i = 0 ; i < len ; i++)
  {
          //iterate over all particles.
      for(j = 0 ; j < len ; j++)
       {
            //calculate forces on i-th particle due to all other particles.
       }
          //update i-th particle properties.
  }
 for( i = 0 ; i < len ; i++)
  {
        //update all particles position.
  }
```

b. Improvements

After implementing the serial version we have realized that we can reduce the memory fetches on retrieving i-th element of array, thus improving the speed. The original code calls same *array[i]* inside the loops for many times which adds unnecessary overhead to the program. By simply assigning array[i] to a global variable, we have been able to increase the running speed for small amount.

```
particle par = array[i]; //par is a struct of particle type.
```

serial2.c contains the code for this improvement.

Another improvement we have made on the serial code was to do a loop unrolling. We have unrolled the inner loop 2 and 4 times to achieve improvements on the serial code. This technique reduces in general reduces branch penalties thus giving better performance. We have observed that unrolling the loop beyond 4 times did not result any improvements, contrary it reduced the running speed. This is possibly the increased cache usage too much which reduces the performance. Files *serial3.c* and *serial4.c* contain the 2 times and 4 times unrolled versions of the serial code respectively.

We present our results for naive implementation and improved serial implementations in the following section.

c. Experiments and Results

We have run our code on the machine with the following specifications:

Intel(R) Xeon(R) CPU W3505 @ 2.53GHz Dual Core.

We have set our element size range between 5000 elements and 55000 elements, with delta = 5000. We have decided that this range of size for the experiments is adequate enough since that the overall shape of the results are exponential, thus it does not greatly change for larger sets of elements.

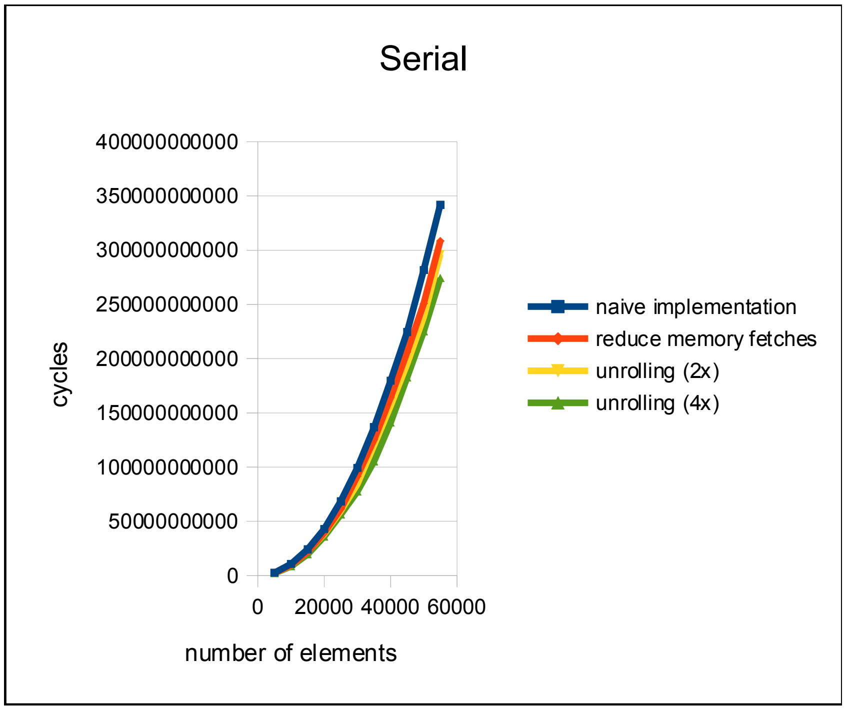


Fig3. Results for various serial implementations.

As it can be seen from figure3 our first naive implementation has the worst performance and although the improvements on serial code actually affected the overall running speed of the simulation, it did not greatly reduced it. This is due to the nature of the implementation which has a complexity of O(N^2) and also due to the fact that the code is serial and every element has to wait for all the other elements position to be calculated. We can greatly increase the performance by implementing a threaded version of this algorithm which we will discuss in chapter 3.

3.Parallel (CPU)

At every single timestep, each particle in the system of N-Bodies updates their current position due to the other particles' gravitational force at that point of time. And it does not depend on the previous calculated force or a neighbor such as in SOR, therefore we can easily partition the problem into blocks and assign each thread into these blocks to work on. After all blocks finish calculating the new properties we can then again update the new positions as it was done in the serial code.

a. Implementation

Following snippet of code shows the serial part of the code which is basically the creation of threads and then joining them after they are all done.

```
for (i = 0; i < NUM_THREADS; i++) {
    end = (len / NUM_THREADS)*(i+1);
    start = (len / NUM_THREADS)*i;
       //create threads
   }
  for (i = 0; i < NUM_THREADS; i++)
  {
       //join threads
  }
```

We pass the *start* and *end* index of blocks into threads using a struct called *threadarg*. The following code is the work thread where each thread executes from the their start of the array to the end of the array. The code's first barrier is where each thread waits for all the other threads to update the i-th particle property before it is allowed to update all particles positions. The second barrier is there to ensure that all the updation is completed before the program exits.

```
void *work(void * threadarg)
{
for(i = start ; i < end ; i++)
    {
       //iterate over all particles for this thread.

       for(j = 0 ; j < len ; j++)
        {
                 //calculate forces on i-th particle due to all other particles.
        }
            //update i-th particle properties.
    }

  //wait for all threads

  pthread_barrier_wait(&barr);

     for( i = start ; i < end ; i++)
  {
       //update all particles positions
  }
  //wait for all threads

  pthread_barrier_wait(&barr);
}
```

b. Experiments and Results

We have run our pthread experiments on the same set as the serial versions ranging from 5000 elements to 55000. We have experimented with the computers on the engineering grid as well as the lab machines. The machine info for grid computers are as follows;

bme.q = AMD FX(tm)-8150 Eight-Core Processor cpu MHz: 3612.465
me.q = 32-Core Processor 2.1 GHz

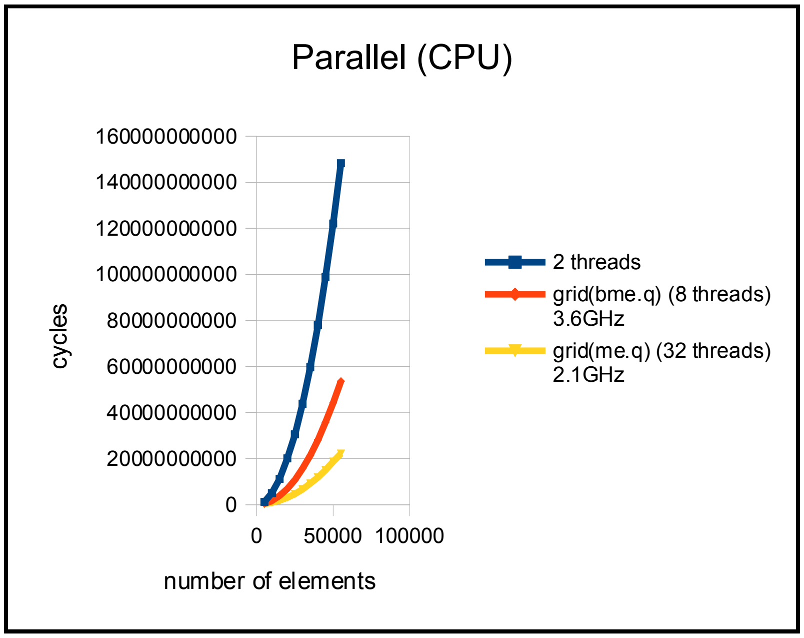


Fig4. Results of Parallel CPU code

2 thread version was run on the lab machine that is dual core. 8 and 32 thread versions were run on the eng-grid (bme.q and me.q)

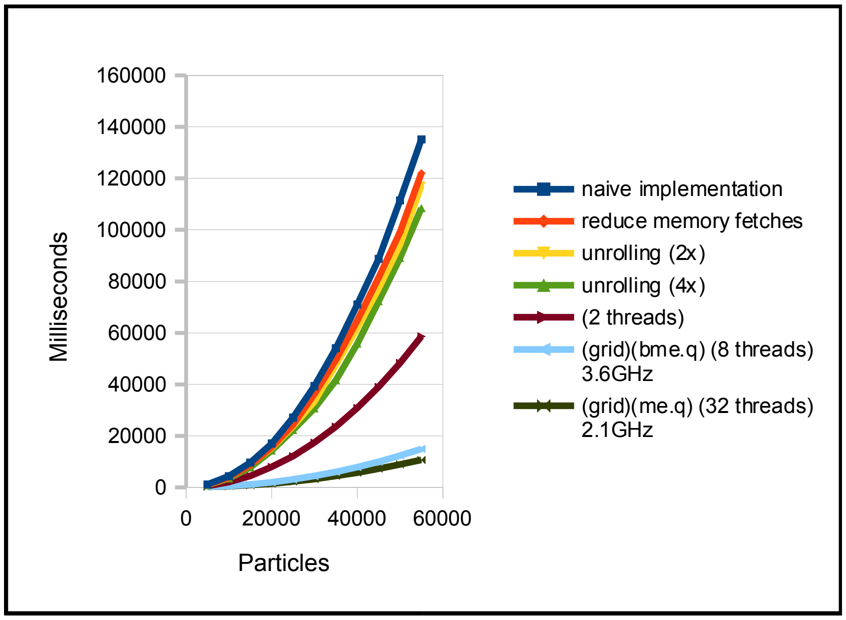


Fig5. Combined results of Serial and Parallel code

4.GPU (Cuda)

We initialize the particle array on the host CPU and then copy the contents over to the CUDA memory as a 2-D array. This 2-D particle array is present in the global memory and shared by all the threads.

The threads work on and update particles present in a square block assigned to them (blocks assigned according to their IDs) . But, all the threads read all the particles and their properties to evaluate the forces being exerted on the particle they are evaluating.

We synchronize the threads in order to let all of them finish storing the new positions of the particles in a temporary array after which we allow them to update the original particle array. This is done in order to maintain a constant scene of evaluation for all the particles.

The implementation of the N-Body simulation is in-place. Therefore, the array of particles can be left intact on the GPU memory and further iterations can be carried out without any transfers to the host memory.

a. Implementation

The Kernel's pseudo code for the CUDA implementation of N-Body simulation:

```
__global__ void cudaNBody(particle_ptr array, temp_ptr temp_array)
{
    long int len = (long int) LENGTH; // The length of the particle_array

    // Holds the address at which the thread's current particle resides
    long int addr;

    // Length of the square block the thread is operating on
    int square_len = len/blockDim.x;

    // x-index of the thread's block
    long int x = threadIdx.x*square_len;

    // y-index of the thread's block
    long int y = threadIdx.y*square_len;

    // for the i-th iteration, the i-th particle's position in the particle array
    addr = k * LENGTH + i;

    // Serial Implementation of the N-body
        // Operate on each thread's 'Block' serially
        …
        // Hold all the new positions in a temporary array
    // End serial implementation
    __syncthreads();

    // Transfer the new positions over to the particle array
    __syncthreads();
}
```

b. Experiments and Results

We ran the experiments on a large range of particle array sizes.

for 256*256 particles;
Max Error = 0.455
Error Percentage = 0.927 %

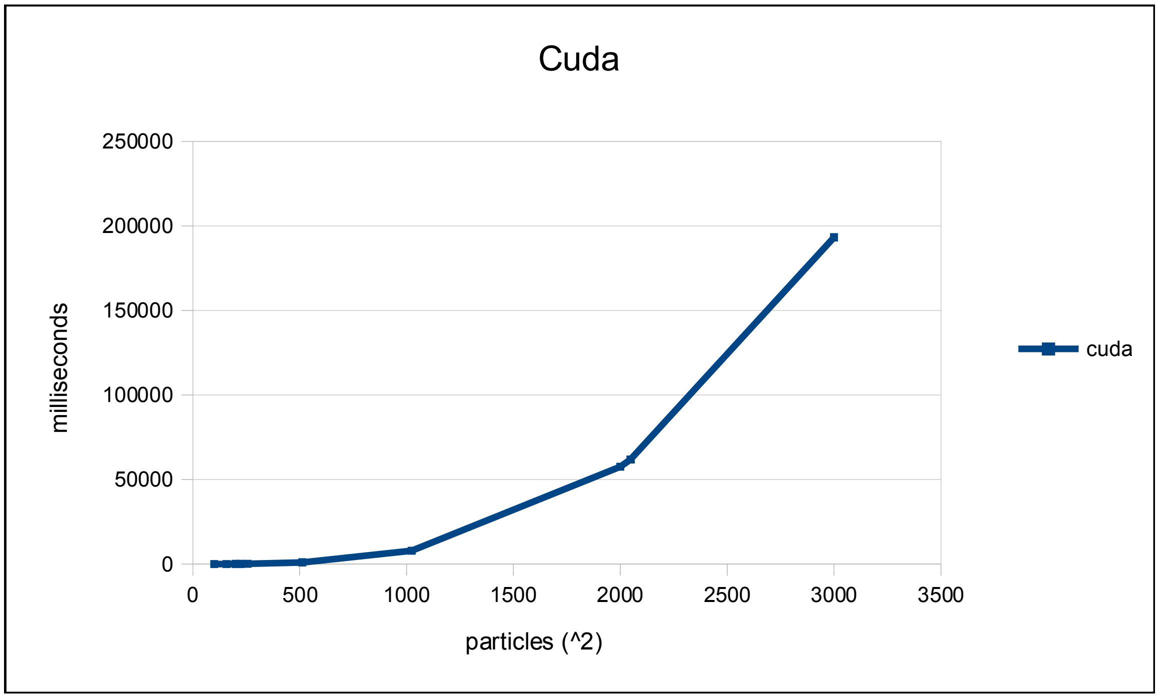


Fig6. Results for Cuda implementation

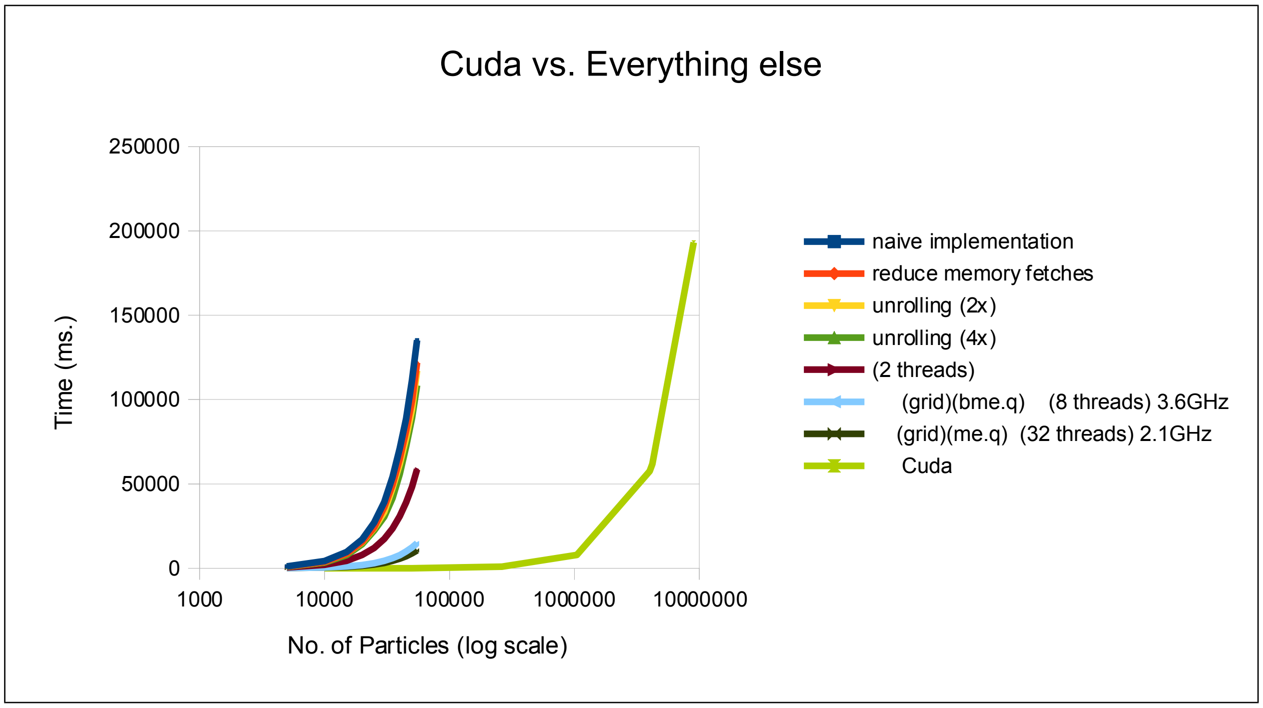


Fig7. Combined results for Serial, Pthread and Cuda implementations

elements	naive implementation	reduce memory fetches	unrolling (2x)	unrolling (4x)	(2 threads)	(grid)(bme.q) (8 threads) 3.6GHz	(grid)(me.q) (32 threads) 2.1GHz	Cuda
5000	1064	958	921	843	481	123	128	
10000	4282	3839	3804	3377	1988	483	406	8.8
15000	9562	8641	8741	7776	4419	1087	844	
20000	16986	15641	15250	14240	7945	1932	1459	31.57
25000	27075	24280	23617	22274	12081	3022	2237	
30000	39267	36216	33339	30599	17323	4364	3194	
35000	54057	49529	47632	41630	23594	5960	4401	
40000	71036	64852	61863	55781	30802	7811	5661	66.49
45000	88765	81543	77263	72249	39057	9929	7169	
50000	111406	99112	94258	89024	48273	12286	8900	85.15
55000	135145	121975	117164	108442	58637	14880	10580	
65536								134.79
262144								1013.78
1048576								7853.52
4000000								57580.46
4194304								61797.39
9000000								193220.36

Table1. Timing results of Serial, Pthread and Cuda. (Time is in ms)

The above table displays the timing results for various implementations that we have discussed so far in this paper. All figures are in milliseconds. As you can see Cuda is clearly the fastest implementation. Next chapter we will discuss Barnes-Hut algorithm and its performance against the other implementations.

5.Barnes-Hut Algorithm

This is an O(n log n) algorithm that computes interactions between distant particles by means of the first order approximation. This technique uses a tree-structured hierarchical subdivision of space into cubic cells, each of which is recursively divided into eight subcells (3D) whenever more than one particle is found to occupy the same cell.

There are three main phases of this algorithm.

- Construct the octree.
- Compute the center of mass for all cells.
- For each body, traverse the tree and calculate the force on that body.

The figure below represents the 2D version of the algorithm. In the first phase, the algorithm recursively divides the n bodies into groups by storing them in a quad-tree. (in 3D, an octree). The root of this tree represents a space cell containing all bodies in the system. The tree is built by adding particles into the initially empty root cell, and subdividing a cell into four children as soon as it contains more than a single body.

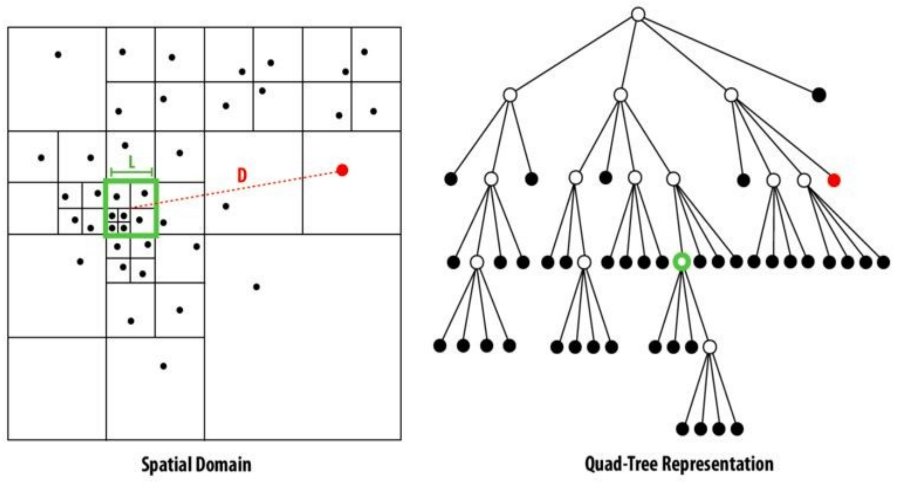


Fig8. Barnes-Hut Algorithm 2D Representation

Source: http://dolores.sp.cs.cmu.edu/15418_spr13/index.php/article/18

Then, it computes the net force acting on each body, traversing the tree starting from the root and conducting the following test on each cell it visits: If the center of mass of the cell is far enough away from the body, the entire subtree under that cell is approximated by a single particle at the center of the cell, and it computes the force contributed by this cell; if the center of mass is not far enough, it goes into the subtrees under that cell and recursively apply this algorithm on them. A cell is determined to be "far away enough" to be considered as a whole if the following condition is satisfied: $l/d<\Theta$. Setting value of Θ from 0 to 1 reduces the complexity from O(n^2) to O(n log n).

In this part of the project we have implemented the 3D version of the Barnes-Hut algorithm. We will present our implementation in the next section and our results later.

a. Implementation

The implementation we have carried out has the following components:

Structures:

1. Particle element:
   - *x, y* and *z* : coordinates
   - *vx, vy* and *vz* : x, y and z components of velocity
   - *m* : Mass
   - *container :* Pointer to node containing the particle
2. Node element:
   - *leaf* (boolean indicator whether node is a leaf)
   - *range* (the offset of the faces of the cube that the node influences)
   - *center* (of the cube that the node represents)
   - *properties (a pointer to a particle)*
     - *the particle would contain the particles in the scene if the node is it's container*
     - *otherwise, the particle would contain information on the node's Center of Mass (COM), cumulative mass etc.*
   - *parent* (a pointer to the node's parent node)

- *child* (an array of eight pointers to the children nodes)
- *children* (a counter of the current number of children of the node)
- *position* (the position at which the node resides in its parent's child array)

Octree and N-Body simulation steps (in order):

1. `init_particle_array( node *particle_array, particle *particle_array )`
   This function binds a randomly initialized particle array to an array of nodes. These nodes will be the 'containers' of the particles that they have in their *properties*. These container nodes do not change throughout the program execution.
   After the particles have been bound to their respective container nodes, the nodes are inserted into the ROOT node and they find their position in the tree.

2. `insert_node( node *parent, node *child )`
   Tries to insert the *child* into the *parent* node. This is done by first finding the position (octant, position in the *parent*'s *child[]* array) at which the *child* fits. If there is another child present at that position, the function calls itself trying to insert *child* into the *parent.child[position]* node. This goes on recursively till the *child* finds a free place to settle in.
   The function considers leaf and non-leaf nodes and dynamically creates more nodes to accommodate the leaves if required.

3. `nbody( node *node_array, point *temp_array, node *root )`
   Carries out the n-body run on the particles (contained in *node_array* elements). Here, the *Multi-pole Acceptance Criterion* is evaluated for every node (starting from the root and moving down the tree) and accordingly the particle is updated. It calculates and assigns the new positions that the particles should be at after a *dt* time-step in the simulation.

4. `particles_update( node *node_array )`
   Iterates over all the particles and finds the current nodes that they should reside in. This means that it checks if the particle has moved out of it's parent's domain, and if it has, finds another parent in which this particle should reside in.

5. `update_nodes_properties( node *root )`
   Starts from the ROOT and digs down till it finds parents with particles in them. Then it iteratively updates the parent's COM and mass. This recursively goes upwards in the tree till all the parent's properties have been updated. This is a necessary step as parent's properties govern the outcome of the *Multi-pole Acceptance Criterion* on which the Barnes-Hut algorithm obtains its speedup.

b. Experiments and Results

Floating point and Double precision results. After 65,536 the code would give segmentation fault for float type, since this is the maximum number that it can represent. That is, for particles very close to each other, they would not be able to find distinct nodes to reside in, as the node's ranges cross (become zero) the maximum precision possible with floating type.

Error percentage is ~ 0.033 %

Tolerance: 0.0001

elements	serial (double)	Barnes-hut (float)	Barnes-Hut (double)
5000	1176.35	34.92	35.44
10000	4496.94	80.45	88.97
15000	10537.87	137.26	143.36
20000	18329.48	191.95	225.8
25000	29327.71	268.98	295.63
30000	40699.47	331.3	368.26
35000	55591.2	405.1	445.4
40000	73668.55	501.75	550.75
45000	95328.67	566.31	636.33
50000	116534.26	651.51	767.61
55000	143039.52	776.05	848.21
65536			1016.75
200000	1791963.19		3964.82
262144			5543
1048576			29511.22
4000000			155098.79

Table2. Serial vs Barnes-Hut Results. (Time is in ms.)

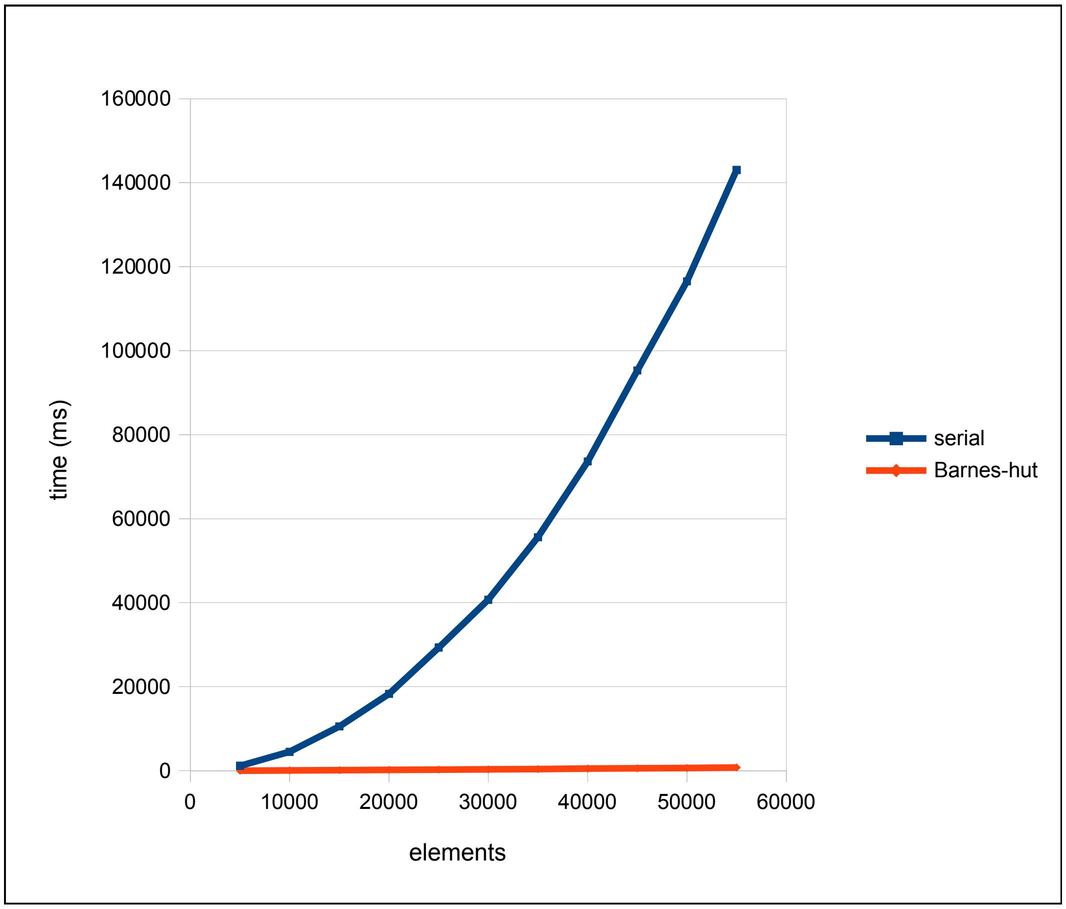


Fig9. Graphical Representation of Table2.

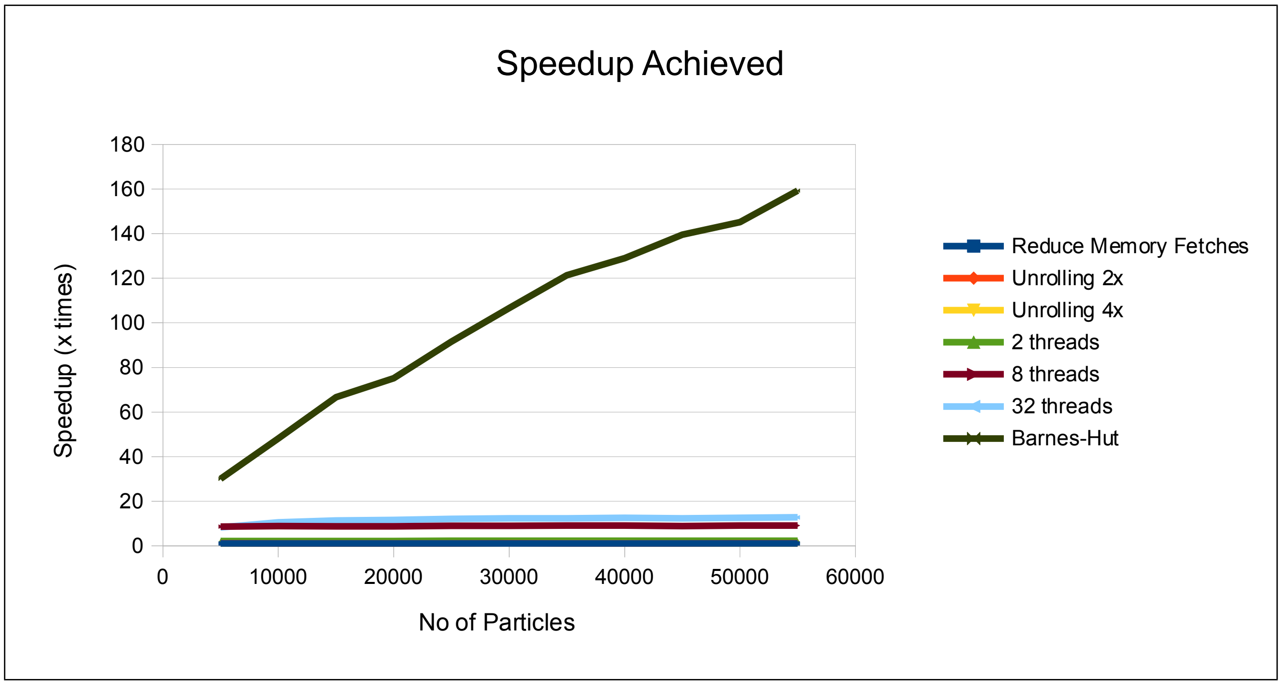


Fig10. Speedup with Barnes-Hut with respect to other serial versions

6. Conclusions

After the comparisons between the different versions of N-Body Simulation, we can conclude that using an algorithmic improvement to the N-body problem pays off very impressively with larger particle numbers. The implementation of Barnes-Hut N-Body decomposition algorithm is highly effective in reducing the iteration time of the simulation.
The CUDA implementation although inefficient in itself, outperforms all others due to the computational power of the GPU with multiple threads.

The Barnes-Hut serial implementation outperforms all other CPU implementations that we tried with a maximum error of 0.03%.

A CUDA implementation of the Barnes-Hut according to us will be the most effective way to perform an N-Body simulation. There are ways to integrate the rendering (OpenGL) with CUDA using a common memory buffer which gets swapped after every iteration.

7. OpenGL Rendering

We used the Glew/Glut libraries for OpenGL and rendered the particles as points in 3-D space. This was done in order to obtain a visual confirmation of the correctness of our programs.

OpenGL allows the user to initialize a 3D scene and draw vector points in it. We used this method to render the particles and passed the actual x, y and z coordinates of the particles as the coordinates of the vector points on OpenGL.

The OpenGL function drawScene() allows the points to be initialized in the scene. The code in the drawScene() function is shown below:

```
void drawScene(void)
{
    long int len = (data_t) NO_OF_PARTICLES;
    long int i;

    // Clear the rendering window
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // Set the drawing color to white
    glColor4f(1.0,1.0,1.0,0.75);

    //Set the drawing mode to points
    glBegin(GL_POINTS);

    for( i = 0 ; i < len ; i++)
    {
        // Insert the points into the scene and normalize the coordinates based on
the limits of the scene

        glVertex3f((particle_array[i].x/INIT_HIGH),
(particle_array[i].y/INIT_HIGH), (particle_array[i].z/INIT_HIGH)*3);

    }

    // End adding points to a scene
    glEnd();

    // Flush the contents onto the screen renderer
    glFlush();

    // Finish rendering
    glFinish();
}
```

A few illustrations of the OpenGL implementation is provided in the next page.

Fig10. Initial Positions of particles.

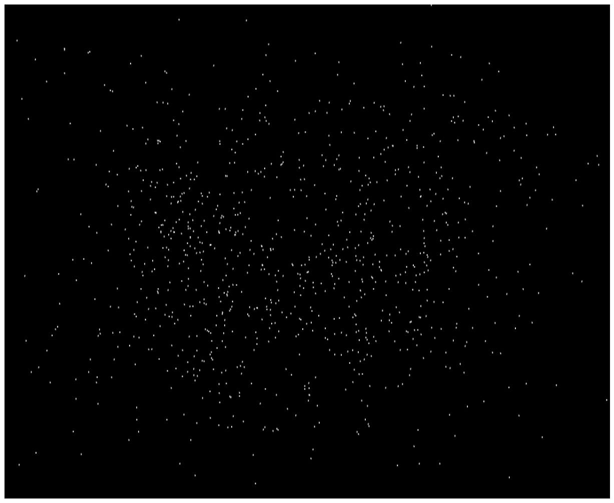


Fig11. Particles have started to interact.

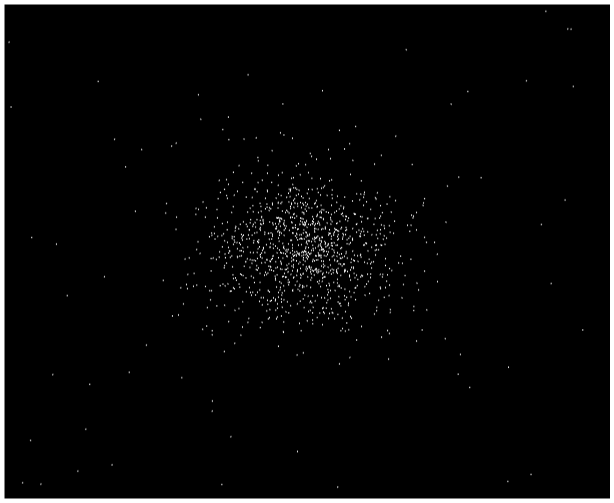


Fig12. Particles have settled into their orbit.

References

1.N-Body Simulation (http://en.wikipedia.org/wiki/N-body_simulation)
2.Astrodynamics/N-Body Problem (http://en.wikibooks.org/wiki/Astrodynamics/N-Body_Problem)
3.Parallel N-Body Simulations (http://www.cs.cmu.edu/~scandal/alg/nbody.html)
4.Fast N-Body Simulation with Cuda (http://http.developer.nvidia.com/GPUGems3/gpugems3_ch31.html)
5.Barnes-Hut Simulation (http://en.wikipedia.org/wiki/Barnes%E2%80%93Hut_simulation)
6.O(N log N) force-calculation algorithm (http://www.nature.com/nature/journal/v324/n6096/abs/324446a0.html)
7.Parallel Computer Architecture and Programming (http://dolores.sp.cs.cmu.edu/15418_spr13/)