ID1217, CONCURRENT PROGRAMMING

The gravitational N-body problem

Evaluation of parallel programs using naive and Barnes-Hut implementations

Frederick Ceder and Joakim Uddholm

3/17/2013

Contents

1. Introduction

The gravitational n-body problem describes the problem of simulating the force of gravity that body masses exert on each other. One typical scenario of this is our solar system, where the earth and other planets go around the sun and moons go around planets. This is generally seen as something that occurs due to the gravitation force.

In this report we present 4 different program implementations that solve the gravitational n-body problem. We then evaluate and discuss which of the version is the fastest. We also look at the speedup gained when using parallel programming. Furthermore we include a graphical version using the LWJGL library that shows the simulation in action.

2. Background

2.1 Physics

In space two bodies or entities (as they will be referred to for the rest of the report) always apply a gravitational force on each other, which can be calculated according to Newton's law of gravitational force. Thus, entities in space will exert force on each other, accelerating or decelerating their movement speed and thus position.

2.2 Accurate algorithm

We use two different algorithms for simulating the gravitational force. The first algorithm runs in $O(n^2)$ time and checks each entity **e** against all other entities to calculate the net force they apply on **e**. We call this the accurate algorithm.

2.3 Barnes-Hut algorithm

The second algorithm uses an implementation of the Barnes-Hut algorithm, which is considered as an approximation algorithm. It runs in O(n log n) time and uses a tree structure to calculate force values. The Barnes-Hut algorithm is theoretically faster as it approximates the net force on each entity, while the accurate algorithm actually calculates for every entity. Depending on the granularity and the width parameters given to the algorithm, the force exerted by one or more entities against a certain entity may instead be approximated instead of calculated. The algorithm does this by grouping entities close to each other and instead calculating a combined force using the center of mass and the combined mass of the group if and only if the entities of that group are considered too far away. Thus the force exerted by entities too far away from each other is considered negligible.

The granularity is the ratio between the width and the distance magnitude, thus the granularity value is a way of setting an approximation level, if width is constant, altering the distance. [1]

Example 1 - granularity in Barnes-Hut:

Width **w** is the maximum span of the space that objects exist in.

If we consider a granularity **g** of value 0.5, then we could evaluate the following:

w/**d** = **g**, where **d** is distance.

If **w** is 10000 units, then we could easily evaluate **d**:

d = **w**/**g** = 10000/0.5 = 20000 Units

Thus if a center of mass of a group of entities is further away than **d**, 20000 Units, from an entity then these could be neglected when calculating the force values, which saves calculations.

It must be stressed that the Barnes-hut algorithm doesn't calculate exact force values, except if the granularity is 0, rather center mass points of a group of entities whose distance away from the relative entity is not greater than **d**.

3. Implementation

3.1 General Structure

Our main program structure consists of the Engine class as well as the EntityUpdater and EngineOutputListener interfaces. The engine class initiated with a list of EntityUpdaters and can optionally be attached to EngineOutputListeners. Using the run method the engine runs the EntityUpdaters' update function once per timestep, which is passed as an argument. EntityUpdaters could be any implementation that changes an entities state, like for example a gravitational force exerted by all other entities in the system. After each EntityUpdater has been run on the list of entities the Engine runs each entities *update* method which updates the Entity's state such as position, velocity and acceleration. If there are any EngineOutputListeners attached they are called after this update method to display the Entities.

Using the interfaces EntityUpdater and EngineOutputListener instead of defined classes we enable a certain modularity which allows us to define different pipelines for the engine

to process. Diagram A provides a rough overview of how the engine works.

Diagram A. Engine workflow

For example different EntityUpdaters were created depending on which gravitation algorithm was to be used, i.e. Barnes-Hut and the "accurate" algorithm, which we will discuss in more details. In addition, a 2D visualizer was created as an EngineOutputListener to display the simulation using LWJGL which we used to debug and display implementation of our algorithms. Furthermore, we created a simple collision detection EntityUpdater which prevents Entities from clipping inside each other. Since the main aspect of our report and project is the force of gravity we will not go into details on the collision algorithm used.

3.2 Entity

Mass bodies are represented by the Entity class. An Entity instance contains many trivial attributes, such as position, velocity, acceleration, radius, mass and forces which can be seen in appendix under **Table 4**.

The **forces** attribute is probably the less trivial one to understand. This array changes, depending on the gravitational force implementation used, into an array of size one or the number of entities present in the system.

In the Barnes-Hut algorithm the single and threaded implementation sets the **forces** array of size 1 and appears as a buffer that holds all the accumulated forces applied by other entities on this entity. The accumulated value is used in every time step to update the entities **acceleration**, **velocity** and lastly **position** (in exactly that order).

A single and multi-threaded “accurate algorithm” implementation will function differently as these implementations follow a bag of task concept. The **forces** list has the size equivalent to the number of entities in the system. Every entity can apply a force on another entity and is considered a task, which has its own index in the **forces** list per entity, or in other words an own position in a force matrix. Thus, there will be no thread-safety issues in multi-threaded environment as threads working with a task, entity, will not write in the same memory space. After a time step the **forces** list is summed up during the update call (see appendix **Table A5**) to find a net force, which is affecting the respective entity.

(See **Accurate force algorithm** and **Barnes-Hut algorithm** for more details regarding this).

3.3 AbstractEntityUpdater

The AbstractEntityUpdater implements the **EntityUpdater** interface and functions as a coordinator for a thread pool and should thus only be extended by a multi-threaded updater. The class will every time an update is called do the following:

```
tasklist = getTasks(allEntities);
for each task in tasklist
        put task in threadpool
```

Tasks pushed into the threadpool will be executed by worker threads.

It should be noted that getTasks is an abstract method and should thus be overridden accordingly. (see Appendix **Table A6**).

3.4 Accurate force algorithm

In general, the affecting force of the entity is in the accurate implementation dependent on all other entities in the system. Thus, in order to calculate the force of an entity **e**, it must be compared to all other in the system. This creates the perfect “for each entity, do calculation force-calc for each entity”.

The accurate algorithm is implemented as an EntityUpdater. Two different versions were created for sequential and multithreaded processing.

3.4.1 Sequential and Multi-threaded implementation

This implementation uses the following algorithm to do force calculations, where a unique int defines a loop of calculations for each entity.

```
start = 0;
// Note start == tasked
for each Entity e1 in system
    accumulateForce = new Vector

    // start - avoids redundant calculations
    for i = start -> entitycount do
        if( e1 == entity[i] )
            skip entity
        else entity[i].setForce(start, gravityforce)
             accumulateForce += gravityforce

    e1.setForce(start, accumulateForce)
    start++;
```

This way, sequential runs will not do redundant force calculations of entities as this algorithms makes sure to skip entities that have already updated themselves and all other entities with respect to itself. Using the bag of bag of task model, the *start* value can be considered as a *taskid* and makes it possible to create tasks that are inserted into a threadpool works with tasks (See **AbstractEntityUpdater**).

3.5 Barnes-Hut algorithm

The Barnes-Hut algorithm is implemented as an EntityUpdater. Two different versions were created for multithreaded and sequential processing. The main crux of the algorithm lies in the DimensionTree implementation which is used to calculate the gravitational force. The EntityUpdater instance creates the DimensionTree, inserts all entities and then runs the *getForce* method for each entity on the DimensionTree sequentially or multithreaded depending on which version is chosen.

For the multithreaded version it divides the list of entities into separate chunks for each of the worker thread to process. Thus each task given to the **AbstractEntityUpdater** is a chunk of entities.

3.5.1 DimensionTree

The DimensionTree is a data structure implemented as general tree structure divided into geometrical sections depending on the given number of dimensions and width. It divides the area into equal binary partitions for each dimension. This results in each node having a total of 2^n sub-nodes, where n is the number of dimensions.

To use this data structure for calculating gravity we add two attributes to each node. A vector describing center of mass and a double representing the total mass. Nodes are divided into two different types; inner and external. An inner node is a leaf and always contains only one Entity object. An external node contains works as a branch, always linking to one or more single nodes. Adding to the tree places the Entity in an inner node, going all the way down and splitting existing inner nodes into external ones if necessary.

To obtain the gravitational force the tree is traversed using a depth-first search (DFS), adding the force calculated for each inner node within the permitted granularity (width / distance) or the approximated force calculated using the center of mass and mass variables instead.

4. Method

Using the described implementation we created different engines described in the table below. We ran the engines for 500 time steps in different runs for 120, 180 or 240 randomly generated entities. The processor chip used was an Intel Core i7 - 2630 QM which has 4 cores and hyperthreading, resulting in 8 that threads that can be run simultaneously. It should also be noted that the tests we run in a virtual machine with operating system Debian 3.2.35-2 x86_64 GNU/Linux and java version 1.7.0.7.

Name	**Description**
GravitationalForce Sequential	Runs the accurate algorithm to simulate gravitational force on a single thread.
GravitationalForce (n)	Runs the accurate algorithm to simulate gravitational force on n threads. We tested this EntityUpdater using 2,4 and 8 threads.
Barnes-Hut Sequential	Runs the Barnes-Hut algorithm to simulate gravitational force on a single thread. We used a granularity value of 0.5.
Barnes-Hut (n)	Runs the Barnes-Hut algorithm to simulate gravitational force on n thread. We used a granularity value of 0.5. We tested this EntityUpdater using 2,4 and 8 threads.

Worth mentioning is that our measurement of time will be in terms of timesteps. One timestep is in fact when one update actually occurs. This simplifies the calculations made when updating an entities acceleration, velocity and position, which is dependent on the resulting force on the entity, as time will simply be interpreted as 1.

Example:

In a one dimensional space we would thus interpret the following using only a resultant force in one timestep snapshot:

$a = F/m$ // a =acceleration, f = force and m = mass

$v = u+a*t = u+a$ // v = final velocity, u = initial velocity and t = 1 timestep

$p_{new} = p_{old} + v$ // p = position

The above is of course applicable in “n” dimensional space, where force, acceleration and velocity are vectors and the position is a position vector.

5. Results

The raw result data is available in the appendix. The following graphs summarize the Results:

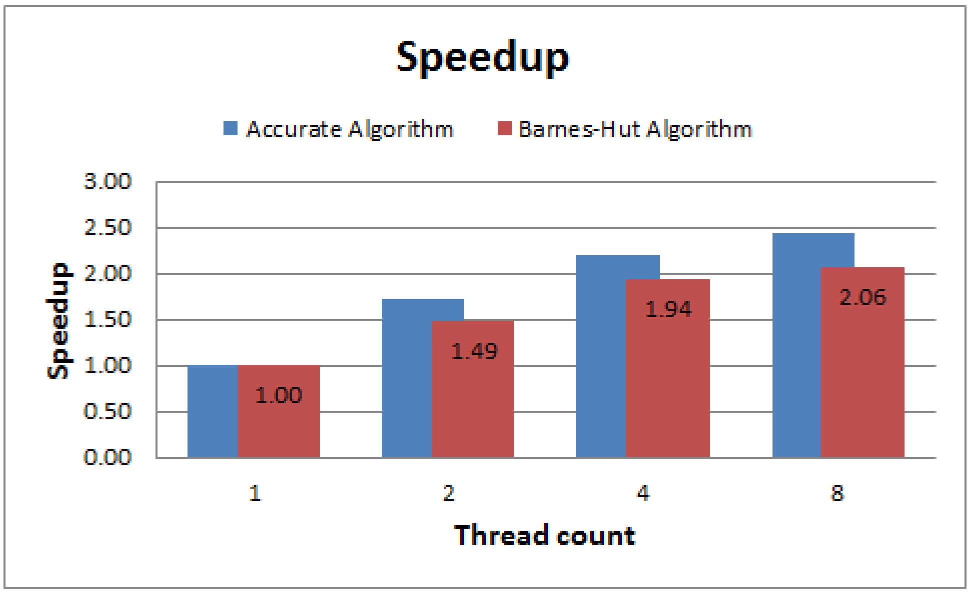

Figure 1: Speedup graph

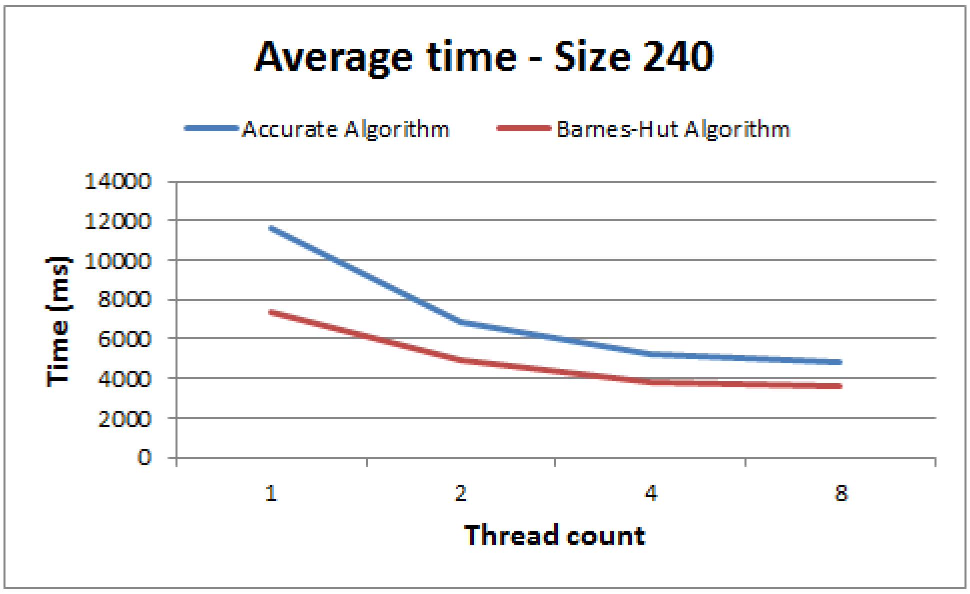

Figure 2: Average time graph

6. Discussion

The results seem to illustrate that Barnes-Hut implementation on every account seems to be the faster algorithm as can be seen in Figure 2: Average time graph. This is probably due to the fact that the Barnes-Hut algorithm does a lot of estimations (the center of mass for an entity group), especially with its relatively high granularity value and thus has a lot less calculations to do compared to the accurate algorithm. It becomes clear why the difference in time complexities exist.

Interestingly though, is that the accurate algorithm has a greater speedup, see Figure 1: Speedup graph, and if we were to extrapolate from the data it could end up being faster than the Barnes-Hut implementation if we were to use more threads (see the lines converging in the Average Time graph). We believe that the speedup is greater on the accurate algorithm because it has less critical sections than our Barnes-Hut implementation. Also, the Barnes-Hut implementation could probably be optimized further for more parallelization. The Barnes-Hut implementation builds a tree data structure at the start which it uses to calculate the force. We only parallelize the lookups in the tree, the actual building of the tree is still sequential. Using a concurrent tree build we could probably gain more performance (higher speedup) on multiple threads.

7. Conclusion

For our results the Barnes-Hut algorithm using multithreading ran the quickest on our machine. From this, we can conclude that at least a speedup general is possible for the gravitational n-body problem when running this on multiprocessors. There might, though, be cases where our implementations of the accurate algorithm will utilize multiple cores better than our implementation of Barnes-Hut, when the amount of simultaneously runnable threads exceed 8, as mentioned in our discussion.

References

[1] “The Barnes-Hut algorithm” http://arborjs.org/docs/barnes-hut

Appendix

Table A1 - Engine

Methods(input)	**Return value**
Engine(List<Entity>, List<EntityUpdater>)	-
addOutputListener(EngineOutputListener)	void
run(int timesteps)	*void*

Table A2 - EntityUpdater (interface)

Methods(input)	**Return value**
update(List<Entity>)	*void*

Table A3 - EngineOutputListener

Methods(input)	**Return value**
handleOutput(List<Entity>)	*void*

Table A4 - Entity Attributes

Attributes	**Type**
Unique id	*int*
Position	*Vector*
Velocity	*Vector*
Acceleration	*Vector*
Radius	*double*
Mass	*double*
forces	*Vector[]*

Table A5 - Entity Methods

Methods(input)	**Return value**
update()	*void*

Table A6 - AbstractEntityUpdater Methods

Methods(input)	**Return value**
update(List<Entity>)	*void*
abstract getTasks(List<Entity>)	List<Runnable>

Raw Result Data

```
GravitationalForce Sequential 120:
4466 ms
4003 ms
3891 ms
4035 ms
4082 ms
4281 ms
4053 ms
4223 ms
4009 ms
4200 ms
4270 ms
4063 ms
3994 ms
3982 ms
3927 ms
GravitationalForce Sequential 180:
9257 ms
9290 ms
9235 ms
9267 ms
9547 ms
9382 ms
9156 ms
81193 ms
9218 ms
9196 ms
9493 ms
9290 ms
9397 ms
9320 ms
9130 ms
GravitationalForce Sequential 240:
16377 ms
16290 ms
16684 ms
16403 ms
16492 ms
16516 ms
17032 ms
```

```
16569 ms
16486 ms
16539 ms
16722 ms
16417 ms
16601 ms
16650 ms
17096 ms
GravitationalForce (2) 120:
2778 ms
2708 ms
2674 ms
2744 ms
2748 ms
2709 ms
2655 ms
2660 ms
2643 ms
2673 ms
2685 ms
2652 ms
2673 ms
2823 ms
2899 ms
GravitationalForce (2) 180:
6368 ms
6621 ms
6147 ms
6113 ms
6170 ms
6162 ms
6321 ms
6422 ms
6491 ms
6324 ms
6389 ms
6561 ms
6181 ms
6302 ms
6258 ms
GravitationalForce (2) 240:
11367 ms
11512 ms
11530 ms
11743 ms
11618 ms
11583 ms
11330 ms
11246 ms
11334 ms
11460 ms
11406 ms
11209 ms
11226 ms
11308 ms
11139 ms
GravitationalForce (4) 120:
2052 ms
2168 ms
2330 ms
2147 ms
2180 ms
2172 ms
2187 ms
2173 ms
2286 ms
```

```
2160 ms
2106 ms
2168 ms
2216 ms
2089 ms
2190 ms
GravitationalForce (4) 180:
5107 ms
4894 ms
4956 ms
4967 ms
4916 ms
4967 ms
4970 ms
5282 ms
4951 ms
4929 ms
4889 ms
4850 ms
4850 ms
4872 ms
4900 ms
GravitationalForce (4) 240:
8687 ms
8649 ms
8710 ms
9037 ms
8671 ms
8595 ms
8637 ms
8814 ms
8703 ms
8690 ms
8954 ms
8697 ms
8659 ms
8644 ms
8623 ms
GravitationalForce (8) 120:
1970 ms
1901 ms
1922 ms
1931 ms
1949 ms
1949 ms
1971 ms
1911 ms
1937 ms
1929 ms
1971 ms
2001 ms
1982 ms
2079 ms
2011 ms
GravitationalForce (8) 180:
4650 ms
4479 ms
4465 ms
4523 ms
4394 ms
4439 ms
4448 ms
4364 ms
4534 ms
4620 ms
4545 ms
```

```
4553 ms
4471 ms
4440 ms
4727 ms
GravitationalForce (8) 240:
7942 ms
7979 ms
8026 ms
8016 ms
8153 ms
7973 ms
7925 ms
8212 ms
8098 ms
7930 ms
8081 ms
7983 ms
7959 ms
7965 ms
7995 ms
Barnes-Hut Sequential 120:
5683 ms
5734 ms
5271 ms
5256 ms
5123 ms
5018 ms
4972 ms
4910 ms
4853 ms
4759 ms
4751 ms
4726 ms
4722 ms
4647 ms
4833 ms
Barnes-Hut Sequential 180:
8144 ms
7992 ms
7807 ms
7629 ms
7558 ms
7633 ms
7476 ms
7361 ms
7612 ms
7240 ms
7385 ms
7110 ms
6998 ms
6974 ms
6780 ms
Barnes-Hut Sequential 240:
10506 ms
10316 ms
9749 ms
9907 ms
9754 ms
9610 ms
9531 ms
9846 ms
9705 ms
9344 ms
9435 ms
9380 ms
9320 ms
```

```
9188 ms
9068 ms
Barnes-Hut (2) 120:
3759 ms
3662 ms
3397 ms
3503 ms
3466 ms
3511 ms
3452 ms
3531 ms
3562 ms
3698 ms
3627 ms
3457 ms
3484 ms
3389 ms
3381 ms
Barnes-Hut (2) 180:
5255 ms
5153 ms
5187 ms
5235 ms
4960 ms
4926 ms
4885 ms
4837 ms
4755 ms
4724 ms
4642 ms
4594 ms
4610 ms
4565 ms
4578 ms
Barnes-Hut (2) 240:
6809 ms
7009 ms
6688 ms
6565 ms
6351 ms
6268 ms
6302 ms
6365 ms
6230 ms
6146 ms
6135 ms
6348 ms
6133 ms
5971 ms
5942 ms
Barnes-Hut (4) 120:
2766 ms
2749 ms
2692 ms
2669 ms
2605 ms
2604 ms
2611 ms
2575 ms
2547 ms
2616 ms
2548 ms
2531 ms
2609 ms
2552 ms
2506 ms
```

```
Barnes-Hut (4) 180:
4087 ms
4168 ms
4180 ms
3871 ms
3784 ms
3763 ms
3755 ms
3746 ms
3736 ms
3636 ms
3626 ms
3647 ms
3614 ms
3637 ms
3604 ms
Barnes-Hut (4) 240:
5218 ms
5246 ms
5408 ms
5323 ms
4997 ms
5006 ms
4943 ms
4974 ms
4907 ms
4712 ms
4748 ms
4793 ms
4760 ms
4812 ms
4674 ms
Barnes-Hut (8) 120:
2629 ms
2679 ms
2785 ms
2539 ms
2521 ms
2463 ms
2441 ms
2462 ms
2433 ms
2348 ms
2338 ms
2391 ms
2652 ms
2421 ms
2395 ms
Barnes-Hut (8) 180:
4093 ms
4076 ms
3892 ms
3622 ms
3545 ms
3545 ms
3489 ms
3418 ms
3490 ms
3691 ms
3474 ms
3374 ms
3331 ms
3325 ms
3249 ms
Barnes-Hut (8) 240:
4972 ms
```

```
4875 ms
4840 ms
4816 ms
4650 ms
4694 ms
4560 ms
4507 ms
4350 ms
4978 ms
4598 ms
4421 ms
4686 ms
4308 ms
4186 ms
```