# Parallel N-Body simulation with Barnes-Hut algorithm

Manos Chatzakis
csd4238@csd.uoc.gr

June 2021

## 1. Introduction

This report is a part of 4th assignment of course HY342 - Parallel Programming of Computer Science Department, University of Crete. It presents an implementation of Barnes-Hut algorithm for N-Body simulation and proposes methods to parallel this algorithm using C++ and Java. The following report is organized as follows: Section 2 Describes the input, Section 3 is about the implementation and Section 4 presents measurements and comparisons among the different programming languages and configurations.

## 2. Input

By default, input is represented as a physical system in an external file containing description about the entities. This description informs us about the name of each entity, the starting coordinates and initial speed on regarding the 2D environment and their corresponding mass. It also configures the size of the universe. These bodies are saved into a dynamic list structure upon the start of the simulation. After the simulation is ended, their final coordinates and speed are printed in an output dataset file following the same format as the input.

## 3. Implementation

In this section the implementation of the algorithm and the parallelism method are described. The Barnes-Hut approximation algorithm runs for a fixed number of iterations, with each iteration representing the time difference of 1 time unit: dt = 1. For every iteration, the algorithm has three phases: Phase 1 is about the BHTree construction, phase 2 calculates the Sum of all powers acting on every entity using the Newton's Gravitational Law, while phase 3 calculates the new positions of the bodies for the current iteration. Thus we proceed by describing each one of these phases:

### 3.1 BHTrees

BHTrees are the core of the efficiency of Barnes-Hut, as they reduce the calculations needed to be done by splitting the universe recursively in quads and subquads, till every leaf has at most one entity. Thus, inner nodes represent the mass center of all bodies below them, allowing us to approximate the total force acting on a specific entity by considering that entities belonging to another quad of the universe to be a general body generated by the mass center of all the bodies above it.

#### 3.1.1 BHTree Structure

The structure of BHTree contains some other supporting structures:

```
class Point {
    double x;
    double y;
}

class Region {
    Point center;
    double dimension;
}

class Entity {
```

```
    double Vy;
    double Vx;
    double mass;
    double SFx = 0;
    double SFy = 0;
    String name;
    Point point;
    Region currentRegion;
}

class BHTree {
    Region region;

    //When the current node is not leaf represents the mass center of the region
    Entity *entity = NULL;

    BHTree *quad1 = NULL;
    BHTree *quad2 = NULL;
    BHTree *quad3 = NULL;
    BHTree *quad4 = NULL;
}
```

### 3.1.2 BHTree Insert

The most powerful yet confusing BHTree functionality is the insert:

```
procedure insertEntity(Entity *_entity){
    if (!region.containsPoint(_entity->getPoint()))
        return false;

    if (isLeaf()){
        //empty leaf case, inserting safely!
        if (entity == NULL) {
            _entity->setCurrentRegion(region);
            entity = _entity;
            return true;
        }

        patchIfCollision();

        //else divide the tree
        createSubQuads();

        if (quad1->insertEntity(entity) ||
                quad2->insertEntity(entity) ||
                quad3->insertEntity(entity) ||
                quad4->insertEntity(entity)) {
            //success
        }

    }

    if (quad1->insertEntity(_entity) ||
        quad2->insertEntity(_entity) ||
        quad3->insertEntity(_entity) ||
        quad4->insertEntity(_entity)) {
```

```
        entity = massCenter(entity, _entity);
        return true;
    }
}

procedure createSubQuads() {
    quad1 = new BHTree(Region(Point(xCenter + dim / 2, yCenter + dim / 2), dim / 2));
    quad2 = new BHTree(Region(Point(xCenter - dim / 2, yCenter + dim / 2), dim / 2));
    quad3 = new BHTree(Region(Point(xCenter - dim / 2, yCenter - dim / 2), dim / 2));
    quad4 = new BHTree(Region(Point(xCenter + dim / 2, yCenter - dim / 2), dim / 2));
}
```

### 3.2. Calculating Netforce

The netforce on each body is calculated for every single entity by traversing the BHTree, as following:

```
procedure netForce(Entity *e, BHTree *bh) {
    if (bh == NULL)
        return;

    if (bh->isLeaf()) {
        Entity *body = bh->getEntity();
        if (body != e && body != NULL) {
            double r = distance(*e, *body);
            double f = F(*e, *body, r);

            double fx = Fx(*e, *body, f, r);
            double fy = Fy(*e, *body, f, r);

            e->addToSFx(fx);
            e->addToSFy(fy);
        }

        return;
    }

    BHTree *quads[4] = {bh->getQuad1(), bh->getQuad2(), bh->getQuad3(), bh->getQuad4()};
    BHTree *quadToGo = NULL;

    for (int i = 0; i < 4; i++) {
        if (quads[i] != NULL) {
            Entity *cent = quads[i]->getEntity();
            Region reg = quads[i]->getRegion();

            if (cent == NULL)
                continue;

            if (reg.containsPoint(e->getPoint(), i + 1))
                quadToGo = quads[i]; //saving the destination quad
            else {
                /*In this case, entity cent represents the mass center of the current adjacen
                double s = 2 * e->getCurrentRegion().getDimension();
                double r = distance(*e, *cent);

                if (r > s || quads[i]->isLeaf()) {
                    double f = F(*e, *cent, r);
                    double fx = Fx(*e, *cent, f, r);
```

```
                    double fy = Fy(*e, *cent, f, r);

                    e->addToSFx(fx);
                    e->addToSFy(fy);
                } else {
                    BHTree *children[4] = {quads[i]->getQuad1(), quads[i]->getQuad2(), quads[
                    for (int k = 0; k < 4; k++)
                        if (children[k] != NULL)
                            netForce(e, children[k]);
                }
            }
        }
    }

    if (quadToGo != NULL)
      netForce(e, quadToGo);
}
```

As stated above, to calculate the netforce on a body, the Newton's Gravitational Law is Used:

$$F = G\frac{m_1 m_2}{r^2}$$

where

$$G = \frac{6.67}{10^{11}}\frac{Nm^2}{kg^2}$$

and from trigonometry and Pythagorean theorem we have

$$F_x = F\frac{x_2 - x_1}{r}$$

and

$$F_y = F\frac{y_2 - y_1}{r}$$

### 3.3. Calculating the New Positions

The most powerful yet confusing BHTree functionality is the insert:

```
procedure calcNewPos(Entity *e, double dt, double dims) {
    double SFy = e->getSFy();
    double SFx = e->getSFx();
    double m = e->getMass();

    double oldVx = e->getVx();
    double oldVy = e->getVy();
    double oldX = e->getPoint().getX();
    double oldY = e->getPoint().getY();

    double Ax = SFx / m;
    double Ay = SFy / m;

    double newVx = oldVx + Ax * dt;
```

```
    double newVy = oldVy + Ay * dt;
    double newX = oldX + newVx * dt;
    double newY = oldY + newVy * dt;

    patchIfOutOfUniverse();

    e->setPoint(Point(newX, newY));

    e->setSFx(0);
    e->setSFy(0);
    e->setVx(newVx);
    e->setVy(newVy);
}
```

It's clear that we are able to calculate the new positions we need to use:

$$a_x = \frac{F_x}{m}, a_y = \frac{F_y}{m}$$

thus

$$V_x = V_x + \Delta t \times a_x, V_y = V_y + \Delta t \times a_y$$

meaning that the new positions are:

$$x = x + \Delta t \times V_x, y = y + \Delta t \times V_y$$

## 3.4. Parallelization

### 3.4.1 Parallelization Method

The way we described the implementation leads us to think that phase two and three can be paralleled. The most simple and efficient way to do this, it to parallel the loop of phase 2, without the deal of data race as every single thread will modify the total force only for the chunk of bodies that it edits, and wont modify any coordinates or speed. Then, using a barrier between phase two and three, we can also parallel the new position calculation loop for every body, without the fear for data race, for the same reason as above.
Thus, for C++ using TBB we could implement the following:

```
for (int i = 0; i < iterations; i++) {
    bh = createBHTree(entities, dims);

    parallel_for(blocked_range<size_t>(0, size),
        [&](const blocked_range<size_t> &r) -> void {
            for (size_t i = r.begin(); i != r.end(); i++) {
                netForce(entities[i], bh);
            }
    }); //barrier

    parallel_for(blocked_range<size_t>(0, size),
         [&](const blocked_range<size_t> &r) -> void {
            for (size_t i = r.begin(); i != r.end(); i++) {
                newPosition(entities[i], dt, dims);
            }
    }); //barrier
```

```
    freeBHTree(bh);
}
```

and for Java Threads we have:

```
for (int k = 0; k < iters; k++) {
try {
    BHTree bh = BHUtils.createBHTree(entities, dims);
    barrier.await();

    for (int i = from; i < to; i++) {
        BHUtils.netForce(entities.get(i), bh);
    }

    barrier.await();

    for (int i = from; i < to; i++) {
      BHUtils.newPosition(entities.get(i), dt, dims);
    }
    barrier.await();
} catch (Exception e)
    e.printStackTrace();
}
```

for each thread.

### 3.4.1 Load Balancing

Considering C++: As this algorithm is implemented with the help of TBB, we could just use two parallel for generics to create the above idea. Also, doing measurements it's obvious that automated grain size of TBB could provide better results by manually setting the chunk size or grain size for each thread.
Considering Java: Using Java threads, we could create the chunks manually and synchronize the threads with a custom barrier. We could also use parallel streams of Java 8. Considering Java threads, a chunk size can be considered as the fraction of number of total bodies and the threads number.

### 3.4.2 BHTree as a parallel container

As insertion on BHTree probably needs to extensively divide the universe more and more, paralleling this procedure would need a great amount of synchronization among the different thread, resulting in a high overhead. For this reason this structure is implement sequentially.

## 3.5 Collision and Error Handling

Its possible for collisions to occur, and there are also cases that a body can leave the region of the universe, mainly when the dimensions are small. Such cases are patched differently: When 2 bodies collide (aka they have the same coordinates) the coordinates of one body are patched by a small dx,dy, so that the two bodies will never be considered to have the same coordinates. A possible variation of the algorithm could consider calculating the collision outcome, other than patching the coordinates. On the other hand, when a body gets out of bounds, it's kept at the border so as to show that it got out of bounds during the simulation. Note that such cases are extremely rare for actual big datasets (e.g. for datasets simulation solar systems), but they are common for small testing datasets.

# 4. Measurements

In this sections, the execution time results are presented for all configurations. These execution times where gathered and calculates using an automated Python script.

## 4.1 Speedup and Standard Deviation

For speedup, two versions of the calculations are used, SpeedUp(C) and SpeedUp(N). SpeedUp(C) is the speedup in comparison with the sequential code (compiled with 0 threads arg), while SpeedUp(N) is the speedup between thread N and N-1 (etc. the speedup between thread 2 and thread 4):

$$SpeedUp(C) = \frac{t_{seq}}{t_{par}} = \frac{Sequential_{average}}{threadN_{average}} = C$$

$$SpeedUp(N) = \frac{t_{seq}}{t_{par}} = \frac{threadN-1_{average}}{threadN_{average}} = N$$

Standard Deviation is calculated for each configuration and input size using the equation:

$$Stdev = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2} = Vseconds$$

## 4.2 C++ Implementation using TBB

### 4.2.1 Input1

- 10000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.138 | 0.141 | 0.14 | 0.139 | 0.14 | 0.1396 | 0 | 0 | 0.00114 |
| 1 | 0.174 | 0.169 | 0.168 | 0.166 | 0.173 | 0.17 | 0.82118 | 0.82118 | 0.00339 |
| 2 | 0.148 | 0.146 | 0.146 | 0.146 | 0.217 | 0.1606 | 0.86924 | 1.05853 | 0.03154 |
| 4 | 0.205 | 0.188 | 0.135 | 0.138 | 0.137 | 0.1606 | 0.86924 | 1.00000 | 0.03334 |

- 100000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1.351 | 1.265 | 1.215 | 1.25 | 1.241 | 1.2644 | 0 | 0 | 0.05171 |
| 1 | 1.583 | 1.514 | 1.482 | 1.481 | 1.534 | 1.5188 | 0.83250 | 0.83250 | 0.04230 |
| 2 | 1.407 | 2.05 | 1.325 | 1.405 | 1.32 | 1.5014 | 0.84215 | 1.01159 | 0.30951 |
| 4 | 1.711 | 1.537 | 1.358 | 2.02 | 1.379 | 1.601 | 0.78976 | 0.93779 | 0.27390 |

### 4.2.2 Input2

- 10000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 6.482 | 6.265 | 6.402 | 6.319 | 6.296 | 6.3528 | 0 | 0 | 0.08830 |
| 1 | 6.899 | 6.682 | 6.757 | 6.845 | 6.687 | 6.774 | 0.93782 | 0.93782 | 0.09616 |
| 2 | 5.471 | 4.803 | 4.679 | 4.687 | 4.698 | 4.8676 | 1.30512 | 1.39165 | 0.34103 |
| 4 | 3.456 | 3.395 | 3.39 | 3.91 | 3.416 | 3.5134 | 1.80816 | 1.38544 | 0.22323 |

- 100000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 64.985 | 62.992 | 63.186 | 62.641 | 61.268 | 63.0144 | 0 | 0 | 1.33253 |
| 1 | 67.879 | 67.588 | 67.37 | 67.353 | 68.062 | 67.6504 | 0.93147 | 0.93147 | 0.31330 |
| 2 | 47.153 | 46.422 | 46.041 | 48.97 | 52.443 | 48.2058 | 1.30720 | 1.40337 | 2.62272 |
| 4 | 40.794 | 38.583 | 39.613 | 37.83 | 35.834 | 38.5308 | 1.63543 | 1.25110 | 1.87482 |

#### 4.2.3 Input3

- 10000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 5.539 | 5.391 | 5.501 | 5.396 | 5.391 | 5.4436 | 0 | 0 | 0.07105 |
| 1 | 5.993 | 5.872 | 5.924 | 5.789 | 5.771 | 5.8698 | 0.92739 | 0.92739 | 0.09275 |
| 2 | 3.725 | 3.947 | 4.001 | 4.172 | 3.963 | 3.9616 | 1.37409 | 1.48167 | 0.15970 |
| 4 | 2.906 | 2.856 | 2.878 | 2.856 | 3.254 | 2.95 | 1.84529 | 1.34292 | 0.17118 |

- 100000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 53.686 | 52.782 | 52.928 | 53.62 | 53.118 | 53.2268 | 0 | 0 | 0.40756 |
| 1 | 57.698 | 57.543 | 57.428 | 59.305 | 57.097 | 57.8142 | 0.92065 | 0.92065 | 0.86213 |
| 2 | 41.105 | 40.006 | 40.167 | 39.822 | 39.82 | 40.184 | 1.32458 | 1.43874 | 0.53473 |
| 4 | 33.895 | 29.655 | 34.099 | 31.822 | 33.173 | 32.5288 | 1.63630 | 1.23534 | 1.83748 |

#### 4.2.4 Input4

- 10000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 85.419 | 82.537 | 82.31 | 82.595 | 82.941 | 83.1604 | 0 | 0 | 1.28266 |
| 1 | 83.144 | 82.349 | 82.834 | 82.045 | 84.205 | 82.9154 | 1.00295 | 1.00295 | 0.83671 |
| 2 | 56.978 | 56.447 | 56.295 | 55.894 | 57.764 | 56.6756 | 1.46731 | 1.46298 | 0.72182 |
| 4 | 42.007 | 41.399 | 42.194 | 42.825 | 42.002 | 42.0854 | 1.97599 | 1.34668 | 0.51065 |

- 100000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 850.73 | 913.379 | 886.605 | 868.92 | 848.703 | 873.6674 | 0 | 0 | 27.00128 |
| 1 | 848.109 | 850.635 | 922.492 | 913.214 | 920.318 | 890.9536 | 0.98060 | 0.98060 | 38.12386 |
| 2 | 600.91 | 607.797 | 593.318 | 598.404 | 602.097 | 600.5052 | 1.45489 | 1.48367 | 5.29006 |
| 4 | 451.665 | 454.782 | 449.434 | 456.092 | 448.046 | 452.0038 | 1.93288 | 1.32854 | 3.42106 |

#### 4.2.5 Input5

- 10000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 111.618 | 112.925 | 115.301 | 112.727 | 111.119 | 112.738 | 0 | 0 | 1.61861 |
| 1 | 110.397 | 111.592 | 111.304 | 110.809 | 111.155 | 111.0514 | 1.01519 | 1.01519 | 0.46200 |
| 2 | 68.641 | 67.859 | 69.499 | 71.967 | 68.373 | 69.2678 | 1.62757 | 1.60322 | 1.62153 |
| 4 | 48.205 | 47.683 | 48.299 | 47.925 | 48.419 | 48.1062 | 2.34352 | 1.43989 | 0.29863 |

- 100000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1111.05 | 1113.23 | 1110.34 | 1079.19 | 1084.44 | 1099.65 | 0 | 0 | 16.42109 |
| 1 | 1121.85 | 1094.34 | 1096.85 | 1111.38 | 1095.19 | 1103.922 | 0.99613 | 0.99613 | 12.19745 |
| 2 | 668.853 | 686.979 | 683.197 | 685.753 | 654.066 | 675.7696 | 1.62726 | 1.63358 | 14.13657 |
| 4 | 453.219 | 468.754 | 464.745 | 465.6 | 466.599 | 463.7834 | 2.37104 | 1.45708 | 6.09233 |

## 4.3 Java Implementation using Java Threads

### 4.3.1 Input1

- 10000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.098 | 0.089 | 0.113 | 0.108 | 0.088 | 0.0992 | 0 | 0 | 0.01117 |
| 1 | 0.119 | 0.11 | 0.117 | 0.117 | 0.117 | 0.116 | 0.85517 | 0.85517 | 0.00346 |
| 2 | 0.335 | 0.327 | 0.373 | 0.35 | 0.355 | 0.348 | 0.28506 | 0.33333 | 0.01794 |
| 4 | 0.691 | 0.729 | 0.685 | 0.636 | 0.67 | 0.6822 | 0.14541 | 0.51011 | 0.03376 |

- 100000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.484 | 0.482 | 0.478 | 0.469 | 0.478 | 0.4782 | 0 | 0 | 0.00576 |
| 1 | 0.516 | 0.492 | 0.494 | 0.479 | 0.504 | 0.497 | 0.96217 | 0.96217 | 0.01386 |
| 2 | 1.895 | 1.862 | 1.893 | 1.747 | 1.796 | 1.8386 | 0.26009 | 0.27031 | 0.06499 |
| 4 | 4.713 | 4.663 | 4.809 | 4.681 | 4.702 | 4.7136 | 0.10145 | 0.39006 | 0.05670 |

### 4.3.2 Input2

- 10000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1.121 | 1.122 | 1.065 | 1.114 | 1.089 | 1.1022 | 0 | 0 | 0.02471 |
| 1 | 1.093 | 1.088 | 1.096 | 1.107 | 1.114 | 1.0996 | 1.00236 | 1.00236 | 0.01064 |
| 2 | 1.195 | 1.128 | 1.132 | 1.106 | 1.144 | 1.141 | 0.96599 | 0.96372 | 0.03317 |
| 4 | 1.962 | 1.832 | 1.732 | 1.748 | 1.795 | 1.8138 | 0.60767 | 0.62907 | 0.09175 |

- 100000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 8.075 | 7.934 | 7.863 | 7.851 | 7.857 | 7.916 | 0 | 0 | 0.09503 |
| 1 | 7.996 | 7.958 | 8.089 | 8.025 | 8.11 | 8.0356 | 0.98512 | 0.98512 | 0.06342 |
| 2 | 7.606 | 7.613 | 7.621 | 7.557 | 7.447 | 7.5688 | 1.04587 | 1.06167 | 0.07252 |
| 4 | 9.584 | 9.619 | 9.443 | 9.648 | 9.619 | 9.5826 | 0.82608 | 0.78985 | 0.08127 |

### 4.3.3 Input3

- 10000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.921 | 0.894 | 0.889 | 0.894 | 0.894 | 0.8984 | 0 | 0 | 0.01282 |
| 1 | 0.928 | 0.909 | 0.913 | 0.908 | 0.883 | 0.9082 | 0.98921 | 0.98921 | 0.01621 |
| 2 | 0.946 | 0.978 | 0.936 | 0.975 | 0.956 | 0.9582 | 0.93759 | 0.94782 | 0.01817 |
| 4 | 1.581 | 1.359 | 1.389 | 1.419 | 1.319 | 1.4134 | 0.63563 | 0.67794 | 0.10073 |

- 100000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 6.277 | 6.206 | 6.21 | 6.259 | 6.214 | 6.2332 | 0 | 0 | 0.03252 |
| 1 | 6.498 | 6.4 | 6.338 | 6.351 | 6.392 | 6.3958 | 0.97458 | 0.97458 | 0.06290 |
| 2 | 6.2 | 6.327 | 6.249 | 6.146 | 6.136 | 6.2116 | 1.00348 | 1.02965 | 0.07885 |
| 4 | 8.856 | 9.32 | 8.702 | 8.83 | 9.174 | 8.9764 | 0.69440 | 0.69199 | 0.25900 |

#### 4.3.4 Input4

- 10000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 11.389 | 10.778 | 11.039 | 10.629 | 10.781 | 10.9232 | 0 | 0 | 0.29926 |
| 1 | 11.041 | 11.064 | 10.741 | 11.081 | 11.175 | 11.0204 | 0.99118 | 0.99118 | 0.16429 |
| 2 | 8.909 | 8.703 | 8.578 | 8.515 | 8.648 | 8.6706 | 1.25980 | 1.27101 | 0.15098 |
| 4 | 8.698 | 9.792 | 9.504 | 9.185 | 9.136 | 9.263 | 1.17923 | 0.93605 | 0.41212 |

- 100000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 104.807 | 103.116 | 103.264 | 103.09 | 103.15 | 103.4854 | 0 | 0 | 0.74178 |
| 1 | 104.381 | 103.112 | 102.364 | 102.655 | 103.293 | 103.161 | 1.00314 | 1.00314 | 0.77451 |
| 2 | 79.973 | 77.141 | 79.678 | 81.017 | 77.268 | 79.0154 | 1.30969 | 1.30558 | 1.72694 |
| 4 | 72.352 | 73.713 | 72.725 | 70.233 | 71.706 | 72.1458 | 1.43439 | 1.09522 | 1.29290 |

#### 4.3.5 Input5

- 10000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 11.986 | 12.163 | 12.148 | 12.035 | 12.115 | 12.0894 | 0 | 0 | 0.07609 |
| 1 | 12.098 | 11.867 | 11.959 | 11.926 | 11.94 | 11.958 | 1.01099 | 1.01099 | 0.08548 |
| 2 | 9.084 | 9.044 | 9.463 | 9.088 | 9.087 | 9.1532 | 1.32078 | 1.30643 | 0.17416 |
| 4 | 8.966 | 9.285 | 8.208 | 8.201 | 8.158 | 8.5636 | 1.41172 | 1.06885 | 0.52554 |

- 100000 iterations

| Threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Speedup(C) | Speedup(N) | Stdev |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 115.758 | 110.113 | 113.002 | 116.448 | 115.902 | 114.2446 | 0 | 0 | 2.66957 |
| 1 | 114.71 | 114.07 | 115.393 | 115.129 | 115.504 | 114.9612 | 0.99377 | 0.99377 | 0.58450 |
| 2 | 85.386 | 85.223 | 85.724 | 85.998 | 85.261 | 85.5184 | 1.33591 | 1.34429 | 0.33292 |
| 4 | 72.98 | 73.558 | 74.892 | 76.848 | 73.396 | 74.3348 | 1.53689 | 1.15045 | 1.57693 |

### 4.4 Comparing the Implementations

From the results above we conclude that JVM Warm-ups and optimizations could result in execution speeds even higher than C++ although the thread speedup is less. In both implementations, we notice that we don't always have speedups. This is normal, as the speedup relies on the number of different bodies a simulation needs to process. When a dataset has many bodies (eg. Dataset input5) we notice that using threads to split the calculations can be really efficient. In any case, someone could conclude that C++ multi-threading using TBB is better than Java multi-threading based on the results presented above. Although TBB is a great tool for concurrent programming, that result is not fully true, as the Java implementation used manual chunk creation and classic Java threads, which means that this implementation is not the best possible.
SOLUTION: Java 8 provides parallel streams, which are really useful and efficient to parallel simple for loops, like this:

```
public static void BarnesHutStream(ArrayList<Entity> entities, int iters, double dims, int th
    ForkJoinPool customThreadPool = new ForkJoinPool(threadsNum);
    for (int i = 0; i < iters; i++) {

    BHTree bh = BHUtils.createBHTree(entities, dims);

    customThreadPool.submit(() -> entities.parallelStream().forEach(
```

```
            e -> { BHUtils.netForce(e, bh); }));

        customThreadPool.submit(() -> entities.parallelStream().forEach(
            e -> { BHUtils.newPosition(e, dt, dims); }));

        }

        customThreadPool.shutdownNow();
}
```

However CSD computers do not support these streams, so the measurements could not be done. Also, there is another problem, which could occur if oneTBB was used in the place of classic TBB: Although these streams offer methods to customize the thread pool, it's the schedulers choice to run on as many threads as the optimal solution could be. That means that measurements to compare difference among thread running times could be impossible.