Parallel computing Report on: N-Body in Intel TBB

Konstantinos Theodorakos

University of Antwerp

Konstantinos.Theodorakos@student.uantwerpen.be

---

1. Assignment

Simulations of dynamical systems of particles are often used in physics to predict behavior of planets, stars, galaxies, gas particles... The interaction between particles is described by physically sound equations and "integrated" in time to predict the outcome of the simulation.

Write a parallel version of the N-body algorithm in C++ using Intel's Threading Building Blocks library for the parallel code. A straightforward approach is to use the "naive" approach described above by using the three nested loops (over j, i and t, from inner to outer).

2. Implementation

2.1. General

Four versions of the source code where created:

- *Serial*: Naive algorithm $O(n^2)$, native *C++ 11*, using STD and STL.
- *Parallel*: Naive pair-wise algorithm $O(n*(n-1))$, *Intel Thread Building Blocks* with concurrent containers.
- *Serial*: *Barnes-Hut* algorithm $O(n \log n)$, native *C++ 11*, using STD and STL.
- *Parallel*: *Barnes-Hut* algorithm $O(n \log n)$, *Intel Thread Building Blocks* with concurrent containers.

The source code for the GCC version is located in the following repository, branch "feature/dev-build-linux":

https://bitbucket.org/universiteitantwerpen/n-body/branch/feature/dev-build-linux.

2.2. Code Structure

The Particle class, stores information of every simulated body. It stores position, velocity, mass and acceleration.

As also read from other sources, it is better to calculate accelerations directly, rather than the force vectors [1]. This is due to the fact that using the acceleration uses calculations that are "less" expensive in execution time terms.

The *add acceleration* and *add acceleration pairwise* are the methods that cumulatively apply forces on the bodies. The pairwise addition provides a small speed up $O(n * (n-1))$ over the "naive" $O(n^2)$. This can be done by just changing the sign on a force calculation. Two interacting particles have the same force applied to each other. So we don't have to recalculate it.

The advance function, applies acceleration and velocity updates on the particle. Then it forwards the new position of the particle by using the time stamp. The times tap acts as an interval.

ParticleHandler is a purely static class. It implements helper methods that allocate random particles in a collection. It converts *std::vector* collections into *cache aligned* concurrent vector or QuadParticleTree nodes. It also assists in the assertion of the results.

Simulation specific settings/options:

- Particle count.
- Time step.
- Total Time steps.
- Gravitational constant.

- Max ranges on Mass, Speed, Acceleration and random numbers in general.
- Uniform or Normal distributions for random number sampling.
- Universe size X and Y. This helps in "containing" the particles in a specific area. In case the particles reach the limits of the area, they can be set to either "bounce" back or teleported onto the other side of the contained area.
- *Theta* parameter for the *Barnes-Hut* trees. It is being used for the comparison against s/d of a particle interacting with the center of mass of a node.
- Min distance, which is used to *soften* the distance square of closely interacting particles.
- Max tree depth. It limits the depth of the *Barnes-Hut* tree.

The first parts of the source code were implemented using *MSBuild* and *Visual Studio Community 2015*. In addition, the *Intel VTune* application (Figure 1) was used to profile some of the hot spots of the application. The Release x64 configuration with /O3 options was used to profile the execution times. Later on, the source code was ported in order to run on a *GCC/G++* compiler. Thus, it was deemed necessary to make the core of the application, as portable as possible. For that reason, as many Thread Building Block constructs were utilized as possible.

Initially, double precision floating point numbers were used. But, later on, all the constructs were converted into *single precision* because they can scale better. In addition, they use less memory, *64 bits* versus *32 bits*.

Rarely a precision of more than 7 decimal points is required. Examples are the VGA industry: NVidia in each recent GPU hardware, provides native support only in single precision floating point numbers. Another example is the GPS location system. Using more than 7 decimal points, we can achieve an precision of +/- 10cm. But is not applicable with today's commercial GPS receivers, because their actual accuracy lies within a margin of +/- 10 meters. In rare occations, it can be +/- 1-5 meters, but only by utilizing expensive professional equipment and base-station infrastructures.

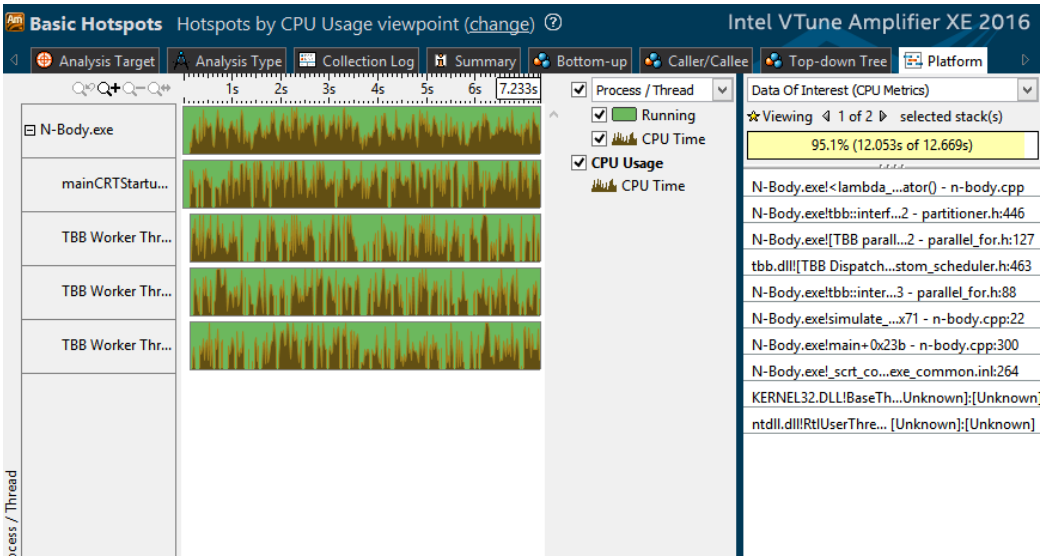

Figure 1: Snapshot of the *Intel VTune Amplifier* plug in that was used along with Visual Studio Community 2015. In this case, the individual thread utilization of a "parallel for" is being shown. The *task_scheduler_init* was called with 4 threads, that spawned directly within the first $100ms$ of the *parallel for* execution (Main thread + 3 "worker" threads).

Achieving as much *thread locality* as possible was the most determining factor in parallelizing the source code. While profiling, it was noticed that a call to get the *size()* of a vector would hinder the performance greatly. Using a *scalar* variable of type size_t inside the parallel body of the execution removed that slowdown. The TBB's *blocked_range* helped in automatically defining the work-size ranges for each worker.

Another feature of the integrated *Task Scheduler* of TBB that was noticeable was Thread Stealing: In the pairwise calculation of accelerations, which normally results in a non-equally distributed work size for each thread.

A benefit of the pairwise calculation is the reduction of branching. Branching can be a major factor in slowing down single and parallel code execution. Especially in a for loop. The processing core "has to guess" beforehand if an operation can be executed directly or there must be a check for the branching. In case it misses, there is a major slowdown applied.

Simplifying calculations and storing variables as *reusable placeholders* is another helping factor. In practice, re-using the value $distance * distance$

was found to be faster than directly calculating the square root. Also using $distance * distance$ rather than "power of 2" is faster in practice, because a simple multiplication "costs" less. As a tribute to the past, the "Fast Inverse Square root" method [2] was compared versus the regular $std :: sqrt$ operation from *C++ 11*. The Fast Inverse Square Root was a technique to accelerate the square root for a 32-bit single precision floating point number by performing a logical right shift and a "magic number" subtraction. However, the latest *C++11 std::sqrt* is faster than this "hack" in practice.

For the debugging, three methods were being used:

1. By static image comparison of the "universes". It was achieved by exporting the body locations into a 2 dimensional PNG image space (Figure 2).
2. Computationally, by comparing the resulting locations of the particles, the particle size of the collection, the overall mass of the bodies (especially useful in the *Barnes-Hut* implementation).
3. With animation in 3D time-space. An *OpenGL* version of the implementation (Figure 3) was used to ensure that the particles "behave" as they should.

It was easy to compare the N-body behaviour by adding one additional dimension. In practice, a bug was found and fixed in the TBB implementation. One specific particle was always "roaming around", without having any cumulative force applied to it. It was due to a mistyping of an index variable, inside the inner for loop that was calling the *add acceleration pairwise* method.

2.3. Barnes-Hut algorithm

The *Barnes-Hut* [3] is an n-body optimized algorithm. In the case of 2 dimensional simulations, it divides the bodies into quad trees and for 3 dimensional in oct trees.

One important factor is the quotient s/d. s is the "width" of the area represented by the current node of the tree. Now d is the distance of the current particle from the center of mass of the node. s/d can be compared against a fixed value, called *theta*. When s/d is zero, the quad tree is fully

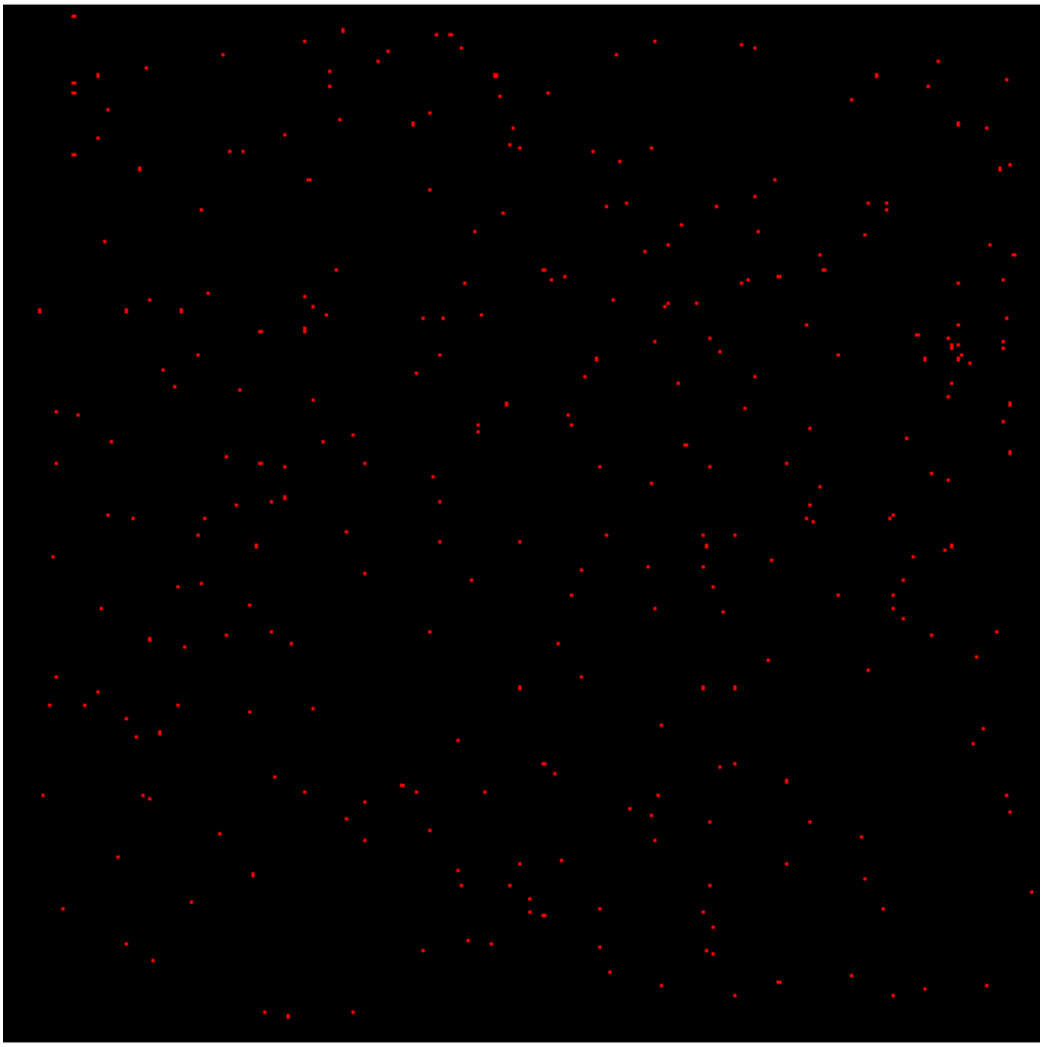

Figure 2: A "universe snapshot" of an N-Body simulation. In this case, a grid 300x300 was used with 300 particles. This is the final result after 100 district time steps using the TBB Parallel N-Body pairwise algorithm with 4 threads.

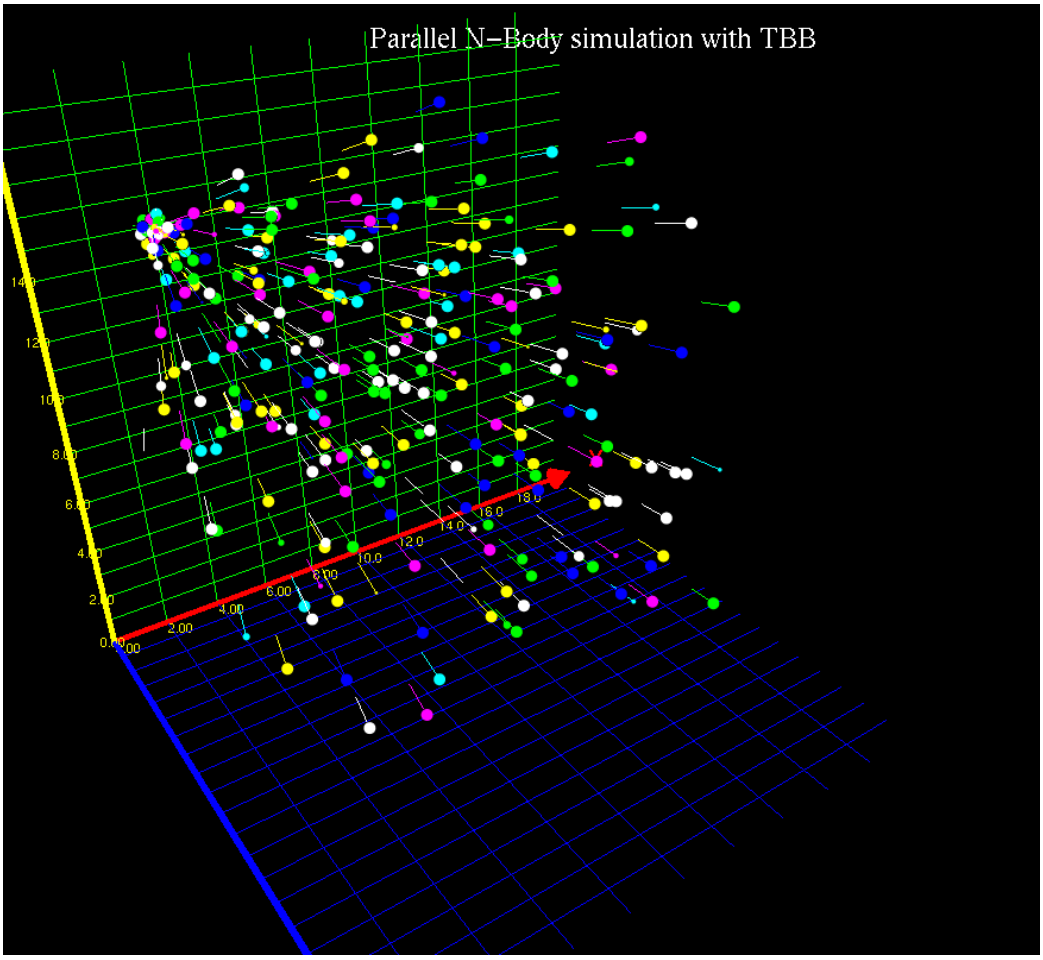

Figure 3: Snapshot of the *OpenGL* N-Body sub-application. A *third dimension* was added in order to make it easier to debug the N-Body algorithms. Each body is scaled by its mass. The line attached to each particle is the spatial representation of the *velocity vector*.

traversed. In the test setup, a value of $theta = 0.5$ was used. QuadParticleTree is the class that implements nodes for the *Barnes-Hut* tree. Every node has 4 children, which are pointers for other QuadParticleTree objects. The Particle objects *origin* and $halfDimension$ are being used to define the side size of the current node, and also in defining the dimensions of any future children.

The *insert* method is used to populate the quad tree. It first checks if the current node is a leaf node. If it is, then the particle that we want to insert, can be safely stored in the corresponding children pointer placeholder. On the other hand, if there are data already stored, new children are being allocated. The insert method is then called again recursively. We have to note that, every time we "pass through" a node, we must update the center of mass of that node (x, y positions and *total mass*).

The *apply method* is used to sum up all the forces applied to a specific particle. It is being called recursively if the s/d distance of the current particle is greater than the *theta* value, in order to go "deeper" into the tree structure.

3. Experiments

3.1. Experimental Setup

The test machine that was used for the experiments, was a x64 desktop machine:

- Processor: 1 CPU 3.0 GHz AMD Phenom II X4 940, 4 cores, L2 Cache: 4 x 512KB, L3 Cache: 6MB
- Memory: 8GB DDR2 800MHz 64bit
- GPU: Nvidia GeForce 6800GS 256MB
- Operating System: Ubuntu Linux 14.04 x64 3.19.0-39-generic

Compilers:

- Gcc version 4.8.4
- Intel TBB Interface version 7

Parameter ranges:

- Timestep: 0.01
- Final time step: 10.0 (so 1000 distinct simulation steps).
- Bodies count: 10, 35, 100, 350, 500, 1000
- Threads: 1, 2, 4
- Repeat rounds per experiment case: 1-10

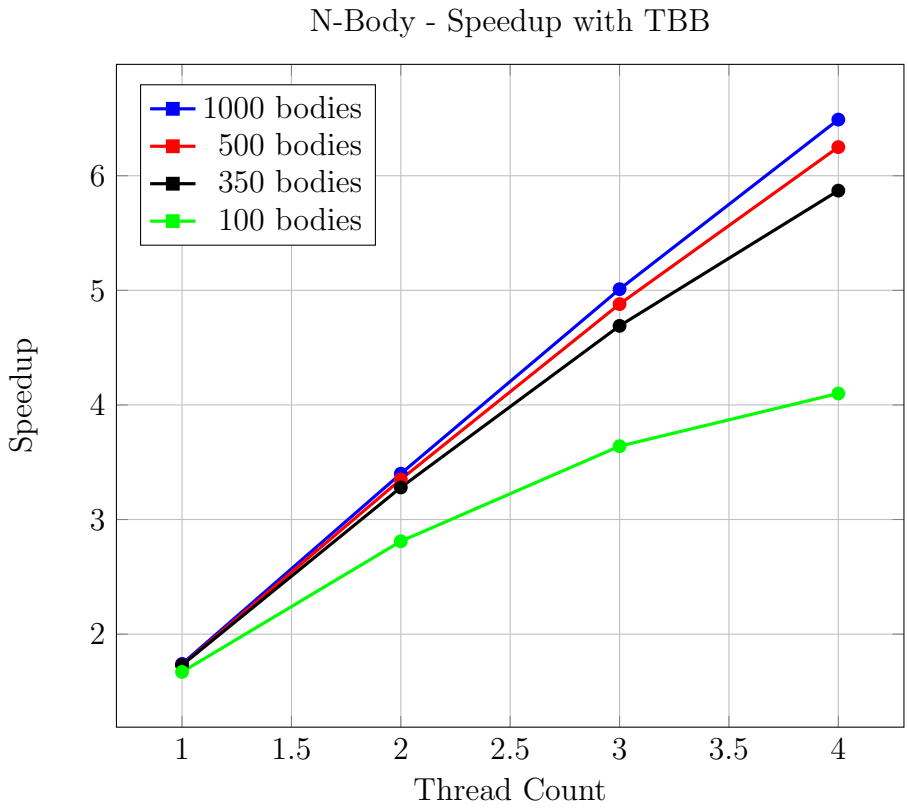

Figure 4: N-Body Simulation. Time step: 0.01, final time step: 10.0. Machine used: X64 Ubuntu Server (Processor: 1 CPU 3.0 GHz AMD Phenom II X4 940, Memory: 8GB DDR2 800MHz 64bit, GPU: Nvidia GeForce 6800GS 256MB)

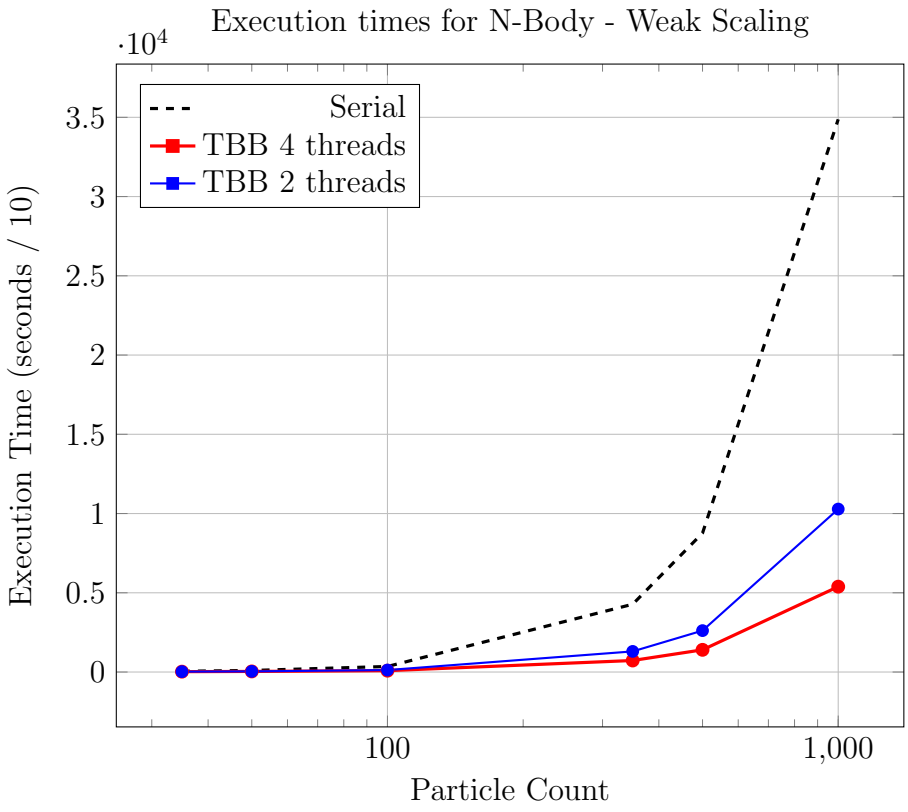

Figure 5: N-Body Simulation. Time step: 0.01, final time step: 10.0. Machine used: X64 Ubuntu Server (Processor: 1 CPU 3.0 GHz AMD Phenom II X4 940, Memory: 8GB DDR2 800MHz 64bit, GPU: Nvidia GeForce 6800GS 256MB)

4. Conclusions

One of the missing features of *Intel Thread Building Blocks* libraries is to directly apply up *SIMD* (Single Instruction Multiple Data) commands, especially on vector operations. However, this technique is being applied by the TBB libraries automatically, especially when applying *thread-local* variables.

I found in practice that it is better to invest on "unrolling" multiple loops, avoiding branching and especially, avoiding the need to use mutexes/critical regions.

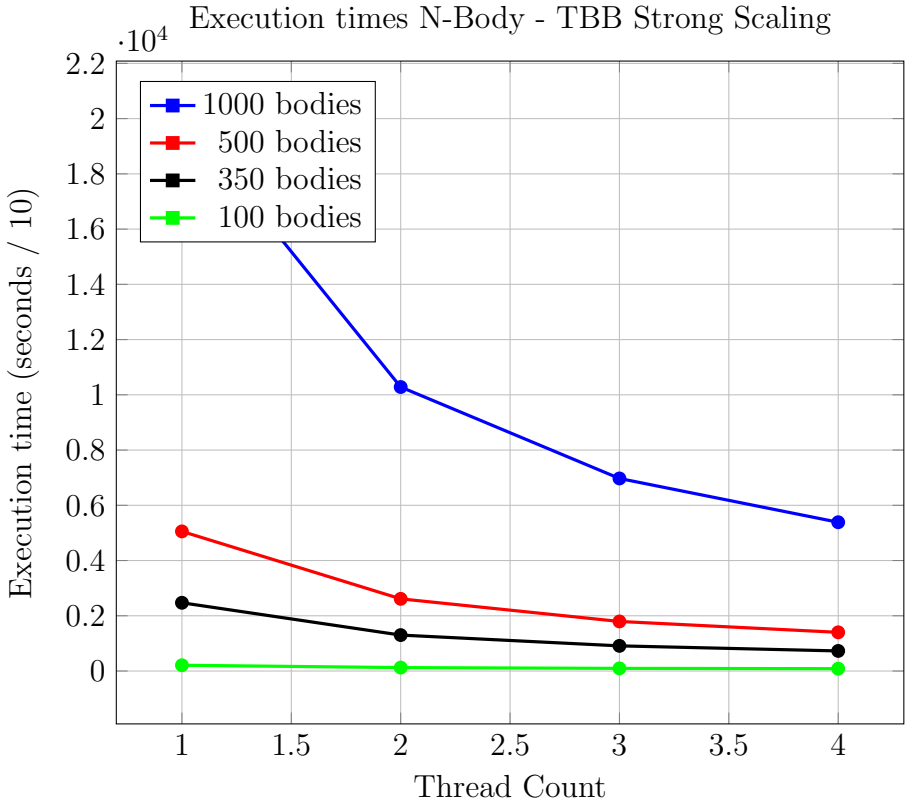

Figure 6: N-Body Simulation. Time step: 0.01, final time step: 10.0. Machine used: X64 Ubuntu Server (Processor: 1 CPU 3.0 GHz AMD Phenom II X4 940, Memory: 8GB DDR2 800MHz 64bit, GPU: Nvidia GeForce 6800GS 256MB)

In the case of the *Barnes-Hut*, it was necessary to provide some locking mechanism, so the TBB's *atomic* container was used for the tree node pointers. Via profiling, it was noticed that a small overhead was applied in the execution times. But this was a small and fixed/constant slowdown, barely noticeable, especially in large simulations.

Concurrent vectors, with *cache_aligned* allocators [4] in particular, really do provide a solid speed up. It was noticed in practice, that the *false sharing* protection does provide speed up, even when comparing different platforms (Intel i5 4 core Windows 8.1 x64 versus AMD Phenom II Black edition and Ubuntu 14.04 x64). But in some cases, especially when the collection is

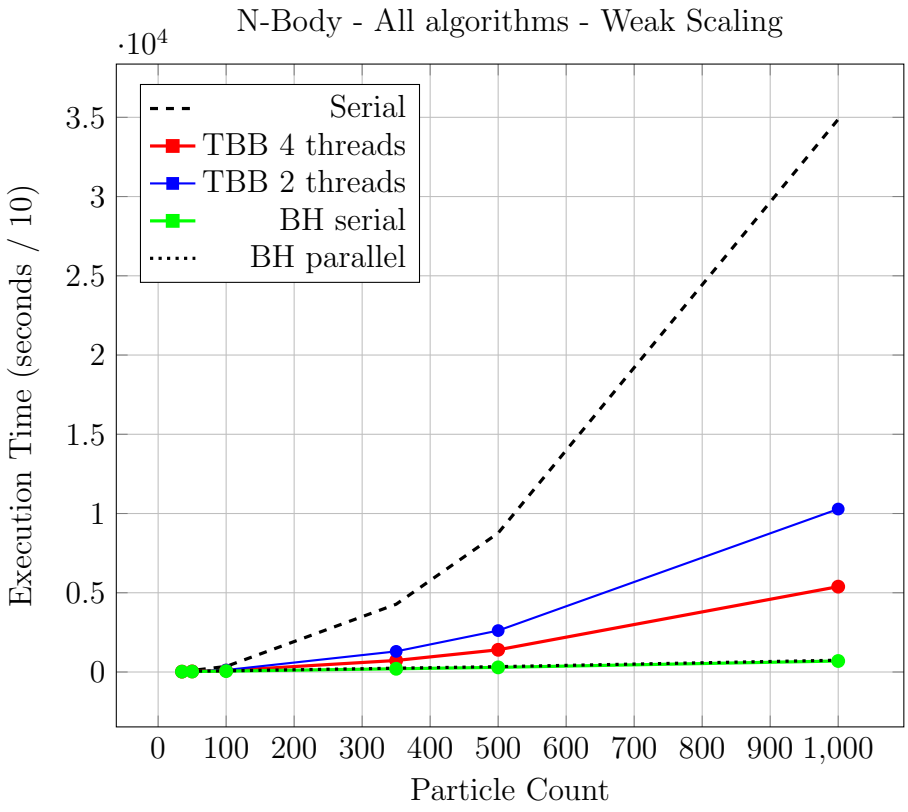

Figure 7: N-Body Simulation. Time step: 0.01, final time step: 10.0. Machine used: X64 Ubuntu Server (Processor: 1 CPU 3.0 GHz AMD Phenom II X4 940, Memory: 8GB DDR2 800MHz 64bit, GPU: Nvidia GeForce 6800GS 256MB)

small, they are not applicable.

Worth mentioning is that the regular *scalable allocator* didn't provide any benefit versus the *cache_aligned*. This was due to the fact that the collections of data in an N-Body simulation don't scale/change rapidly in size, which is one of the benefits over the cache_aligned allocator.

My final conclusion is that:

- Compute bound algorithms like $O(n^2)$ are easy to implement, have some margin of speed up (depending on the number of cores), easy to debug, especially when applying parallelism.

- Memory bound algorithms, like *Barnes-Hut* can be hard to implement, even harder to debug. Applying parallelism isn't trivial. But if done correctly, they can scale very well, especially when increasing the problem size/*weak scaling*.

References

[1] *Intel Developer zone - n-bodies: a parallel TBB solution: computing accelerations? or forces?* - https://software.intel.com/en-us/blogs/2009/09/22/n-bodies-a-parallel-tbb-solution-computing-accelerations-or-forces

[2] *Wikipedia: Fast Inverse Square Root* - https://en.wikipedia.org/wiki/Fast$_i nverse_s quare_r oot$

[3] *Wikipedia: Barnes-Hut simulation* - https://en.wikipedia.org/wiki/Barnes-Hut_simulation

[4] *Tutorial: Scalable Memory Allocator* - https://www.threadingbuildingblocks.org/tutorial-intel-tbb-scalable-memory-allocator