Benchmarking different hardware architectures using a parallel implementation of the N-Body Simulation algorithm

Jan MRÁZEK, mrazeja7@ksu.edu

***Abstract*—This is a write-up paper which describes my efforts when working on a course project for the Advanced Computer Architecture Experiments course at Kansas State University during the Fall 2017 semester.**

First, the N-Body Simulation problem is described. As there are multiple ways of simulating a system of large amounts of space bodies, two different algorithms are presented and their pros, cons, accuracy, time complexity and implementation difficulty are compared.

One of the algorithms is then implemented in a simple, naive way and then optimized, parallelized and custom-tuned to run on computer chips of very different architectures.

This optimization is done in multiple iterations, each time with explanations as to the relative speedups and overall performance.

Finally, multiple different versions of the algorithm are run on computer hardware of different architectures, and the resulting data is analyzed and commented on. Performance is measured in the amount of billions of floating-point operations per second, and the efficiency of each computer chip is addressed. An overall conclusion is drawn which compares the results gathered during the benchmarking phase of the project.

I. PROJECT PROPOSAL

In my course project, I explored the ins and outs of efficient parallel design and better cache-friendliness when it comes to programming techniques when applied on different CPU architectures. After briefly describing the N-Body simulation and two different algorithms which can be used to solve it, I chose one and implemented it in the C++ language. I then fine-tuned the source code to better suit each CPU architecture that was tested. Then I benchmarked all of the different implementations that I created and commented on the (sometimes vast) performance differences and speedup yields between them.

II. PROBLEM DEFINITION

The N-Body simulation

The N-body simulation is a way to simulate the continuous gravitational interactions among potentially very large amounts of galactic bodies, solar systems and similar particles, where every body influences the positions and velocity vectors of all other bodies. It is a tool that is used widely and frequently in the fields of astronomy and astrophysics to help researches reach a better understanding of the evolution of a large-scale structure of the universe.[?]

III. PROBLEM BACKGROUND AND SCOPE

There are many ways to go about the problem of implementing a suitable algorithm for the simulation of galactic particle movements.

The first thing that we have to think about when it comes to making a choice in this matter is the trade-off between a relatively reasonable time-complexity of the algorithm and the scientific accuracy of the algorithm according to the laws of physics that is required.

If we needed to run a truly massive number (hundreds of millions to billions of space bodies) N-Body movement simulation for a high amount of iterations, it would be best to choose the Barnes-Hut algorithm. If instead we prioritize complete accuracy of all computations above all and don't really mind extremely long running times on the same amount of bodies for the same amount of iterations, it is best to stick with an algorithm that solves the simulation in a brute-force manner.

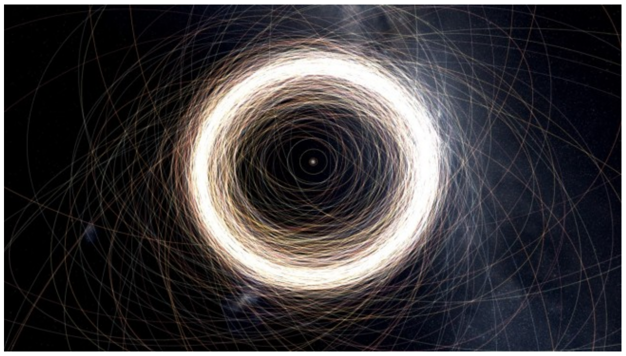

Figure 1: An illustration of an orbital timelapse in *Universe Sandbox*^{2}

[?]

Brute-Force algorithm

The most obvious and most accurate algorithm of solving the N-body simulation is simply a brute-force approach. This algorithm makes sure to compute the forces that are being applied between all pairs of space particles within a given system, thus ensuring scientifically correct movements. This algorithm is also sometimes called ”the all-pairs N-Body simulation algorithm.”

In the computation of each iteration, or time step, we calculate the force vectors $\mathbf{f}_{ij}$, which signifies the force that is applied on body i caused by its gravitational pull to a different body j.

Given **n** bodies with an initial position vector $\mathbf{p}_i$ for $1 \leq i \leq \mathbf{n}$, the force vector $\mathbf{f}_{ij}$ is given by the following formula [**?**]:

$$\mathbf{f}_{ij} = K \frac{m_i m_j \cdot r_{ij}}{||r_{ij}||^2} \tag{1}$$

where m_i and m_j are the respective masses of bodies i and j; r_{ij} is the length vector between bodies i and j, i.e. $r_{ij} = x_j - x_i$, and K is the gravitational constant used in the simulation. As you can see from the equation above, the force vector is not linear; instead, the relative force pulling on body i diminishes with the square of distance between these two bodies (or the vector r_{ij}).

The total force that acts on body i - $\mathbf{F}_i$ while it interacts with all other $\mathbf{n}-1$ bodies, is given by the sum of all particular force vectors:

$$\mathbf{F}_i = \sum_{j=1}^{N} \mathbf{f}_{ij} = K m_i \sum_{j=1}^{N} \frac{m_j \cdot r_{ij}}{||r_{ij}||^2} \tag{2}$$

Of course, being a brute-force algorithm, the asymptotic bound on computation time is not very good. Because the algorithm needs to touch all particle pairs during a single iteration of the computation, the time complexity is bounded by $O(k \cdot n^2)$, where k is the number of iterations, or time steps we simulate for, and n stands for the number of bodies there are in the given system.

The Barnes-Hut algorithm

The Barnes-Hut algorithm offers a very different approach to solving the simulation in a timely, yet less accurate manner. This algorithm is a mere approximation of the N-Body simulation problem, which implies that it is best used when the running time of the computation is a significant constraint and the physical accuracy of the simulation is maybe not as important.

The Barnes-Hut algorithm divides the whole space of the given system into different quadrant cells. The algorithm then only computes the forces between particles within a single cell, not taking into consideration any other single particles. For quadrant cells where the number of bodies occupying it is particularly small, the brute-force algorithm is used to solve the simulation instead, as it is inefficient to divide the space within the quadrant cell any more.

All other cells are then treated as one, massive body, which has the mass of all particles contained within a given cell. The position of this dense uber-body is centered around the center of mass of all neighboring particles within that cell. Other than this important modification, the algorithm still works very similarly to the brute-force algorithm.

The fact that only a single massive body is sometimes used instead of many other smaller bodies in the computation drastically reduces the number of pairs whose influences have to be calculated. The time complexity of the Barnes-Hut algorithm is thus bounded by $O(k \cdot n \cdot logn)$, where k is the number of time steps we simulate for, and n is the number of bodies there are in the given system.

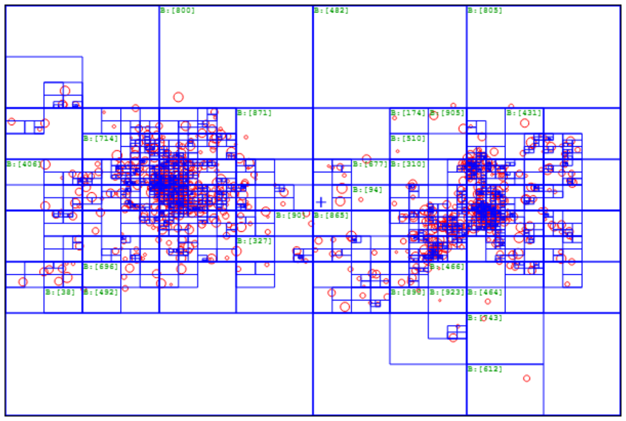

Figure 2: An illustration of the Barnes-Hut quadrant division [**?**]

It is important to also mention the drawbacks of this algorithm. Because of the approximated mass of a majority of the particles during a single iteration, the Barnes-Hut algorithm is not very accurate in the scientific sense.

Apparently, the recursive nature of the algorithm also makes it somewhat unsuitable for parallel implementations, meaning that the brute-force algorithm might actually perform better on a massively multi-threaded machine for systems with reasonably low numbers of space bodies.

These reasons are why I chose to pursue the implementation, optimization and parallelization of the brute-force algorithm over the Barnes-Hut algorithm.

IV. Related work in literature

Typical N-Body simulation problems can be traced all the way back to the 1960s. The first astrophysical simulation of this kind was performed by Sebastian von Hoerner in 1960 for a dynamic system consisting of just 16 particles. [**?**]

At that time, the lack of computational facilities meant that scientists had to decide between performing fewer interations of the simulation with the highest amount of particles (n) possible, or perform a much more in-depth study of significantly smaller systems.[**?**]

A milestone of an even amount of $n = 100$ bodies was reached by S. J. Aarseth in 1962. Another milestone of $N = 500$ for spherical liquid bodies was reached in 1964 by A. Rahman. [**?**]

Since those times, it can be said that the performance of N-Body simulations has been limited by Moore's law and therefore advancements in moden computer architectures. A significant single-machine speedup was reached with the introduction of multi-core processors. Another fairly large (and fairly recent) leap in computational performance was caused by the dawn and spread of graphics processing units (GPUs) and their use for scientific calculations. Technologies like *OpenCL* and *CUDA* have been extremely helpful in all kinds of topics in the field of scientific computation.

N-Body simulations have been at the forefront of high-performance computing in the terms of *FLOP/s* (or floating point operations per second) that they deliver.[**?**] This is especially true for the brute-force (or all-pairs) N-Bbody algorithm, as it responds fairly positively to parallelization efforts.

There have been discussions whether the *FLOP/s* metric is even useful in the comparison of these algorithms, or whether the "amount of science done" should somehow be considered instead. Other algorithms, such as the Barnes-Hut algorithm, may be less efficient theoretically, but they can deliver more "information" in the same amount of computation time.

Nowadays, it is possible to simulate dynamic systems with as many as 10^{11} space particles. In 2010, a team of programmers at Nagasaki University in Japan created a parallel implementation of the N-body problem using NVIDIA's CUDA technology. When the massively parallel implementation was ran on a supercomputer containing a total of 576 *GT200* (Tesla 2.0 architecture) GPUs, they measured a performance of 190.5 *TFLOP/s*. [**?**]

The theoretical processing power of such a supercomputer should be in the neighborhood of about 580 *TFLOP/s*, as one *GT200* chip can perform as many as (depending on core configuration and clock speeds) 1010 *GFLOP/s*.[**?**]

The resulting 190.5 *TFLOP/s* is a very respectable number, especially since the code that was used to run the computation used the Barnes-Hut algorithm, instead of the "faster" (*FLOP/s*- or performance-wise) brute-force method.

V. Methods and facilities used

I will try to create an efficient implementation of the brute-force algorithm with regards to cache-friendliness, then I will parallelize it using OpenMP and possibly CUDA, if time constraints permit. I will then measure the computation times on a multitude of different devices:

- a typical personal computer CPU architecture, i.e. Intel's x86-64. I will use Kansas State University's Beocat supercomputer to run the tests on this architecture.
- an ARM-architecture based computer. I will use my own OrangePi One micro-computer, which is currently setup as a media center in my home. This micro-computer uses a single Allwinner H3 SoC (system-on-chip) with a quad-core processor and an integrated GPU. The CPU's clock speed is 1200 MHz. I will install a fresh distribution of ARMbian on it and run my tests that way.
- a computer equipped with Intel's Xeon Phi coprocessor card. I hope to confirm my theoretical knowledge I have about this coprocessor card – that it is extremely suitable for simple, embarrassingly parallel computations and vector computing. I hope to see a massive speedup compared to Intel's x86-64 architecture. I will run my tests on a Xeon Phi-equipped computer at my home university (CTU in Prague, CZ).
- if time permits, I would like to run a CUDA implementation on a computer equipped with an nVidia graphics processor. I would like to run these benchmarks on nodes that are part of Beocat and have dedicated GPUs just for this purpose.