1. Introduction

I. Why Parallel Computing?

During the period from 1986 to 2002, the average performance increase recorded in microprocessors was nearly 50% per year. This allowed software developers to simply wait until the release of next generation microprocessors to increase the performance of their programs, leaving the task of increasing the performance of computers for hardware developers for the most part. The Moore's Law states that the number of transistors that fit on a chip will double roughly every 24 months. Theoretically, this implies that the speed will also double. However, the need for increasing the power usage efficiency became a problem. According to a study done by Edgar Grochowsky, a Research Scientist at Intel, the power consumption of new chips grew almost quadratically through the years, revealing a completely unsustainable power model–the power wall. What ensued was a radical redesign and optimization of the chip architecture for power.

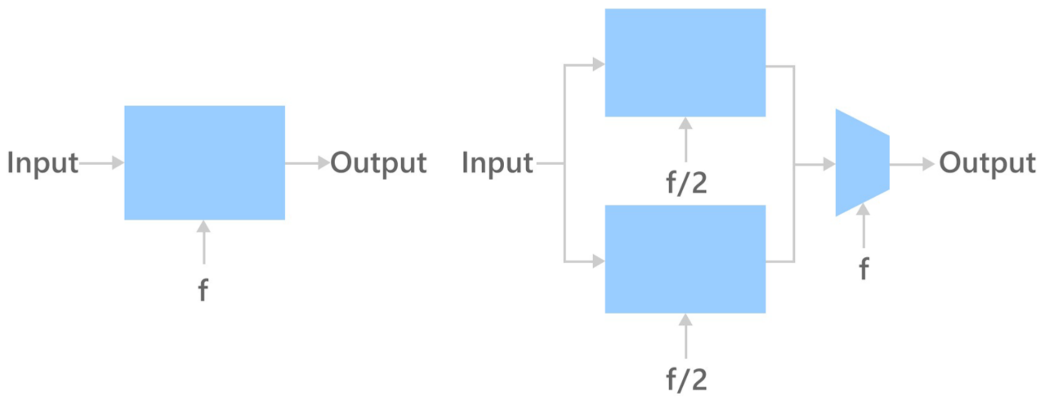

Figure 1.1: Two chip architectures.

In the left one of above two architectures, some input is driven through a chip with some output at frequency F, requiring power P.

$$P = CV^2F \tag{1}$$

On the other (right), two cores are present on the same chip, but we are driven at half the frequency (F / 2). However, since more connections (wires) are also involved, the capacitance needs to be increased a little higher than just doubling (C * 2.2). Frequency scales with voltage, but since there are leakages and the related, assume the voltage is not half, but a little more than that (V * 0.6). Therefore, according to Eq. (1), the power usage decreases in the half-frequency case by about 40% (P * 0.396).

That means, for a given input F with one core versus F / 2 with two cores we are getting the same output. But with two cores we are getting 40% power. This is why parallel computing is important, it gives us the ability to get the same amount of work done with multiple cores at a lower frequency with a tremendous power saving.

II. N-body Simulation

In an n-body problem, we need to find the positions and velocities of a collection of interacting particles over a period of time. For example, an astrophysicist might want to know the positions and velocities of a collection of stars, while a chemist might want to know the positions and velocities of a collection of molecules or atoms. An n-body solver is a program that finds the solution to an n-body problem by simulating the behavior of the particles. The input to the problem is the mass, position, and velocity of each particle at the start of the simulation, and the output is typically the position and velocity of each particle at a sequence of user-specified times, or simply the position and velocity of each particle at the end of a user-specified time period.

For the sake of explicitness, let us write an n-body solver that simulates the motions of planets or stars. We will use Newton’s second law of motion and his law of universal gravitation to determine the positions and velocities. Thus, if particle q has position sq(t) at time t, and particle k has position sk(t), then the force on particle q exerted by particle k is given by

$$\mathbf{f}_{qk}(t) = -\frac{Gm_q m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right] \qquad (2).$$

Here, G is the gravitational constant ($6.673 \times 10-^{11}m^3/(kg \cdot s^2)$), and m_q and m_k are the masses of particles q and k, respectively. The notation $|\ S_q(t) - S_k(t)\ |$ is representing the distance from particle k to particle q.

If Formula (2) is used for finding the total force on any particle, it is obtained by adding the forces due to all the particles. Therefore, assuming n particles numbered from 0 to n − 1, the total force on particle q would be,

$$\mathbf{F}_q(t) = \sum_{\substack{k=0 \\ k\neq q}}^{n-1} \mathbf{f}_{qk} = -Gm_q \sum_{\substack{k=0 \\ k\neq q}}^{n-1} \frac{m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right] \quad (3).$$

A naïve approach to compute the forces would be along the lines of following pseudo-code.

```
for each particle q {
   for each particle k != q {
      x_diff = pos[q][X] − pos[k][X];
      y_diff = pos[q][Y] − pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      forces[q][X] −= G*masses[q]*masses[k]/dist_cubed * x_diff;
      forces[q][Y] −= G*masses[q]*masses[k]/dist_cubed * y_diff;
   }
}
```

Figure 1.2: A naïve approach in pseudo-code.

However, according to Newton's third law of motion, for every action there is an equal and opposite reaction. Therefore, it is possible to reduce the total number of calculations required for the forces by half. That is, if the force on particle q due to particle k is f_{qk}, then the force on k due to q would be $-f_{qk}$. Using this simplification, we can modify our code to compute forces, as shown in Figure 1.3. For a better understanding of the presented pseudocode, imagining the individual forces as a two-dimensional array would be helpful.

$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\ -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\ -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} & -\mathbf{f}_{1,n-1} & -\mathbf{f}_{2,n-1} & \cdots & 0 \end{bmatrix}$$

Figure 1.4: Individual forces as a two-dimensional array.

Following is the reduced algorithm.

```
for each particle q
   forces[q] = 0;
for each particle q {
   for each particle k > q {
      x_diff = pos[q][X] − pos[k][X];
      y_diff = pos[q][Y] − pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
      force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff

      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] −= force_qk[X];
      forces[k][Y] −= force_qk[Y];
   }
}
```

Figure 1.5: Reduced algorithm for computing n-body forces.

If we are to use a numerical method for estimating the position and the velocity of each particle at each time frame, we’ll be approximating the values of the position and velocity at relevant timestamps during the simulation. While there exist multiple numerical methods at our disposal for accomplish the above, the Euler’s method (named after the famous Swiss mathematician Leonhard Euler) was used. The basic idea here is that the value of a function $g(t_0)$ at time 0, along with its derivative $g'(t_0)$at timet0, can be used to approximate its value at time $t_0+\Delta t$ by using the tangent line to the graph of $g(t_0)$.

The data (mass, coord_x, coord_y, coord_z, velocity_x, velocity_y, velocity_z) bodies for the simulation were read from a csv file, while using multiple csv files for multiple data counts. Six datasets were used with body counts of 5, 6, 7, 8, 9, and 10–in thousands. 50 frames for each body count were simulated and the executions were timed, as discussed in more detail in coming sections for each implementation. All celestial bodies were considered to be particle objects while friction in medium was discarded and the time duration between two in-simulation frames was set as 0.01 seconds (100 frames/sec.). The simulation was implemented in a Euclid space with regard to Newton’s Universal Constant of Gravity.

2. Results

Note that all time values presented are in seconds where not specified otherwise.

I. Serial Implementation

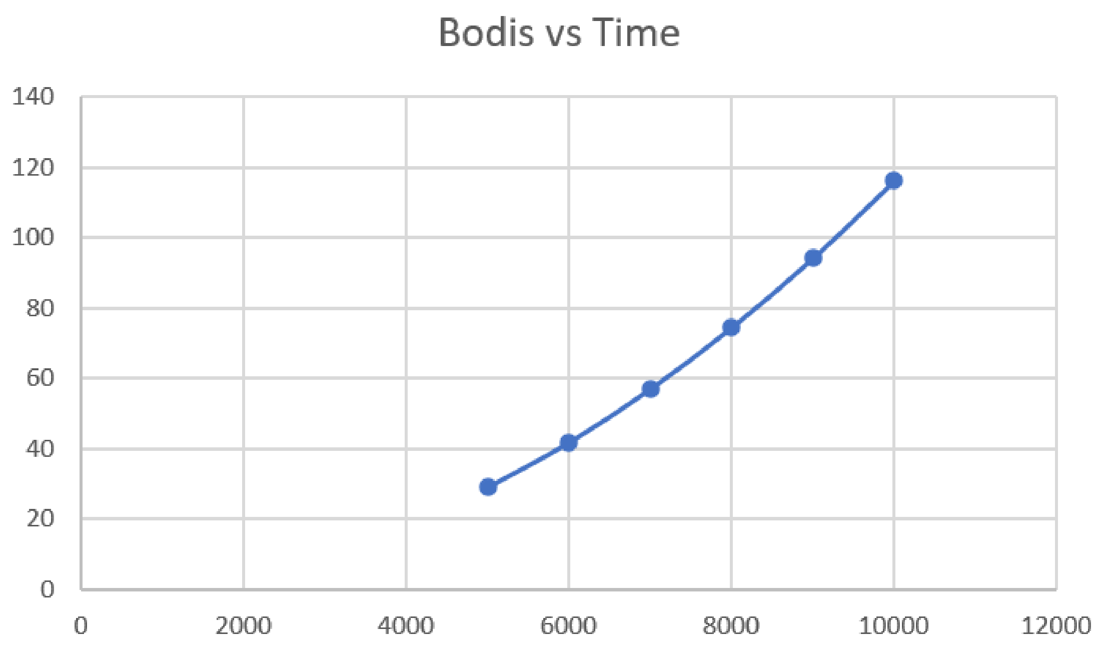

Chart 1.1: Serial - Bodies vs Time

II. OpenMP Implementation

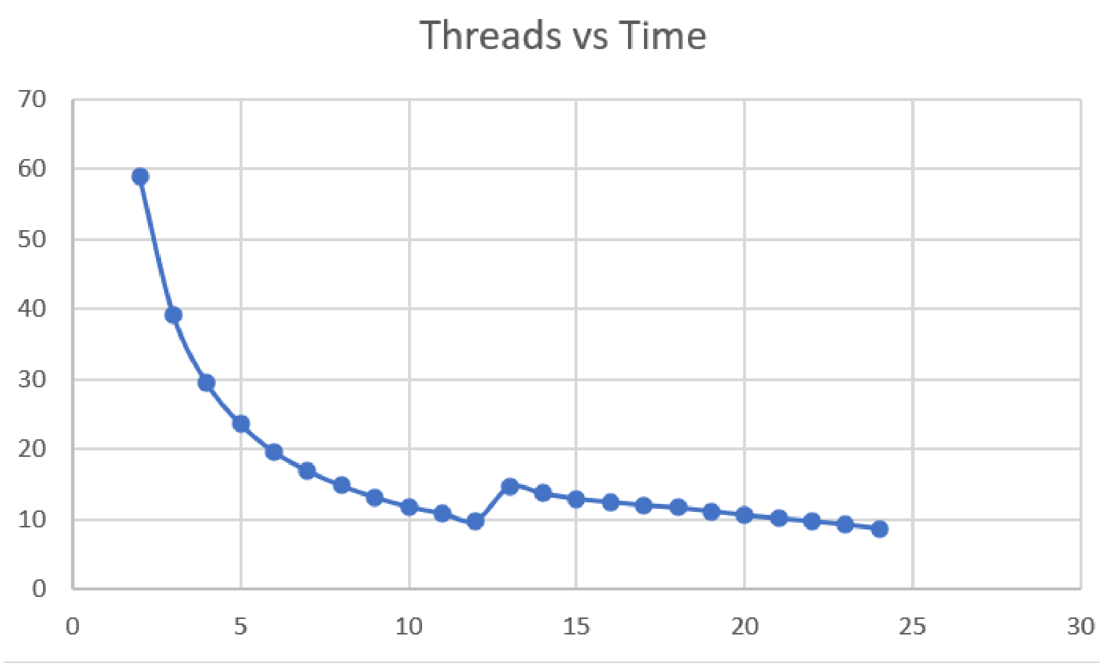

Chart 2.1: OpenMP - Threads vs Time

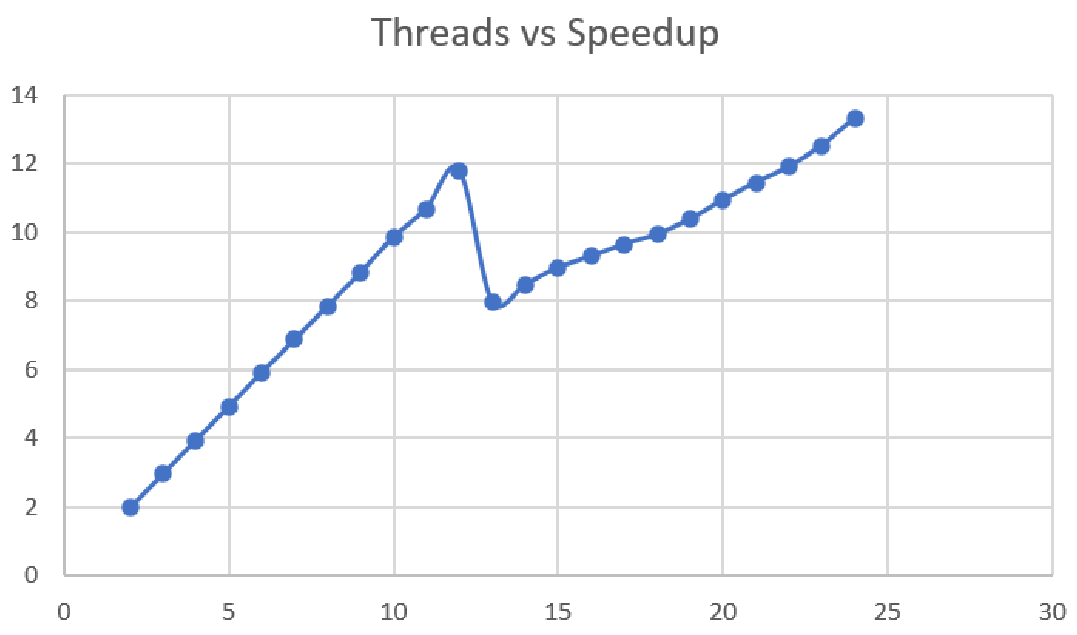

Chart 2.2: OpenMP - Threads vs Speedup

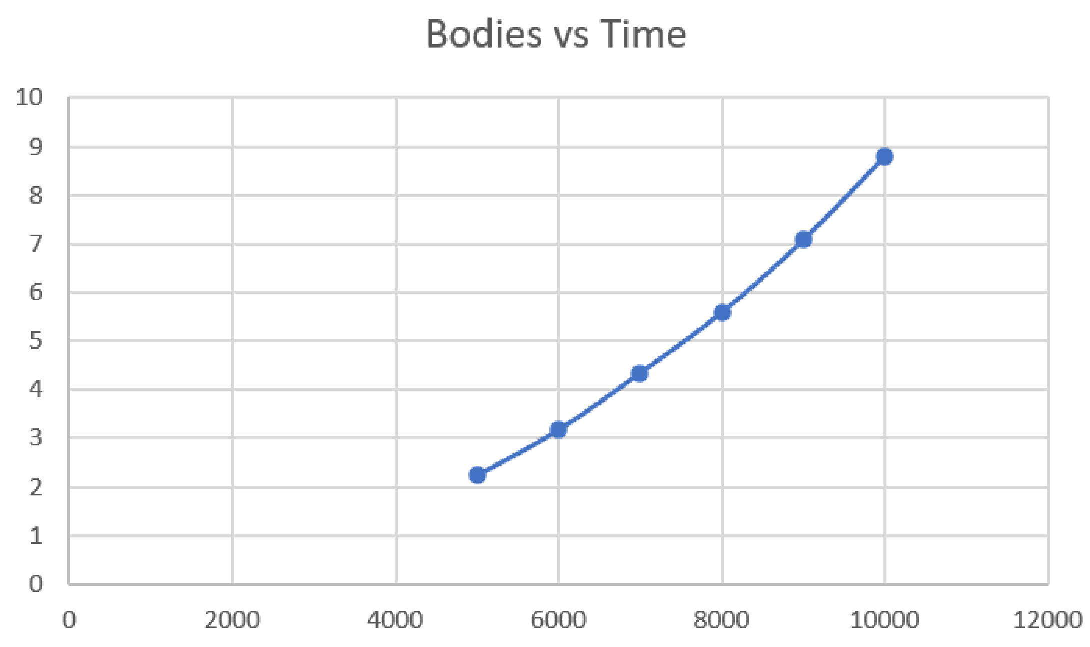

Chart 2.3: OpenMP - Bodies vs Time

III. MPI Implementation

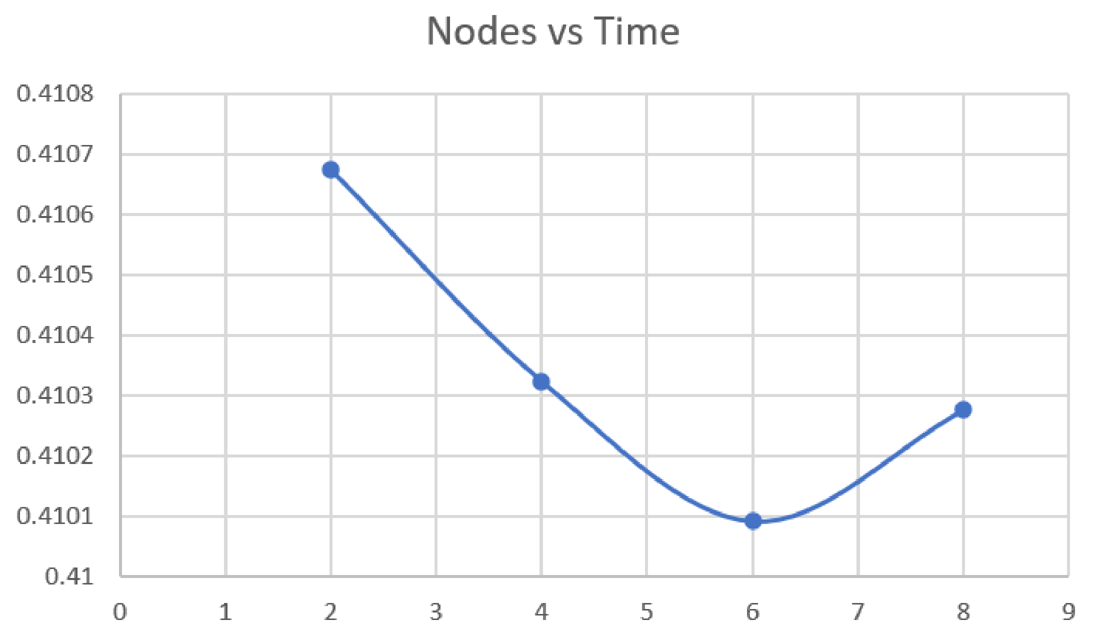

Chart 3.1: MPI - Nodes vs Time

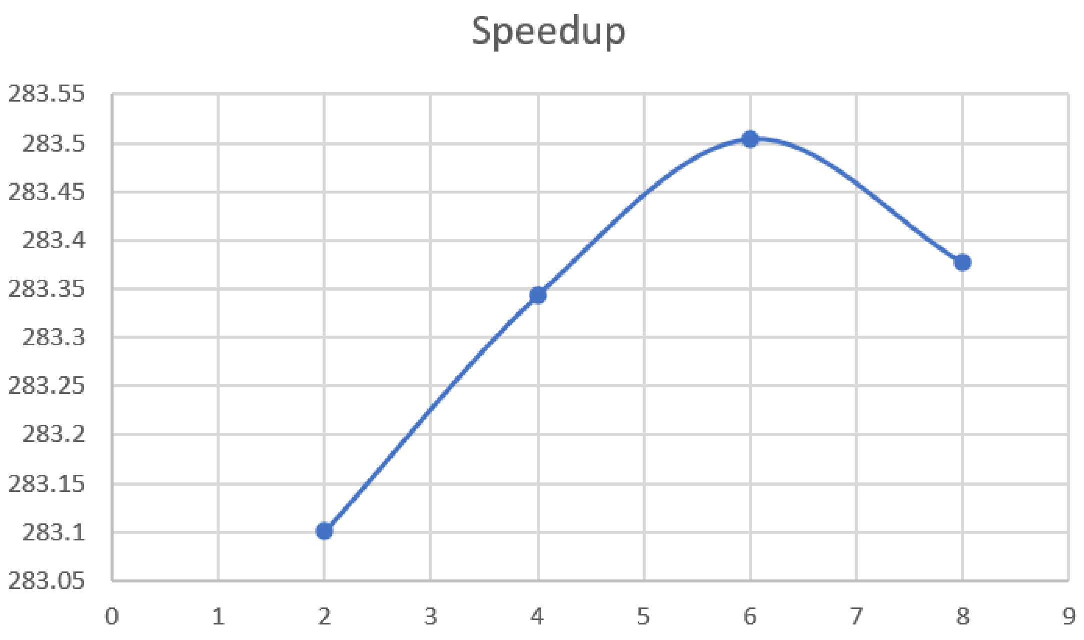

Chart 3.2: MPI - Nodes vs Speedup

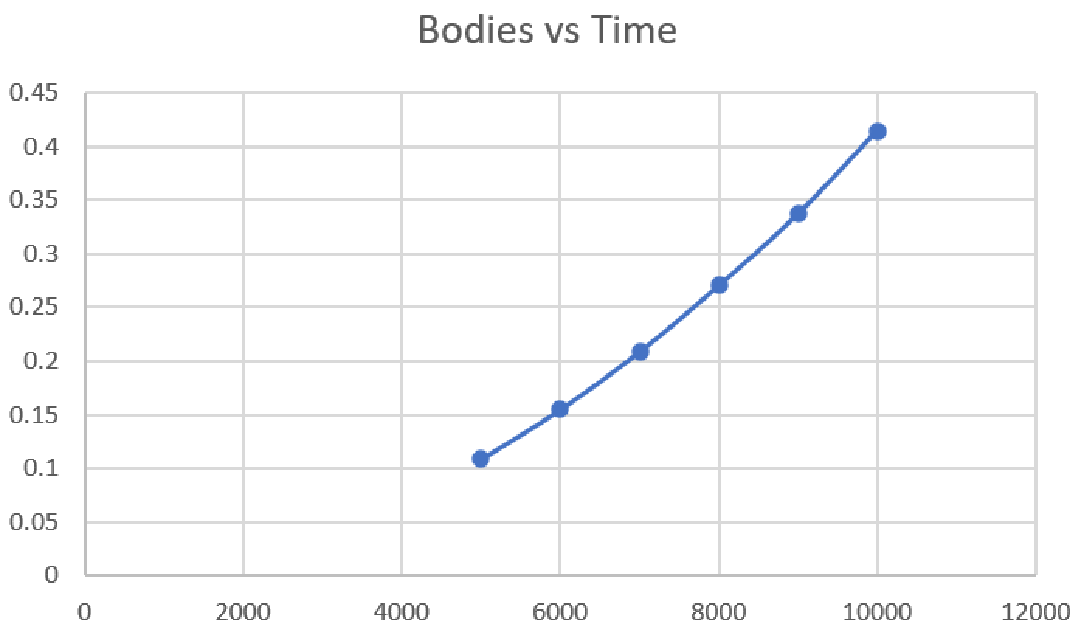

Chart 3.3: MPI - Bodies vs Time

IV. CUDA Implementation

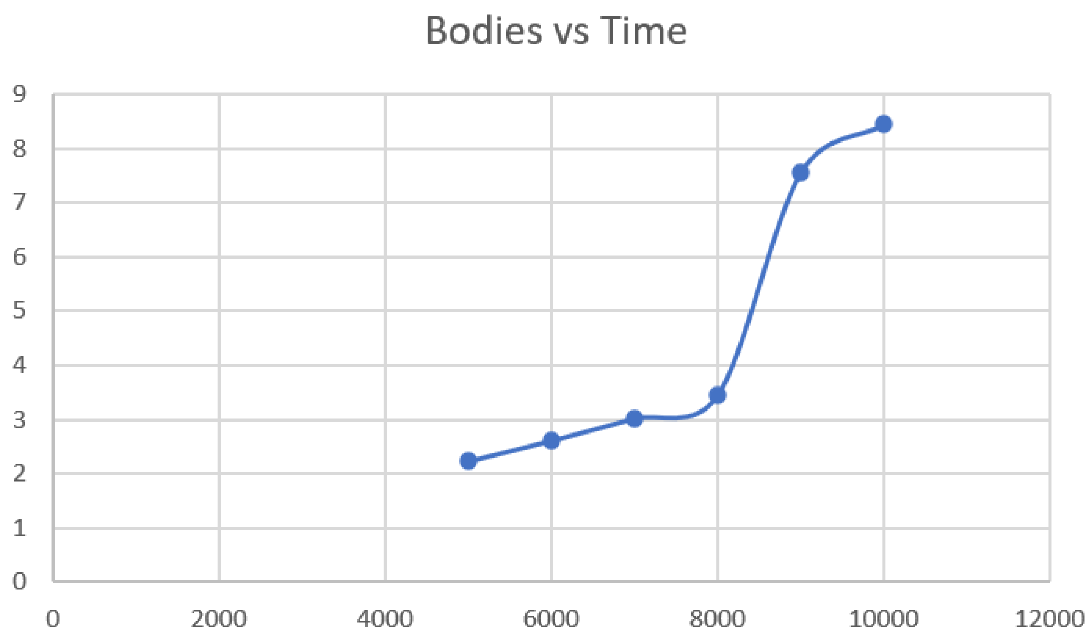

Chart 4.1: CUDA - Bodies vs Time

When executing the simulation with CUDA, cores were not variated. An speedup of 13.76556446 was recorded for CUDA.

3. Discussion

I. Serial Implementation

Tested serial version of the n-body solver was developed to be scalable, with a minimal performance compromise. A custom struct was defined to hold each body with the data within consisting of 7 float variables for holding the data read from the csv file for each celestial body. A pointer variable was created, and adequate memory was allocated for holding an array of structs, on which the body structures were stored–before, during, and after calculations.

II. OpenMP Implementation

The C code developed for the assignment with the aid of the OpenMP library uses a parallelization strategy of calculating the velocities, and forces (in x, y, z dimensions) using a dedicated thread spawned solely for the task. This calculation is carried out for each body in the n-body pool for each frame during the simulation. Updating the values (velocities) in the shared array of structs was carried out in a critical section residing within the scope of a #pragma omp critical compiler directive. The positional value updates were carried out serially since the task contained minimal calculations and the values resided in shared memory (process heap) as well. It is worth noting that if this were to be parallelized, it would have caused a degrade of performance owing to the overhead penalties such methods ensue.

Multiple research papers were found on the subject to be using similar parallelization strategies, with the most notable being [1] written by Andrey Vladimirov of Colfax International and Vadim Karpusenko of Stanford University in 2013 on testing a then-new version of Intel Xeon Phi coprocessors including comparisons with multiple capable and competing processors at the time.

Ability for adding directives incrementally–gradual parallelization should be considered an advantage in OpenMP. While the library eases the implementation of POSIX threads, the lack of support for distributed memory computers was experienced, however, this should not be considered an actual drawback since the task could be better parallelized by integrating with a specific library (i.e. MPI) supporting said architecture.

III. MPI Implementation

The MPI implementation leverages the in-built scatter function to evenly scatter the bodies among the processes running in the cluster (with one or more processes running on each processor, with multiple processors available in each node). The data passed using the scatter function was not necessary to be serialized since the struct containing the bodies comprises of multiple elements of the same data type (seven float variables). The bodies were evenly scattered among multiple processes spawned by available processors. Each process spawned in the simulation was used for calculating the assigned subset of bodies, and the final velocities and forces calculated were collected by the root process (rank=0) using the gather function provided by MPI (in instances where the number of processes is a factor of celestial bodies being simulated). It should be noted that the time spent on scattering and gathering of data was not included in the measured process times.

Research conducted on the topic was found out to have used a similar direct approach in simulating n-bodies [2]. According to the paper, the particle-particle codes using direct summation methods are considered a good example of algorithms where parallelization can speed up the computation in an efficient way.

Since distributed memory computers are less expensive than large, shared memory computers, this method of parallelization may be favored over more expensive alternatives. It is also worth mentioning that MPI can handle processes on both shared and distributed memory architectures. One of the main limitations identified on the high-performance cluster where the execution was done happens to be the unavailability of more than 8 processes at runtime–this is due to a resource allocation limitation. Also, the performance is constrained by the communication network between nodes.

IV. CUDA Implementation

The bodies are stored in an array of structs, which in turn gives the benefit of accessing consecutive elements by incrementing/decrementing the pointer along the array. For calculating the velocities and forces for each body in the simulation (in each frame), a separate thread is assigned. This resulted in a scalable solution where the thread count could be increased/decreased according to the body count requirements. A block size of 256 (threads) was used, and the number of blocks were dynamically determined by dividing the number of celestial bodies available for simulation by the block size for each execution. The bodies were allocated memory on the host first and then copied to device memory. Also, while there exist methods for mapping both host and device memory to a single memory space, such methods were not utilized in the simulation. The delays occurred by copying data from host to device and vice versa were not included in the measured execution times.

Multiple research papers were available that used the algorithm described above. The paper by Lars Nyland et al. discusses multiple methods for writing and optimizing an n-body solver implementation leveraging the power of parallel processing inherent to GPUs with CUDA.

Scattered reads, the ability for code to read from arbitrary addresses in memory is considered an advantage in CUDA. Unified memory is also available in newer versions of the library (version 4.0 and above). However, although partially alleviated with asynchronous memory transfers handled by the GPU’s Direct Memory Access engine, copying between host and device memory may incur a major performance hit on execution of programs with CUDA.

4. References

[1] Test-driving Intel® Xeon Phi™ coprocessors with a basic N-body simulation. 2013. Online: https://colfaxresearch.com/test-driving-intel-xeon-phi-coprocessors-with-a-basic-n-body-simulation/

[2] Pereira, Nuno Sidónio Andrade. "A parallel N-body integrator using MPI." In International Conference on Vector and Parallel Processing, pp. 627-639. Springer, Berlin, Heidelberg, 1998.

[3] Nylons, Lars, Mark Harris, and Jan Prins. “Fast n-body simulation with Cuda”. 2007.

[4] Pacheco, P., “An Introduction to Parallel Programming”. Elsevier. 2011

[5] Quinn, Michael J., “Parallel Programming in C with MPI and OpenMP”. MGH Higher Education. 2003.

[6] Kirk, David B., and W. Hwu Wen-Mei. “Programming Massively Parallel Processors: a Hands-on Approach.” Morgan Kaufmann, 2016.