



School of Computer Science and Engineering
J Component report

Programme : B.Tech
Course Title : Parallel and Distributed Computing
Course Code : CSE4002
Slot : F2

**Title: The Gravitational N-Body Simulation analysis using
Sequential, OpenMP and CUDA**

Team Members : Abhishek N.N.(20BCE1025)
Mayank Gupta(20BCE1538)

Faculty: Dr. Kumar R

Sign:
Date:

DECEMBER 2022



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

DECLARATION

We hereby declare that the project report entitled “**The Gravitational N-Body Simulation analysis using Sequential, OpenMP and CUDA**” undertaken by me under the supervision of **Dr. Kumar R**, School of Computer Science and Engineering, Vellore Institute of Technology, Chennai 600127 in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology – Computer Science and Engineering** is a record of bonafide work carried out by us. We further declare that the work reported in this report has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma of this institute or of any other institute or university.

Signature

Mayank Gupta Abhishek N.N.
(20BCE1538) (20BCE1025)



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

CERTIFICATE

This project report for the course “**The Gravitational N-Body Simulation analysis using Sequential, OpenMP and CUDA**” is prepared and submitted by **Abhishek N.N. (Register No: 20BCE1025) Mayank Gupta (Register No: 20BCE1538)**. It has been found satisfactory in terms of scope, quality and presentation as partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology – Computer Science and Engineering** in Vellore Institute of Technology, Chennai, India.

Examined by:

Examiner I

Examiner II

ACKNOWLEDGEMENT

This is an acknowledgement to VIT Chennai as a whole to motivate and encourage me to develop a curious mind that never stops learning. This lab has only been possible with the inordinate and meticulous efforts VIT has put in while shaping my journey of becoming more diligent and capable than I already was.

We extend my gratitude to the professor Dr. Kumar R., Associate Professor B.Tech Computer Science and Engineering , SCOPE, VIT Chennai who took our CSE3505 course. We truly appreciate him and his time he spent helping and pushing us to be good learner. He definitively love to teach. We thank him for taking the course and guiding us. We thank our teammates who worked with us and helped us grow during this course. We also extend our gratitude to our parents who constantly supported us throughout the journey.

CONTENTS

Chapter	Title	Page
	Title Page	i
	Declaration	ii
	Certificate	iii
	Acknowledgement	iv
	Table of contents	v
	Introduction	
	i) Objective and goal of the project	
	ii) Problem Statement	
	iii) Motivation	
	iv) Challenges	
	Literature Survey	
	Requirements	
	Methodology Used	02
	i) All Pair Algorithm (N^2)	
	ii) Barnes Hutt Algorithm ($N\log(N)$)	
	Performance Analysis	
	Conclusion	04
	References	09
	10

Introduction

Computational methods to track the motions of particles which interact with one another, and possibly subject to an external field as well, have been the subject of extensive research for many years. So-called “N-body” methods have been applied to problems in astrophysics, semiconductor device simulation, molecular dynamics, plasma physics, and fluid mechanics. In this paper we consider the example of gravitational N-body simulation.

An N-body simulation numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body. A familiar example is an astrophysical simulation in which each body represents a galaxy or an individual star, and the bodies attract each other through the gravitational force, as in Figure 31-1. N-body simulation arises in many other computational science problems as well. For example, protein folding is studied using N-body simulation to calculate electrostatic and van der Waals forces. Turbulent fluid flow simulation and global illumination computation in computer graphics are other examples of problems that use N-body simulation.

The all-pairs approach to N-body simulation is a brute-force technique that evaluates all pairwise interactions among the N bodies. It is a relatively simple method, but one that is not generally used on its own in the simulation of large systems because of its $O(N^2)$ computational complexity. Instead, the all-pairs approach is typically used as a kernel to determine the forces in close-range interactions. The all-pairs method is combined with a faster method based on a far-field approximation of longer-range forces, which is valid only between parts of the system that are well separated. Fast N-body algorithms of this form include the Barnes-Hut method (BH) (Barnes and Hut 1986), the fast multipole method (FMM) (Greengard 1987), and the particle-mesh methods (Hockney and Eastwood 1981, Darden et al. 1993).

The all-pairs component of the algorithms just mentioned requires substantial time to compute and is therefore an interesting target for acceleration. Improving the performance of the all-pairs component will also improve the performance of the far-field component as well, because the balance between far-field and near-field (all-pairs) can be shifted to assign more work to a faster all-pairs component. Accelerating one component will offload work from the other components, so the entire application benefits from accelerating one kernel.

All-Pairs N-Body Simulation

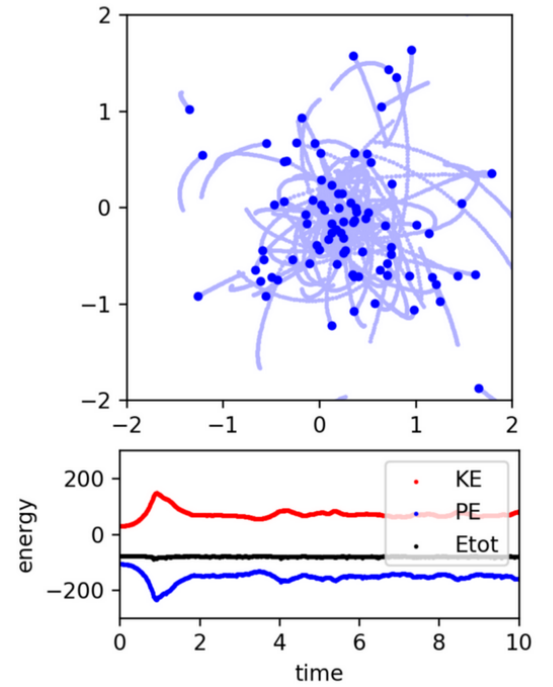
We use the gravitational potential to illustrate the basic form of computation in an all-pairs N-body simulation. In the following computation, we use bold font to signify vectors (typically in 3D). Gravitational attraction to body j is given by the following:

$$\mathbf{f}_{ij} = G \frac{m_i m_j}{\|\mathbf{r}_{ij}\|^2} \cdot \frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|},$$

from body i to body j ; and G is the gravitational constant. The left factor, the magnitude of the force, is proportional to the product of the masses and diminishes with the square of the distance between bodies i and j . The right factor is the direction of the force, a unit vector from body i in the direction of body j (because gravitation is an attractive force)

Literature Survey

They created a simulation of a dynamical system of particles interacting with each other gravitationally. Each particle feels the gravitational attraction of all the other particles according to Newton's law of universal gravitation (the famous 'inverse square-law'). The softening parameter in the code is a small number added to avoid numerical issues when 2 particles are close to each other, in which case the acceleration from the 'inverse square-law' goes to infinity. The evolution is performed in the code using a For-loop and our function for the acceleration from earlier: The only thing left to carry out a simulation is to specify the initial positions and velocities of the particles at time $t=0$. They initialize a random Gaussian distribution of positions and velocities for $N=100$ particles, but feel free to construct your own. Their code computes these quantities and keeps track of the total energy to make sure it is being approximately conserved by the numerical method. Running the code allows you to visualize the simulation in real time and will yield the figure:



Methodology used

Grouping close bodies and treating them as one body is key to accelerating the brute force n-body algorithm. We can approximate the gravitational effects of the group by using its centre of mass if it is far enough away.

The average placement of a body inside a group of bodies, weighted by mass, is the group's centre of mass. Formally, the total mass and centre of mass of two bodies with locations (x_1, y_1) and (x_2, y_2) and masses (m_1, m_2) are given by:

- $m = m1 + m2$
- $x = (x1*m1 + x2*m2) / m$
- $y = (y1*m1 + y2*m2) / m$
-

A creative method for assembling bodies that are close enough together is the Barnes-Hut algorithm. By keeping them in a quad-tree, it separates the set of bodies into groups in a recursive manner. Similar to a binary tree, a quad-tree differs in that each node has four offspring (some of which may be empty). A section of the two-dimensional space is represented by each node.

The four offspring of the top-most node, which stands for the entire space, correspond to the four spatial quadrants. The space is recursively divided into quadrants as depicted in the diagram until each subdivision has 0 or 1 body (some regions do not have bodies in all of their quadrants). A few internal nodes therefore have less than four non-empty children.

The screenshot shows a C++ IDE with a file explorer on the left. The main editor displays a C++ file named 'createVideo.cpp' with code for calculating particle interactions and rendering. The output console at the bottom shows the execution of 'createVideo.bash', displaying system information and the output of the 'ffprobe' command, which shows video details like resolution (1624x1624) and frame rate (45 tbr).

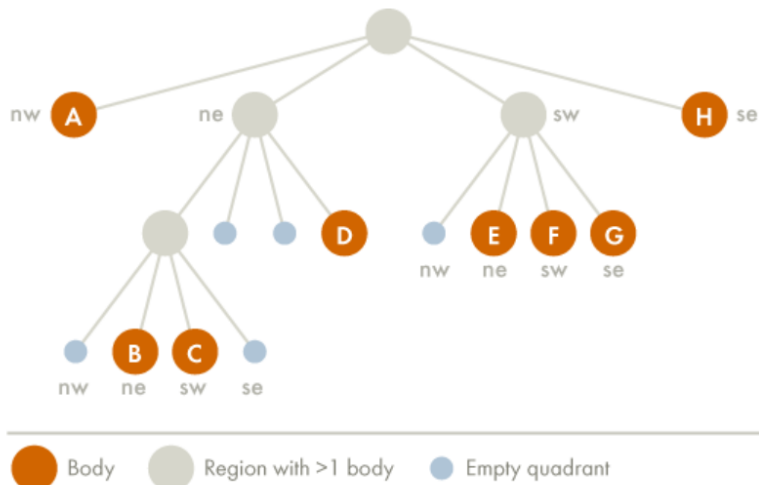
BRANES HUTT

ALGORITHM TO CONSTRUCT THE TREE

1. If node x does not contain a body, put the new body b here.
2. If node x is an internal node, update the center-of-mass and total mass of x . Recursively insert the body b in the appropriate quadrant.
3. If node x is an external node, say containing a body named c , then there are two bodies b and c in the same region. Subdivide the region further by creating four children. Then, recursively insert both b and c into the appropriate quadrant(s). Since b and c may still end up in the same quadrant, there may be several subdivisions during a single insertion. Finally, update the center-of-mass and total mass of x .

ALGORITHM TO CALCULATE FORCE ON TO THE BODY

1. If the current node is an external node (and it is not body b), calculate the force exerted by the current node on b , and add this amount to b 's net force.
2. Otherwise, calculate the ratio s/d . If $s/d < \theta$, treat this internal node as a single body, and calculate the force it exerts on body b , and add this amount to b 's net force.
3. Otherwise, run the procedure recursively on each of the current node's children.



A. CPU Serial

The gravitational N-Body problem relies on two of Newton's Laws– Newton's Law of Gravitation, as in (1); two bodies, B_i and B_j of mass m_i and m_j are attracted to each other by a force F_{ij} , which is inversely proportional to the square of the distance between them, r_{ij} ,

Algorithm 4: Sequential Barnes-Hut Algorithm

```

1: Function build_tree() is
2:   Reset Tree
3:   foreach  $i$ : particle do
4:      $root\_node \rightarrow insert\_to\_node(i)$ 

5: Function insert_to_node(new_particle) is
6:   if  $num\_particles > 1$  then
7:      $quad = get\_quadrant(new\_particle)$ 
8:     if subnode(quad) does not exist then
9:        $create\_subnode(quad)$ 
10:     $subnode(quad) \rightarrow insert\_to\_node(new\_particle)$ 
11:   else if  $num\_particles == 1$  then
12:      $quad = get\_quadrant(new\_particle)$ 
13:     if subnode(quad) does not exist then
14:        $create\_subnode(quad)$ 
15:      $subnode(quad) \rightarrow insert\_to\_node(existing\_particle)$ 

16:      $quad = get\_quadrant(new\_particle)$ 
17:     if subnode(quad)  $\neq NULL$  then
18:        $create\_subnode(quad)$ 
19:      $subnode(quad) \rightarrow insert\_to\_node(new\_particle)$ 
20:   else
21:      $existing\_particle \leftarrow new\_particle$ 
22:    $num\_particles++$ 

23: Function compute_mass_distribution() is
24:   if  $new\_particles == 1$  then
25:      $center\_of\_mass = particle.position$ 
26:      $mass = particle.mass$ 
27:   else
28:     forall child quadrants with particles do
29:        $quadrant.compute\_mass\_distribution$ 
30:        $mass += quadrant.mass$ 
31:        $center\_of\_mass = quadrant.mass * quadrant.center\_of\_mass$ 
32:      $center\_of\_mass /= mass$ 

33: Function calculate_force(target) is
34:   Initialize  $force \leftarrow 0$ 
35:   if  $num\_particles == 1$  then
36:      $force = gravitational\_force(target, node)$ 
37:   else
38:     if  $1/D < \theta$  then
39:        $force = gravitational\_force(target, node)$ 
40:     else
41:       forall  $node : child\ nodes$  do
42:          $force += node.calculate\_force(node)$ 

43: Function compute_force() is
44:   forall particles do
45:      $force = root\_node.calculate\_force(particle)$ 

```

The screenshot shows a terminal window titled 'cpu_serial.ipynb'. It displays the command 'mnt > E > cpu_serial.ipynb > from' followed by a table of execution times:

Category	Time
real	208m21.194s
user	204m28.857s
sys	1m31.900s

$$\vec{F}_{ij} = \frac{Gm_i m_j}{r_{ij}^2} \hat{r}_{ij}$$

$$\vec{a}_i = \frac{\vec{F}_i}{m_i}$$

Algorithm 1: Sequential All-Pairs Algorithm

```

1: Function calculate_force() is
2:   foreach  $i$ : body do
3:      $find\_force(i, particles)$ 

4: Function find_force(i: body, particles) is
5:   foreach  $j$  in particles do
6:     if  $j \neq i$  then
7:        $d\_sq = distance(i, j)$ 
8:        $ans[i].x += d\_x * mass(i) / d\_sq^3$ 
9:        $ans[i].y += d\_y * mass(i) / d\_sq^3$ 

```

B. CPU OMP Barnes-Hut Nlog (N)

In computing the center of mass of the nodes, even though the presence of data level dependencies restricts parallelizing, some level of parallelism can still be introduced as the computation done for each quad is independent of the other. So, all these computations can occur in parallel; this speeds up the process significantly. So, in the Algorithm described below only the center of mass computation has been parallelized leaving the force computation as it is. The graphs plotted have been shown in the sections that follow:

Algorithm 5: Parallel Barnes-Hut Algorithm (OpenMP)– Force Computation is Parallelized

```

1: Function compute_force() is
2:   #pragma omp parallel for
3:   forall particles do
4:     force = root_node.calculate_force(particle)

5: Function calculate_force(target_body) is
6:   force = 0
7:   if num_particles == 1 then
8:     force = gravitational_force(target_body, node)
9:   else
10:    if l/D <  $\theta$  then
11:      force = gravitational_force(target_body, node)
12:    else
13:      #pragma omp parallel for
14:      forall node : child nodes do
15:        #pragma omp critical
16:        force += node.calculate_force(node)

```

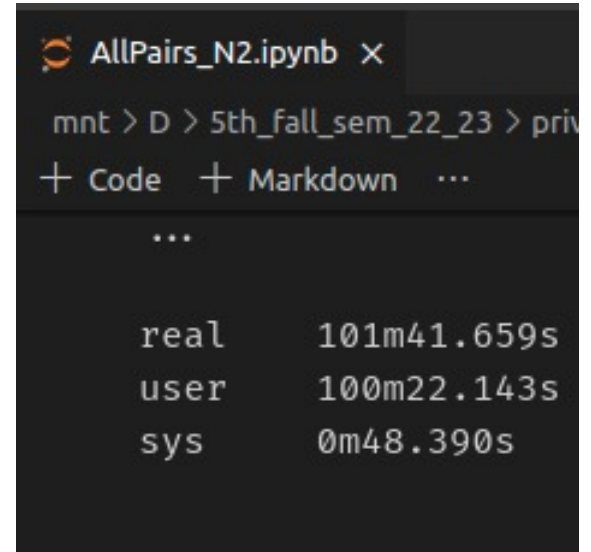
The screenshot shows a terminal window titled 'cpu_parallel_nlogn.ipynb'. The prompt is 'mnt > D > 5th_fall_sem_22_23 > priv'. Below the prompt, there are three lines of output: 'real 184m50.684s', 'user 312m29.057s', and 'sys 1m48.974s'. The window also shows a 'Done' status and a 't' character.

C. GPU CUDA All-Pairs N^2

We may think of the all-pairs algorithm as calculating each entry f_{ij} in an $N \times N$ grid of all pairwise forces.¹ Then the total force F_i (or acceleration a_i) on body i is obtained from the sum of all entries in row i . Each entry can be computed independently, so there is $O(N^2)$ available parallelism. However, this approach requires $O(N^2)$ memory and would be substantially limited by memory bandwidth.

Instead, we serialize some of the computations to achieve the data reuse needed to reach peak performance of the arithmetic units and to reduce the memory bandwidth required. Consequently, we introduce the notion of a computational tile, a square region of the grid of pairwise forces consisting of p rows and p columns. Only $2p$ body descriptions are required to evaluate all p^2 interactions in the tile (p of which can be reused later).

These body descriptions can be stored in shared memory or in registers. The total effect of the interactions in the tile on the p bodies is captured as an update to p acceleration vectors. To achieve optimal reuse of data, we arrange the computation of a tile so that the interactions in each row are evaluated in sequential order, updating the acceleration vector, while the separate rows are evaluated in parallel. In Figure 31-2, the diagram on the left shows the evaluation strategy, and the diagram on the right shows the inputs and outputs for a tile computation. In the remainder of this section, we follow a bottom-up presentation of the full computation, packing the available parallelism and utilizing the appropriate local memory at each level.



The screenshot shows a Jupyter Notebook terminal window titled "AllPairs_N2.ipynb". The terminal output displays the following statistics:

```

...
real    101m41.659s
user    100m22.143s
sys      0m48.390s

```

Algorithm 3: Parallel All-Pairs Algorithm (CUDA)

```

1: Function calculate_force() is
2:   foreach  $i$ : body do
3:     find_force <<< BLOCKS,
       THREADS_PER_BLOCK >>> (index,
       particles, ans, size)

4: Function find_force(index, particles, ans, size) is
5:    $j = \text{particles}[\text{treadIdx.x} + \text{blockIdx.x} * \text{blockDim.x}]$ 
6:   if  $j \neq i$  then
7:      $d\_sq = \text{distance}(i, j)$ 
8:      $\text{ans}[i].x += d\_x * \text{mass}(i) / d\_sq^3$ 
9:      $\text{ans}[i].y += d\_y * \text{mass}(i) / d\_sq^3$ 

```



Figure 2 - A schematic figure of computational diagram

D. GPU CUDA Barnes-Hut Nlog(N)

In computing the center of mass of the nodes, with the CUDA programming using google colab, as the computation done for each quad is independent of the other it took 17 minute 17 seconds. So, all these computations can occur in parallel; this speeds up the process significantly. So, in the graph described below only the center of mass computation has been parallelized leaving the force computation as it is. The graphs plotted have been shown in the sections that follow:

The force calculation is performed for each body. It recursively finds nodes in the tree which are considered to be far enough away to perform an interaction with. The calculation to decide whether a node is far enough away is called the opening condition. It is important because it decides how many bodies can be grouped together as a pseudo-body;

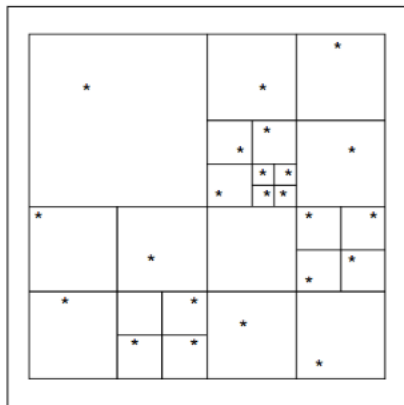


Fig. 1. Barnes-Hut tree structure

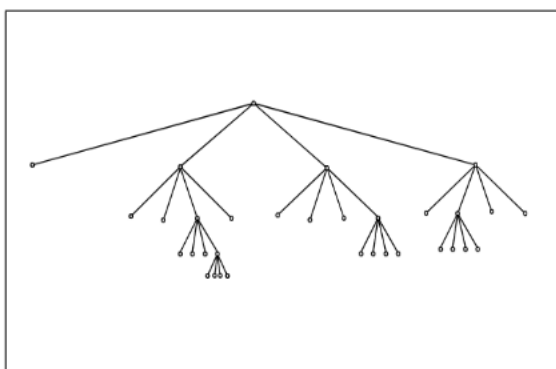


Fig. 2. Barnes-Hut domain decomposition

```
DynamicPar_NlogN.ipynb X
mnt > D > 5th_fall_sem_22_23 > priv
+ Code + Markdown ...

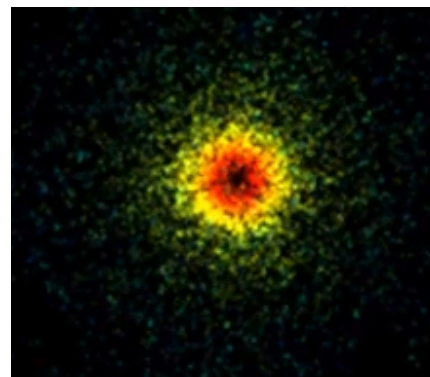
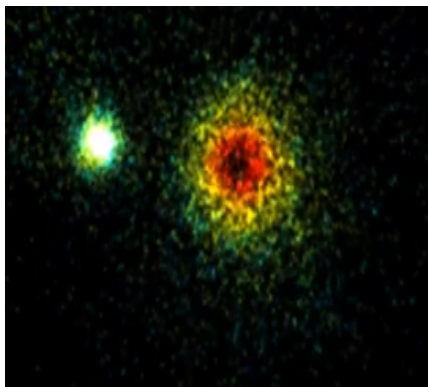
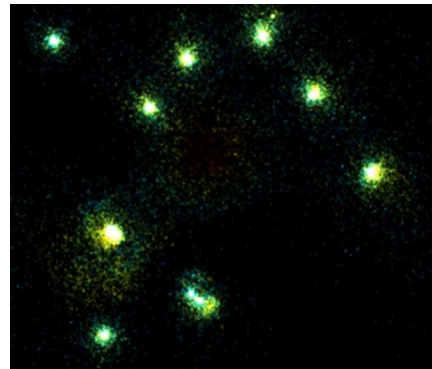
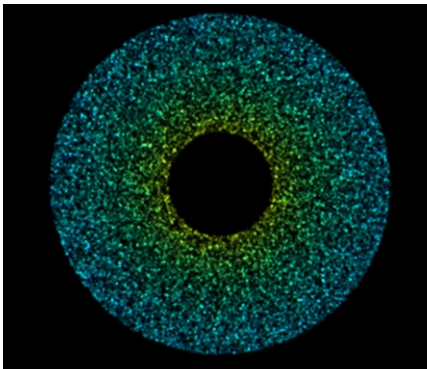
...

real    17m17.755s
user    15m24.098s
sys     1m2.648s
```

WORK DONE AND RESULT ANALYSIS

The sequential All-Pairs algorithm is implemented in C++. The parallelization of the algorithm is performed using OpenMP and CUDA. OpenMP is a usage of multi- threading [11]; an expert string forks a predetermined number of slave strings and the framework separates an errand among them. The strings then run simultaneously, with the runtime environment assigning strings to distinctive processors. The segment of code that is intended to keep running in parallel is stamped likewise, with a preprocessor order that will bring about the strings to shape before the segment is executed. Each string has an id appended to it which can be acquired utilizing a method `omp_get_thread_num()`. After the execution of the parallelized code, the strings join over into the master string, which proceeds with forward to the end of the system.

CUDA is an extension of the C that allows the program- mer to take advantage of the massive parallel computing power of an Nvidia graphics card in order to do general purpose computation [12]. In order to run efficiently on a GPU, you need to have many hundreds of threads. Generally, the more threads you have, the better. If you can break the problem down into at least a thousand threads, then CUDA probably is the best solution. When something extremely computationally intense is needed, the problem can simply call the CUDA kernel function written by the user. GPUs use massive parallel interfaces in order to connect with it's memory; is approximately 10 times faster than a typical CPU to memory interface



SN	Performance Evaluation						
		Serial		OpenMP		CUDA	
	<i>Time used in algorithms</i>	<i>N² All Pair</i>	<i>NLog(N) Barnes</i>	<i>N² All Pair</i>	<i>NLog(N) Barnesz</i>	<i>N² All Pair</i>	<i>Nlog(N) Barnes</i>
1	Total Time	441.80	208.04	301.08	184.50	101.41	17.17

TABLE 8 - COMPARATIVE PERFORMANCE OF ALGORITHM

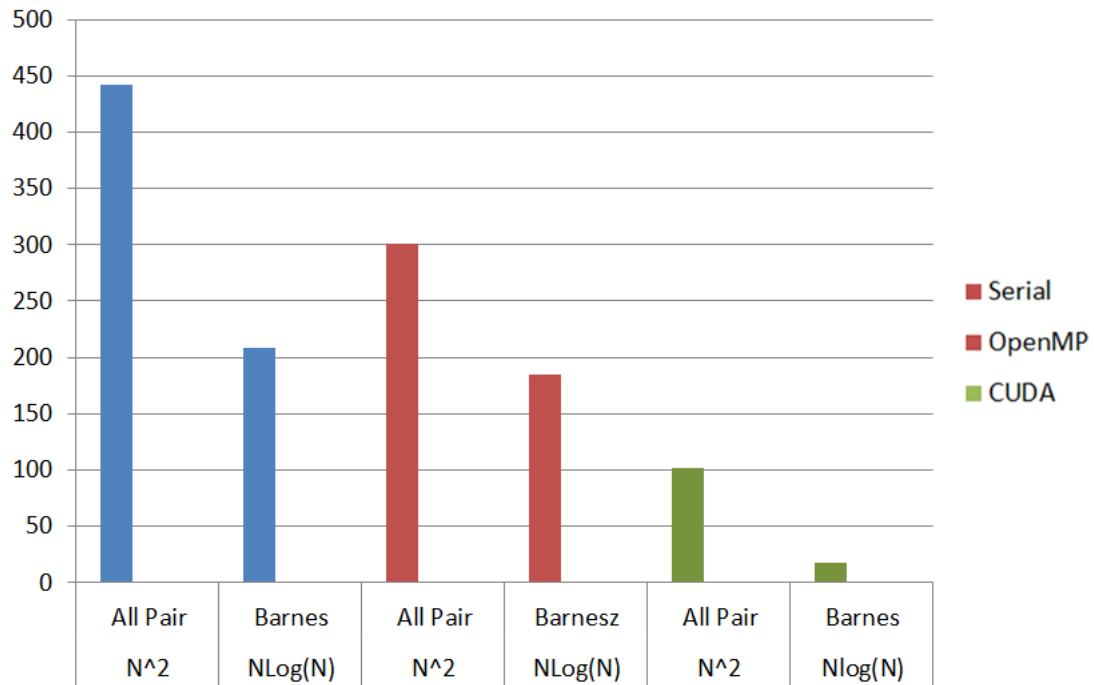


FIGURE 7 - COMPARATIVE EXECUTION TIME OF ALGORITHM

Google Colab used CPU and GPU Specification –

CPU Model	Frequency	No. Cores	Family	Ram	Disk Space
Intel(R) Xeon(R)	2.30GHz	2	Haswell	12GB	25GB

GPU	Memory	Clock	Performance	Mixed Precision	Release Year	No. CPU Cores	RAM	Disk Space
Nvidia K80 / T4	12GB / 16GB	0.82GHz / 1.59GHz	4.1 TFLOPS / 8.1 TFLOPS	No / Yes	2014 / 2018	2	12GB	358GB

CONCLUSION

As part of our project, we analyzed two algorithms to solve the classical N-Body problem– the naive All-Pairs Algorithm and quad-tree based Barnes-Hut Algorithm in Serial, OpenMP and CUDA. Compared to the sequential execution we noticed a decrease in execution time in parallelization of sequential algorithm. The performance of these algorithms can be further bettered by running the algorithms on a processor with a higher multiprocessing support. The parallel direct method scaled linearly with respect to the number of processes. The communication overhead for the parallel direct method is negligible as the number of stars is so small, but as more processes are added the algorithm becomes plausible for larger numbers of N , but with increasing N will come increasing communication overheads. Due to limitations with time this project only implemented a simple parallel version of the Barnes- Hut algorithm that contains no load balancing. The computation time of the Parallel Barnes-Hut scaled almost linearly, but with an increasing number of processes came an increasing communication overhead which soon out- weighed the benefit seen due to the increase in computation time. The Barnes-Hut algorithm can offer substantial increases in running time depending on the choice of θ . This shows how well the naive method is improved by parallel computing.

References

- [1] I. A. Mir and S. N. Dhage "Diabetes Disease Prediction Using Machine Learning on Big Data of Healthcare" Proceedings - 2018 4th International Conference on Computing Communication Control and Automation ICCUBE6 2018 Jul. 2018.
- [2] I. G. Angus and W. T. Thompkins Data storage concurrency and portability: An object oriented approach to fluid mechanics.
- [3] A. W. Appel "An efficient program for many-body simulation" SIAM J. Sci. Stat. Comput. vol. 6 1985.
- [4] J. Barnes and P. Hut "A hierarchical $O(N \log N)$ force-calculation algorithm" Nature vol. 324 pp. 446-449 1986.
- [5] S. Bhatt M. Chen C-Y Lin and P. Liu Abstractions for parallel N-body simulations 1992.
- [6] M. Chen and J. Cowie "Prototyping Fortran-90 Compilers for Massively Parallel Machines" ACM SIGPLAN'92 PLDI Conference 1992.
- [7] A. Chien and W. Dally "Experience with concurrent aggregates (CA): Implementation and programming" 5th DMCC 1990.
- [8] https://kazemnejad.com/blog/how_to_do_deep_learning_research_with_absolutely_no_gpus_part_2/
- [9] <https://www.nvidia.com/en-in/data-center/tesla-t4/>