Analyzing the network traffic in a local area network using the Cisco Packet tracer

Aadhaar Koul, Arjun Charak , Anil Kumar , Shobit Kitchloo and Sidharth Bhawani
Department of Computer science and Technology
Model institute of Engineering and Technology ,
Kot Bhalwal , Jammu , Jammu and Kashmir , India
{2020a1r040, 2020a1r057, pratyushprakash47, suraj1997pisces}@gmail.com

vishalika Sharma
Department of Computer Science and Technology
National Institute of Technology Karnataka
Surathkal, Mangaluru 575025, Karnataka, India
geethav@nitk.edu.in

***Abstract*— [1][2]In our new era PCs become our part of life for every personal and professional requirement. Majority of organizations depend on the finest possible working of their systems for correspondences, organization, mechanization, online business solutions, and so on. LAN is the best fundamental and significant PC system claimed by discrete organizations and might be utilized for interconnection of wide region systems. A LAN provides effective cost sharing of fast processing information handling gear, for example, mass stockpiling media, centralized server PCs or tiny computers and various types of printers. Asset sharing is generally similar as significant where a Local Area Network (LAN) serves as the entrance path for an Internet. In view of this, framework supervisor's requirement professional tools to help them with the motivation of improvement of QoS and maintenance of LANs. So in our project, a LAN system is structured utilizing Cisco Packet Tracer. This project explains just how the apparatus can be used to build up a reenactment model of the Local Area Network (LAN) for College of Engineering which contains a department like Bio Technology (BT), Civil, Mechanical, ECE and EEE or any. The examination gives a knowledge into different ideas such as IP address setup, topology plan and how to send data as packets in a solitary network and for the usage of Virtual Local Area Networks to isolate the heavy traffic produced by various systems.**

***Index Terms*— N-Body, All-Pairs, Barnes-Hut, Parallelization, OpenMP, CUDA**

I. INTRODUCTION

[4]The requirement for PC systems administration was an effect of the requirement to use PCs for exchanging information in an association in form of messages or packets, exchanging documents and data bases, etc. Regardless of whether the organization is situated in one structure or spread over a huge grounds, the requirement for systems administration the computers cannot be over underscored. As the name assumes, a Local Area Network (LAN) connects PCs in a limited physical territory . It gives high-data transfer capacity correspondence over cheap transmission media .The corporate LAN has developed from an easy basis business segment to a profoundly vibrant, noticeable core asset that activities depend on to help everyday tasks to their market accomplishment. E-Governance is a system of open segment order and is a significant advance in the adjustment of metropolitan organization, with E-Governance joins the utilization of ICT's by government's association. The anticipated calculation utilizes insight of calculation for security of substance in e-governance executing a standard based methodology from computational Knowledge and client's present purpose of area data. On a work area PC, a recreation model had been actualized and assessment utilizing meandering client's continuous position-based data exhibits that proposed system can capably preserve wandering client position secrecy while giving better execution, ensured position privacy, and better nature of administration in e-Governance.

II. FRAMEWORK

A. Background

Cisco Packet Tracer is designed to be used as multitasking, that's been won't to organize and examine varied network exercises like application of dissimilar topologies, development of apt servers, subnetting and study of different network setups, configuration and different troubleshooting defined commands.

To initialize communication among two networking devices i.e., user networking devices and to organize a network, we intend to demand to pick applicable networking devices like switches , routers and interconnecting devices and build physical change of integrity by connecting cables, quick local area network seaports from the module list of packet tracer. Internet working devices square measure costly and thus it's well to perform 1st on the packet tracer to recognize the conception, performance of the designed network.

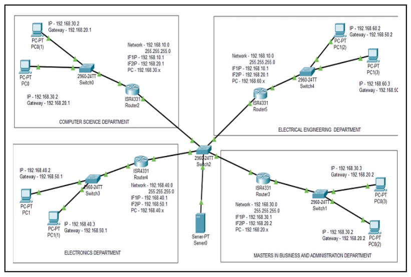

Fig. 1. Barnes-Hut tree structure

B. *Framework Continued*

Framework The graph of Fig. 1 is the finished graph of the LAN and at the center it connected to switch, switch and the servers framing the Network Operating Center and every one of the different departments in College are only a simple expansion of the system at the center. The allotted IP address picked to the inside system is 192.168.0.0 and it has been sub netted to acquire IP address obstructs that are allocated to various divisions and segments of this prescribed LAN.

III. LAN Simulation Model

A. *Sequential All-Pairs Algorithm*

[4]We require at least 252 hosts for every subnet the quantity of unmasked bits in the subnet mask is 8. Which infers that the amount of masked bits are 8.

Create and assign IP/subnet mask for VLANs:

In this VLAN, we are assigning the below gate ways to all the VLANs with ip address and subnet mask (255.255.255.0). Which is configured in the main switch of VLAN.

Algorithm 1 given below is the serial All-Pairs algorithm used to compute forces on all bodies as explained above.

Algorithm 1: Parallel All-Pairs Algorithm (OpenMP)

```
Function calculate_force() is
    #pragma omp parallel for
    foreach i: body do
        find_force(i, particles)

Function find_force(i: body, particles) is
    #pragma omp parallel for reduction (+ : ans[i].x, ans[i].y)
    foreach j in particles do
        if $j \neq i$ then
            d_sq = distance(i, j)
            ans[i].x += d_x * mass(i) / d_sq^3
            ans[i].y += d_y * mass(i) / d_sq^3
```

Algorithm 2: Parallel All-Pairs Algorithm (CUDA)

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

Function find_force(index, particles, ans, size) is
    j = particles[treadIdx.x + blockIdx.x * blockDim.x]
    if $j \neq i$ then
        d_sq = distance(i, j)
        ans[i].x += d_x * mass(i) / d_sq^3
        ans[i].y += d_y * mass(i) / d_sq^3
```

The brute force algorithm is very easily parallelized as it is known in advance exactly how much work needs to be done. The work can be partitioned easily among processes with a block partitioning strategy. The number of bodies is known and to update each body takes the same amount of calculation. Assigning each process to calculate a block of bodies each $\frac{number of planets}{number of processors}$ in size will mean each process performs the same amount of calculations. Therefore partitioning the workload effectively; depicted by Algorithm 2.

B. *Parallelization of All-Pairs Algorithm (CUDA)*

Algorithm 3 given below is the parallel All-Pairs algorithm used to compute forces on all bodies using Nvidia CUDA.

C. *Sequential Barnes-Hut Algorithm*

The Barnes-Hut algorithm [8] approximates a solution to the gravitational N-Body problem by clustering groups of distant bodies together as a single pseudo-body. Each has an overall mass and center of mass based on the individual bodies it contains. It achieves this by creating a tree structure where each node has four children and each node has a center of mass and total mass based on that of its children. When created, this tree describes the whole system where each internal node represents a pseudo-body. Each star then uses the tree to work out the forces it experiences. The algorithm to realize the spatial system into a tree structure is achieved as follows:

- Divide the whole domain into four square regions of equal size.
- If any of these regions contains more than one body, recursively divide that region into four more squares. Continue until each square contains maximum of one body.
- Once the tree is created perform a recursive walk to calculate the center of mass, $\vec{c}$, as in (1), where m_i is the mass of a node´s i^{th} child and $\vec{c}_i$ is the center of mass of a node´s i^{th} child. .

$$\vec{c} = \frac{\sum_1^i \vec{c}_i m_i}{\sum_1^i m_i} \tag{1}$$

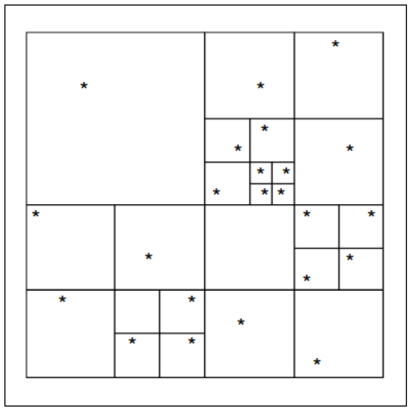
Fig. 2. Barnes-Hut tree structure

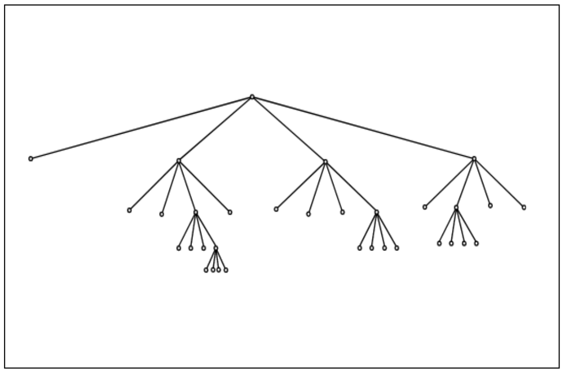
Fig. 3. Barnes-Hut domain decomposition

This creates a tree where the root node (Fig. 1) contains the whole system. Each node has four children (Fig. 2) and the leaves of the tree are the individual bodies [9]. The construction of the tree can be done with $O(N\log(N))$ runtime. Each body now uses this tree to calculate the acceleration it experiences due to every other body. 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; the more bodies which are grouped together, the less accurate the calculations will be. The opening condition is a simple relationship, $\frac{l}{D} < \theta$, where θ is the fixed accuracy parameter which is positive, l is the width of the current internal node and D is the distance of the body from the center of of mass of the current node to the body the force is being calculated for. The sequential algorithm for the same is given as Algorithm 4.

D. Parallel Barnes-Hut Algorithm using OpenMP– Force Computation is Parallelized (Method-1)

Two major issues involved in the analysis of a parallel algorithm include *Decomposition*– divide the problem amongst available processes and *Communication*– need to communicate between processes to assure they have data they need. Decomposition is associated with load

Algorithm 3: Sequential Barnes-Hut Algorithm

```
1:  Function build_tree() is
2:      Reset Tree
3:      foreach i: particle do
4:          root_node→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)→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)→insert_to_node(existing_particle)

16:         quad = get_quadrant(new_particle)
17:         if subnode(quad) ≠ NULL then
18:             create subnode(quad)
19:         subnode(quad)→insert_to_node(new_particle)

20:     else
21:         existing_particle ← 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 ← 0
35:     if num_particles == 1 then
36:         force = gravitational_force(target, node)
37:     else
38:         if l/D < θ 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)
```

balancing while Communication bottleneck is a major issue; need for minimization of communication volume. Barnes-Hut algorithm construction is as follows:

- Build the quad-tree.
- Calculate the center of mass for all cells.
- Traverse the quad-tree.
- Calculate the force on the nodes.

Building the quad-tree needs synchronization. Since the computation of the center of mass depends on the center of masses of corresponding sub-cells, data dependencies are introduced– can be parallelized per level. To compute the force, we need other particles' center of mass, but we don't need to modify this information– can be parallelized. The value of θ plays a crucial role- a higher value of θ implies that fewer nodes are considered for the calculation of force and hence increasing the window for error. Algorithm 5 given below, depicts the parallelized version of Barnes-Hut algorithm in the way described above.

Algorithm 4: 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 < θ 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)
```

E. Parallel Barnes-Hut Algorithm using OpenMP– Mass Distribution is Parallelized (Method-2)

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-6 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.

Parallelization of the Barnes-Hut algorithm has many issues making it more complex than anything so far seen in this project. The main issue is the lack of prescience of the number of calculations done by each process. With the increase in the tree traversal depth, the number of force calculations increase and the exact depth is dependent on the position of the current body, which is random.

Algorithm 5: Parallel Barnes-Hut Algorithm (OpenMP)– Mass Distribution is Parallelized

```
1:  Function compute_mass_distribution() is
2:      if new_particles == 1 then
3:          center_of_mass = particle.position
4:          mass = particle.mass
5:      else
6:          #pragma omp parallel for
7:          forall child quadrants with particles do
8:              quadrant.compute_mass_distribution
9:              #pragma omp critical
10:             mass += quadrant.mass
11:             center_of_mass = quadrant.mass *
                  quadrant.center_of_mass
12:         center_of_mass /= mass
```

IV. Work Done and Results 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 multithreading [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 programmer 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.

This section focuses on running each algorithm in serial and in parallel. Testing Speedup, Cost and Efficiency (see equations (2), (3), (4)). All the tests for All-Pairs algorithm (openMP) are ran on nearly identical machines with the following specification:

- Model– Asus
- Processor– i5 7200U @ 4x 3.1GHz
- Memory– 8GB DDR3 at 1333MHz
- Network– 10/100/1000 Gigabit LAN Connection
- Operating System– Arch Linux

All the tests for All-Pairs algorithm (CUDA) are ran on nearly identical machines with the following specification:

- Nvidia Tesla Server
- Operating System– Ubuntu Linux

All the tests for Barnes-Hut algorithm (openMP, both methods) are ran on nearly identical machines with the following specification:

- Model– HP
- Processor– i5 6200U
- Memory– 8GB DDR3 @ 2.8GHz
- Network– 10/100/1000 Gigabit LAN Connection
- Operating System– Ubuntu Linux

$$Speedup(S) = \frac{TimeforSerialExecution}{TimeforParallelExecution} \tag{2}$$

$$Cost(C) = ParallelExecution \times NumberofProcessors \tag{3}$$

$$Efficiency(E) = \frac{Speedup}{TimeforParallelExecution} \tag{4}$$

A. Result and Analysis of Parallel All-Pairs algorithm in OpenMP and Nvidia CUDA

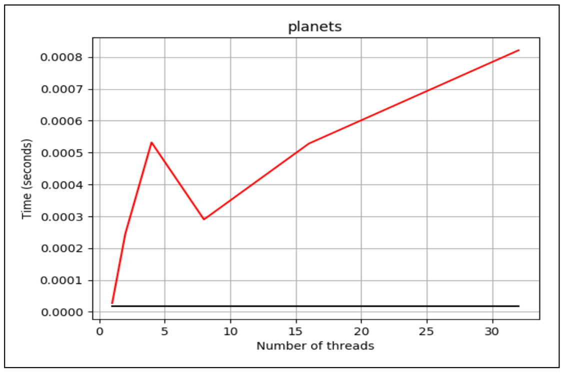

Fig. 4. Serial (black) vs. Parallel (red) execution for *Planets.txt* [4] for 5 bodies using OpenMP (Algorithm 2)

For inputs with a small number of planets we find sequential execution to be faster than parallelized OpenMP code (Fig. 3). This supports the fact that threads have a high cost of initialization, which outweighs the execution

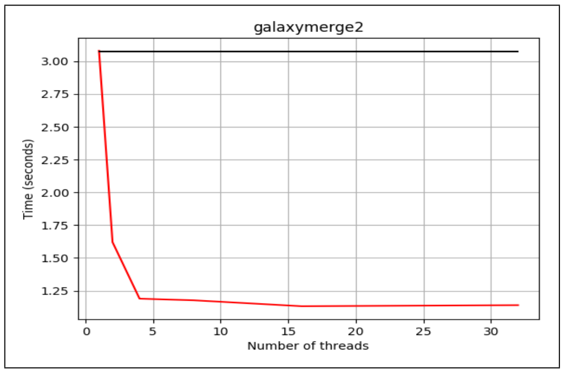

Fig. 5. Serial (black) vs. Parallel (red) execution for *galaxymerge2.txt* [4] for 4000 bodies using OpenMP (Algorithm 2)

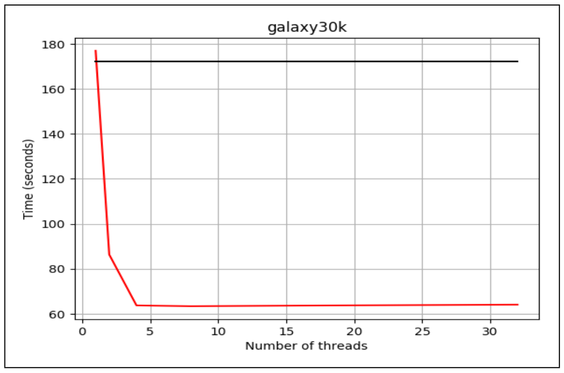

Fig. 6. Serial (black) vs. Parallel (red) execution for *galaxy30k.txt* [4] for 30002 bodies using OpenMP (Algorithm 2)

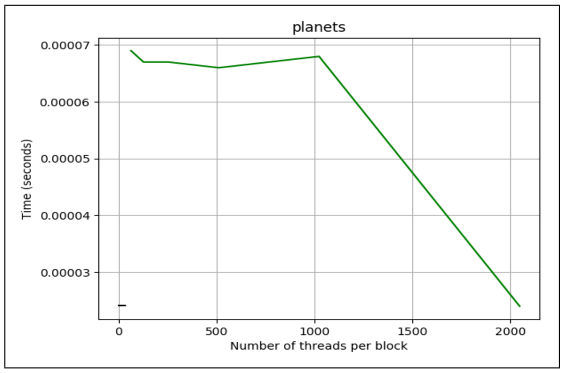

Fig. 7. Serial (black) vs. Parallel (green) execution for *planets.txt* [4] for 5 bodies using CUDA (Algorithm 3)

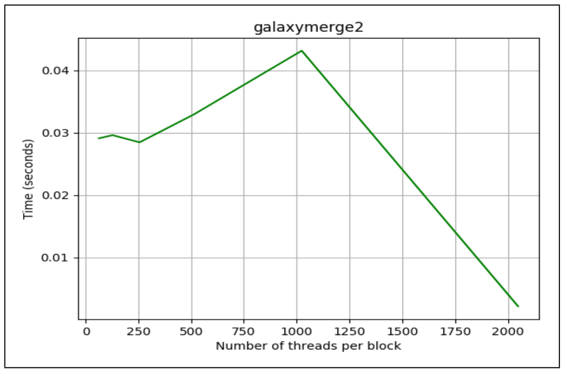

Fig. 8. Serial (black) vs. Parallel (green) execution for *galaxymege2.txt* [4] for 4000 bodies using CUDA (Algorithm 3)

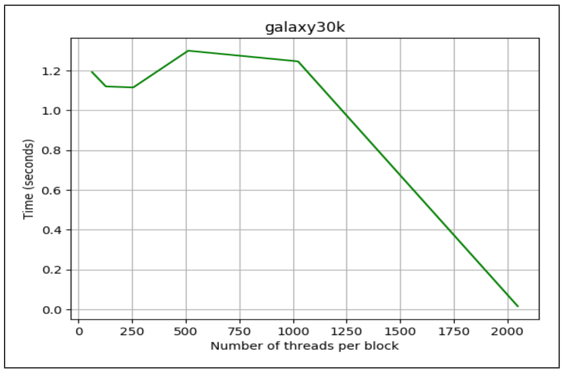

Fig. 9. Serial (black) vs. Parallel (green) execution for *galaxy30k.txt* [4] for 30002 bodies using CUDA (Algorithm 3)

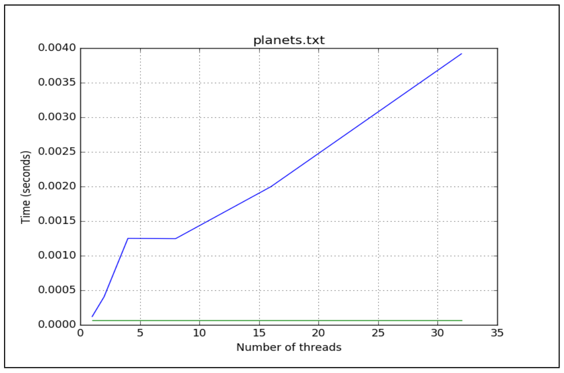

Fig. 10. Serial (green) vs. Parallel (blue) execution for *Planets.txt* [4] for 5 bodies using OpenMP (Algorithm 5)

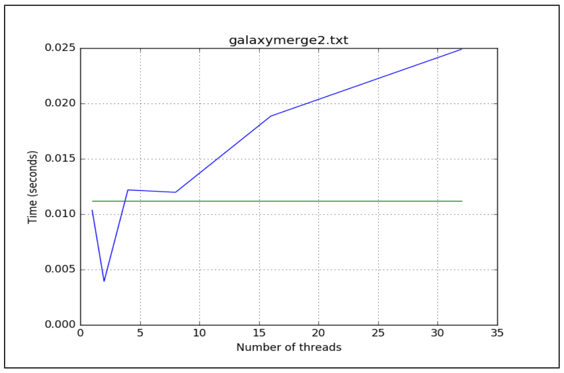

Fig. 11. Serial (green) vs. Parallel (blue) execution for *galaxymerge2.txt* [4] for 4000 bodies using OpenMP (Algorithm 5)

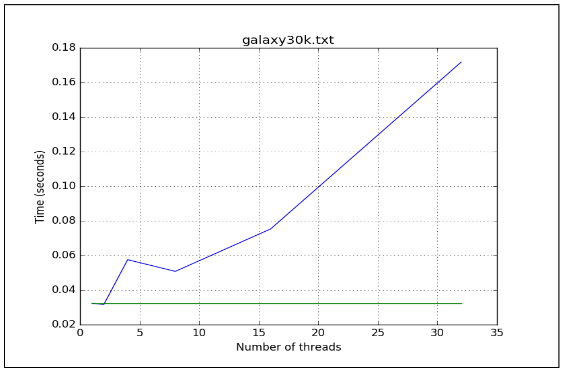

Fig. 12. Serial (green) vs. Parallel (blue) execution for *galaxy30k.txt* [4] for 30002 bodies using OpenMP (Algorithm 5)

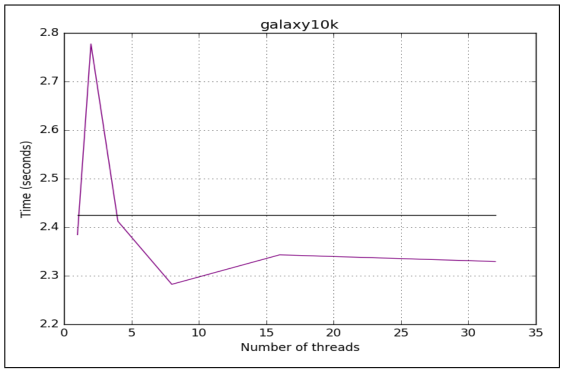

Fig. 13. Serial (black) vs. Parallel (purple) execution for *galaxy10k.txt* [4] for 10001 bodies using OpenMP (Algorithm 6)

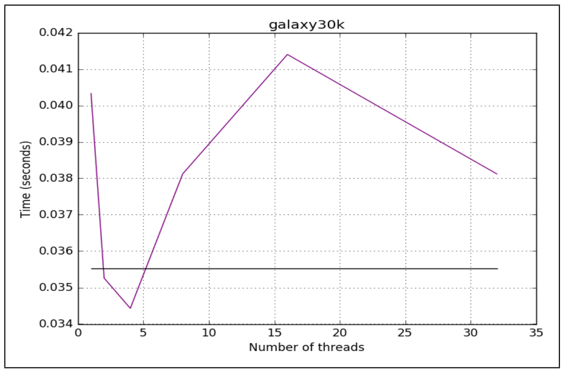

Fig. 14. Serial (black) vs. Parallel (purple) execution for *galaxy30k.txt* [4] for 30002 bodies using OpenMP (Algorithm 6)

time since it is smaller in comparison. When we increase the input size, the parallel code runs much faster than the sequential counterpart. Since the input is large, the time of execution is also larger than the tread spawn overheads. (Fig. 4). However we notice that increasing the threads beyond a certain value does not cause any further decrease. This is because the CPU on the test machine cannot support more than 4 threads. Therefore we see a flat curve after. For an input file with very large inputs the graph remains the same as the previous graph. However there is a larger speedup (Fig. 5).

From Figures 3, 4 and 5 we observe that there is a

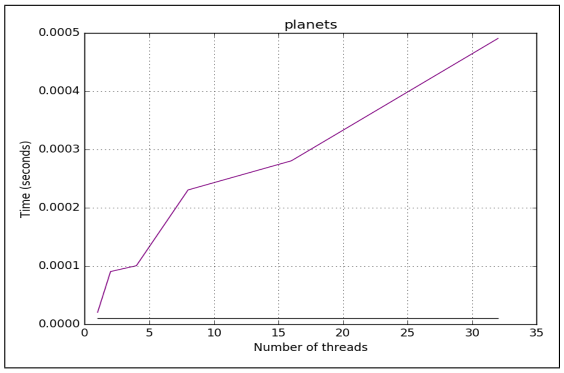

Fig. 15. Serial (black) vs. Parallel (purple) execution for *planets.txt* [4] for 5 bodies using OpenMP (Algorithm 6)

large speedup for input files with a large size and this speedup is limited to the number of threads on the test machine. For small inputs, sequential execution remains faster than parallelization with OpenMP. In the case of parallelization with CUDA, we see an exponential decrease in CUDA time. This follows from the fact that the GPU has an exponentially larger thread pool as compared to the CPU (Figures 6, 7, 8).

For a very small input the communication over PCI lanes is the bottleneck as execution time is negligible. Hence we see that sequential is comparable to the CUDA program. In the case of Fig. 6 we find parallel execution with CUDA an order of 100 times faster than sequential. As the number of threads per block is increased beyond a certain value the execution time decreases drastically. Fig. 8 shows CUDA execution times for an input file of very large size. The speedup of CUDA with 2048 threads per block over sequential is approximately 350. This means with heavy parallelization of the GPU, we can achieve the results much quicker than doing the same on the CPU. This confirms the fact that GPUs typically have more multiprocessing capabilities than the CPU.

B. Result and Analysis of Parallel Barnes-Hut Algorithm using OpenMP

In Fig.9, the sequential algorithm proved to be efficient as compared to OpenMP. One explanation for the same would include the communication overhead and thread overhead for such small dataset (5 bodies). In Fig. 10, it was observed that the OpenMP implementation performed better in the case of 2 and 4 threads but the overheads increased as the number of threads increased from 4. In the case of Fig. 11, the Barnes-Hut implementation in OpenMP algorithm performed better in the case of 2 threads but increased progressively before reducing once in 8 threads and increasing again.

In Fig. 12, the time reduced drastically for 8 threads and above. Although, an anomaly was observed for 4 threads. In case of Fig. 13, it was observed that for *galaxy30k* performed better than the sequential for 4 and 8 threads and then increased linearly with the number threads, again changing the trend from 16 threads onwards. As observed in Fig. 14 for *planets.txt* the overhead increases progressively. Hence, we can infer that the dataset *planets.txt* has data which does not go well with the tree building methodology.

The value of θ determines how deep the Barnes-Hut algorithm traverses the tree. The smaller the value the deeper it goes, increasing accuracy but at the cost of an increased number of calculations and therefore a slower running time. Experimentally it can be seen that as θ tends to 0 the number of calculations increases, as expected. As θ reaches 0.3 the number of calculations begin to rise quickly and at 0.1 the larger problems have to computer many more calculations. When θ is 0 the number of calculations is equal to the number of bodies in the set. This shows the algorithm has become just as complex as the All-Pairs method, with each body needing to calculate forces due to every other star.

V. CONCLUSIONS

In this paper, we analyzed two algorithms to solve the classical N-Body problem– the naive All-Pairs Algorithm and quad-tree based Barnes-Hut Algorithm in OpenMP and CUDA. Compared to the sequential execution we noticed a decrease in execution time till a certain level of parallelization, after which the time either remained the same or increased. The performance of these algorithms can be further bettered by running the algorithms on a processor with a higher multiprogramming 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 outweighed 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.

APPENDIX

The appendix shows the analysis of the Barnes-Hut algorithm implemented in Parallel using OpenMP (method-1). The Sequential and Parallel times have been shown in Table 1; for all the galactic datasets [4] with number of bodies ranging from 5 to 30002.

ACKNOWLEDGMENT

We would like to thank Dr. Geetha V for her valuable comments and suggestions to improve the quality of the paper. We are also grateful to Miss Archana for helping us review our work regularly. We would also like to thank the Department of Information Technology, NITK Surathkal for providing us with Tesla GPU for us to test our code.

REFERENCES

[1] En.wikipedia.org. (2018). N-body problem. [online]. Available: `https://en.wikipedia.org/wiki/N-body_problem`. [Accessed 7 Jan. 2018].

[2] Carugati, N. J. (2016). The Parallelization and Optimization of the N-Body Problem using OpenMP and OpenMPI.

[3] 15418.courses.cs.cmu.edu. (2018). The Barnes-Hut Algorithm : 15-418 Spring 2013. [Online]. Available: `http://15418.courses.cs.cmu.edu/spring2013/article/18`. [Accessed: 07- Jan- 2018].

[4] Cs.princeton.edu. (2018). COS 126 Programming Assignment: N-Body Simulation. [Online]. Available: `http://www.cs.princeton.edu/courses/archive/fall04/cos126/assignments/nbody.html`. [Accessed: 07- Jan- 2018].

TABLE I

PERFORMANCE OF SERIAL CODE VS. PARALLEL CODE ON GALACTIC DATASETS [4] OF BARNES-HUT ALGORITHM IN OPENMP (METHOD-1)

Dataset	**Number of Particles**	**Serial Time (seconds)**	**Parallel Time (seconds)**					
			Number of Threads					
			1	**2**	**4**	**8**	**16**	**32**
asteroids1000.txt	1000	0.023097	0.020348	0.021905	0.030464	0.063325	0.121116	0.221256
cluster2582.txt	2582	0.004927	0.005837	0.005042	0.011231	0.008328	0.011733	0.014243
collision1.txt	2000	0.004917	0.004829	0.004447	0.006030	0.005751	0.009468	0.012608
collision2.txt	2002	0.006227	0.006008	0.006098	0.006309	0.006821	0.009951	0.013182
galaxy1.txt	802	0.015414	0.015616	0.015217	0.020928	0.045315	0.072090	0.110689
galaxy2.txt	652	0.012274	0.012615	0.014664	0.023931	0.028826	0.040485	0.072064
galaxy3.txt	2001	0.091639	0.087738	0.094466	0.141264	0.264529	0.488200	0.975077
galaxy4.txt	502	0.012875	0.013325	0.010431	0.012065	0.027304	0.037397	0.051786
galaxy10k.txt	10001	2.325312	2.357691	2.422882	3.557520	6.697054	13.312913	27.061886
galaxy20k.txt	20001	13.663441	15.492622	16.259973	23.813991	45.588013	88.301931	160.741782
galaxy30k.txt	30002	0.032405	0.032411	0.031647	0.057545	0.050811	0.075314	0.171779
galaxyform2500.txt	2500	0.007052	0.005922	0.006162	0.006707	0.008641	0.011501	0.016563
galaxymerge1.txt	2000	0.004920	0.005160	0.004812	0.006784	0.006742	0.008701	0.018789
galaxymerge2.txt	4000	0.011205	0.010364	0.003930	0.012193	0.011976	0.018860	0.024891
galaxymerge3.txt	2901	0.009433	0.009095	0.009045	0.015460	0.011692	0.012852	0.019202
planets.txt	5	0.000070	0.000120	0.000406	0.001250	0.001246	0.001997	0.003918
saturnrings.txt	11987	0.024471	0.024749	0.020095	0.025863	0.032043	0.038763	0.064468
spiralgalaxy.txt	843	0.017879	0.017627	0.023605	0.024740	0.052584	0.091534	0.166260

[5] Damgov, V., Gotchev, D., Spedicato, E., & Del Popolo, A. (2002). N-body gravitational interactions: a general view and some heuristic problems. arXiv preprint astro-ph/0208373.

[6] Beltoforion.de. (2018). The Barnes-Hut Galaxy Simulator. [Online]. Available: `http://beltoforion.de/article.php?a=barnes-hut-galaxy-simulator`. [Accessed: 07- Jan- 2018].

[7] Wwwmpa.mpa-garching.mpg.de. (2018). Cosmological simulations with GADGET. [Online]. Available: `http://wwwmpa.`

mpa-garching.mpg.de/gadget/. [Accessed: 07- Jan- 2018].

[8] Barnes, J., & Hut, P. (1986). A hierarchical O (N log N) force-calculation algorithm. nature, 324(6096), 446.

[9] Burtscher, M., & Pingali, K. (2011). An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. GPU computing Gems Emerald edition, 75.

[10] C. Swinehart. Arborjs.org. (2018). The Barnes-Hut Algorithm. [Online]. Available: http://arborjs.org/docs/barnes-hut. [Accessed: 07- Jan- 2018].

[11] Computing.llnl.gov. (2018). OpenMP. [Online]. Available: https://computing.llnl.gov/tutorials/openMP/. [Accessed: 07- Jan- 2018].

[12] Docs.nvidia.com. (2018). Programming Guide :: CUDA Toolkit Documentation. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-general-purpose-parallel-computing-architecture. [Accessed: 07- Jan- 2018].