Parallel OpenMP and CUDA Implementations of the *N*-Body Problem

Tushaar Gangavarapu[0000−0002−0489−9573]⋆, Himadri Pal, Pratyush Prakash, Suraj Hegde, and Geetha V[0000−0002−1230−8729]

Department of Information Technology,
National Institute of Technology Karnataka, Surathkal, Mangaluru, India
{tushaargvsg45,himadripal37,pratyushprakash,suraj1997pisces}@gmail.com,
geethav@nitk.edu.in

Abstract. The N-body problem, in the field of astrophysics, predicts the movements of the planets and their gravitational interactions. This paper aims at developing efficient and high-performance implementations of two versions of the N-body problem. Adaptive tree structures are widely used in N-body simulations. Building and storing the tree and the need for work-load balancing pose significant challenges in high-performance implementations. Our implementations use various cores in CPU and GPU via efficient work-load balancing with data and task parallelization. The contributions include OpenMP and Nvidia CUDA implementations to parallelize force computation and mass distribution, and achieve competitive performance in terms of speedup and running time which is empirically justified and graphed. This research not only aids as an alternative to complex simulations but also to other big data applications requiring work-load distribution and computationally expensive procedures.

Keywords: All-Pairs Algorithm · Barnes-Hut Algorithm · CUDA · N-Body Simulations · OpenMP · Parallel Processing · Performance

1 Introduction

The N-body problem in astrophysics is the problem of predicting the individual motions of a group of celestial bodies, interacting gravitationally [3]. Scientific and engineering applications of such simulations to anticipate certain behaviors include biology, molecular and fluid dynamics, semiconductor device simulation, feature engineering, and others [18, 14, 15]. The gravitational N-body problem [23] aims at computing the states of N bodies at a time T, given their initial states (velocities and positions). The naive implementation of the N-body problem has a complexity of $O(N^2)$ which is inefficient in terms of both power consumption and performance, leaving much room to improve the effectiveness of the execution of these simulations using data and task parallelism, aided with

⋆ Corresponding author.

utilities such as OpenMP (distribution among processors) [16, 9] and Nvidia CUDA (distribution among Graphical Processing Units (GPUs)) [2].

With Appel [7] and Barnes-Hut [8] algorithms, the N-body simulation is significantly faster, with the time complexity of $O(N)$ for Appel and $O(NlogN)$ for the Barnes-Hut algorithm. Even with such adaptive tree optimizations, significant improvement in the performance can be seen when implemented in parallel. In this paper, we review existing All-Pairs and Barnes-Hut algorithms to solve the N-body problem, and then propose our method of parallelization to achieve work-load balancing using data and task parallel approaches effectively. We then draw conclusions from the presented results to assess the potential of parallelization in terms of running time and speedup. The key contributions of this paper are mainly three-fold:

- Design of OpenMP and CUDA implementations of the All-Pairs algorithm, to parallelize force computation.
- Design of OpenMP implementation of the Barnes-Hut algorithm, to parallelize both force computation and mass distribution.
- We present a detailed evaluation of the performance of the parallel algorithms in terms of speedup and running time on galactic datasets [1] with bodies ranging from 5 to $30,002$.

The rest of this paper is structured as follows: Section 2 gives a detailed overview of the All-Pairs and Barnes-Hut algorithms to solve the N-body problem. Section 3 reviews relevant existing works in the field of parallel N-body simulations. Section 4 explains our proposed approaches to parallelize the All-Pairs and Barnes-Hut simulations and presents their implementations using OpenMP and CUDA. Section 5 presents the evaluation of the proposed approaches and Section 6 reviews the significant implications of such parallelization and concludes with future enhancements.

2 The Gravitational *N*-Body Problem

The gravitational N-body problem [23] is concerned with interactions between N bodies (stars or galaxies in astrophysics), where each body in a given system of bodies, affects every other body. The creation of galaxies, effects of black holes, and even the search for dark matter are associated with the N-body problem [17], thus making it one of the most widely experimented problem.

The Problem: Given the initial states (velocities and positions) of N bodies, compute the states of those bodies at time T using the instantaneous acceleration on every body at regular time steps.

In this section, we present two commonly used algorithms: 1) *All-Pairs*, which is best suited for a smaller number of bodies and 2) *Barnes-Hut*, which scales efficiently to a large number of bodies (e.g., molecular dynamics); to solve the N-body problem.

2.1 The All-Pairs Algorithm

The traditional All-Pairs algorithm is an exhaustive brute-force that computes instantaneous pair-wise acceleration between each body and every other body. Any two bodies (B_i, B_j) in the system of N bodies are attracted to each other with force ($\overrightarrow{F_{ij}}$) that is inversely proportional to the square of the distance (r_{ij}) between them (see Equation 1, G is the universal gravitational constant).

$$\overrightarrow{F_{ij}} = \frac{Gm_i m_j}{r_{ij}^2} \hat{r_{ij}} \tag{1}$$

Also, a body (B_i) of mass (m_i) experiences acceleration ($\overrightarrow{a_i}$) due to the net force acting on it ($F_i = \sum_{j \neq i} \overrightarrow{F_{ij}}$) from $N-1$ bodies (see Equation 2).

$$\overrightarrow{F_i} = \overrightarrow{a_i} m_i \tag{2}$$

From Equation 1 and Equation 2, the instantaneous pair-wise acceleration ($\overrightarrow{a_i}$) acting on a body (B_i) due to another body (B_j) can be given by Equation 3 (G is the universal gravitational constant).

$$\overrightarrow{a_i} = \frac{Gm_j}{r_{ij}^2} \hat{r_{ij}} \tag{3}$$

The All-Pairs method given by Algorithm 1 essentially computes pair-wise accelerations and updates the forces acting on all bodies, thus updating their state. Such an update can be programmatically achieved by using an in-place update of a double nested loop, thus resulting in a time complexity of $O(N^2)$. Usually, the All-Pairs method is not used on its own, but as a kernel to compute forces in close-range interactions [27]. Since the All-Pairs algorithm takes substantial time to compute accelerations, it serves as an interesting target for parallelization.

Algorithm 1: Sequential All-Pairs Algorithm (2 Dimensions)

```
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 ≠ i then
7:             d_sq = distance(i, j)
8:             force[i].x += d_x * mass(i) / d_sqˆ3
9:             force[i].y += d_y * mass(i) / d_sqˆ3
```

2.2 The Barnes-Hut Algorithm

The Barnes-Hut algorithm [8, 5, 33, 4] is one of the most widely used approximations of the N-body problem, primarily by clustering groups of distant close bodies together into a "pseudo-body." Empirical evidence proves that the Barnes-Hut heuristic method requires far fewer operations than the All-Pairs method, thus useful in cases of a large number of bodies where an approximate but efficient solution is more feasible.

Each pseudo-body has an overall mass and center of mass depending on the individual bodies it contains (its children). The Barnes-Hut algorithm uses an adaptive tree structure (quad-tree for 2D or oct-tree for 3D)[1]. A tree structure is created with each node bearing four children (see Fig. 1).

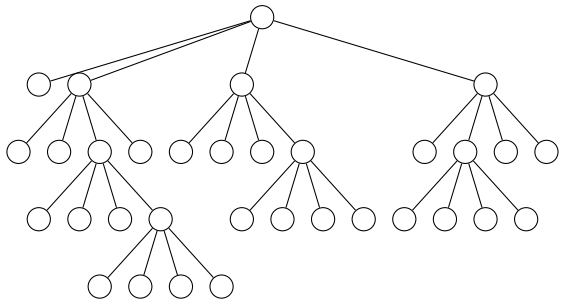

Fig. 1: Example of a quad-tree used in the Barnes-Hut algorithm.

Once built, the tree describes the whole system, with internal nodes representing pseudo-bodies and leaf nodes representing the N bodies [13]. The tree is then used in computation and updating of a body's state. The Barnes-Hut method given by Algorithm 2 implements the following steps to achieve the realization of a spatial system into a quad-tree:

- Division of the whole domain into four square regions (quads).
- If any of these quads contain more than one body, recursively divide that region into four square regions until each square holds a maximum of one body.
- Once the tree is built, perform a recursive walk to calculate the center of mass ($\overrightarrow{c}$) at every node as $\sum_i \overrightarrow{c_i} m_i / \sum m_i$ (i is a child of the node).

Each body uses the constructed tree to compute the acceleration it experiences due to every other body. The Barnes-Hut algorithm approximates bodies that are too far away using a fixed accuracy parameter (threshold (θ)), and the approximation is called the opening condition. Based on the center of mass, the opening condition is given by $l/D < \theta$ (see Fig. 2, blue body represents the body

[1] In this paper, we have considered quad-tree to implement the Barnes-Hut algorithm.

under consideration) where l is the width of the current internal node and D is the distance of the body from the center of mass of the pseudo-body. Threshold determines the number of bodies to be grouped together and thus determines the accuracy of computations. Heuristics show that Barnes-Hut method can approximate the N-body problem in $O(NlogN)$ time. Depending on the dispersion of bodies in the system an adaptive tree is constructed (usually unbalanced) which poses challenges of building, storing, and work-load balancing.

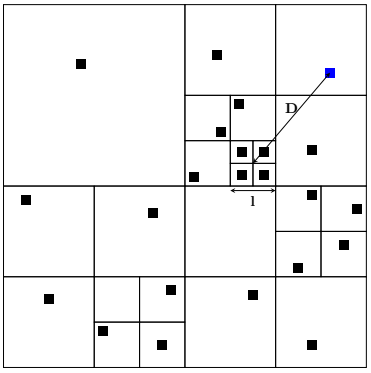

Fig. 2: An example of the pseudo-bodies used in the Barnes-Hut algorithm.

3 Related Work

In 1994, the Virgo Consortium was founded to perform cosmological simulations such as universe formation, tracking the creation of galaxies and black holes on supercomputers; and the most significant problem worked on by them till date is the "Millennium Run" [6]. Their simulations traced around 10 billion particles (each particle represented 20 million galaxies) using code called GAlaxies with Dark matter intEracT (GADGET) [31] along with MPI-HYDRA and FLASH, initially written sequentially but has since been developed to run in parallel to model a broad range of astronomical problems. MPI-HYDRA simulates galaxy and star formations while FLASH simulates thermonuclear flashes seen on the surface of compact stars.

There exist several works in the literature to optimize the N-body problem. Starting with Appel [7], and Barnes and Hut [8] who optimized the N-body problem using adaptive tree structures from $O(N^2)$ to $O(N)$ and $O(NlogN)$ time complexities respectively. An $O(N)$ fast multipole method was developed by Greengard and Rokhlin [21, 22], and they showed the fast multipole approach (FMM) to be accurate to any precision. This was further extended by Sundaram [32] to allow updating of different bodies at different rates which further reduced the time complexity. However, adaptive multipole method in 3 dimensions is complex and has an issue of large overheads.

Algorithm 2: Sequential Barnes-Hut Algorithm (2 Dimensions)

```
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)
```

Various approaches to parallelize the algorithms mentioned above have been developed over the years. Salmon [29] used multipole approximations to implement the Barnes-Hut algorithm on NCUBE and Intel iPSC, message passing architectures. An impressive performance was reported from extensive runs on the 512 node Intel Touchstone Delta by Warren and Salmon [36]. Singh [30] implemented the Barnes-Hut algorithm for the DASH, an experimental prototype. Bhatt *et al.* [10, 19] implemented the filament fluid dynamic problem using the Barnes-Hut method. 16 Intel Pentium Pro processors were used by Warren *et al.* [35] to obtain a sustained performance. Blelloch and Narlikar [11] both implemented and compared in NESL (parallel programming language) the Barnes-Hut algorithm, FMM, and the parallel multipole tree algorithm.

Board *et al.* [12] implemented an adaptive FMM method in three dimensions on shared memory machines. Zhao and Johnsson [38] described a non-adaptive version of Greengard and Rokhlin's method in three dimensions on the Connection Machine CM-2. Mills *et al.* [25] prototyped the FMM in Proteus (an architecture-independent language) using data parallelization, which was then implemented by Nyland *et al.* [26]. Pringle [28] implemented the FMM both in two and three dimensions on the Meiko Computer Surface CS-1 which is a parallel computer with distributed memory and explicit message passing interface.

Liu and Bhatt [24] explained their experiences with parallel implementation of the Barnes-Hut algorithm on the Connection Machine CM-5. A highly efficient, high performance and scalable implementation of the N-body simulation on FPGA was demonstrated by Sozzo *et al.* [18]. Totoo and Loidl [34] compared the parallel implementation of the All-Pairs and Barnes-Hut algorithms in Haskell, a functional programming language. The Tree-Code Particle-Mesh method developed by Xue [37] combines the particle-mesh algorithm with multiple tree-code to achieve better solutions with low computational costs. Nylons [27] accelerated the All-Pairs algorithm using CUDA and presented a sustained performance. Burtscher and Pingali [13] implemented the Barnes-Hut algorithm with irregular trees and complex traversals in CUDA.

4 Proposed Methodology

There are many considerations such as storage, load-balancing, and others in parallelizing the algorithms to solve the N-body problem. Following subsections elucidate on the challenges in parallelization and relevant parallel considerations to overcome those challenges.

4.1 Parallel All-Pairs Algorithm in OpenMP and CUDA

The traditional brute-force All-Pairs algorithm, with $O(N^2)$ time complexity serves as an interesting target for parallelization. This approach can be easily parallelized, as it is known in advance, precisely how much work-load balancing is to be done. The work can be partitioned using a simple block partition strategy since the number of bodies is known and updating each body takes the same

Algorithm 3: OpenMP Implementation of the All-Pairs Algorithm

```
1:  Function calculate_force() is
2:      #pragma omp parallel for
3:      foreach i: body do
4:          find_force(i, particles)
5:  Function find_force(i: body, particles) is
6:      #pragma omp parallel for reduction (+ : force[i].x, force[i].y)
7:      foreach j in particles do
8:          if j ≠ i then
9:              d_sq = distance(i, j)
10:             force[i].x += d_x * mass(i) / d_sqˆ3
11:             force[i].y += d_y * mass(i) / d_sqˆ3
```

amount of calculation. Algorithm 3 provides pseudo-code of the All-Pairs algorithm in parallel using OpenMP. We assign each process a block of bodies each *numPlanets*/*numProcessors* in size, to compute forces acting on those bodies (all processes perform the same number of computations). Thus, the work-load is equally and efficiently partitioned among processes. Algorithm 4 reports the pseudo-code of the All-Pairs algorithm in CUDA.

Algorithm 4: CUDA Implementation of the All-Pairs Algorithm

```
1:  Function calculate_force() is
2:      foreach i: body do
3:          find_force <<< BLOCKS, THREADS_PER_BLOCK >>>
              (index, particles, force, size)
4:  Function find_force(i: body, particles, force, size) is
5:      j = particles[treadIdx.x + blockIdx.x * blockDim.x]
6:      if j ≠ i then
7:          d_sq = distance(i, j)
8:          force[i].x += d_x * mass(i) / d_sqˆ3
9:          force[i].y += d_y * mass(i) / d_sqˆ3
```

4.2 Parallel Barnes-Hut Algorithm in OpenMP

The Barnes-Hut algorithm poses several challenges in the parallel implementation of which, decomposition and communication have a severe impact. To begin with, the cost of building and traversing a quad-tree can increase significantly when divided among processes. The irregularly structured and adaptive nature of the algorithm makes the data access patterns dynamic and irregular, and the nodes essential to a body cannot be computed without traversing the quad-tree. Decomposition is associated with work-load balancing while communication bottleneck is a severe issue that requires the need for minimization of communication volume.

Algorithm 5: Force Computation Parallelization of Barnes-Hut Algorithm

```
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)
```

Building a quad-tree needs synchronization. Since the computation of the center of mass of a pseudo-body depends on the center of masses of corresponding sub-bodies, data dependencies are predominant and thus implying tree level-wise parallelization. For force computation, we need other particles' center of mass, but we do not modify the information and thus can be parallelized. The value of the fixed accuracy parameter (θ) plays a prominent role and must be optimized. Higher values of θ imply that fewer nodes are considered in the force computation thus increasing the window for error; while lower values of θ will bring the Barnes-Hut approximation time complexity closer to that of the All-Pairs algorithm. Algorithm 5 presents pseudo-code of force parallelization (as explained above) of the Barnes-Hut algorithm.

In computing the center of mass of the nodes, some level of parallelization can be achieved in-spite of data dependencies as the computation for each quad in the tree is independent of the other which speeds up the process significantly. Algorithm 6 depicts the parallelization of the center of mass computation.

Parallelization of the Barnes-Hut algorithm has many issues, the significant issue being the lack of prescience on the number of computations per process; which make it a complex parallelization problem. It can be observed that, with the increase in the quad-tree traversal depth, the number of force calculations increases significantly and the exact depth is dependent on the position of the current body.

5 Evaluation, Results, and Analysis

The sequential All-Pairs algorithm was implemented in C++, with parallelization in OpenMP and CUDA. OpenMP's multi-threading [9] fork-join model was used to fork a number of slave threads and separate the errand among them.

Algorithm 6: Mass Distribution Parallelization of Barnes-Hut Algorithm

```
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
```

The separated tasks are then performed simultaneously, with run-time environment assigning threads to distinct tasks. The segment of code intended to run in parallel is stamped likewise with a preprocessor order that is used to join the outputs of the processes in order after they finish execution of their corresponding task. Each thread can be identified with an ID, which can be acquired using OpenMP's `omp_get_thread_num()` method. CUDA allows the programmer to take advantage of the massive parallel computing power of an Nvidia graphics card to perform any general-purpose computation [2, 20]. To run efficiently on CUDA, we used hundreds of threads (the more the number of threads, the faster the computation). Since the All-Pairs algorithm can be broken down into hundreds of threads, CUDA proves to be the best solution. GPUs use massive parallel interfaces to connect with their memory and are approximately ten times faster than a typical CPU-to-memory interface.

This section focuses on running the algorithms described in the above section, in serial and in parallel. All the algorithms were tested using data with a number of bodies ranging from 5 to 30,002 in the galactic datasets [1]. All tests for sequential and OpenMP implementations were performed on nearly identical machines with the following specifications:

- *Processor*: i5 7200U @ 4× 3.1 GHz
- *Memory*: 8 GB DDR3 @ 1333 MHz
- *Network*: 10/100/1000 Gigabit Local Area Network (LAN) Connection

All tests for CUDA implementations were performed on a server with the following specifications:

- *Processor*: Intel Xeon Processor @ 2× 2.40 GHz
- *Memory*: 8 GB RAM
- *Tesla GPU*: 1× TESLA C-2050 (3 GB Memory)

Speedup (S) is a measure of the relative performance of any two systems, here parallel implementations over sequential implementations. Speed up can be estimated using Equation 4.

$$S = \frac{\text{Time}_{\text{sequential}}}{\text{Time}_{\text{parallel}}} \tag{4}$$

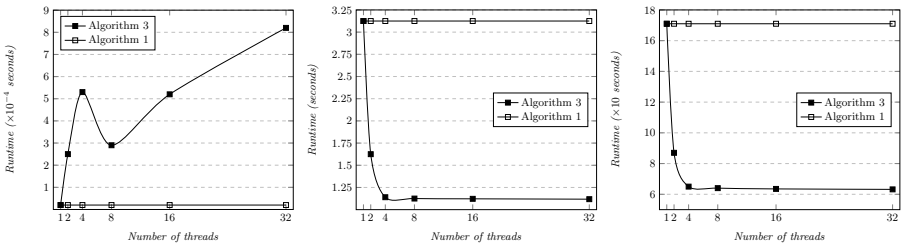

Fig. 3: Parallel vs. sequential running time for (left) 5 bodies in *Planets* [1], (center) 4,000 bodies in *galaxymerge2* [1], and (right) 30,002 bodies in *galaxy30k* [1] using Algorithm 3.

Note that the execution times collected to measure the performance and impact of parallelization is collected five times to overrule bias caused by any other system processes that are not under the control of the experimenter. In every run for time measurement, the order of experimentation for a given dataset is shuffled. The individual measurements are then averaged to represent the running time taken by the parallel algorithm accurately.

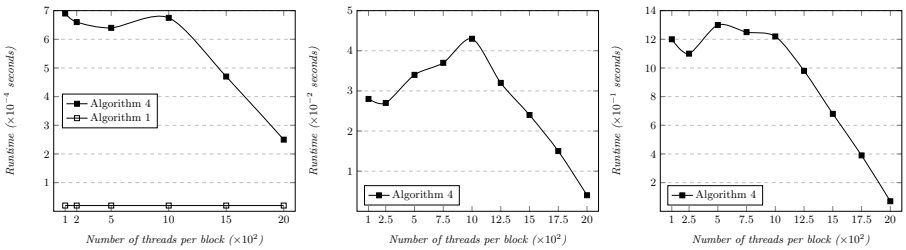

Fig. 4: Parallel vs. sequential running time for (left) 5 bodies in *Planets* [1], (center) 4,000 bodies in *galaxymerge2*[2] [1], and (right) 30,002 bodies in *galaxy30k*[2] [1] using Algorithm 4.

In this paper, we graphed (see Fig. 3 to Fig. 6) the parallel implementations against their respective sequential implementations to visualize the effect of speedup. We also present the results of the performance of the Barnes-Hut

[2] Serial execution takes more than 100 times the parallel execution time and hence is not graphed.

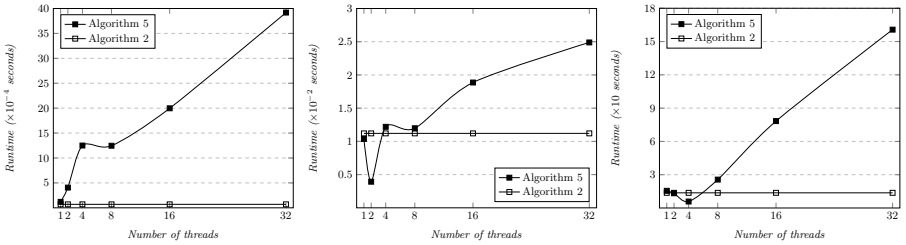

Fig. 5: Parallel vs. sequential running time for (left) 5 bodies in *Planets* [1], (center) 4,000 bodies in *galaxymerge2* [1], and (right) 30,002 bodies in *galaxy30k* [1] using Algorithm 5.

algorithm when both force computation and mass distribution are parallelized (see Table 1). Also, we present the effect of the fixed accuracy parameter (θ) on the Barnes-Hut algorithm (see Fig. 7).

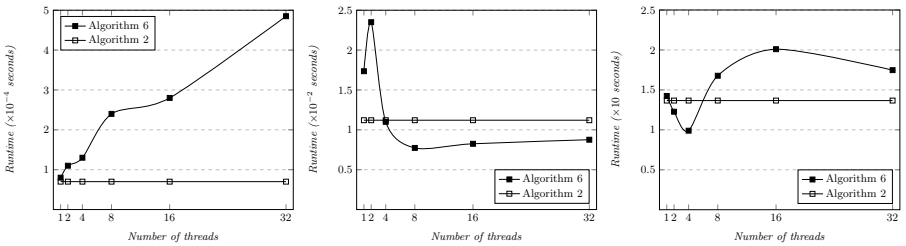

Fig. 6: Parallel vs. sequential running time for (left) 5 bodies in *Planets* [1], (center) 4,000 bodies in *galaxymerge2* [1], and (right) 30,002 bodies in *galaxy30k* [1] using Algorithm 6.

For inputs with a smaller number of celestial bodies, we observe that the sequential execution is faster than parallelized OpenMP code in case of both All-Pairs and Barnes-Hut algorithms (see Fig. 3 (left), Fig. 5 (left), and Fig. 6 (left)). Such behavior can be attributed to the high cost of initialization of threads and their communication overheads, which outweighs the execution time for a lesser number of bodies. With the increase in the number of bodies, the OpenMP parallel implementation runs faster than its sequential counterpart which is attributed to the fact that execution time is larger than the thread spawn overheads (see Fig. 3 (center), Fig. 3 (right), Fig. 5 (center), Fig. 5 (right), Fig. 6 (center), and Fig. 6 (right)). However, it can also be noticed from the graphs that increasing the threads beyond four, either does not change (see Fig. 3 (center), Fig. 3 (right), and Fig. 6 (center)) or increases (see Fig. 5 (center), Fig. 5 (right), and Fig. 6 (right)) the running time. This is because the CPU on the testing machine does not support more than four cores. Greater speedups are observed with the increase in the number of bodies. Also, it is interesting

Table 1: Performance of the OpenMP parallelization with force computation and mass distribution of the Barnes-Hut algorithm on various galactic datasets [1].

Dataset	**Size**	**Serial time (s)**	**Parallel time per thread count (s)**					
			1	**2**	**4**	**8**	**16**	**32**
asteroids1000	1,000	0.023097	0.020348	0.021905	0.030464	0.063325	0.121116	0.221256
cluster2582	2,582	0.004927	0.005837	0.005042	0.011231	0.008328	0.011733	0.014243
collision1	2,000	0.004917	0.004829	0.004447	0.006030	0.005751	0.009468	0.012608
collision2	2,002	0.006227	0.006008	0.006098	0.006309	0.006821	0.009951	0.013182
galaxy1	802	0.015414	0.015616	0.015217	0.020928	0.045315	0.072090	0.110689
galaxy2	652	0.012274	0.012615	0.014664	0.023931	0.028826	0.040485	0.072064
galaxy3	2,001	0.091639	0.087738	0.084466	0.141264	0.264529	0.488200	0.975077
galaxy4	502	0.012875	0.013325	0.010431	0.012065	0.027304	0.037397	0.051786
galaxy10k	10,001	0.032405	0.032411	0.031647	0.027545	0.050811	0.075314	0.171779
galaxy20k	20,001	2.325312	2.357691	1.422882	0.557520	3.697054	7.312913	17.061886
galaxy30k	30,002	13.663441	14.492622	8.259973	3.813991	22.588013	48.301931	90.741782
galaxyform2500	2,500	0.007052	0.005922	0.006162	0.006707	0.008641	0.011501	0.016563
galaxymerge1	2,000	0.004920	0.005160	0.004812	0.006784	0.006742	0.008701	0.018789
galaxymerge2	4,000	0.011205	0.010364	0.003930	0.012193	0.011976	0.018860	0.024891
galaxymerge3	2,901	0.009433	0.009095	0.009045	0.015460	0.011692	0.012852	0.019202
planets	5	0.000070	0.000160	0.000526	0.002250	0.002246	0.002997	0.004918
saturnrings	11,987	0.024471	0.024749	0.020095	0.025863	0.032043	0.038763	0.064468
spiralgalaxy	843	0.017879	0.017627	0.023605	0.024740	0.052584	0.091534	0.166260

to note that with better hardware that supports a greater number of cores, the results can be bettered further.

While the OpenMP implementations have a bottleneck over the number of cores on the test machine, parallelization with CUDA outperforms any such limitations (see Fig. 4 (left), Fig. 4 (center), and Fig. 4 (right)). It can be observed that CUDA implementation provides an exponential decrease in the running time, which is because GPUs has an exponentially larger thread pool as compared to CPUs. It can be seen from Fig. 4 (left) that, for a smaller number of bodies, the parallel running time gradually decreases with the increase in the number of threads per block due to the communication overhead over the peripheral component interconnect lanes. For $4,000$ bodies (see Fig. 4 (center)), the speedup of parallel implementation was observed to be 100 and for $30,002$ bodies (see Fig. 4 (right)), the speedup was approximately 250. These results establish the potential of CUDA in parallel programming and multi-processing support over traditional CPUs.

Table 1 presents the results of the OpenMP Barnes-Hut implementation with both force parallelization and mass distribution. The results show that for a large number of bodies (e.g., *galaxy30k* with $30,002$ bodies), the method proved to be superior as compared to Algorithm 5 and Algorithm 6. However, for a smaller number of bodies (e.g., *planets* with 5 bodies), the method had a massive bottleneck of communication overheads and thread spawn initialization.

The effect of the fixed accuracy parameter used for approximation in the Barnes-Hut algorithm on running time is shown in Fig. 7. The value of θ determines the depth of traversal of the quad-tree. Smaller values of θ mean deeper

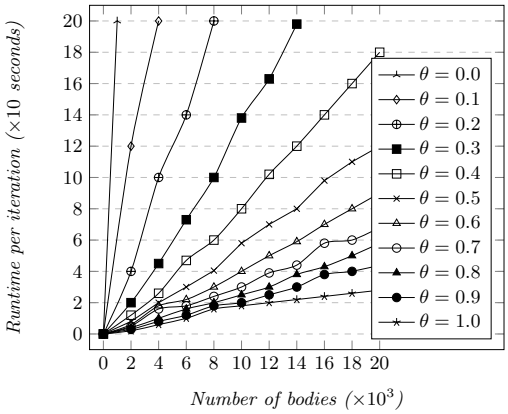

Fig. 7: Effect of the fixed accuracy parameter on the Barnes-Hut algorithm.

traversals, implying larger running times while larger values of θ imply lower accuracy and lower running time. It was observed that the number of computations increased as $\theta \approx 0.0$. We experimentally found that with $\theta = 0.4$, an efficient trade-off between the running time and accuracy can be achieved (all results presented above use $\theta = 0.4$).

6 Conclusions

In this paper, we analyzed two prominent approaches to solve the gravitational N-body problem: the naive All-Pairs approach and an adaptive quad-tree based Barnes-Hut approach. We presented data and task parallel implementations of the algorithms considered, using OpenMP and CUDA. We evaluated the challenges in the parallelization of the Barnes-Hut algorithm and two significant parallel considerations. It was observed that until a certain level of parallelization the running time decreases and then increases afterward. The performance analysis of these methods establishes the potential of parallel programming in big data applications. We achieved a maximum speedup of approximately 3.6 with OpenMP implementations and about 250 with CUDA implementation. The OpenMP implementations experienced a massive bottleneck of the number of cores on the testing machine. Also, we experimentally determined the optimal value of the fixed accuracy parameter for an efficient trade-off between the running time and accuracy.

In the future, we aim at extending the Barnes-Hut implementation and FMM approach to CUDA and message-passing clusters over the LAN, each node parallelized using OpenMP; and also evaluate their performance in terms of speedup and running time. We also aim at considering even larger samples of bodies to evaluate our proposed parallel implementations effectively.

References

1. COS 126 Programming Assignment: N-Body Simulation (Sep 2004), http://www.cs.princeton.edu/courses/archive/fall04/cos126/assignments/nbody.html, [Online; accessed 07. May. 2018]
2. CUDA C Programming Guide (Oct 2018), https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-general-purpose-parallel-computing-architecture, [Online; accessed 07. May. 2018]
3. n-body problem - Wikipedia (Oct 2018), https://en.wikipedia.org/wiki/N-body_problem, [Online; accessed 7. May. 2018]
4. The Barnes-Hut Algorithm : 15-418 Spring 2013 (Oct 2018), http://15418.courses.cs.cmu.edu/spring2013/article/18, [Online; accessed 07. May. 2018]
5. The Barnes-Hut Galaxy Simulator (Oct 2018), http://beltoforion.de/article.php?a=barnes-hut-galaxy-simulator, [Online; accessed 07. May. 2018]
6. World's Largest Supercomputer Simulation Explains Growth of Galaxies (Oct 2018), https://phys.org/news/2005-06-world-largest-supercomputer-simulation-growth.html, [Online; accessed 07. May. 2018]
7. Appel, A.W.: An efficient program for many-body simulation. SIAM Journal on Scientific and Statistical Computing **6**(1), 85–103 (1985)
8. Barnes, J., Hut, P.: A hierarchical o (n log n) force-calculation algorithm. nature **324**(6096), 446 (1986)
9. Barney, B.: OpenMP (Jun 2018), https://computing.llnl.gov/tutorials/openMP, [Online; accessed 07. May. 2018]
10. Bhatt, S., Liu, P., Fernandez, V., Zabusky, N.: Tree codes for vortex dynamics: Application of a programming framework. In: In International Parallel Processing Symposium. Citeseer (1995)
11. Blelloch, G., Narlikar, G.: A practical comparison of tv-body algorithms. Parallel Algorithms: Third DIMACS Implementation Challenge, October 17-19, 1994 **30**, 81 (1997)
12. Board Jr, J.A., Hakura, Z.S., Elliott, W.D., Rankin, W.T.: Scalable variants of multipole-accelerated algorithms for molecular dynamics applications. Tech. rep., Citeseer (1994)
13. Burtscher, M., Pingali, K.: An efficient cuda implementation of the tree-based barnes hut n-body algorithm. In: GPU computing Gems Emerald edition, pp. 75–92. Elsevier (2011)
14. Carugati, N.J.: The parallelization and optimization of the n-body problem using openmp and openmpi (2016)
15. Chanduka, B., Gangavarapu, T., Jaidhar, C.D.: A single program multiple data algorithm for feature selection. In: Intelligent Systems Design and Applications. pp. 662–672. Springer International Publishing, Cham (2018)
16. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering **5**(1), 46–55 (1998)
17. Damgov, V., Gotchev, D., Spedicato, E., Del Popolo, A.: N-body gravitational interactions: a general view and some heuristic problems. arXiv preprint astro-ph/0208373 (2002)
18. Del Sozzo, E., Di Tucci, L., Santambrogio, M.D.: A highly scalable and efficient parallel design of n-body simulation on fpga. In: Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International. pp. 241–246. IEEE (2017)

19. Fernandez, V.M., Zabusky, N.J., Liu, P., Bhatt, S., Gerasoulis, A.: Filament surgery and temporal grid adaptivity extensions to a parallel tree code for simulation and diagnosis in 3d vortex dynamics. In: ESAIM: Proceedings. vol. 1, pp. 197–211. EDP Sciences (1996)
20. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel computing experiences with cuda. IEEE micro (4), 13–27 (2008)
21. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. Journal of computational physics **73**(2), 325–348 (1987)
22. Greengard, L., Rokhlin, V.: A new version of the fast multipole method for the laplace equation in three dimensions. Acta numerica **6**, 229–269 (1997)
23. Heggie, D., Hut, P.: The gravitational million-body problem: a multidisciplinary approach to star cluster dynamics (2003)
24. Liu, P., Bhatt, S.N.: Experiences with parallel n-body simulation. IEEE Transactions on Parallel and Distributed Systems **11**(12), 1306–1323 (2000)
25. Mills, P.H., Nyland, L.S., Prins, J.F., Reif, J.H.: Prototyping n-body simulation in proteus. In: Parallel Processing Symposium, 1992. Proceedings., Sixth International. pp. 476–482. IEEE (1992)
26. Nyland, L.S., Prins, J.F., Reif, J.H.: A data-parallel implementation of the adaptive fast multipole algorithm. In: Proceedings of the DAGS93 Symposium (1993)
27. Nylons, L.: Fast n-body simulation with cuda (2007)
28. Pringle, G.J.: Numerical study of three-dimensional flow using fast parallel particle algorithms. Ph.D. thesis, Napier University of Edinburgh (1994)
29. Salmon, J.K.: Parallel hierarchical N-body methods. Ph.D. thesis, California Institute of Technology (1991)
30. Singh, J.P.: Parallel hierarchical n-body methods and their implications for multiprocessors (1993)
31. Springel, V., Yoshida, N., White, S.D.: Gadget: a code for collisionless and gasdynamical cosmological simulations. New Astronomy **6**(2), 79–117 (2001)
32. Sundaram, S.: Fast algorithms for n-body simulation. Tech. rep., Cornell University (1993)
33. Swinehart, C.: The Barnes-Hut Algorithm (Jan 2011), http://arborjs.org/docs/barnes-hut, [Online; accessed 07. May. 2018]
34. Totoo, P., Loidl, H.W.: Parallel haskell implementations of the n-body problem. Concurrency and Computation: Practice and Experience **26**(4), 987–1019 (2014)
35. Warren, M.S., Becker, D.J., Goda, M.P., Salmon, J.K., Sterling, T.L.: Parallel supercomputing with commodity components. In: PDPTA. pp. 1372–1381 (1997)
36. Warren, M.S., Salmon, J.K.: A parallel hashed oct-tree n-body algorithm. In: Proceedings of the 1993 ACM/IEEE conference on Supercomputing. pp. 12–21. ACM (1993)
37. Xue, G.: An o(n) time hierarchical tree algorithm for computing force field in n-body simulations. Theoretical Computer Science **197**(1-2), 157–169 (1998)
38. Zhao, F., Johnsson, S.L.: The parallel multipole method on the connection machine. SIAM Journal on Scientific and Statistical Computing **12**(6), 1420–1437 (1991)