Dynamic Load Balancing in Parallel N-Body Treecodes

Alexander Brandt
Department of Computer Science
University of Western Ontario
London, Canada
Email: abrandt5@uwo.ca

Abstract

The N-body problem is a classic problem comprising of N mutually interacting discrete bodies evolving in a dynamical system. In the case of gravitational N-body, the specific problem explored herein, no general analytic solution exists for $N > 2$ and simulations must be employed. However, this problem poses computational challenges where the number of interactions scales quadratically. Computational power has thus limited direct simulations of practically meaningful size. Advancements in computer hardware, particularly support for parallelism, has led to vast improvements in simulation capability. Unfortunately, N-body simulations constitute the class of dynamic irregular parallelism, meaning parallel speed-up does not scale with the number of processors. Nonetheless, parallel extensions of classic algorithms, like the Barnes-Hut algorithm, have been developed. Following the work of Singh et al. we explore the use of space-filling curves and "costzones" in an effort to achieve dynamic work load balance in gravitational N-body simulation. Our from-scratch implementation in C/C++ is complimented with a 3D OpenGL visualization tool.

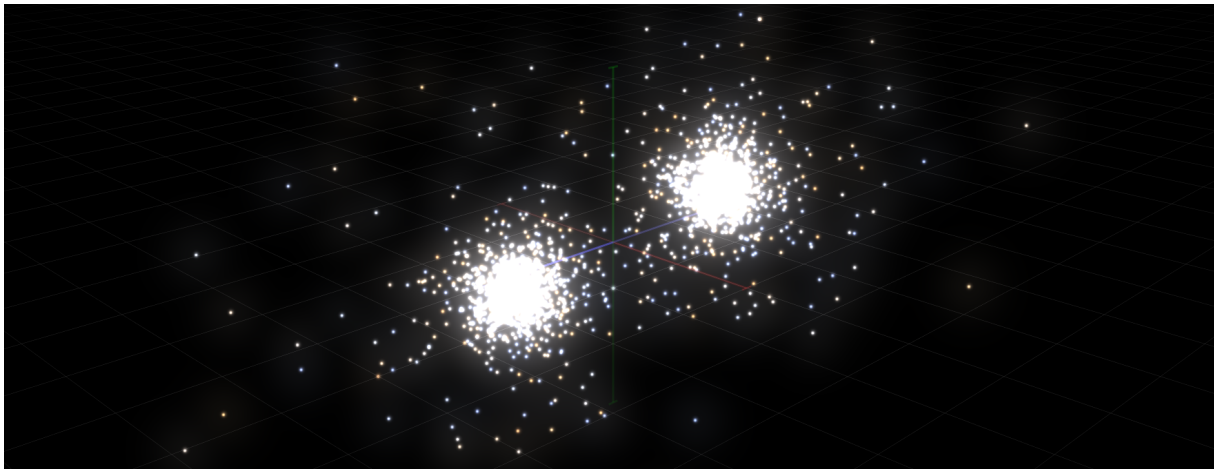


Fig. 1: Simulating the collision of two star clusters, each of consisting of 1000 stars.

I. Introduction

N-body simulations encompass a large collection of simulations involving many discrete bodies under mutually influential physical forces together in a dynamical system. Molecular dynamics, fluid dynamics, and gravitationally-bound systems are all specific problems areas making us of N-body simulations. The N-body problem, meanwhile, historically refers to the dynamics of bodies under the influence of gravity, such as in solar system dynamics (celestial mechanics), or star clusters and galactic formation (known collectively as stellar dynamics).

Beginning generally, each body is depicted as a point mass, and may be described as a point, a particle, or a body. The underlying model in these systems is often a second order dynamical system where the acceleration of a body is usually determined via the computation of a force or potential which involves contributions from the remaining $N - 1$ bodies. As the number of bodies in the system increases, so too does the number of equations in the dynamical systems.

For gravitational simulations, depending on the scale of the system, each body in the system may be considered a single star, such as in globular clusters containing 10^4–10^6 stars, or may represent an

entire galaxy, such as in galaxy clusters containing up to 1000 gravitationally-bound galaxies. The latter, however, must also account for dark matter and relativistic behaviour (see [21] and references therein). The former is dominated rather by Newtonian dynamics, posing less of a mathematical challenge and more of a computational one, due to the increased number of bodies under investigation. We focus on stellar dynamics throughout this paper.

Independent of the underlying model, N-body simulations face two major issues. First, analytic solutions are prohibitively hard to determine for increasing values of N. For example, in quantum mechanics, exact solutions are known up to the order 10 particles, albeit in less than 3 dimensions [19]. The story is much worse for the gravitational N-body problem. A general solution is only known for $N = 2$ while for $N = 3$ only a few solutions are known for very specific initial conditions [5, Ch. 5]. The need for simulations is obvious. However, even direct simulations quickly become intractable. The second issue is thus *scalability*. Acceleration of a single particle[1] has contributions from every other particle. Hence, not only does the number of equations describing the dynamical system grow linearly in N, but so too does the number of terms in each acceleration equation. That is to say, the number of interactions in the system increases as N^2, resulting in a prohibitively large computational time.

Research in practical N-body simulation has been ongoing for decades, where techniques and algorithms have developed alongside advancements in computational power. Throughout the 20th century the greatest leap in scale has been the result of algorithmic advancements which dropped the computational complexity of the simulation from $O(N^2)$ (by the direct method, i.e., *particle-particle method*) to $O(N \log N)$ or even $O(N)$ using approximation algorithms. Here, approximation algorithms do not mean that simulations are approximate due to numerical or truncation errors in the integration (which is always inherent), but rather further algorithmic simplifications to remove some interactions entirely.

Three main methods have arisen from this research, particle-mesh methods [10, Chapter 1], fast multipole methods [3], and Barnes-Hut tree methods [2]. The first is best suited for sparse systems where distances between particles are large, while the second has the lowest asymptotic computational complexity but very large coefficients in the complexity analysis in more than two dimensions. The third method is our method of choice, see Section II.

Fast multipole methods and Barnes-Hut tree methods are both examples of hierarchical tree-based methods, referred to as *treecodes*. Treecodes have received the majority of attention due to many reasons, including being most easily decomposed into large-scale parallel tasks. On modern computer architectures, parallelism is a key requirement to achieve computational performance and scalability (particularly as Moore's law is failing in recent years) [8, Chapter 1]. These approximation methods employed rely on the fact that well-separated bodies are easier to approximate than clusters of bodies. Unfortunately, bodies naturally tend to form clusters as the simulation evolves in time. N-body simulations therefore elicit *irregular parallelism*—tasks and data do not split in a consistent or even way and performance does not scale with the number of processors (see Section III).

Still, much work has been dedicated to obtaining effective parallel schemes for N-body simulation. At the 1993 ACM/IEEE conference on Supercomputing, two pioneering works were presented which tackle parallelism in N-body simulations. The hashed octree method of Warren and Salmon [20] used a hashing function for geometrical coordinates to map particles and tree cells to distributed memory locations for supercomputer scale computations. Decades later, this work is still being researched and refined [21]. The other work by Singh, Holt, Hennessy, and Gupta [17] employs so-called *costzones* to evenly divide the geometrical space containing the bodies into zones of equal work and operate on those zones concurrently. This scheme attempts to maintain data locality through the use of space-filling curves to organize an otherwise scattered list of particles into geometrically meaningful groups.

In this report we follow the costzones technique to implement an effective and dynamic load balancing parallel gravitational N-body simulation based on the Barnes-Hut algorithm. We begin in Section II

[1]Throughout this text we interchange the terms bodies, particles, points, etc. to keep the writing varied and interesting. The differences in terminology only come from different problem domains and bear little difference in interpretation or understanding.

with a review of the classic Barnes-Hut algorithm, and Section III with a review of some fundamental considerations for parallel computing. Section IV examines the model of the gravitational N-body problem. Sections V and VI describe some algorithms underlying our implementation: the numerical integrator, and parallel treecode, respectively, while Section VII describes the implementation itself. In that latter section we also describe a 3D visualization tool for our N-body simulation making use of OpenGL, a screenshot of which is shown in Figure 1. Finally, our implementation is tested for parallel performance on a 12-core node (24 physical threads with simultaneous multithreading) in Section VIII. Our implementation is freely and openly available at https://github.com/alexgbrandt/Parallel-NBody/.

II. Hierarchical N-body Methods

In N-body simulations the numbers in the equations describing the dynamical system scales linearly, where each particle's motion is influenced by the $N-1$ other particles. The result is a quadratically increasing number of interaction terms to be computed. The direct method for simulating the evolution of such a system is simple. Time is discretized into small steps where, for each particle at each time-step, the forces acting on it are computed and then the particle's velocity and position are updated for that small time interval. Algorithm 1 depicts this simple scheme.

In this process, the updating of each particle's position is easy and quick using simple kinematics, requiring only $O(N)$ work. The true troubles lie in the force computations, or equivalently, the computation of the accelerations. Since there are a total of $N(N-1)/2$ interactions to be computed at every step of the simulation, it quickly becomes computationally infeasible for any substantial simulation time or practical number of bodies. Much research has been dedicated to reducing the number of interactions computed while maintaining accuracy in the simulation. These methods only modify the force calculation loop of the simulation, leaving the position update or *numerical integration method* unchanged. We introduce some of these approximation methods now, leaving Sections IV and V to describe the particular interaction equations and numerical integration, respectively.

Algorithm 1 Direct N-Body Simulation

The general scheme loops over small time intervals, first computing the forces on and the acceleration of each body for that interval, then updating each body's position. P is the list of bodies and t_{end} and Δt are simulation parameters for the total simulation time and time-step, respectively.

```
for t from 0 to t_end by Δt do
    for each particle p in P do                          ▷ Force Calculation
        Set the acceleration of p to 0⃗.
        for each particle q in P \ {p} do
            Add the force contribution of q acting on p to the acceleration of p.
    for each particle p in P do                          ▷ Position Update
        Update position and velocity of p using its current acceleration.
```

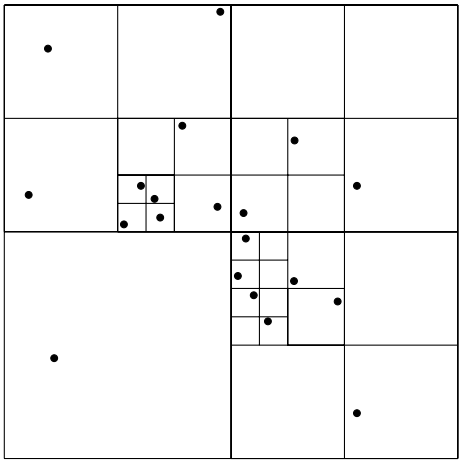
In the late 1980s two methods arose to reduce the computational efforts required in the force calculation step of N-body simulations. The first method to appear was by Barnes and Hut [2] and eventually named the Barnes-Hut method. The second method of Greengard and Rokhlin is the Fast Multipole Method (FMM) [6]. Both of these approximation algorithms—approximate in the way force acting on a body is calculated, not by numerical or truncation errors—are based on a hierarchical tree representation of the geometric space containing the bodies.[2] Thus, they are called *treecodes*.

The key idea in treecodes is to describe an ensemble of bodies by a single one, and then use that single representative pseudo-body in place of its constituents when computing long-range interactions. Applying this technique hierarchically, the further away a target point is from a group of points, the more of those points can be included in an ensemble to further reduce the number of interactions to be computed for the target point. Whenever two points are considered to be too close to be approximated in this way, computations fall back to the direct method of computing pair-wise interactions. This scheme drastically reduces the number of interactions needed to compute. The Barnes-Hut and FMM methods

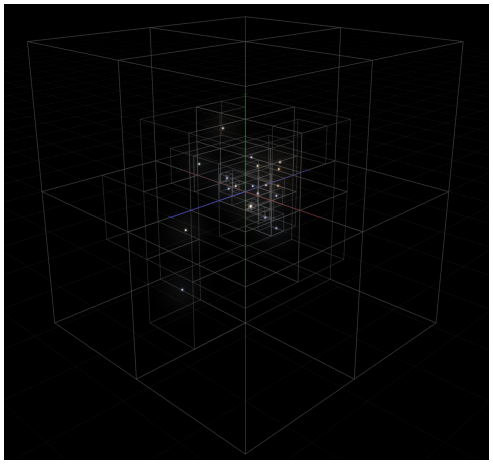
[2]Tree here refers to the computer science notion of a tree data structure, a particular kind of directed acyclic graph.

differ in how an interaction is determined to be "long-range" and how those long-range interactions are specifically computed. However, let us begin with the commonalities.

Both methods begin by constructing a hierarchical tree representing the entire geometric space which bounds the particles being simulated. The root of this tree is precisely the bounding box of all the particles. The root is then split in half in each of its dimensions. In two dimensions, a *quadtree* has each parent node split into 4 children, the area of each being $1/4$ of its parent. In three dimensions, an *octree* splits each of its nodes into 8 children, the volume of each being $1/8$ of its parent. This splitting continues recursively until each leaf node contains exactly one particle. A quadtree is shown in Figure 2a while an octree is shown in Figure 2b. The process of creating and filling a tree is shown in the first part of Algorithm 2 below.



(a) The particle distribution of 20 points in two dimensions contained in a quadtree.



(b) The particle distribtuion of 20 points in three dimensions contained in an octree.

With each particle assigned to a leaf node, each method now beings parsing the tree in a bottom-up fashion. Starting from the leaves, for each level in the tree, a single representative particle is computed for the ensemble of particles in the child cells below it. To compute this representative particle, one can simply place it at the the centre of mass of the child particles and set its mass equal to the sum of masses of its children. This is the original formulation of the Barnes-Hut method [2]. In the FMM method, the representative particle is instead described by a *multipole expansion* of the gravitational potential of the ensemble of points (see Section IV-A).

For the purposes here, a multipole expansion can be seen as simply a series expansion with increasing accuracy as more terms are included. The centre of mass calculation of the original Barnes-Hut method is in fact the first term of the multipole expansion and, moreover, Barnes and Hut themselves indicate that one should use multipole expansions in future. When including terms in the multipole expansion beyond the centre of mass, this "representative particle" at each level is no longer really a particle but rather a formula expressing the mass distribution of its children. For a better description we then refer to it as simply a cell, aligning with the notion of a node or cell within the tree structure.

With the tree structure fully constructed, it is used for force calculations for each body. Recall that the key behaviour of treecodes is to approximate the interaction between a body and an ensemble of bodies. Hierarchical ensembles are precisely what are encoded as the internal cells of the tree. To accurately approximate the interaction between a body and an internal cell, they must be "far enough away" or "well-separated". This notion depends on the particular method.

(i) In the FMM method, a cell d is statically determined to be well-separated from another cell c if the separation is greater than the side length of c. Note that this definition is not symmetric.

(ii) In the Barnes-Hut method, a parameter θ, called the *opening angle* or *multipole acceptance criterion*, controls whether a cell should be considered well-separated. If the ratio between a

cell's side length ℓ and the separation d between a point and the cell's centre of mass is less than θ (i.e. $\ell/d < \theta$), then they are well-separated. This is depicted in Figure 3.

These two definitions also imply the second major difference between the two methods. While both methods directly compute the so-called particle-particle interactions when two particles are close together, the approximation of long-range interactions differ. In the Barnes-Hut method, all long-range interactions are computed as particle-cell interactions. That is, a cell contributes to the net force acting on a particle. In contrast, in FMM cell-cell interactions are also included. This additional type of interaction is what allows this method to reach an asymptotic complexity of $O(N)$ despite being mathematically more challenging and more difficult to code [18]. Since we are focused on the Barnes-Hut method, we refer the reader to [6, 17, 1] for further information on the FMM method.

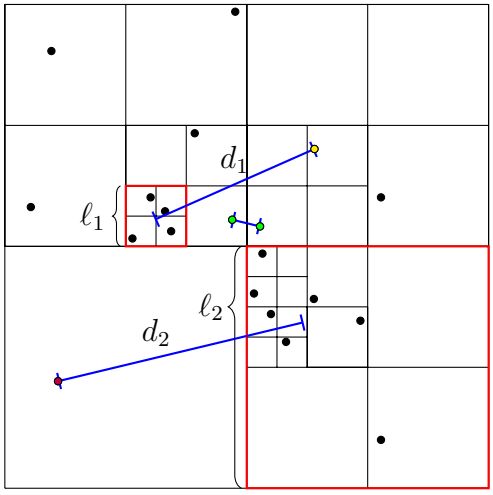


Fig. 3: Determining particle-particle and particle-cell interactions in the Barnes-Hut method. The yellow point is well-separated from the cell with side length ℓ_1 if $\ell_1/d_1 < \theta$. The purple point is well-separated from the large cell with side length ℓ_2 if $\ell_2/d_2 < \theta$. The non-particle endpoints of the two line segments are at the centre of mass of the respective ensemble of points. The two green points are not well-separated and their close-range interaction must be computed directly.

In the Barnes-Hut method, the forces acting on each body are computed via a top-down tree traversal which can be implemented recursively, starting at the root node. Let ℓ and d be defined as before for the current node. If the multipole approximation is acceptable, $\ell/d < \theta$ (i.e. the particle is far enough away from the cell), then a particle-cell interaction is computed and the traversal along this branch of the tree is complete. Otherwise, the cell is "opened" and the traversal continues down each of the current node's children. If a leaf node is encountered, a typical particle-particle interaction is computed. This force calculation process is described in the middle portion of Algorithm 2.

We conclude the discussion of the classic Barnes-Hut method with a final remark. The essential benefit of the Barnes-Hut method is that to traverse the tree and compute the forces on a single particle in this way takes time $O(\log N)$. Thus, to compute force for all N particles is $O(N \log N)$. First, notice the constructed tree has height on the order of $O(\log N)$. This arises from the fact that the average size of a leaf node is on the order of the average inter-particle spacing. The best case is a uniform distribution of particles which creates a balanced tree with $\approx \log N$ levels. Then, traversing the tree to compute the forces of a particle is also on the order of $O(\log N)$. In the worse case, the target particle is "close" to every other particle and no approximations can be made, thus computing N interactions for the target particle. However, it is not possible for every particle to be close to every particle. Indeed, for branches of the tree not including the target particle, it is much more common for those branches to be approximated with a single particle-cell interaction. Since well-separateness is controlled by the parameter θ, the amount of work spent in the force calculation phase is dependent on θ. It is found that the force calculation for a single particle typically scales as $1/\theta^2 \log N$ [9]; a smaller value of θ requires more cells to be opened and thus more interactions to be computed.

Algorithm 2 BARNES-HUT SIMULTATION LOOP BODY

Given a list of particles P and a MAC criterion θ, compute forces and update positions for the current time interval.

```
Build Tree:
Create a tree T with a single root node that is the bounding box of all particles.
for each particle p in P do
    Insert p into the leaf node L of T which encloses the point's coordinates.
    if L now holds two particles then
        Split L in half in each of its dimensions to create children, L is now an internal node.
        Insert each particle of L into the appropriate child.
        Repeat until each particle is in a separate leaf node.
for each node in T from leaves to root do
    Compute sum of masses of contained points, centre of mass, and multipole values.
Compute Forces:
for each particle p in P do
    Node := the root node of T
Compute Cell Force:
    for each child cell C of Node do
        if C contains 1 particle then                                   ▷ C is a leaf
            Add to p's acceleration the contribution from C's particle. ▷ particle-particle
        else
            ℓ := C's side length
            d := distance between C's c.o.m. and p's position.
            if ℓ/d < θ then
                Add to p's acceleration the contribution from C.        ▷ particle-cell
            else
                Compute Cell Force with Node := C.                      ▷ recursive call
Integration step (update velocities and positions):
for each particle p in P do
    Update p's position and velocity using its acceleration.
```

III. HIGH-PERFORMANCE AND PARALLEL COMPUTING

In order to better understand the design and implementation of a parallel Barnes-Hut method, we must discuss aspects of high-performance computing and parallel computing. In this section we give a brief overview of some core concepts including multithreading, data locality, parallel task granularity and load balancing, and irregular parallelism. For further details readers should see [16] and [8].

1) Multithreading: The multithreading programming and execution model is a way for programs to achieve thread-level parallelism (TLP). TLP is achieved by creating multiple independent and concurrent *threads* of execution, each usually executing an independent task within the program. True concurrency, although, is only achieved where the computer hardware executing a program supports multithreading through multiple *physical threads*. Typically, one physical thread is one core (i.e. processor) within a multi-core processor or a multiprocessor. With simultaneous multithreading (SMT), each physical core can support several physical threads (often only 2). However, simultaneous multithreading does not provide the same level of parallelism as distinct cores, and brings additional overheads. SMT with two physical threads on a single core can give a parallel speedup between 0.9 (a slowdown) and 1.6 for various applications [8, Ch. 3]. Overall, SMT can give a slight boost to parallel speedup over the same number of cores without SMT. This parallel speedup is defined as the ratio between a completely serial execution and the parallel execution. *Ideal speedup*, or linear speedup, is obtained when the parallel speedup equals the number of physical threads.

2) Data Locality: On modern computer architectures with a cache memory hierarchy, a key consideration for program performance is data locality. This refers to the way a program accesses and traverses data during the program's execution. Two types of locality should be considered:

(i) *temporal locality* refers to when data which was recently accessed is accessed again, and
(ii) *spatial locality* refers to when data is accessed that is close to recently accessed data.

Here, data being close refers to their memory positions, e.g. adjacent elements in an array. Good data locality avoids expensive *cache misses* where data must be retrieved from main memory as opposed to the processor's local cache of data. The *processor-memory gap* [8, Ch. 5], tells us that modern processors

are exponentially faster than modern random-access memory, and thus data locality is important for performance. Key considerations to obtain data locality include accessing an array in a linear order (spatial locality), and accessing only a small amount of the overall data at one time (temporal locality). Data locality is made further important in the case of multithreading. Independent threads should access independent sections of memory. Otherwise overhead is introduced where coherency of data must be maintained by constantly sharing data between each core's local cache.

3) Task Granularity, Load Balancing: An important consideration when implementing thread-level parallelism is task granularity and load balancing. Task granularity refers to the amount of work to be executed by a thread. Since creating or *spawning* a thread takes time, as does scheduling threads for execution and copying data into each core's local cache, the amount of work to perform must be substantial to warrant those parallelism overheads. To achieve optimal parallel speedup, the amount of work performed by each thread should also be equal. Otherwise, the overall program performance is limited by the thread performing the most amount of work. The act of spreading work evenly between available threads is known as *load balancing*.

4) Irregular Parallelism: Lastly, the parallel execution of a program can be categorized into one of *regular parallelism* or *irregular parallelism* [16, Ch. 2]. Regular parallelism describes situations where tasks to be executed in parallel are similar and predictable. Each task performs a known amount of work and load balancing is easy. In contrast, dissimilar tasks with unpredictable amounts of work lead to irregular parallelism. Problems with irregular parallelism are difficult to load balance. Moreover, the ability of a program to *scale*—to increase parallel speedup given more processors— is further challenging. In treecodes, the number of interactions to be computed for each body changes at every time-step; exhibiting irregular parallelism. In contrast, updating positions and velocities requires a consistent and known amount of work per body, yielding regular parallelism.

IV. The Gravitational N-Body Model: Force, Potential, and Energy

Using Newtonian dynamics combined with Newton's Law of Gravity yields the equations of motions for a system of bodies moving under their mutual gravity. For a system of N bodies there exists $3N$ second-order differential equations describing the motion of the bodies. Using vector notation for simplicity, the position of the body with index i, $\boldsymbol{r}_i$, evolves as:

$$\ddot{\boldsymbol{r}}_i = \boldsymbol{a}_i = \sum_{\substack{j=1 \\ j \neq i}}^{N} \boldsymbol{a}_{ij} = -G \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{m_j(\boldsymbol{r}_i - \boldsymbol{r}_j)}{\|\boldsymbol{r}_i - \boldsymbol{r}_j\|^3}. \tag{1}$$

This equation is derived from Newton's second law, $\boldsymbol{F} = m\boldsymbol{a}$, along with Newton's Law of Gravity which gives the force of gravity exerted on mass m_i by mass m_j as:

$$\boldsymbol{F}_{ij} = \frac{G m_i m_j}{\|\boldsymbol{r}_i - \boldsymbol{r}_j\|^2} \cdot \frac{-(\boldsymbol{r}_i - \boldsymbol{r}_j)}{\|\boldsymbol{r}_i - \boldsymbol{r}_j\|}. \tag{2}$$

Determining the interaction between two particles i and j is simply the computation of this $\boldsymbol{F}_{ij}$ or equivalently, the acceleration $\boldsymbol{a}_{ij}$. This defines a so-called particle-particle interaction. Summing (2) over all other bodies and dividing through by m_i yields the acceleration $\boldsymbol{a}_i$ in (1).

Due to the finite and numerical nature of the simulation, it is useful to introduce a so-called *softening factor* to a particle-particle interaction. Adding a small value ϵ when computing particle separations helps reduce the effects of close-encounters. One simply replaces $\|\boldsymbol{r}_i - \boldsymbol{r}_j\|$ with $(\|\boldsymbol{r}_i - \boldsymbol{r}_j\|^2 + \epsilon^2)^{1/2}$. The softening avoids both numerical errors associated with singularities, as well as preventing odd behaviour arising from discretizing close encounters [1, Ch. 2].

We now look to define the particle-cell interaction approximation from the previous section. We know that to compute $\boldsymbol{F}_{ij}$ for every pair of points quickly becomes infeasible. The fundamental observation

of the Barnes-Hut method is that for some subset of bodies J, with centre of mass $\boldsymbol{r}_J$ and total mass m_J, the force acting on particle i from this ensemble of points is:

$$\boldsymbol{F}_{iJ} = \sum_{j \in J} \frac{-Gm_i m_j(\boldsymbol{r}_i - \boldsymbol{r}_j)}{\|\boldsymbol{r}_i - \boldsymbol{r}_j\|^3} \approx \frac{-Gm_i m_J(\boldsymbol{r}_i - \boldsymbol{r}_J)}{\|\boldsymbol{r}_i - \boldsymbol{r}_J\|^3}. \tag{3}$$

This approximation approaches equality as the separation distance $\|\boldsymbol{r}_i - \boldsymbol{r}_J\|$ grows larger. However, this approximation can be further refined by using gravitational potential.

A. Gravitational Potential and Multipole Expansion

The gravitational potential Φ relates energy, work, and force. It can be defined as the gravitational potential energy per unit mass at a particular location or, equivalently, as the amount of work per unit mass done against gravity to move a mass from infinity to that location. In the frame of reference of a point mass with mass M inducing a gravitational field, the gravitational potential at position $\boldsymbol{r}$ at a distance r from the point mass is:

$$\Phi(\boldsymbol{r}) = \frac{1}{m}\int_{\infty}^{r} \boldsymbol{F} \cdot d\boldsymbol{r} = \int_{\infty}^{r} \frac{GM}{r^2} dr = -\frac{GM}{r}. \tag{4}$$

Since gravity is an attractive force, gravitational potential is negative while acceleration is positive. The acceleration due to gravity of an object at position $\boldsymbol{r}$ is then given as the negative gradient of the potential:

$$\boldsymbol{a} = -\nabla\Phi(\boldsymbol{r}) = -\frac{GM}{r^3}\boldsymbol{r}. \tag{5}$$

This implies that we can use gravitational potential for the force calculations and interactions in our N-body system. The useful consequence of this is that gravitational potentials can be combined via superposition. Given as subset of points J as before, and a point $\boldsymbol{r}$ external to the points in J, the resulting potential is:

$$\Phi(\boldsymbol{r}) = \sum_{j \in J} -\frac{Gm_j}{\|\boldsymbol{r} - \boldsymbol{r}_j\|} \tag{6}$$

$$\approx -\frac{Gm_J}{\|\boldsymbol{r} - \boldsymbol{r}_J\|} \tag{7}$$

Much like forces, we can approximate the superposition of potentials using their total mass m_J and their centre of mass $\boldsymbol{r}_J$. However, an even better approximation can be obtained using a *multipole expansion*.

For simplicity of notation, let us take $G = 1$ henceforth. A multipole expansion is a series expansion of a function depending on angles. Often, the set of basis functions for this series expansion are the spherical harmonics, a set of orthogonal functions defined on the surface of a sphere. In Cartesian coordinates these functions are the Legendre polynomials. Following [1, Ch. 3], let us denote the Legendre polynomial of order (degree) k as P_k. The multipole expansion of (6) is then given by:

$$\Phi(\boldsymbol{r}) = -\frac{1}{r}\sum_{j \in J} m_j \sum_{k=0}^{n} \left(\frac{r_j}{r}\right)^k P_k(\cos\theta), \tag{8}$$

with $r_j = \|\boldsymbol{r}_j\|$ and $\cos\theta = \boldsymbol{r} \cdot \boldsymbol{r}_j / r r_j$. The direct summation of (6) is approached with higher order n.

In this series expansion the first few terms are called the monopole, dipole, quadrupole terms. Including even higher-ordered terms in the approximation yields diminishing returns since their contribution to the approximation decreases with higher order meanwhile their computational cost increases. Often only up to quadrupole terms are included, yet more recent works have included up to octopole terms [11].

Using the compact notation of [1], the multipole expansion of Φ to order 2 is

$$\Phi(\boldsymbol{r}) = -\frac{M}{r} - \frac{D_a x_a}{r^3} - \frac{Q_{ab} x_a x_b}{2r^5}, \tag{9}$$

where $\boldsymbol{r} = (x_1, x_2, x_3)$ and the summation over indices a and b is implied. The multipole coefficients hide the summation over the bodies in J as:

$$M = \sum_{j \in J} m_j, \qquad D_a = \sum_{j \in J} m_j x_{a,j},$$
$$Q_{aa} = \sum_{j \in J} m_j \left(3x_{a,j}^2 - r_j^2\right), \qquad Q_{ab} = \sum_{j \in J} m_j \left(3x_{a,j} x_{b,j}\right), \quad \text{if } (a \neq b),$$

where $x_{a,j}$ is the ath coordinate of $\boldsymbol{r}_j$.

This multipole expansion can be directly applied to our Barnes-Hut tree to improve the approximation of particle-cell interactions. Notice that the monopole term is precisely the sum of the masses in J as in our initial approximation. Next, notice that taking the frame of reference to be the centre of mass of the points in J causes the dipole term to vanish ($D_a = 0$). Therefore, to better the particle-cell approximation by two orders, we need only compute the quadrupole terms. The quadrupole terms for each cell can easily be computed during tree construction during the phase in which the centres of mass for each cell are being computed. This method is detailed in [1, Ch. 6].

This multipole expansion can now easily be included in the Barnes-Hut method. In terms of the basic Barnes-Hut method described in Algorithm 2, only lines 9 and 19 need to be changed. Line 9 changes where each cell now also computes Q_{ab} from (9) while line 19 changes to compute acceleration by $-\nabla\Phi(\boldsymbol{r})$ using the truncated multipole approximation of Φ in (9).

B. Energy of the System and Standard Units

In order to understand the accuracy of our simulation, and in particular our approximation method, a diagnostic measure which can be evaluated is very useful. In gravitational N-body simulation, this value is the energy. This makes natural sense following the law of conservation of energy. Since our simulation precisely simulates an isolated system, the energy should be conserved. The total energy of an N-body system is given by the kinetic energy and the potential energy over the N particles. For masses m_i, positions $\boldsymbol{r}_i$, and velocities $\boldsymbol{v}_i$, we have:

$$E_{kin} = \sum_{i=1}^{N} \frac{1}{2} m_i \|\boldsymbol{v}_i\|^2, \qquad E_{pot} = \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j \neq i}}^{N} -\frac{G m_i m_j}{\|\boldsymbol{r}_i - \boldsymbol{r}_j\|}, \qquad E_{tot} = E_{kin} + E_{pot}. \tag{10}$$

By computing E_{tot} at the beginning and the end of a simulation (or even throughout), it is possible to obtain a measure of how well the simulation performed. A perfect simulation would perfectly conserve energy. Of course this is not possible due to round-off errors, truncation errors in the numerical integrator (see Section V), and the approximations introduced in the treecode.

Unfortunately, these measures are not completely adequate. For different simulations, with different initial conditions, different systems can have vastly different values for their total energy. We would like a way to measure accuracy agnostic to particular initial conditions. The few practitioners of N-body simulation in the 1980s then decided on the so-called *standard units*, see [7] and [1, Ch. 7].

The standard units define a scaling whereby all simulations have an initial total energy $E_0 = -0.25$. In the standard units the total mass of the system is scaled to 1, the gravitational constant is 1, and the system's coordinates are shifted to be centred at its centre of mass. This latter fact makes the system's centre of mass the origin and the net velocity $\vec{0}$. To scale a system to standard units it is first shifted to its centre of mass frame and then its kinetic energy and potential energy calculated. From the virial theorem, a system in equilibrium has kinetic energy equal to a negative half of its potential energy. One can scale velocities and positions to independently and respectively scale E_{kin} and E_{pot} to obtain $E_{kin} = -1/2 E_{pot}$. Finally, the positions and velocities are scaled together so that $E_0 = -0.25$.

V. Leapfrog Integration

Up to this point we have focused on how to calculate the interactions between bodies in our simulation, and thus each body's acceleration. Now we must use this acceleration to update the positions and velocities of the bodies in our system to that it may evolve over time.

The simplest method is the basic Euler method. Velocities are updated from accelerations and positions from velocities. Let $\boldsymbol{r}_i^{(k)}$ and $\boldsymbol{v}_i^{(k)}$ be the position and velocity, respectively, of the ith particle at the kth time-step. The forward Euler method uses values of derivatives at the current step to approximate values at the next step:

$$\boldsymbol{r}^{(k+1)} = \boldsymbol{r}^{(k)} + \boldsymbol{v}^{(k)}\Delta t, \tag{11}$$

$$\boldsymbol{v}^{(k+1)} = \boldsymbol{v}^{(k)} + \boldsymbol{a}^{(k)}\Delta t. \tag{12}$$

While simple, the forward Euler method is only a first-order method, meaning that the (local) error introduced at each time-step is proportional to $O(\Delta t^2)$ [4]. A higher-order method would see that the error reduces more quickly with smaller time-steps. For example, a common fourth-order method is the *Hermite scheme* [1, Ch. 2]. This scheme is a more complex generalization of the classic and simpler scheme known as *leapfrog*, which is sufficient in many cases.

The leapfrog method is as simple and as computationally expensive as the Euler method, but is a second-order method, thus providing better accuracy. Leapfrog specifically solves second-order differential equations of the form $\ddot{\boldsymbol{r}} = \boldsymbol{a}(\boldsymbol{r})$, or equivalently, $\dot{\boldsymbol{v}} = \boldsymbol{a}(\boldsymbol{r}), \dot{\boldsymbol{r}} = \boldsymbol{v}$. The method gets its name from the way that the two differential equations are solved at interleaved time-steps, so that equation jumps over the other in time. The classic formulation of leapfrog shows this jumping structure:

$$\boldsymbol{v}^{(k+\frac{1}{2})} = \boldsymbol{v}^{(k-\frac{1}{2})} + \boldsymbol{a}^{(k)}\Delta t, \tag{13}$$

$$\boldsymbol{r}^{(k+1)} = \boldsymbol{r}^{(k)} + \boldsymbol{v}^{(k+\frac{1}{2})}\Delta t. \tag{14}$$

One of the key advantages to this scheme is its simplicity and time-symmetry. That is, it provides the same results when run forward in time as when run backward in time. In the context of gravitational simulations, this time-symmetry prevents systematic build-up in error in the total energy of the system over time [13].

These equations can also be written at integer time-steps with a a little rearranging and shifting certain values by a half-step. In particular, we can arrive at the "Kick-Drift-Kick" (KDK) version[3] of leapfrog:

$$\boldsymbol{v}^{(k+\frac{1}{2})} = \boldsymbol{v}^{(k)} + \boldsymbol{a}^{(k)}\frac{\Delta t}{2}, \tag{15}$$

$$\boldsymbol{r}^{(k+1)} = \boldsymbol{r}^{(k)} + \boldsymbol{v}^{(k+\frac{1}{2})}\Delta t, \tag{16}$$

$$\boldsymbol{v}^{(k+1)} = \boldsymbol{v}^{(k+\frac{1}{2})} + \boldsymbol{a}^{(k+1)}\frac{\Delta t}{2}. \tag{17}$$

This variation of leapfrog allows for variable time-steps. That is, changing the value of Δt over the course of a simulation to give more precision during volatile points of the simulation (i.e. close encounters). While the KDK leapfrog in employed in our implementation, we currently only use a fixed time-step.

VI. Parallel Aspects of Treecodes

Traditional knowledge of task parallelism suggests that one could model each point, or a subset of points, as a single thread-level task and then let each thread run its own simulation. This follows coarse-grained parallelism and essentially parallelizes the outer-most simulation loop. Of course, in N-body simulations, the simulation of one particle at a particle time-step is dependent on all other particles at that same time-step. A synchronization between threads is thus needed for each new iteration of the loop

[3]The KDK leapfrog is also known as the velocity Verlet method.

over time. Moreover, as we have seen from the discussion on treecodes and the Barnes-Hut method, the number of interactions computed for a particle at each time-step depends on its physical separation from its neighbours. The number of interactions thus varies over time, and a static assignment of particles to threads would create unbalanced work per thread over time. In practice, this naive method is not useful. Instead, the work performed within each time-step should be parallelized in a way which accounts for dynamic load-balancing as the system evolves.

In this section we detail many important aspects to achieve load-balancing in treecodes based on the work of [17] and [18]. As we have seen in Algorithm 2, each time-step in a treecode has three distinct steps: (i) building the hierarchical tree structure, (ii) computing the force on each particle, and (iii) the integration step which updates each particle's position based on the force computed. Each step will be parallelized independently.

Completing steps (i) and (ii) in parallel is not a straightforward process and is thus detailed below in Sections VI-A and VI-B, respectively. Step (iii), however, is very simple to parallelize. Each particle only needs to evaluate a few formula using the leapfrog integration scheme (Equations (15)–(17)), meaning load balance is simple. For p threads, each thread performs the integration step for N/p particles. For data locality each thread should operate on a single contiguous group of particles from the overall. This becomes further important where the ordering of particles will change due to the parallelization of the other simulation steps. Lastly, it should be noted that on modern architectures supporting vectorized instructions, hand-coded assembly can yield very effective parallelism within a single thread during the integration step (see, e.g., [14]). However, we do not utilize that approach here.

A. Parallel Tree Building

There are two possible approaches to building the tree structure in parallel. The first, and conceptually easier, is to create a global tree and have have multiple threads concurrently insert the particles they are responsible for into the tree. One should realize that having a single global structure which is actively being written to by several threads can create data races. Thus, synchronization and mutual exclusion is needed when multiple writes are attempted simultaneously. As the tree grows, the amount of synchronization is reduced as chances are that different threads will be traversing different branches of the tree and the required synchronization is limited. For a small number of processors, or a multi-core processor with a shared cache, this method can be suitable. However, it does not scale well with an increasing number of processors.

A more scalable method was first suggested in [17]. If each thread built their own local tree for its own group of particles, no synchronization is needed. Then, each local tree could be merged into the global tree. Since entire subtrees are being added to the global tree instead of individual particles, the amount of synchronization needed is greatly reduced. The advantage of this method is strengthened if the group of particles inserted into each local tree are physically close. Each local tree then represents a more or less distinct partition of the global tree, allowing the merge to be done almost for free where local trees act as entire branches of the global tree. The merge algorithm is described precisely in [17, Section 5.3.1]. The grouping of particles into physically close neighbourhoods is precisely the approach of costzones used in the parallel force computation.

B. Parallel Force Computation and Costzones

The key issue in computing the forces acting on each particle in parallel is the fact that each particle experiences a different number of interactions and thus a different amount of work associated with it. Moreover, this number of interactions changes dynamically throughout the simulation. If the number of interactions for each particle were static then load balancing would be trivial. Instead, a method is needed to dynamically balance the amount of work for each thread. That is to say, a method to dynamically partition the particles into groups where the total number of interactions to be computed in each group is the same.

A naive method to achieve this would be to partition the already computed global tree structure, assigning distinct branches of the tree to different threads. Each thread would then compute the interactions for each particle in its branch. This takes advantage of the physical locality of particles, since those are the ones to mutually interact, to yield temporal data locality for each individual thread. This method is satisfactory if particles are uniformly distributed in the physical space and thus evenly distributed across branches of the tree. Unfortunately, as a simulation progresses, clusters of particles inevitably form and cause disproportionate amounts of interactions in different branches of the tree.

In the *costzones* approach [17, 18], the tree is dynamically partitioned into (nearly-)contiguous zones (areas in a quadtree, volumes in an octree) where the total amount of work in each zone is equal. This is achieved through a linearization of the leaves of the tree (i.e. particles) and then partitioning them into groups of nearly equal work. Since the number of interactions for each particle at this step in the simulation has not yet been computed, the number of interactions during the previous step is used as a work estimate. Summing these estimates and dividing by the number of processors gives the target size of each costzone.

The linearization of leaf nodes to achieve this partitioning is not a trivial operation. Any typical tree traversal would return a linearization of leaf nodes that are not necessarily contiguous in physical space. The original formulation in [17] used a traversal where child cells would be explored in an order which depended on the order which the current cell was being visited from its parent. In practice, this is equivalent to the Z-order or Morton order, a particular kind of space-filling curve. For example, this order has been used by [20]. More recent works have used Peano-Hilbert curves for this purpose to achieve even better physical contiguity [15]. Using an ordering induced by a space-filling curve gives an explicit ordering for every particle without relying on the the tree structure itself. In fact, this ordering can be computed before the tree is even built to enable better parallel tree building. The implementation of this is discussed in the next section.

VII. Implementation and Visualization

The implementation of our treecode is written from scratch in the C/C++ language, making use of no external libraries. Nearly everything is written in C, except for the the use of C++11 threads. Our implementation closely follows all of the preceding discussion, including the Barnes-Hut method described in Section II and Algorithm 2, the leapfrog integrator of Section V, and the parallel aspects described in Section VI. We take this section to describe implementation-level details which differ slightly in practice from their conceptual description or from their original description in [17] and [18]. In particular, we detail here our data structures, the use of Morton ordering to compute costzones, our particular tree merging technique, and lastly, our thread implementation using the C++11 built-in *thread support library*. Finally, we conclude with an overview of our OpenGL visualization tool.

Our main data structure is a set of parallel arrays (i.e. structure of arrays) maintaining the masses, positions, velocities, and accelerations of the particles. For a particle of index i, its values are represented in positions $3i$, $3i + 1$, and $3i + 2$ for the x, y, and z components, respectively, for each of the latter three arrays, while mass is a simple array of scalar values. This data layout allows our code to make use of spatial data locality throughout the simulation. In particular, it also allows the use of each type of value independently for even better locality (cf. an array of structures). For example, when building the tree only positions are needed, or when computing forces velocity is not needed. Using this structure for the integration time-step is simple and follows exactly from the equations.

The only other data structure for our simulation is the tree itself. In our case it is an octree in three dimensions. Our octree is a simple recursive data structure with each cell storing several values: (i) its centre in physical space, (ii) its side length, (iii) its centre of mass, (iv) the total mass of its children, (v) the total number of particles contained in its children, (vi) the quadrupole coefficients (See Equation (9)), and (vii) a list of its 8 children if it is an internal cell.

To construct the octree in parallel we first determine the costzones. This begins with an estimate of work for each particle. We augment our structure of arrays with one more array holding the work estimate of each particle. This array is automatically updated during the force calculation of the previous step to count the number of interactions and serve as the estimate of work for the current step.[4]

Next, we must obtain a linear ordering of the particles. Directly using their indices as an order will not account for physical locality as the simulation evolves. We thus employ the Morton ordering. In code this is referred to a *spatial key*. The Morton ordering is a bit-wise interleaving of multi-dimensional integer coordinates to yield a single (one-dimensional) integer which acts as key to give the Morton ordering for the coordinates creating the key. On a 64-bit machine, it is useful for this one-dimensional value to be a single machine word. Thus each key is 64 bits and, in three dimensions, each coordinate is a 21-bit number. We begin by determining the bounding box of all points in the system. Let R be the smallest value such that all points in the system are contained in the cube with extents $[-R, R]$. We then map each particle's floating-point coordinates from the range $[-R, R]$ to $[0, 2R]$ to $[0, 2^{21})$ and finally to an integer by taking its ceiling. Given these three integers, it is now simple to produce the spatial key using Morton order bit-interleave (see [20] for details) or any other space-filling curve. Each array in our structure of arrays is then sorted based on the ordering induced by the spatial keys.

Finally, the costzones for this time-step are determined. One simply iterates through the now sorted array of work estimates, aggregating contiguous partitions so that each partition holds $1/p$ of the total work, given p processors. Not only does this sorting yield an ordering of the particles in physical space, but, since the structure of arrays is sorted by the spatial keys, spatial data locality is obtained within each costzone. As we know, improved data locality yields improved performance. With the costzones computed, we can now build each processor's local octree in parallel.

Building each local tree simply follows the first step of Algorithm 2. Then, a global tree must be computed by merging each local tree. In contrast with the original work in [17], we avoid any use of mutual exclusion to a global data structure through a restructuring of the merge. Whereas all local trees were merged into a single global one, we merge trees pair-wise, in parallel, until only one remains, naming that the global one. The local tree construction and merge thus follows the celebrated map-reduce parallel pattern [16, Ch. 5].

Lastly, we have also added an optimization where trees can be constructed in-place, reusing the existing nodes and structure of the trees from the previous step. In order to employ this, the bounding box of all the points must be scaled up slightly so that the size of the root node does not change at every step. Only when the root node changes size do the trees need to be truly created from scratch again. With the octree built and costzones computed, it is now straightforward to compute the forces on each particle in parallel, using one thread per costzone.

A. C++ Thread Support for Parallelism

For each step in the N-body simulation there are several distinct parallel steps. There is the map-reduce creation of the global octree, the force computation, and the integration step. To spawn and join threads for each of these steps would lead to too much overhead. Instead, we employ a *thread pool*—a collection of long-running threads which wait to be given a task, execute that task, and then return to the pool. This avoids the overhead of repeatedly spawning threads. Often, threads in a pool execute some statically-determined task. However, our threads must be receptive to each of parallel steps. We thus implement *functor*-executing threads. A functor is simply an object encoding a function call. Thus, these threads may execute any function that can be encapsulated as a functor. To facilitate the object passing of functors to threads, we have implemented an asynchronous producer-consumer queue. Nonetheless, these are all standard constructs in modern multithreaded programming, so we exclude further details.

[4]In fact, it counts not only the number of interactions but an integer value representing a work estimate. Since particle-cell interactions use quadrupole moments that particle-particle interactions do not, a particle-cell interaction is counted twice to make up for this work difference.

Our threads, functors, and asynchronous queues are all built using standard features of C++11. In particular, `std::thread` (which wraps `pthread` on Linux), `std:function` for functor objects, and `std::condition_variable` for sleeping and waking threads waiting for a new functor.

B. Three Dimensional Visualization

For both debugging and the natural beauty of it, we have implemented a visualization tool for our three-dimensional N-body simulation using OpenGL. This visualization tool has already been shown in Figure 1 where the collision of two star clusters is being simulated. The overall scene is rather simple. There is a camera whose sight is fixed at the origin, but can rotate and zoom freely. The origin is augmented with axes which act as a reference point during movement. The x-y plane is given a translucent grid to give both scale to the simulation and a reference point in the otherwise black space. The most interesting detail here is the implementation of the bodies in the space.

At each time-step in the simulation the positions of the bodies are passed to the visualization tool for rendering. Each is rendered as single point but using a number of shaders. There are three rendering passes of importance. The first renders a small two-dimensional Gaussian texture at each point. This gives the appearance of *limb-darkening*—where the central part of a star appears brighter than the edges. The next pass slightly blurs each texture for a subtle haze and twinkle around each point. Finally, lighting bloom is calculated where each point acts as a light source, creating the impression of ambient light in the inter-particle space. Lastly, it may appear that the color of each body is random, but it is only to a point. In fact, we interpolate the apparent color of main sequence stars from red through orange and white to blue. In equal-mass simulations, a random color assignment in this interpolated range gives visual interest and the ability to visually track a body over time. Otherwise, one could assign color based on the mapping between mass and temperature.

VIII. Experimentation

We have now seen approximation algorithms, integration schemes, dynamic load balancing considerations, and implementation details for gravitational N-body simulation. The last piece is to actually simulate some stellar dynamics. Following [18] and [2] we simulate the collision of two globular clusters, with equal-mass stars, initially separated in each dimension by a fixed amount and then brought together under their mutual gravity. Each of these globular clusters follows a *Plummer model* [12] where the mass density of bodies about a core of some *Plummer radius* falls off as $x^{\frac{-5}{2}}$. All simulations are run using the standard units (see Section IV-B) and with a multipole acceptance criteria $\theta = 0.5$, a time-step of 0.01, and 1000 total time-steps. Our experiments were performed on a node running Ubuntu 18.04.4 and GCC 7.5.0, with two Intel Xeon X5650 processors each with 6 cores (12 physical threads with hyperthreading; thus 24 threads total) at 2.67 GHz, and a 12x4GB DDR3 main memory at 1.33 GHz.

	Number of processors					
N	1	2	4	8	12	24
10000	0.32	0.34	0.33	0.41	0.34	0.39
20000	0.41	0.35	0.36	0.35	0.35	0.38
30000	0.37	0.36	0.37	0.35	0.36	0.40
40000	0.38	0.39	0.38	0.37	0.38	0.37
50000	0.37	0.37	0.38	0.39	0.37	0.37
60000	0.39	0.38	0.36	0.35	0.38	0.35
70000	0.36	0.38	0.38	0.36	0.38	0.35
80000	0.37	0.37	0.38	0.38	0.36	0.35

TABLE I: Absolute value of percent change in total energy for each experimental trial.

Our simulations ranged over various values of N, i.e. the number of bodies in the simulation, and the number of processors utilized during the simulation. We first present a table for the conservation of energy in our simulations to ensure that our implementation of the Barnes-Hut method and the multipole

approximation is correct. Recall that total energy would be conserved in a perfect simulation, and that standard units dictate it should be equal to -0.25. Table I shows the percent change in energy for each experimental trial. In the original presentation of the Barnes-Hut method, energy was conserved to $\approx 1\%$ using a MAC of $\theta = 1$ for the simulated collision of two globular clusters [2]. Thus, with energy conservation of $\approx 0.35\%$, our implementation appears to be satisfactory.

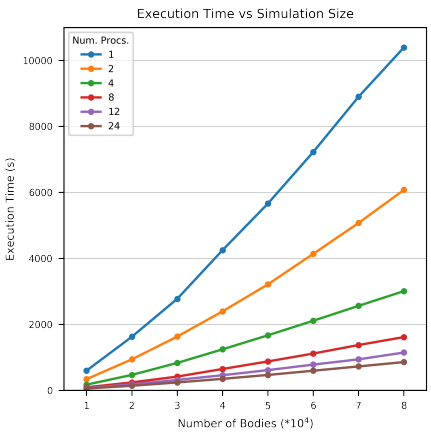


Fig. 4

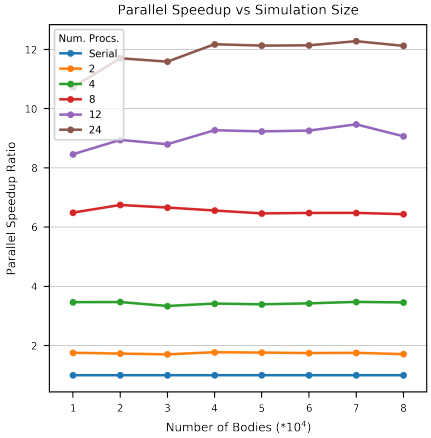


Fig. 5

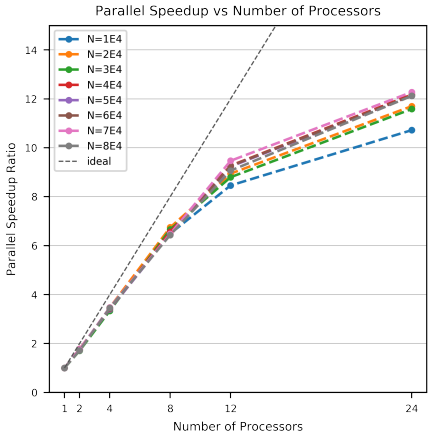


Fig. 6

Now, Figures 4, 5, and 6 show three variations of the same thing: the time to execute each experiment and how it compares to a serial execution. The first and second show execution time and parallel speedup, respectively, as they change with the number bodies in the system for each number of processors used during the experiment. The third shows how parallel speedup changes as the number of processors increase for each simulation size. The takeaway here is that, as the number of processors increases, the larger the simulation size is needed to overcome the overheads in parallelism. From these figures we can see that the speedup obtained is quite good, being at least 80% of ideal up to 8 cores. For 12 cores the speedup is more modest, around 75% of ideal. For 24 processors one should note that large speedups are not expected. The machine running the experiments has only 12 cores but can support 24 physical threads through simultaneous multithreading. SMT can give up 1.6x speedup compared to 1 thread per core. Here, we have obtained about 1.3x speedup with the addition of SMT.

Our results compare well against [18], where roughly 90% of ideal speedup was obtained up to 48 cores. However, we should note that computer architectures have greatly changed since then. Previous experiments were run decades ago on the Stanford DASH, a custom multiprocessor with only single-core processors running at only 33MHz. Hence, the processor-memory gap was not as much of a issue then. Certainly more fine tuning is needed now on modern computer architectures to further account for locality and the memory hierarchy to obtain even better speed-ups.

REFERENCES

[1] S. J. Aarseth. *Gravitational N-body simulations: tools and algorithms*. Cambridge University Press, 2003.

[2] J. Barnes and P. Hut. "A hierarchical O(N log N) force-calculation algorithm". In: *Nature* 324.6096 (Dec. 1986), pp. 446–449.

[3] J. Carrier, L. Greengard, and V. Rokhlin. "A fast adaptive multipole algorithm for particle simulations". In: *SIAM journal on scientific and statistical computing* 9.4 (1988), pp. 669–686.

[4] R. M. Corless and N. Fillion. *A graduate introduction to numerical methods*. Springer, 2013.

[5] H. Gould, J. Tobochnik, and W. Christian. *Computer Simulation Methods*. 3rd. Addison-Wesley, 2006.

[6] L. Greengard and V. Rokhlin. “A fast algorithm for particle simulations”. In: *Journal of Computational Physics* 73.2 (1987), pp. 325–348.
[7] D. Heggie and R. Mathieu. “Standardised units and time scales”. In: *The use of supercomputers in stellar dynamics*. Springer, 1986, pp. 233–235.
[8] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. 4th. Elsevier, 2007.
[9] L. Hernquist. “Hierarchical N-body methods”. In: *Computer Physics Communications* 48.1 (Jan. 1988), pp. 107–115.
[10] R. W. Hockney and J. W. Eastwood. *Computer simulation using particles*. crc Press, 1988.
[11] D. A. Hubber, C. P. Batty, A. McLeod, and A. P. Whitworth. “SEREN–a new SPH code for star and planet formation simulations-Algorithms and tests”. In: *Astronomy & Astrophysics* 529 (2011), A27.
[12] P. Hut and J. Makino. *The Art of Computational Science*. Vol. 11. 2007. URL: http://www.artcompsci.org/.
[13] P. Hut, J. Makino, and S. McMillan. “Building a better leapfrog”. In: *The Astrophysical Journal* 443 (1995), pp. L93–L96.
[14] T. Kodama and T. Ishiyama. “Acceleration of the tree method with an SIMD instruction set”. In: *Publications of the Astronomical Society of Japan* 71.2 (Feb. 2019). 35.
[15] P. Liu and S. N. Bhatt. “Experiences with parallel n-body simulation”. In: *IEEE Transactions on Parallel and Distributed Systems* 11.12 (2000), pp. 1306–1323.
[16] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.
[17] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta. “A parallel adaptive fast multipole method”. In: *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. 1993, pp. 54–65.
[18] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy. “Load Balancing and Data locality in Adaptive Hierarchical N-Body Methods: Barnes-Hut, Fast Multipole, and Rasiosity”. In: *Journal of Parallel and Distributed Computing* 27.2 (1995), pp. 118–141.
[19] B. Sutherland. *Beautiful models: 70 years of exactly solved quantum many-body problems*. World Scientific Publishing Company, 2004.
[20] M. S. Warren and J. K. Salmon. “A Parallel Hashed Oct-Tree N-Body Algorithm”. In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing ’93. Portland, Oregon, USA: Association for Computing Machinery, 1993, pp. 12–21.
[21] M. S. Warren. “2HOT: an improved parallel hashed oct-tree n-body algorithm for cosmological simulation”. In: *Scientific Programming* 22.2 (2014), pp. 109–124.