Parallelization of the Barnes-Hut algorithm for gravitational n-body simulations

David Gichev 196034

Abstract

The Barnes-Hut algorithm is widely used in problems involving a large number of bodies interacting with each other. While being extremely more efficient than the brute force approach, the need for parallelization still exists. This paper first gives a brief overview of the algorithm itself, before getting to the main crux of the problem - its parallelization. An example solution architecture is implemented in C++ using the Open MPI interface. Variations of the approach are looked at, along with analysis and benchmarking for each of the modifications. These help shed light on the underlying nature of the problem and its compatibility with different architectures. We finish by giving a commentary of the Open MPI interface itself, and its applicability to this problem.

1 Introduction

Most simulations consist of objects that affect one another. This can be in the form of a biological simulation in which agents are drawn to particles of food, or a physics simulation of charged particles that repel or attract each other. The main focus of this paper are gravitational simulations. The naive solution to the n-body problem is calculating all of the pairwise forces that act between every pair of particles. This brute-force approach is characterised by a $O(n^2)$ computational complexity. For large particles counts, this quickly becomes impractical. The alternative is approximation. The Barnes-Hut algorithm is based on the following idea: When calculating the gravitational forces that act on Earth, distant galaxies can be approximated by a single point. This point represents the galaxy's center of mass. If done correctly, this approximation reduces the complexity to $O(n \log n)$. The first step is the organization of the particles in a tree structure, known as a quadtree (or octree for 3D spaces). This step is elaborated in section 3.1. Every node represents a square region in space, along with an approximate center of mass. If an object is sufficiently far away from a region then the force acting upon it can approximated. Let d be the distance between our reference object and a given region's center of mass. The region has a width w which is equal to its height. We can now calculate the ratio $\frac{w}{d}$. If the equation $\frac{w}{d} < \theta$ holds (θ is a predefined parameter usually set to 1) we can approximate the region with a single point. This eliminates the need for further

descent into the quadtree, thus reducing the amount of necessary computation. Notice that the threshold can be adjusted according to our needs. Lowering θ corresponds to fewer approximations and more computations, while increasing it will have the opposite effect. Parallizing this is not a trivial task because each point still needs to be run against portions of the quadtree. By using processes that don't have a shared memory, we are essentially simulating a multiprocessor architecture that doesn't operate on a shared memory. Since we don't have the luxury of having a shared global tree, we must essentially distribute it across the processing units. The problems that arise with this approach are pinpointed and discussed in sections 4 and 5. In addition, gravitational simulations are highly prone to clustering and thus display irregular particle distribution. The effect this has on performance is measured in section 4 by comparing uniform and clustered systems.

2 Related Work

The n-body problem consists of calculating the movement of n bodies interacting with each other. Predominately used in physics simulations, this problem has been worked on for decades [5]. One of the most popular approaches is the Barnes-Hut method[4]. Due to the wide scale applications of this problem, solutions and optimisations have been developed in a number of fields, expanding beyond its roots in physics. As a result of the underlying hierarchical data structure, this problem isn't trivially parallelizable. Still, there is a lot of research that demonstrates the potential for parallelization. Multiple papers focus on the data locality aspect of the problem [11][6][14][2]. Load balancing is also a well researched topic [11][1][8]. Signh et al. [11] look at load balancing and data locality in adaptive hierarchical n-body methods. The ORB partitioning technique is explained as a way to partition space while preserving locality. Their proposed partitioning technique, costzones, takes advantage of the existing tree structure. They partition the tree instead of introducing a new data structure. When a given processor needs information regarding a neighboring node that is stored elsewhere, there are two distinct communication schemes: data shipping and function shipping. In the former, the data is transferred to the node that requests it, which later processes it locally. This is in contrast to the latter method, in which the computation is shipped to the processor where the data resides. Grama et al. [5] use the function shipping scheme in all of their parallel formulations. Of the three, the first two employ static partitioning of the domain space, differing in the stasis of the allocation. The last formulation is aimed towards simulations that have a highly unstructured distribution. It uses the costzones scheme as an efficient load partitioning method. Winkel et al. [12] propose a hybrid tree algorithm that decomposes and orders the domain space using the Morton (Z) curve. They later introduce the idea of specialized threads that are only used for communication, enabling asynchrony and increased concurrency. Utilization of the GPU comes up as a frequent challenge [6], in spite of the difficulty associated with the nature of the original data structure [9].

3 Solution Architecture

The proposed solution is an example solution that is based on computing units that don't have a shared memory. Thus, sharing of the quadtree is not possible. We combat this by proposing a shared partial tree technique. This will later turn out to be suboptimal and greater performance benefits will come from optimization of the message exchange.

Every iteration consists of three steps: tree building, force calculation, and making changes. Of the three, the second step is the most computationally intensive, taking up more than 90% of the total execution time. Since the data sets for these types of simulations can get extremely large, splitting the data is required most of the time. This means that all of the three stages must be parallelized. Before the processing can begin, we must first organize and partition the data. This solution works for both 2D and 3D gravitational simulations. We will explain the algorithm in the 2D case, mainly because of the simplified visualizations that come with it. All of the data structures that are used have their 3D counterparts and are completely interchangeable.

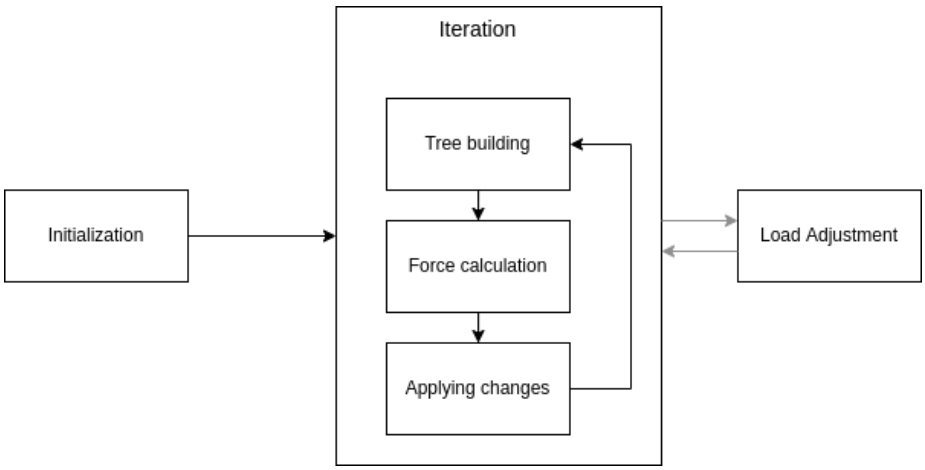

Figure 1: Overview of the proposed algorithm

3.1 Initialization

This is only done once and thus can be done sequentially. We begin by loading all of the data and building a quadtree (or octree for the 3D case). An example of a quadtree can be seen in Fig. 2. We will refer to the non-leaf nodes of the tree as internal nodes, or internal cells. The leaf nodes represent the particles themselves. This is a strict adaptive quadtree, meaning that all of the nodes have a maximum of 4 nodes that are positioned in the four quadrants that the bounding box of the node defines. Every quadrant of an internal node is associated with either a particle (leaf) or another internal node, which in turn, is

a quadtree. Having built the tree, the next step is partitioning it. By traversing it in-order we will effectively end up with an ordered list of points. This ordering loosely preserves the locality of particles. Hence, a pair of particles that are closely positioned in space will most likely be near each other in the resulting array. This helps with data locality and reduces the amount of communication overhead in the force calculation phase. We will now split the array in P parts, where P is the number of processing units. Every processing unit will receive its part of the domain space via broadcast. This step can also be done by ordering the particles using space-filling curves, e.g. Z-Curves (Morton ordering).

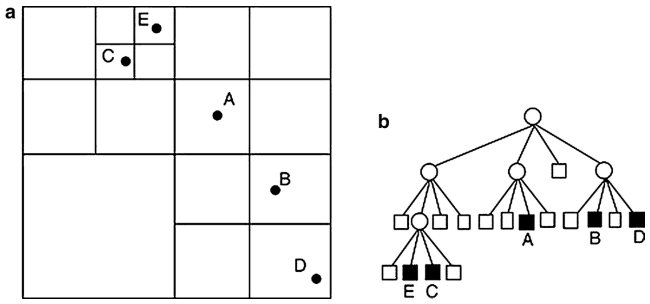

Figure 2: Diagram of a quadtree

3.2 Iteration

Every processing unit will now locally build its own quadtree and broadcast the top m levels of internal nodes. This broadcast can be done in an all-to-all fashion, resulting in each processing node having a vague idea of the space allocated to its coworkers. Usually only the root node of the local tree is advertised, but we decided that sharing the m topmost layers is more beneficial. The precalculation of the center of mass and force exertion for every internal node is implicit in this step.

Now comes the force computation section of the algorithm. This is done in parallel, as is the local tree building. Attraction forces need to be calculated for every particle. We have a rough view of the global tree after composing the broadcast calls in the tree building step. For every particle, we begin the force calculation at the root of the global tree. Every internal node is analyzed along with its distance to the particle. Depending on the calculated distance we determine whether this node needs to be further decomposed or approximated. If the current internal node is not local to the current processing unit, while also being close enough to the particle, we must send a message to the processing unit in which the data resides. We can use either of the two shipping schemes discussed in Section 2. We continue traversing the global tree, taking note of every interaction the particle participates in. This is done by incrementing the counter tied to the internal node. The force calculation phase is now finished and all that is left is to apply the forces to all of the particles. Since the

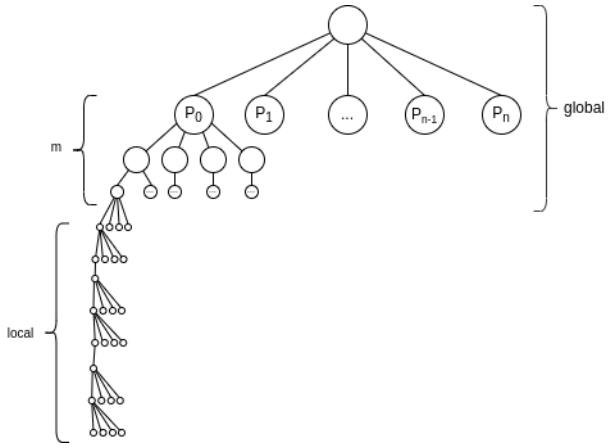

Figure 3: Global and local subtrees

local quadtrees are now outdated, tree rebuilding is mandatory. This is also done locally at the processing unit level. The last step is the broadcast of the updated m topmost levels so that the global tree representation can be refreshed in each of the processing units. As mentioned before, the centers of mass have to be recalculated and assigned to each of the internal nodes.

3.3 Message exchange

Some of the particles will have to be sent for processing on the other units. This amount is highly dependant on the particle distribution of the system. We have tried and tested two methods of message exchange orchestration. The first method processes one pair of units at a time. Given 4 cores, this would mean an exchange between core 0 and core 1, then 0 and 2, etc. This is the easiest to implement, but leaves most of the cores on standby. The second method divides the cores into all possible pairings, and tries to utilize all of the cores. It does this by running all of the pairs of a given configuration simultaneously. The 4 core example would consist of having 3 configurations with 2 pairs each: [(0,1), (2,3)], [(0,2),(1,3)] and [(0,3),(1,2)]. Looking at the first configuration, core 0 and 1 would exchange their points, all while core 2 and 3 do the same thing, concurrently.

4 Results

The solution is implemented in C++, using MPICH. All of the examples are run on a Linux machine with an Intel i5 7th Gen 7200U processor. The points are first sorted using the library libmorton[3], this helps with spacial locality and greatly improves performance. The points are then split in P parts, where P is the number of processing units (cores). After receiving the points, each core builds its own local quadtree and calculates the centers of mass for each of its nodes. The top m layers of the core's tree are to be broadcasted to all the other cores. This is done by taking the first m layers of the quadtree, flattening them into an array, such that we keep only the root's box coordinates, and the center of mass for each of the nodes. We can then calculate the bounds for the remaining nodes. With that, we are finished with the tree building phase. We found that the parameter m only comes into play for very irregularly distributed systems, where the bounds for each quadtree are significantly smaller than the particle space. In situations like these, it usually suffices to set m to 4-6. Below are two example screenshots of the types of systems we will go through. The coloring of the points signifies the processing unit to which the particle belongs. The first system (Fig. 4) is uniformly distributed, making the regions overlap over subsequent iterations. The latter (Fig. 5) is not to scale (the clusters are even tinier) which leads to higher distances between pairs of points that belong on different clusters.

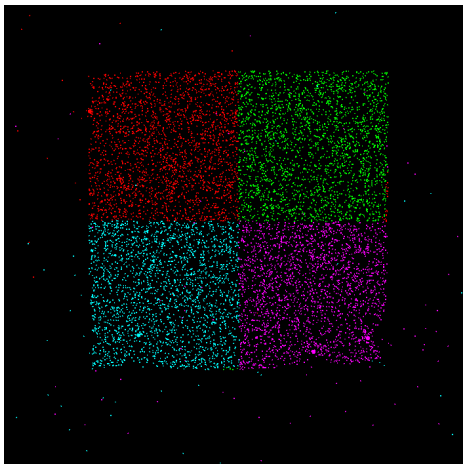

Figure 4: Uniformly distributed system

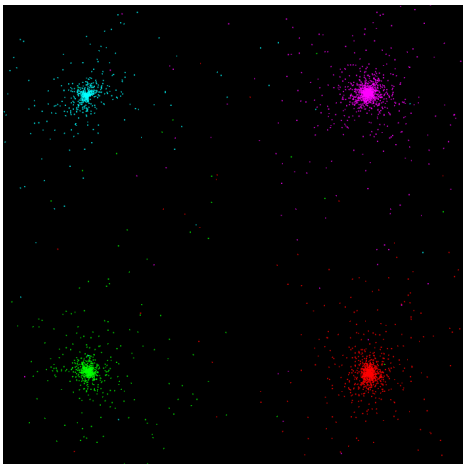

Figure 5: Clustered distribution

4.1 Uniformly distributed particle space

This subset of problems is where the MPI solution suffers the most. This is due to the regions being in close proximity. Almost all of the points have to be sent around as a result. Discarding the top level tree calculation helps with the

performance, since it's just calculation that is done to no avail. Furthermore, we found that the lack of shared memory is most noticed here.

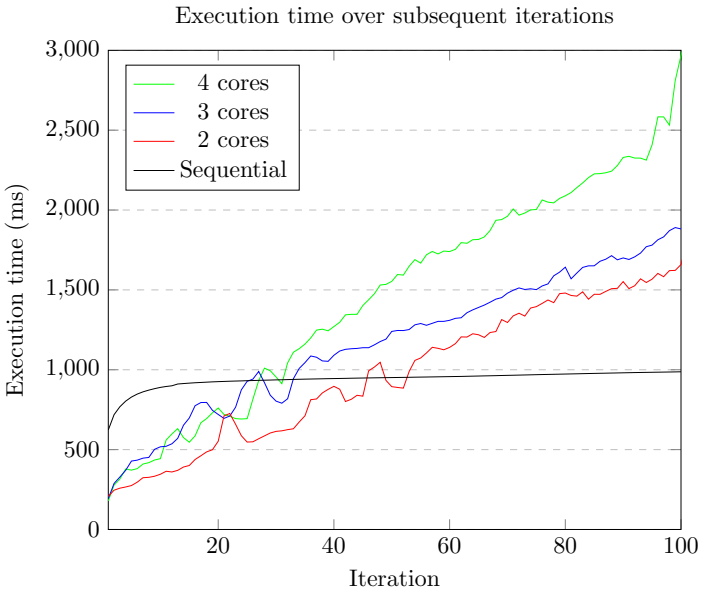

Figure 6: Average execution time over subsequent iterations for a system with 100,000 points (without message exchange optimization)

The execution time suffers as we increase the number of cores. This is counter intuitive at first, but the culprit lies in the message exchange algorithm. This uses the first method (described in section 3.3) and practically utilizes a pair of cores at a time. As we increase the number of cores, the number of messages that have to be passed increases with $O(n^2)$, resulting in a lot of idle time.

4.2 Clustered distribution

Systems that consist of clustered regions of particles allow the top level tree sharing step to come through. The amount of particles being sent around is minimized and as a result more of the computation is local. An improvement of more than 2x is seen across all of the core combinations when compared to the sequential version. However, there isn't a benefit to using more cores, this is because of the synchronisation that happens when pairs of messages get sent. Less points have to go around, but core-to-core communication still needs to be performed.

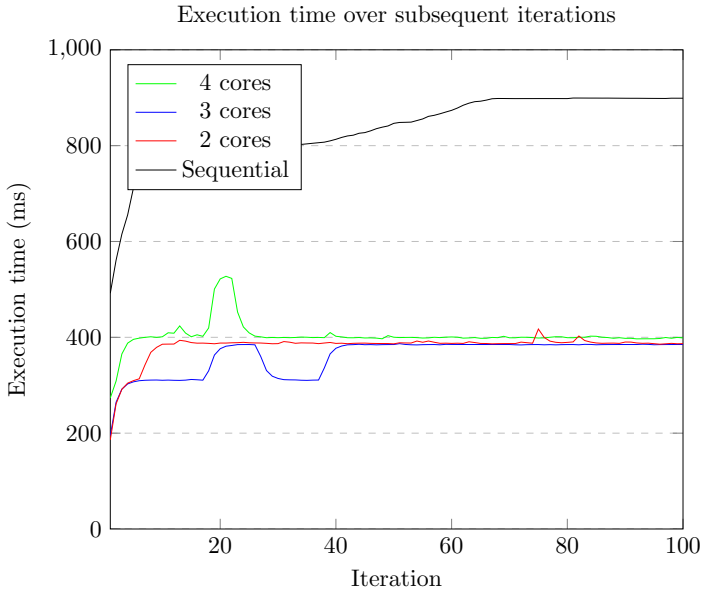

Figure 7: Average execution time over subsequent iterations for a system with 100,000 points

# Points	# Iterations	Sequential	4 cores	4 cores optimized
10,000	1000	71.2 ms	47.8 ms	40.05 ms
100,000	1000	898 ms	445 ms	347 ms
1,000,000	25	12.1 s	4.47 s	4.02 s

Table 1: Comparison of the average iteration time for clustered systems. Optimized means using the second method of message exchange (Section 3.3)

A noticeable speedup can be noted when comparing the sequential algorithm with the paralell algorithm used with 4 cores (Table 1). The largest speedup can be seen in the 1,000,000 points case - the optimized 4 core instance is 2.88x faster than the sequential version. The benefit of using the optimized version is the 15% faster execution time.

5 Conclusion

We have implemented the Barnes-Hut algorithm, both its sequential and parallel version. We have observed that the Open MPI interface is not applicable to data sets that consist of uniformly distributed particles. This is because of its lack of shared memory, which results in the need for constant communication. A large bottleneck that consequently gets created are the synchronisation steps required in the communication phase. Data exchange greatly impacts performance, re-

gardless of whether they are minimal. Still, our work is fruitful with regards to its potential use in systems that have heavily clustered data. Optimization of the message passing pattern has proved to yield noticeably lower execution times and higher processor utilization. All in all, the experimentation we have done with the example solution highlighted the strengths and weaknesses of this type of architecture. Improvement in the uniformly distributed domain can be a topic for future work, along with a threading and shared memory setup.

Visual examples of the solution at work can be seen in Fig. 8. The implementation includes mouse based navigation controls for panning and zooming. The graphics portion is built using the SFML library[10]. This is mainly used for visual verification of the algorithms. Without it, error detection is considerably more difficult. The frame rates are sufficient for noticing inconsistent or weird behaviours.

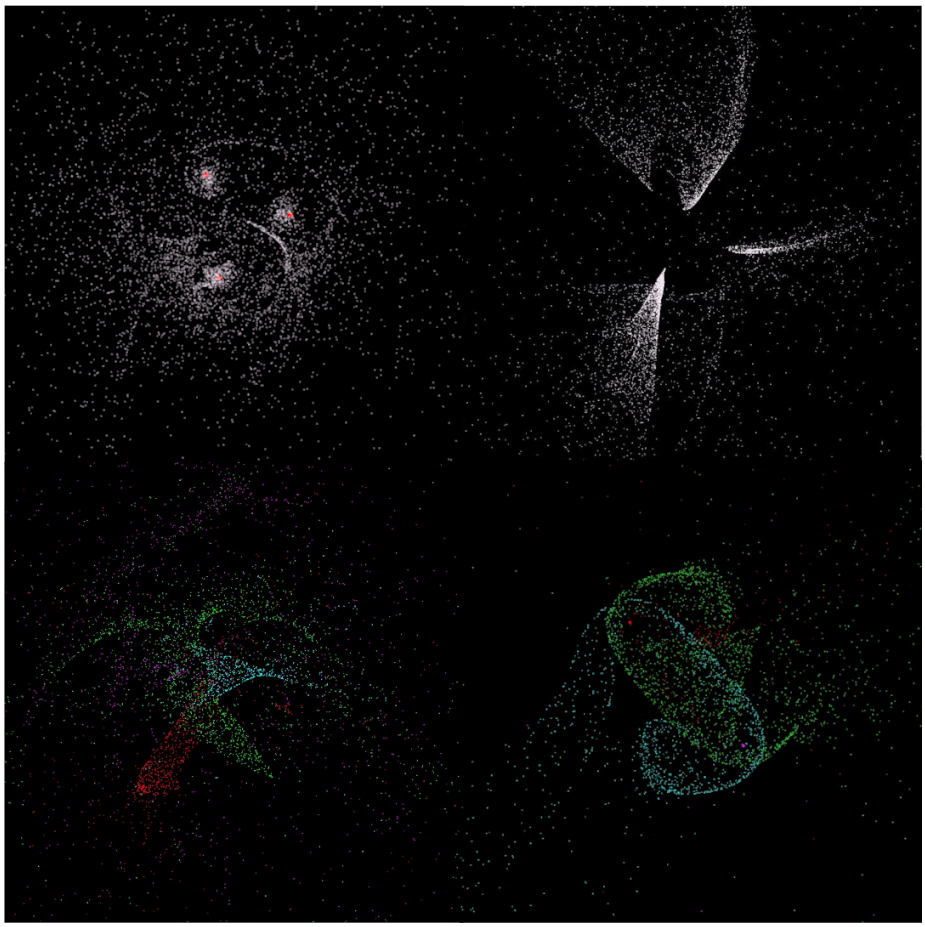

Figure 8: Example screenshots

References

[1] Martin Alt, Jens Müller, and Sergei Gorlatch. Towards high-level grid programming and load-balancing: A barnes-hut case study. pages 391–400, 2005.

[2] M. Amor, F. Argüello, J. López, O. Plata, and E. L. Zapata. A data parallel formulation of the barnes-hut method for n-body simulations. pages 342–349, 2001.

[3] Jeroen Baert. libmorton. https://github.com/Forceflow/libmorton, 2019.

[4] Josh Barnes and Piet Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(6096):446–449, dec 1986.

[5] Ananth Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulations of the barnes–hut method for n-body simulations. 24(5-6):797–822, jun 1998.

[6] Bruno Henrique Meyer, Aurora Trinidad Ramirez Pozo, and Wagner M. Nunan Zola. Improving barnes-hut t-SNE scalability in GPU with efficient memory access strategies. jul 2020.

[7] Badri Munier, Muhammad Aleem, Majid Khan, Muhammad Arshad Islam, Muhammad Azhar Iqbal, and Muhammad Kamran Khattak. On the parallelization and performance analysis of barnes–hut algorithm using java parallel platforms. 2(4), mar 2020.

[8] Olga Pearce, Todd Gamblin, Bronis R. de Supinski, Tom Arsenlis, and Nancy M. Amato. Load balancing n-body simulations with highly non-uniform density. 2014.

[9] Speck Robert, Gibbon Paul, and Hoffmann Martin. Efficiency and scalability of the parallel barnes-hut tree code pepc. *Advances in Parallel Computing*, 19:35–42, 2010.

[10] SFML. Sfml. https://github.com/SFML/SFML, 2018.

[11] J.P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. 27(2):118–141, jun 1995.

[12] Mathias Winkel, Robert Speck, Helge Hübner, Lukas Arnold, Rolf Krause, and Paul Gibbon. A massively parallel, multi-disciplinary barnes–hut tree code for extreme-scale n-body simulations. 183(4):880–889, apr 2012.

[13] Thomas Canhao Xu, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. Evaluate and optimize parallel barnes-hut algorithm for emerging many-core architectures. jul 2013.

[14] Junchao Zhang, Babak Behzad, and Marc Snir. Design of a multithreaded barnes-hut algorithm for multicore clusters. 26(7):1861–1873, jul 2015.