# PERFORMANCE EVALUATION OF TREE STRUCTURES AND TREE TRAVERSALS FOR PARALLEL N-BODY COSMOLOGICAL SIMULATIONS

BY

AMIT SHARMA

B.Tech., Indian Institute of Technology, Kharagpur, 2004

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

# Abstract

Modern parallel cosmological simulators are an important component in the study of the formation of galaxies and planetary systems. However, existing simulators do not scale effectively on more recent machines containing hundreds and thousands of processors. The parallel programming lab at University of Illinois, in collaboration with the Department of Astronomy at University of Washington, has developed a new parallel simulator called ParallelGravity which is based on the Charm++ infrastructure. The work on the simulator has been supported by the National Science Foundation. This simulator provides a powerful runtime system that automatically maps computation to physical processors. The simulator scales to a large number of processors with astronomical datasets containing millions of particles using Charm++ features, in particular its measurement-based load balancers. In this thesis, we describe some optimization techniques that have been implemented as part of the simulator. We implement a new scheme for organizing force computation and new techniques for particle space decomposition. The new force computation scheme uses the idea of an *interaction list* introduced in [12]. The performance comparison of particle decomposition techniques is done and the effect of Charm++ features like run-time load balancing is investigated on different types of particle decompositions. By the addition of features like the ones presented in this thesis, we aim to complete a production version of the code and make ParallelGravity a powerful resource for the astronomy community.

To my parents

# Acknowledgments

I would like to thank Prof. L. V. Kale who advised me on this thesis. His guidance, advice and encouragement have been invaluable over the last two years, and particularly for the work in this thesis.

I thank Filippo Gioachin, my colleague at the Parallel Programming Laboratory, with whom I worked closely during this work. His high energy level, great enthusiasm and cooperative nature made this work easier for me. Celso Mendes helped me with his experience and expertise in all matters whenever I had a question. I have had a very enjoyable stay at the Parallel Programming Laboratory, for which I thank everyone in the lab.

Finally, I would like to thank my parents and my brother for their unconditional support all throughout this work.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In the past, the N-body problem in astrophysics dealt with the evolution of the Solar System. Over the last few decades most of the attention and computing resources in astrophysics has been focused on cosmological N-body simulations with huge number of particles for studying the formation and evolution of large scale structure in the universe.

## 1.1 Motivation

Galaxies are the most distinctive objects in the universe, containing almost all the luminous material. They are remarkable dynamical systems, formed by non-linear collapse and a drawn-out series of mergers and encounters. Since the formation is a highly non-linear process, it is, in general, not analytically tractable. Therefore, the only way to compare the consequences of a theory with the observed distribution of galaxies is via a numerical simulation of structure formation. The widely accepted theory of the formation of structure is the gravitational collapse of initially small fluctuations in the mass density. N-body simulations are commonly used to follow the dynamics of this collapse. These require significant computing resources both in terms of floating-point operations and memory capacity. This motivates the development of improved numerical algorithms as well as the use of parallel computing for performing such simulations.

Thus, parallel cosmological simulators are an important component in the study of the formation of galaxies and planetary systems. Galaxy formation is indeed a challenging computational problem, requiring high resolutions and dynamic timescales. For example, to form a stable Milky Way-like galaxy, tens of millions of resolution elements must be simulated to the current epoch. Locally adaptive timesteps may reduce the CPU work by orders of magnitude, but not evenly throughout the computational volume, thus posing a considerable challenge for parallel load balancing. No existing N-body/Hydro solver can handle this regime efficiently.

The study of planet formation requires modeling the pairwise accretion of smaller bodies into proto-planets. In order to follow this process, it is necessary to predict collisions (and near misses), and resolve the consequences of such collisions, whether they be mergers, fragmentations, or something in between. The computational challenge is to find an adaptive timestep scheme that is stable over millions of dynamical times yet that can handle close encounters accurately and is suitable for simulations involving millions of planetesimals. There are algorithms that are present to solve this problem, but implementing them on existing systems has always been a hard problem.

The scientific payback from such studies can be enormous. There are a number of outstanding fundamental questions about the origins of planetary systems which these simulations would be able to answer. What is the relationship between hot Jupiters and terrestrial-like planets? Do asteroid belts form around other stars (if so, this could be trouble for the Terrestrial Planet Finder, as dust may overwhelm the planet's signal)? How common are other small body reservoirs such as Kuiper Belts and Oort clouds? How effective are planetary systems at cleansing small bodies and thus mitigating sterilizing impacts on any terrestrial planets? What is the origin of the spin of the terrestrial planets? Answering questions like these will open the way to a whole new world of knowledge about the universe.

## 1.2 Overview

To evolve a self gravitating system realized as a large number of particles, we must determine the gravitational acceleration on a given particle due to the mass of all other particles. The simple approach of calculating all pairwise interactions between the particles scales as $O(N^2)$, as $N$ is increased. There have been two basic approaches to reduce this scaling, enabling large scale simulations to take place. One of the approaches rely on the speed of FFT algorithms and include the particle-mesh kind of methods. The second approach uses multipole expansions to approximate the gravitational effect of distant mass.

One of the most widely used methods in the second approach (described above) was proposed by Barnes and Hut [2]. Their scheme uses a hierarchical representation of the mass implemented as some form of tree structure. This tree is traversed and the forces between particles are computed exactly or by approximations, depending on the distance between the given particles. This approach achieves reduction in the complexity of the problem from the original $O(N^2)$ to $O(N \log N)$, where $N$ is the number of particles.

Based on Barnes and Hut method, various cosmological simulators have been created recently. PKDGRAV [4], developed at the University of Washington, can be considered among the state-of-the-art in that area. However, PKDGRAV does not scale efficiently on newer machines with thousands of processors. In this work, we talk about a new N-body cosmological simulator that utilizes the Barnes-Hut tree topology to compute gravitational forces. Our new simulator, named ParallelGravity, is based on the Charm++ runtime system [5]. We leverage the object based virtualization [6] inherent in the Charm++ runtime system to obtain automatic overlapping of communication and computation time, as well as to perform automatic runtime measurement-based load balancing. ParallelGravity advances the state-of-the-art in N-Body simulations by allowing the programmer to achieve higher levels of resource utilization with moderate programming effort. In addition, the use of Charm++ has enabled ParallelGravity to efficiently scale on large machine configurations.

## 1.3 Related Work

There have been numerous studies on the N-Body problem, which involves the evolution of interacting particles that are under the effects of Newtonian gravitational forces. Given the power of hierarchical methods for N-Body simulations, such methods have been adopted for quite some time by the astronomy community [7].

One of the most popular codes currently in the astronomy area is PKDGRAV [4]. PKDGRAV is a parallel hierarchical tree-structured code used to conduct cosmological simulations on shared-memory and distributed-memory systems. It is portable across different communication substrates (e.g. MPI, PVM, etc.), and contains support for adaptive decomposition of work among the processors. In its current production version, PKDGRAV has been used in simulations of systems with millions of particles, and has been shown to scale well on up to hundreds of processors. One restriction in PKDGRAV's current version, however, arises from its limited load-balancing capability. This effectively prevents scaling the code efficiently on newer machines with thousands of processors.

Other cosmological simulators have been in use as well. Among these, two of the major codes are GADGET [10], developed in Germany, and falcON [3], developed at the University of Maryland. However, despite claiming a good scalability with the number of particles, falcON is a serial simulator. Meanwhile, GADGET originally had some of the same limitations of PKDGRAV when scaling to a large number of processors. This has been addressed in a more recent version of their code (GADGET-2), but there are not yet results reported with more than around one hundred processors [11].

## 1.4 Thesis Contribution and Organization

The main contributions of this thesis are:

- *New Scheme for computing forces* We describe a new scheme for computing gravitational forces utilizing the Barnes-Hut tree topology.

- *Space Decomposition Methods* Different schemes for decomposition of the particle space leading to construction of different types of Barnes-Hut tree have been implemented and analyzed.

Chapter 2 talks about the major features of ParallelGravity. It also describes all the optimizations that we have applied to our simulator. Chapter 3 describes the new scheme for computing forces followed by the performance results. Chapter 4 gives a description of the three space decomposition techniques. Performance comparison results have also been presented. Chapter 5 outlines the conclusions and future directions of work.

# Chapter 2

# ParallelGravity

In this chapter, we describe the N-body cosmological simulator called ParallelGravity that we have developed to leverage the features of the Charm++ runtime system. Our goal in developing this new application is to create a full production cosmological simulator that scales to thousands of processors.

This new simulator is capable of computing gravitational forces generated by the interaction of a very large number of particles, integrating those forces over time to calculate the movement of each particle. Since most of the running time of the application is devoted to force computation, our focus has been in optimizing this aspect of the code.

Since the gravitation field is a long range force, the total force applied to a given particle has contributions from all the other particles in the entire space. The algorithm we applied is based on a Barnes-Hut tree topology [2], which enables achieving an algorithmic performance of $O(N \log N)$. The tree generated by this algorithm is constructed globally over all the particles, and distributed across elements that are named *TreePieces*. This distribution depends on the type of decomposition of particle space and on the type of tree constructed. Decomposition of particle space and the types of trees are discussed in detail in chapter 4.

At the leaves of the tree are the particles, which are grouped by spatial proximity into *buckets* of a user-defined size. While walking the tree to compute forces, a single walk is performed for all the particles contained in a given bucket. The new scheme discussed in chapter 3 is different from the original ParallelGravity code in terms of the way in which forces are calculated while walking the tree.

In the following section, we start with a general description of Charm++ features and show how these features were applied to the ParallelGravity code. All the major features implemented in ParallelGravity are then described in section 2.2. This is followed by a detailed description of all the optimizations that have been applied to the original ParallelGravity code (section 2.3). The performance gains (as against the original ParallelGravity version) achieved by making use of these optimizations have also been outlined.

## 2.1 Charm++ and Virtualization

Our new ParallelGravity code is based on the Charm++ [5] infrastructure. Charm++ is a parallel C++ library that implements the concept of *processor virtualization*: an application programmer decomposes his problem into a large number of components, or objects, and the interactions among those objects. The objects, called *chares* in Charm++ nomenclature, are automatically mapped to physical processors by the Charm++ runtime system. Typically, the number of chares is much higher than the number of processors. By making the number of chares independent of the number of existing processors, Charm++ enables execution of the same code on different machine configurations. This separation between logical and physical abstractions provides higher programmer productivity, and has allowed the creation of parallel applications that scale efficiently to thousands of processors, such as the molecular dynamics NAMD code [8].

The execution model of Charm++ is message-driven. This means that computations in Charm++ are triggered based on arrival of associated messages from remote processors. This message-driven execution turns out to be a very useful mechanism for hiding communication latency in the system. It allows adaptive overlap of computation and communication which results in great performance benefits for the parallel applications. Charm++ provides system calls to asynchronously create remote chares and to asynchronously invoke functions on remote chares by sending messages to those chares. Such functions are called *entry methods* in Charm++ terminology. This asynchronous message passing is the basic interprocess communication mechanism in Charm++. There is a scheduler for each Charm++ program which chooses a message from the available pool of messages and executes the computations associated with that message.

The Charm++ runtime system has the ability to migrate chares across processors during execution of an application. This migration capability is used by the powerful measurement-based load-balancing mechanism of Charm++ [13]. The runtime system can measure various parameters in the chares, such as computational load or communication frequency and volume. Charm++ provides a family of load balancers, targeting optimization of a variety of metrics. The user simply needs to select her desired balancers at application launch. The metrics being optimized can be computation load, communication volume or both. During execution, the selected balancers will collect the measured chare values for the appropriate metrics, and dynamically remap chares across the available processors in a way that execution performance is optimized. This dynamic optimization capability is critical for applications such as particle system simulators, where particles can move in space and cause overloading on a given processor as the simulation progresses, while other processors become underutilized.

## 2.2 Major ParallelGravity Features

One of the early decisions in the design of ParallelGravity was to select where to compute the forces applied to a bucket of particles. Historically, there has been two main methods for that: (a) distributing the computation of the forces on that bucket across all processors, with each processor computing the portion of the forces given by its subtrees, or (b) gathering at the processor owning that bucket all the data needed to compute the forces on it. We decided to adopt the second scheme, since the capabilities of Charm++ could be further

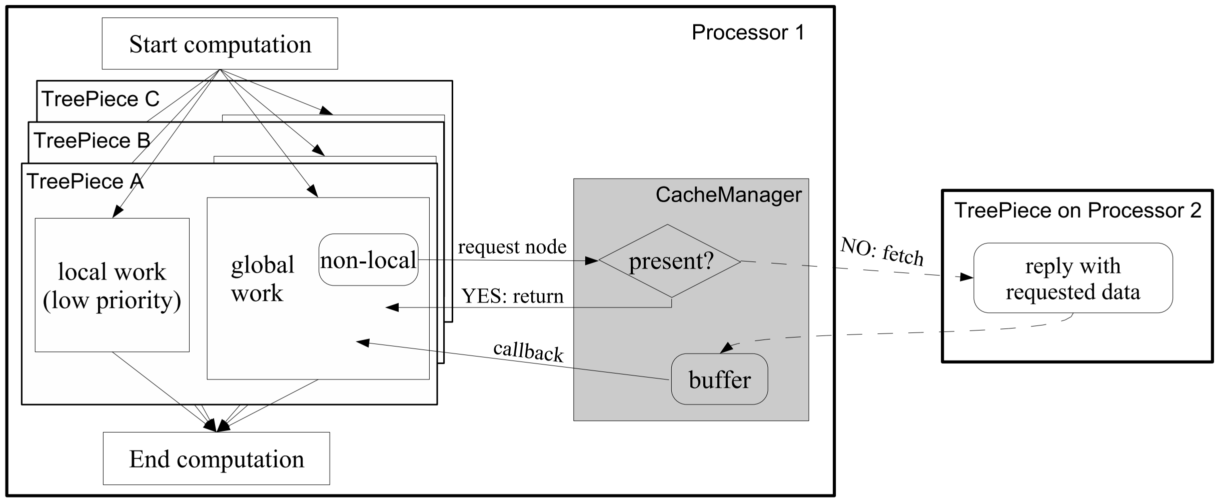


Figure 2.1: Control flow of the execution of an iteration of force calculation

exploited, as explained later in this section.

In our implementation of ParallelGravity, each TreePiece is a Charm++ chare. Thus, TreePieces are dynamically mapped to physical processors by the Charm++ runtime system. The overall structure of how the code works is shown in Figure 2.1, and described in the next paragraphs.

To perform the computation of the forces on its particles, a TreePiece processes its buckets independently. For each bucket, the TreePiece must walk the overall tree and compute the forces due to all other particles. During the walk, visited nodes may be local (i.e. owned by this TreePiece) or non-local. For local nodes, the force computation can proceed immediately. For non-local nodes, a retrieval must be carried out, to bring the corresponding data into the TreePiece. A non-local node may reside either at another TreePiece of the same processor, or at a remote processor. In the first case, we use a direct data transfer between chares. In the second case, data must be requested to the remote processor. While waiting for remote data to arrive, the TreePiece can process other buckets.

Instead of repeating fetches of the same remote node for different bucket walks, we can use the property that buckets close in space will require similar remote portions of data. Therefore, we can buffer the imported data and have it used by all buckets in the TreePiece before discarding it. Because in Charm++ we may have more than one chare in a single processor, we implemented this optimization at the processor level so that remote data is used by the buckets of all the TreePieces. This has been realized using a Charm++ *group*, which we call *CacheManager*.

The purpose of the CacheManager is to serve all requests made by the TreePieces, and provide a caching mechanism to hide the latency of interprocessor data fetching. The CacheManager implements a random access to the cached data through the use of a hash table. To reduce the overhead of table lookup, the imported data is reconstructed into a local tree. Thus, once entering a subtree, TreePieces can iterate over direct pointers, until another cache miss occurs. Upon detecting a miss, the CacheManager will fetch the remote data and use callbacks to notify the requesting TreePiece when the data arrives. More advanced features provided by the CacheManager are presented in the next section.

Because Charm++ executes chare methods in a non-preemptive fashion, a long sequence of consecutive tree walks might potentially prevent a processor from serving incoming data requests from other processors. In order to provide good responsiveness to incoming requests, we partitioned the processing of tree walks with a fine granularity. The grainsize is a runtime option, and corresponds to the number of buckets that will walk the tree without interruption. After that number of walks is performed, the TreePiece will yield the processor, enabling the handling of existing incoming data requests.

While dividing the computation into fine grains, we also distinguish between *local* and *global* computation. Local computation is defined as the interaction with the particles present in the same TreePiece. In contrast, global computation is defined as the interaction with the rest of the tree, i.e. the computation that involves non-local nodes. In particular, because this global computation is performed on the imported sections of the tree, it is on the more critical path. To express this different criticality, we utilized the prioritization mechanism embedded into Charm++. This mechanism allows establishing a total order of priority for the different operations performed by a TreePiece: the highest priority is assigned to accepting requests arriving from other processors, followed by sending replies to such requests, and finally the two types of computation (local and global), with the local one having the lowest priority. The Charm++ runtime system will schedule these operations according to such priorities.

## 2.3 Optimizations

After having a basic version of ParallelGravity in place, we studied its performance and added a number of optimizations to the code. Some of these optimizations were designed to exploit Charm++ aspects that enable high performance, whereas others were aimed at specific characteristics of particle codes. In this section, we describe the various optimizations that we have added, and present, in each case, the performance improvement that we obtained by applying such techniques to real cosmological datasets.

### 2.3.1 Software Cache Mechanism

As mentioned in Section 2.2, the CacheManager not only reduces the number of messages exchanged to fetch remote data, but also hides the latency of fetching data from other processors. We evaluated the effectiveness of the CacheManager on various real datasets running on varying numbers of processors. Our results show that the CacheManager dramatically reduces the number of messages exchanged. The performance improvement due to sending a much lower number of messages, combined with the latency-hiding effects of the CacheManager, produces a sharp reduction in the execution time of ParallelGravity. Thus, the software cache mechanism is absolutely necessary to obtain good parallel performance.

### 2.3.2 Data Prefetching

We talk about two kinds of prefetching that we have implemented in ParallelGravity. The first kind is really not prefetching. It involves fetching something more than what is needed, which will most likely be used sometime in the future. We start with the description of the first kind in the following paragraph.

As in PKDGRAV, we can take the principle of the software cache one step further by fetching not only the node requested by a TreePiece, but pro-actively also part of the subtree rooted at that node. The user can specify the *cache depth* (analogous to the concept of cache line in hardware) as the number of levels in the tree to recursively prefetch. The rationale for this is that if a node is visited, most probably its children will be visited as well. This mechanism of prefetching more data than initially requested helps to reduce the total number of messages exchanged during the computation. Since every message has both a fixed and a variable cost, prefetching reduces the total fixed cost of communication. On the other hand, a cache depth of more than zero might cause some data to be transferred but never used, thus increasing the variable part of the cost. Our performance results show that there is an optimal value of cache depth, at which the execution time is minimal. The optimal point is achieved when the fixed cost and the variable cost of transferring data over the network are in balance.

If a TreePiece requested data to the CacheManager only when required by the tree-walk computation, the CacheManager might not have it. This would trigger a fetch of the data from the remote node, but at the same time it would suspend the computation for the requesting bucket until the moment of data arrival. Both the interruption of the tree walk and the notification from the CacheManager incur an overhead. To limit this effect, we developed a *prefetching phase* which precedes the real tree-walk computation. During this phase, we traverse the tree and prefetch all the data that will be later used during the computation in the regular tree walk. This prefetching phase can work with different cache depths.

From our experiments, we observed that the prefetching phase improves performance for all considered values of cache depth. This is due to the increased hit rate of the cache. The hit rate increases to near 100% with the prefetching phase from about 90% without any prefetching.

We define the *relevance* as the ratio between the number of nodes fetched and used, and the total number of nodes fetched. Ratios closer to 1.0 represent a better relevance. We observe that relevance decreases with increasing cache depth, leading to unnecessarily higher memory consumption. Nevertheless, this higher memory consumption due to caching is limited to a fraction of the total memory footprint for moderate values of cache depth. At a very low value of relevance, the cost of fetching a large amount of extra data is not offset by the benefit of having the data already present in the software cache when it is requested. This is why we observed a rise in the execution time for large values of cache depth. The prefetching phase does not affect the relevance, since it does not change which data items are transferred. Prefetching simply causes those data transfers to occur earlier.

Thus, we see that using the prefetching phase along with a small but non-zero value of cache depth improves performance. In the following subsections, we will assume that the prefetching phase is active, and a reasonable value of cache depth is used.

### 2.3.3 Remote Chunks

We take the idea of prefetching to a higher level for the purpose of performance improvement. Instead of having a single prefetching phase, we'll have multiple prefetching phases.

We divide the remote computation into parts which we call *chunks*. In each prefetching phase, we prefetch a single remote chunk. This is followed by the computation of buckets with that chunk. Prefetching of the next chunk is carried out in parallel with the computation of previous chunk.

Our results show an improvement with remote chunks, but the full effects on the entire execution time are more complex and will require more detailed studies to be fully characterized.

### 2.3.4 Tree-in-Cache

In Section 2.2 we introduced the concept of local and global computation.We pointed that the global work is on the critical path, and that the local work can be used to hide the latency of data transfers. From this, it is clear that we should have as much local work as possible. One point to notice is that in the Charm++ environment we fragment the particle dataset in more TreePieces than the number of physical processors available. This over-decomposition reduces the amount of local work per TreePiece. In some of our experiments, when increasing the number of processors beyond one hundred, the local work became insufficient to maintain the processor busy during the entire computation.

By noticing that during the force computation there is no migration of TreePieces, we can consider collectively all the TreePieces residing on a given processor. We can attribute to local computation not only the work related to nodes/particles present in the same TreePiece, but also the work related to particles and nodes present in other TreePieces in the same processor. This is implemented by having each TreePiece registering to the CacheManager at the beginning of the computation step. The CacheManager will then create a superset tree including all the trees belonging to the registered TreePieces. Each TreePiece will now consider as local work this entire tree. During this operation, only the nodes closest to the root of the tree will be duplicated. According to our tests with datasets of a few million particles, less than one hundred nodes were duplicated.

Our experiments show that the percentages of local and global work changed considerably before and after this optimization. In our tests, this new scheme enabled scaling the computation up to hundreds of processors. However, when reaching the limit of one thousand processors, even the extra work from co-resident TreePieces becomes insufficient. The solution proposed in section 2.3.3, of splitting the global walk into multiple sub-walks (remote chunks), seems to provide the necessary infrastructure to scale beyond the limit of thousand.

# Chapter 3

# New Scheme for organizing force computation

In this chapter, we describe a faster scheme for calculating gravitational forces that has been implemented in ParallelGravity. This scheme has been implemented over and above all the basic features and optimizations that have been described in chapter 2. The implementation is based on the scheme involving *interaction lists* that was introduced in Stadel's thesis [12].

## 3.1 Overview

After having preceded the computation with a prefetching phase and making many types of optimizations to the code, we explored a faster algorithm (presented in [12]) for gravitational force computation. The new algorithm is based on the same principle of the CacheManager: two buckets close in space will tend to interact similarly with a given remote node.

In the regular ParallelGravity algorithm, whenever a bucket walk visits a tree node, a fundamental test is carried out. In this test, we check the spatial position of the bucket in respect to the particles in that node. If the bucket is sufficiently *far* from the node, the forces on the bucket due to the entire subtree rooted at that node are immediately computed, using the subtree's center of mass. Otherwise, ParallelGravity *opens* the node, i.e. it recursively traverses the subtree rooted at that node. Thus, the threshold used to decide if a node is close enough to the bucket represents the *opening criteria* for deciding whether the visited node must be opened or not, for the bucket being considered.

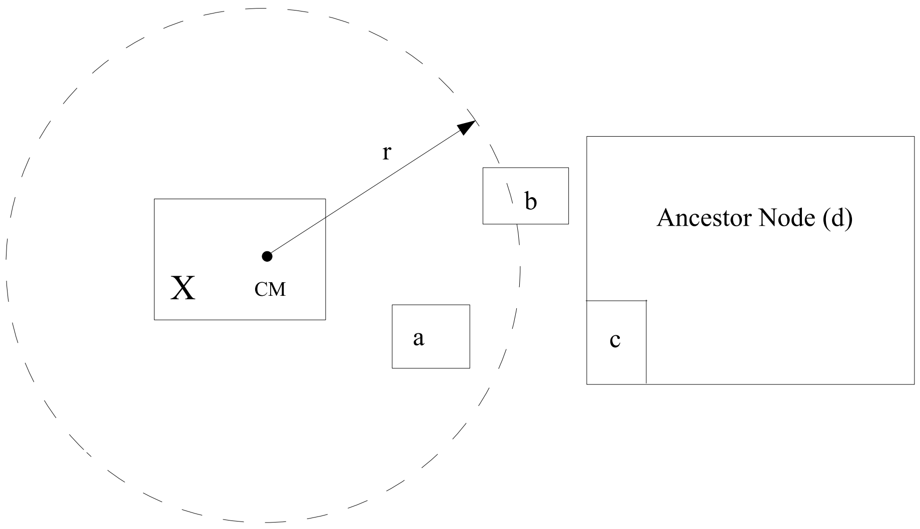


Figure 3.1: Illustration of opening criteria of a node

Figure 3.1 illustrates the opening radius $r$ for a node $X$. The multipole of node $X$ is placed on the interaction list of a bucket, if the bucket lies outside of the opening radius $r$ of the node. Only bucket $c$ satisfies this condition. Both the buckets, $a$ and $b$ satisfy the opening criteria and must open the node into two sub-nodes for future consideration. Bucket $a$ lies completely inside the opening radius whereas bucket $b$ is intersected by the opening radius. Some of the particles in bucket $b$ would have accepted the multipole of node $X$, but the benefits of amortizing the tree walk over all particles of a bucket far outweighs such a consideration. We observe that the multipole of $X$ is acceptable to all the buckets of ancestor node $d$. This observation forms the basis of the algorithm described later. Note that the opening radius of a node is given by some constant factor times the distance from the center-of-mass (CM) to the most distant corner of the node.

Instead of checking the opening criteria at a given node for each bucket independently, we can modify the algorithm and do that check for various local buckets at once. We can do this collective check using the buckets' ancestors in the local tree. These ancestors will be local nodes containing particles which are close in space. If an ancestor needs to open a visited node, that node will be opened for every bucket that is a descendent of such ancestor. On the other hand, if a node is far enough for that ancestor, this node will be far enough for all the ancestor's buckets too. In this second case, we can directly compute the interaction between the node and all these local buckets. Checking the opening criteria can also give a third answer, which is uncertainty, due to the greater distribution of the particles in the space. This means that the node will open for some buckets under this ancestor, and not for others. In this case we will need to split our ancestor and look at a smaller ancestor (containing less number of buckets) to get the answer.

By grouping the checking for various local buckets, we can reduce the total number of checks for opening nodes. A potential problem in this modified algorithm is that it may cause less effective usage of the hardware cache: because the computation of interactions proceeds for various local buckets, one bucket's data may flush another bucket's data from the hardware cache. We can reduce the number of hardware-cache misses by storing all the nodes that interact with a given bucket in a bucket's *interaction list*, and perform the entire computation of forces on that bucket at the end of the tree walk. Note that this is different from the original ParallelGravity code. In the original version, the force computation on a bucket takes place as we walk the tree for that bucket. That is, when we reach a node with whom forces need to be computed, we compute them there itself.

Interaction lists result in performance improvement, also due to the following two reasons. Firstly, modern compilers may keep a particle's data inside CPU registers while computing interactions with the nodes in the list. Also, there is a minimal amount of change in the interaction lists as we move from one bucket to another. This results in an increase in cache utilization of the processor and consequently, better performance.

## 3.2 Basic Algorithm

The entire algorithm is structured as two tree walks. Outer walk is the one over the tree containing all the buckets owned by the local TreePiece (or the local tree) and inner one over the global tree covering the whole space. Our aim is to compute force interactions for buckets of the local tree. Force computation takes place when we reach a bucket in the local tree.

The algorithm is based on the observation that two nearby buckets will show only a small number of differences in their interaction lists, namely in the closest particles and in the closest nodes. The algorithm uses two rules to walk the local tree top down in a single pass for all the buckets, making minimal required number of changes to the interaction lists. The rules being: If an ancestor is contained entirely by the opening radius of a node i.e. the node needs to be opened, the node will be opened for all the descendent buckets of such an ancestor. Also, if an ancestor lies entirely outside the opening radius of a node i.e. node is far enough for that ancestor, then the node is far enough for all the ancestor's buckets too and the *multipole* of the node is acceptable to all the buckets. This means that we can directly compute the interactions between the node and the buckets, due to the subtree rooted at the node, using subtree's center of mass.

The algorithm recursively descends the local tree from the root node to the buckets in a depth-first fashion. This tree walking procedure produces two interaction lists, a *cell list* (c-list) and a *particle list* (p-list), which are subsequently evaluated to determine the gravitational force on the given bucket. Cell list contains all the nodes and particle list contains all the particle buckets with whom force computation will take place. A *checklist* of nodes which still need to be considered for inclusion on either of the interaction lists is processed by the algorithm for an ancestor before descending deeper in the tree. If the opening radius of a node on the check list entirely contains the current ancestor we are considering then we remove the node and add its two child nodes to the end of the check list. If the opening radius of the node does not intersect the ancestor, it is removed from the check list and added to the current interaction list. If neither of these two cases apply, then the node remains on the check list and is considered when we proceed deeper in the local tree. Once we have processed whole of the check list in this fashion, we recursively descend the tree to current ancestor's children. When we have descended to a bucket, we open the check list node if the opening radius of the node intersects the bucket. This means if the bucket lies entirely within or is intersected by the opening radius, the node is opened. This condition ensures that we end up removing all the nodes from the check list and we have a complete interaction list for the bucket at the end.

Algorithm 1 presents pseudo-code for the recursive form of the algorithm described above. The for loop in the algorithm is not fixed since checkList gets modified within the loop itself. Note that 'interactList.plist' stands for the particle interaction list and 'interactList.clist' stands for the node interaction list. Also, $ancestor \rightarrow lower$ and $ancestor \rightarrow upper$ refer to the two children of ancestor. Other functions used in the algorithm are briefly described below.

- intersect(x,y): It finds out if the opening radius of node y intersects node x.
- isBucket(x): It returns true if the node x is a bucket.
- contained(x,y): It returns true if node x is contained entirely within the sphere inscribed by the opening radius of y.
- calculateGravity(part,interact): It carries out actual force computation between the particles of the bucket (part) and the interaction lists.

**Algorithm 1:** Walk(ancestor,checkList,interactList)

```
begin
    for $c \in checkList$ do
        if $intersect(ancestor, c)$ then
            if $isBucket(ancestor) \bigvee contained(ancestor, c)$ then
                $checkList = checkList - c$;
                if $isBucket(c)$ then
                    $interactList.plist = interactList.plist + particles(c)$;
                else
                    $checkList = checkList + children(c)$;
        else
            $checkList = checkList - c$;
            $interactList.clist = interactList.clist + moments(c)$;
    end
    if $isBucket(ancestor)$ then
        $calculateGravity(particles(ancestor), interactList)$;
    else
        $Walk(ancestor \rightarrow lower, checkList, interactList)$;
        $Walk(ancestor \rightarrow right, checkList, interactList)$;
end
```

## 3.3 Implementation

The new algorithm for force computation using interaction lists has been implemented as part of ParallelGravity, the parallel N-body cosmological force simulator developed using Charm++. The new version of ParallelGravity has a compile-time option, by specifying which, we can compile the new algorithm instead of the original force computation algorithm of ParallelGravity. The implementation of the algorithm is iterative in nature, as against the recursive form of the algorithm presented in algorithm 1. The iterative implementation is somewhat complex as compared to the straightforward recursive algorithm, but has gains associated with it. It avoids memory explosion and does away with the function calling overhead of a recursive implementation. Moreover, with the existing implementation of tree walk in ParallelGravity, we found it easier to implement our algorithm in an iterative fashion.

The basic top-level flow of our implementation is similar to the original algorithm. Like before, the force computation is divided into *local computation*, interaction with particles in the same TreePiece and *global computation*, interaction with the rest of the global tree. So, we have separate check lists and interaction lists for local computation and global computation. From now onwards, we'll have a single interaction list to stand for both node and particle interaction lists.

By keeping only one check list and one interaction list for the entire TreePiece, it is not possible to implement our algorithm iteratively. The reasoning for this is described here. As we go from one bucket to another while walking the local tree in our recursive algorithm, both check list and interaction list undergo changes. For example, when we go from bucket node $A$ to bucket node $B$ in figure 3.2, we undo all the insertions and deletions that take place in the check list and the interaction list in the path from $A$ to internal node $C$ and then, make all the changes in the lists while walking from $C$ to $B$. To keep track of all these changes, we need to keep a check list and an interaction list at each level of the tree. The check list and the interaction list for a child are constructed by making changes to the check list of the parent.

As we walk the local tree, new nodes are added to the check list which is at the current level of the tree. The check list of level $l$ is constructed by making changes to the check list of level $l-1$. When we reach level $l$, we parse the nodes contained in the check list at level $l-1$, either adding them to the check list at level $l$ or to the interaction list at level $l$. When we

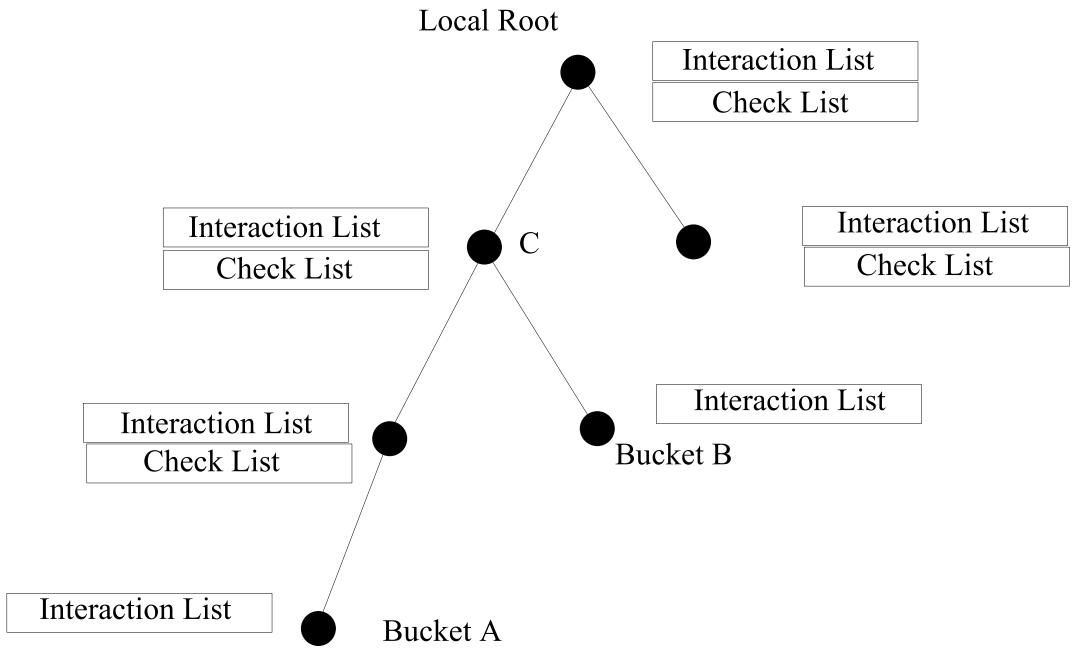


Figure 3.2: Implementation of Interaction list scheme

reach a bucket, the check list at the previous level is parsed to add nodes (or particles) only to the interaction list at the bucket level. Figure 3.2 shows a tree owned by a TreePiece with both check lists and interaction lists at all the levels. When going from bucket $A$ to bucket $B$, we go up to their common ancestor $C$ and then, walk down the branch to $B$ starting with the check list present at $C$. To calculate the actual forces on reaching a bucket, we traverse the interaction lists at all the levels of the tree starting from the root to the bucket.

We process the buckets in order in our implementation to prevent the memory from exploding. Since we need to build the entire list of interaction that a bucket requires for the computation, if we are to process more than one bucket simultaneously, the memory consumption will quickly explode. For this reason, we process the buckets in depth-first order, as the tree walk visits them, and for each we add to its interaction list only the nodes that are present in the local processor. Those that are missed will be computed later during the regular callback from the cache. This allows us to reduce the memory to a single interaction list at a time. It is clear that the best case is obtained when no nodes are missed

Table 3.1: Characteristics of the parallel systems used in the experiments

| System Name | Number of Procs | Procs per Node | CPU Type | CPU Clock | Memory per Node | Type of Network |
|---|---|---|---|---|---|---|
| Tungsten, NCSA | 2,560 | 2 | Xeon | 3.2 GHz | 3 GB | Myrinet |
| BlueGene/L, EPCC | 2,048 | 2 | Power440 | 700 MHz | 512 MB | Torus |
| HPCx, HPC-UK | 1,536 | 16 | Power5 | 1.5 GHz | 32 GB | Federation |

during the walk, and further justify the previous optimizations (discussed in chapter 2).

## 3.4 Results

In this section, we present results for our new scheme of force computation which uses interaction lists. We compare the results with the results from the original version of ParallelGravity.

In our experiments, we used the parallel systems described in Table 3.1, and the following particle datasets:

- **lambs**: Final state of a simulation of a $71Mpc^3$ volume of the Universe with 30% dark matter and 70% dark energy. $144^3$ particles, i.e. nearly three million particles, are used (3M). Three subsets of this dataset are obtained by taking random subsamples of size thirty thousand (30K), three hundred thousand (300K), and one million particles (1M), respectively.

- **dwarf**: A snapshot at $z = .3$ of a multi-resolution simulation of a dwarf galaxy forming in a $28.5Mpc^3$ volume of the Universe with 30% dark matter and 70% dark energy. The effective resolution in the central regions is equivalent to $2048^3$ particles in the entire volume. The total dataset size is nearly five million particles.

First, we performed small experiments to compare the open criterion calls for original algorithm and algorithm with interaction lists. These experiments were performed on a single processor of the HPCx system. Note that the number of opening criterion calls are

Table 3.2: Number of checks for opening criteria, in millions

| | lambs 1M | dwarf 5M |
|---|---|---|
| Original algorithm | 120 | 1,108 |
| Modified algorithm | 66 | 440 |

independent of the number of processors, so they don't change with increase in the number of processor, though minor variations in the number do occur. Table 3.2 shows the number of checks that are observed with the two algorithms, executing on the HPCx system with our two datasets. As claimed in section 3.1, we see that the number of open criterion calls decrease with the new interaction lists version of the code. The decrease in the number of calls is pretty significant. For the lambs dataset, the decrease is about 45% and for the dwarf dataset, the decrease is about 60%.

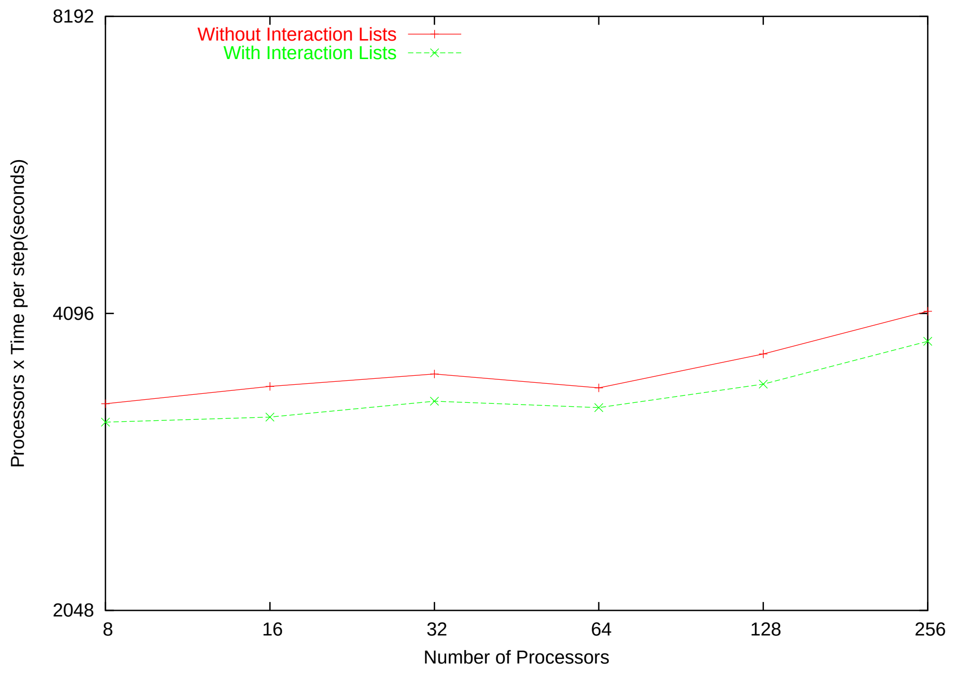


Figure 3.3: Comparison between regular ParallelGravity and ParallelGravity with interaction lists on BlueGene with the dwarf dataset

We performed comparisons between the regular ParallelGravity version and the ParallelGravity version with interaction lists on large parallel machines like BlueGene/L and HPCx. We show the results obtained on these machines in the following plots for both our datasets, the lambs 1M and the dwarf 5M particle dataset. The vertical axis of the shown plots is the product of the execution time per iteration and the number of processors in the simulation. The horizontal axis is the number of processors. Horizontal lines represent perfect scalability. So, scalability of the code is dependent on how horizontal the line is, in the plot. Also, note that both the axis in the plots have log scales.

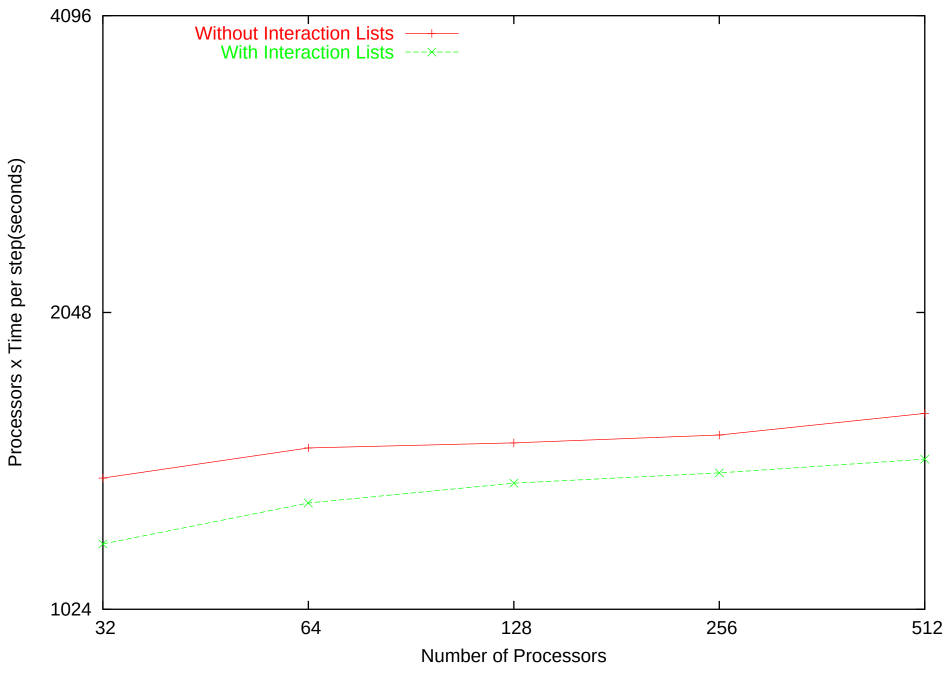


Figure 3.4: Comparison between regular ParallelGravity and ParallelGravity with interaction lists on HPCx with the dwarf dataset

Figure 3.3 plots the execution times of both the algorithms on BlueGene/L for the dwarf 5M dataset. We see that both the algorithms scale pretty well and the version with interaction lists scales as well as the original version on BlueGene/L. Also, ParallelGravity with interaction lists has a performance improvement over the entire range of processors. We don't show beyond 256 processors because both the algorithms don't scale well after that, since there is not adequate work available for each processor and hence, the gain is reduced. Also, when going from 128 processors to 256 processors, we can clearly see the scaling becoming worse. We observe that the performance improvement for interaction lists is around 10% for the entire range which is pretty good.

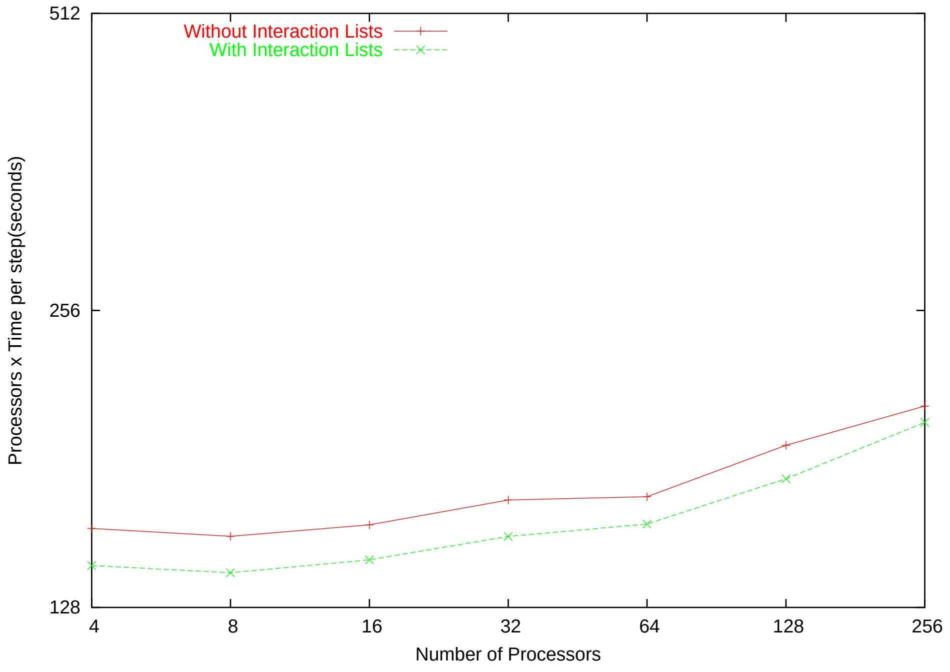


Figure 3.5: Comparison between regular ParallelGravity and ParallelGravity with interaction lists on HPCx with the lambs dataset

Figures 3.4 and 3.5 show the execution times of both the versions of the code on HPCx for the two datasets, dwarf 5M and lambs 1M. Both the codes pretty well on HPCx too. As seen for BlueGene/L, interaction lists version has a constant performance improvement over the original version for the entire range of processors. We don't scale beyond 256 processors for lambs 1M dataset and 512 processors for the dwarf 5M dataset since we run out of work. In fact, the scaling begins to worsen towards the end when going from 128 to 256 processors in figure 3.5. In both these plots too, the performance improvement for interaction lists is around 10%.

Hence, all the results show pretty decent improvement for the new scheme over the original force computation scheme. This performance improvement is partly due to the good cache utilization achieved with interaction lists, partly due to up-gradation of bucket data in the processor registers and partly due to reduction in the number of open criterion calls.

# Chapter 4

# Comparison of Particle Decomposition Techniques

This chapter describes three different particle-space decomposition techniques which lead to construction of different types of trees and compares them. These techniques, along with the trees, have been implemented in ParallelGravity. The user can choose between these different techniques by specifying a runtime option to ParallelGravity.

The program flow of ParallelGravity has a well-defined structure. Firstly, the particle data is *loaded* onto the processors. Since the number of particles run into millions, it is usually not feasible to load all the data onto a single processor. So, the data is split among all the TreePieces and loaded. The loading of the data is followed by *space domain decomposition* of particle-space. The third phase is the building of the Barnes and Hut trees for all the TreePieces. This chapter talks about the second and the third phase. After tree building finishes, we go into the gravitational force computation phase which is the core of the simulator. In the last phase, we update the particle positions, velocities and energies. This sequence of phases forms first *iteration* of our simulator. In the subsequent iterations, we carry out all but the particle loading phase. They are carried out in the same order. We need to perform particle decomposition and tree building again and again, since the spatial positions of the particles change after every iteration and consequently, the previous space decomposition becomes invalid making the tree incorrect too.

We see that particle-space decomposition along with tree-building forms a very significant part of our simulator. Hence, we have implemented various space decomposition techniques and tree-building techniques and studied their performance on different kinds of input datasets.

The central data structure in ParallelGravity is a tree which forms the hierarchical representation of the mass (or particle) distribution in space. Each node of this tree represents a rectangular sub-volume of the total simulation volume, containing the particles, center-of-mass, and higher moments of the mass distribution in the enclosed volume. The *root-node* represents the entire simulation volume, and as we proceed deeper into the tree we get a finer representation of the mass distribution where children represent smaller sub-volumes of the total volume. Each node is divided into sub-nodes until we reach the leaves of the tree which are buckets and contain only a few particles. The tree we construct, in our case, is a binary tree, although some tree-codes use a *oct-tree*. We intend to implement oct-trees in ParallelGravity in future.

The space decomposition techniques that we have implemented are: Space Filling Curve (SFC) Decomposition, Oct-tree Decomposition, and Orthogonal Rectangular Bisection (ORB) Decomposition. There are two types of binary trees implemented by us: Spatial Binary trees and ORB trees. Note that in the following discussion, we'll use the word *domain decomposition* interchangeably with *space decomposition*.

During domain decomposition, we divide the particles into spatially local regions of approximately equal work. As outlined in [12], using a data structure for domain decomposition that does not coincide with the hierarchical tree for gravity calculation, leads to poor memory scaling with large number of processors and/or difficult book-keeping. Hence, the efficiency is greatly improved if the tree data structure matches the domain decomposition structure. So, we use ORB trees with ORB decomposition and Spatial Binary Trees with SFC and Oct-tree decomposition.

SFC decomposition along with the spatial binary tree was implemented as part of the original version of ParallelGravity. Oct-tree decomposition, ORB decomposition and ORB Trees have been implemented as part of this thesis. We now describe each of the decomposition techniques in detail followed by our experimental results.

## 4.1 SFC Decomposition

SFC stands for Space-Filling Curve. A Space-Filling Curve [9] is a continuous mapping from a $d$-dimensional space to a 1-dimensional space written as

$$f : N^d \rightarrow N$$

The $d$-dimensional space (or cube) is mapped onto a line such that the line passes through each point in the volume of the cube, entering and exiting the cube only once. A point in the cube can be described by its spatial coordinates, or by the length along the line, measured from one of its ends. In our case, we map our 3-dimensional space to the SFC curve. A particle in the space is described by its 3-dimensional coordinates. SFC imposes a total ordering on the particles based on the particle keys we generate from the spatial position of the particles. In our implementation, particle keys are 63-bit numbers in which 21 bits are derived from spatial position of the particle in each dimension. The key is constructed by mixing these 21 bits from each dimension.

The hierarchical tree structure which is suitable for SFC decomposition is the *Spatial Binary Tree.* A spatial binary tree is the tree constructed when we spatially bisect the bounding box containing all the particles in the volume. Since the box is spatially bisected, the number of particles in the resulting sub-boxes are usually not equal. One of the resulting sub-boxes can even be empty. The spatial binary tree goes well with SFC decomposition as well as Oct-tree decomposition of the particles which is discussed in the next section.

The keys assigned to the particles have a very nice property. From the prefix of the

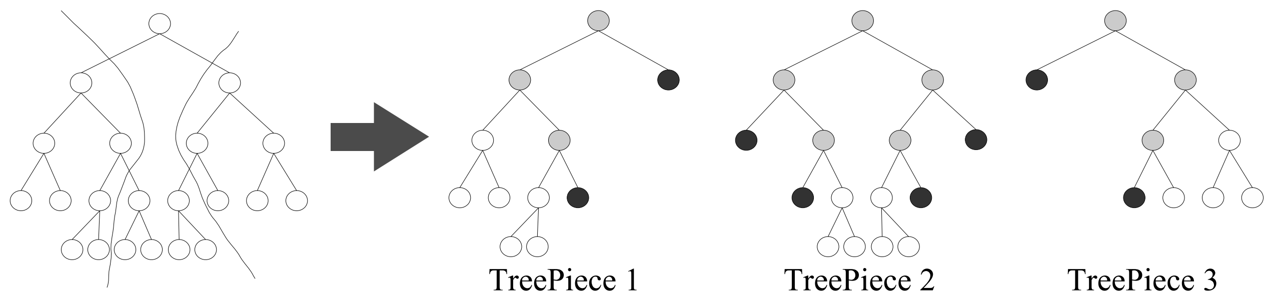


Figure 4.1: Distribution of a tree across TreePieces (top levels). White nodes are owned by one TreePiece, black nodes are placeholders for remote nodes, gray nodes are shared among multiple TreePieces.

particle keys (discussed above), we are able to determine which node the particle belongs to. The particle key (or a prefix of the key) traces the path of the particle from the root of the global spatial binary tree to the bucket.

A contiguous portion of the SFC curve is assigned to each TreePiece. The number of particles assigned to each TreePiece is the same. The resulting tree is shown in figure 4.1. The figure shows the global tree constructed over all the particles and the distribution of the particles among TreePieces. We see that some of the internal nodes are replicated in more than one TreePiece.

Figure 4.2 shows a simple example of a SFC assigning particles to TreePieces. As mentioned above, a contiguous portion of the curve is assigned to each TreePiece. In the figure, SFC curve is shown with the structure of the *spatial binary tree* in the background. The SFC curve is divided into 4 portions which are equal in terms of the number of particles. The tree goes deeper only in 2 of the 4 top-level boxes since it needs to determine the SFC curve splitting points. Those two boxes have a denser concentration of particles as compared to the other two. The figure shows each portion of the curve which is assigned to a different TreePiece in a separate pattern. The different patterns are shown in the legend.

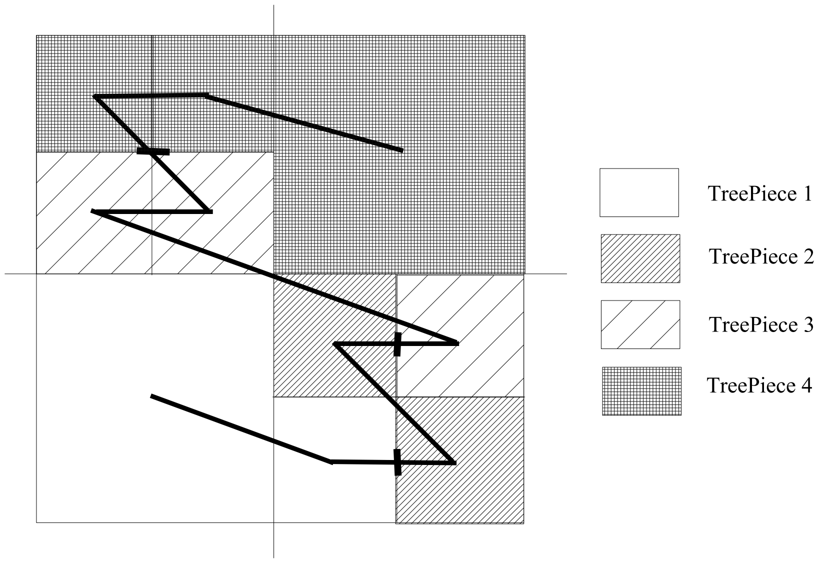


Figure 4.2: Assignment of particles to TreePieces according to Space Filling Curve (SFC)

## 4.2 OCT-Tree Decomposition

In *Oct-tree* decomposition, the particles are decomposed based on the nodes of a *spatial binary tree* covering the entire space. So, the decomposition data structure coincides completely with the hierarchical gravity tree data structure. We perform particle decomposition, first in $x$-dimension, followed by $y$-dimension and finally, in $z$-dimension. The decomposition tries to distribute the particles approximately equally among all the TreePieces. To do this, it bisects some parts of the space, containing higher densities of particles, more than the others. Note that we call this Oct-tree decomposition just to denote that it is based on the decomposition of a spatial tree. It doesn't mean that an *oct-tree* is constructed out of the decomposed sets of particles in the tree-building phase.

Figure 4.3 shows an example of a simple Oct-tree decomposition. The space bisecting

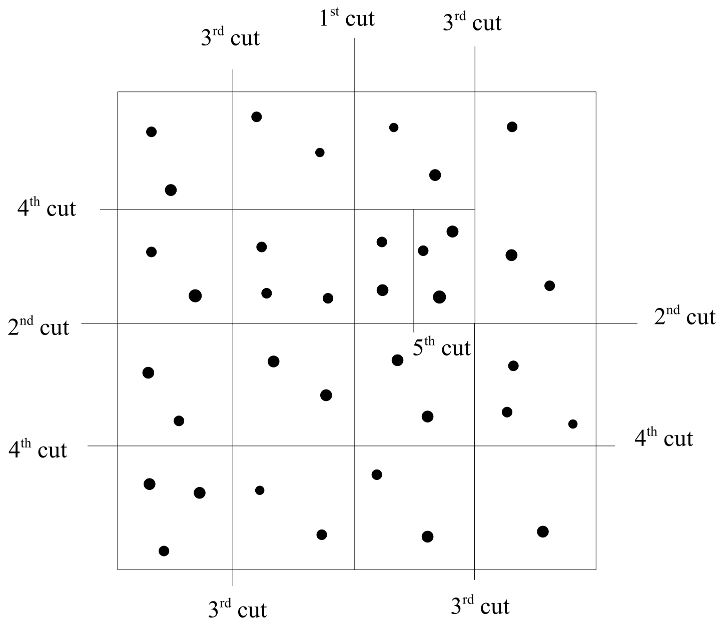


Figure 4.3: Assignment of particles to boxes according to Oct-tree Decomposition

*cuts* are also shown. The number of particles in each of the sub-boxes are not equal, as expected. Note that there is a $5^{th}$ cut for one of the boxes which has 5 particles which is much higher then the combined total of 3 particles which the neighboring box has after $3^{rd}$ cut. The $5^{th}$ cut bisects the box so that the two sub-boxes have 2 and 3 particles. There is no cut after the $3^{rd}$ cut for the neighboring box as the number of particles is already low.

A convenient property that is present in Oct-tree decomposition and absent in SFC decomposition is described here. For the spatial binary tree built out of the particle distribution derived from Oct-tree decomposition, an entire subtree always belongs to the same TreePiece and nothing else goes into that TreePiece. This can be seen in figure 4.4, which shows the distribution of particles for Oct-tree decomposition. As a result of this convenient property, only some top-level internal nodes are replicated in more than one TreePiece. The number of

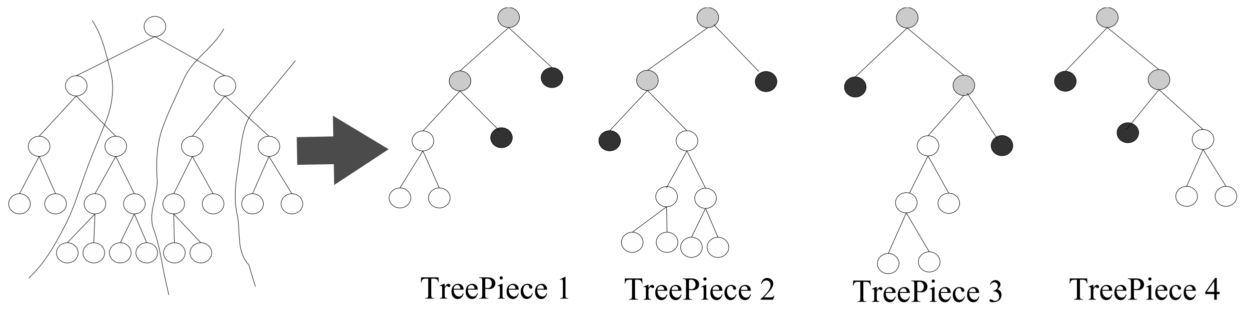


Figure 4.4: Distribution of a tree across TreePieces (top levels). White nodes are owned by one TreePiece, black nodes are placeholders for remote nodes, gray nodes are shared among multiple TreePieces.

internal nodes replicated in this case are not as many as is the case with SFC decomposition.

The particle keys used in Oct-tree decomposition are the same ones that we used in SFC decomposition. There are two purposes that a particle key solves in Oct-tree decomposition. Firstly, a particle key identifies the path from the root of the tree to the bucket in which the particle is contained. Secondly, a particle key also makes the sorting of the particles implicit when spatially decomposing a box in Oct-tree decomposition. One just needs to sort the particle keys once in the beginning. During Oct-tree decomposition, no more sorting phase is needed since the particles are always sorted spatially after the initial sorting.

To implement Oct-tree decomposition in ParallelGravity, we implemented a *Oct decomposer* algorithm which carries out the Oct-tree decomposition and tries to balance the particles in the sub-boxes. Our algorithm obtains a loose bound on balancing of the particle distribution. The algorithms which can guarantee a tighter bound are more complex to implement and involve a larger running overhead.

The outline of the algorithm we employ is shown as a pseudo-code in algorithm 2. The algorithm calculates a set of splitter keys based on an initial estimate of the sub-boxes. The keys are broadcasted to all the TreePieces and the total number of particles within each sub-box are evaluated using a global reduction. There is a Charm++ *chare* object which controls this broadcast and reduction of data. Once we have the counts within each box, a function called *weightBalancer* (shown in algorithm 3) is invoked, that recalculates the new boxes balancing the particle counts. The function balances the boxes of the entire space in many iterations. In each iteration, the function tries to find the *heaviest* box and two sibling boxes that are the *lightest*. The *heaviest* box is the box with the maximum number of particles and the *lightest* siblings are the sibling boxes in which the sum of the number of particles is minimum. If the function succeeds in finding both the heaviest box and the lightest sibling boxes and if the number of particles in the heaviest box is greater than the number in lightest siblings, the heaviest box is bisected and the two sibling boxes are coalesced together. The weightBalancer continues doing this in all subsequent iterations until the heaviest box it finds has lower number of particles than the lightest siblings.

**Algorithm 2:** OctDecomposer(nodeKeys)

```
begin
    while 1 do
        //Convert the Node Keys to spatial boundary keys to be broadcasted;
        $boundaryKeys = convertNodesToSplitters(nodeKeys)$;
        //Evaluate the particle counts in all the TreePieces and do the reduction;
        $binCounts = treeProxy.evaluateBoundaries(boundaryKeys)$;
        $newNodeKeys = weightBalancer(nodeKeys, binCounts)$;
        if $newNodeKeys == nodeKeys$ then
            $break$;
        else
            $nodeKeys = newNodeKeys$;
end
```

Convergence is guaranteed. If some box got bisected in one step, it was the heaviest at that point of time. No other box, including the lightest box being created, was heavier than it. The function doesn't converge if there are oscillations, which means the bisected box should get joined at some later point of time. But this can't happen since the box will definitely be heavier (or at most equal in weight) than any heaviest box at a later time. This means that the box won't get joined at a later time at all. Similar argument is valid for the lightest box too. Therefore, there are no oscillations. The weightBalancer gives a good balancing of the particle counts in practice, though it is tough to arrive at a theoretical bound on the distribution of the particles.

**Algorithm 3:** weightBalancer(nodeKeys,weights)

```
begin
    while 1 do
        $heaviest = findHeaviest(nodeKeys, weights)$;
        for $n \in nodeKeys$ do
            $curLightest = findLightestParent(curLightest, n, n.next)$;
        if $heaviest > curLightest$ then
            $nodeKeys.erase(heaviest)$;
            $nodeKeys.add(heaviest.left, heaviest.right)$;
            $nodeKeys.erase(curLightest.left, curLightest.right)$;
            $nodeKeys.add(curLightest)$;
        else
            $break$;
end
```

## 4.3 ORB Decomposition

ORB stands for Orthogonal Rectangular Bisection. An ORB decomposition step divides a simulation volume into two sub-volumes having approximately equal number of particles. ORB decomposition decomposes the entire simulation volume recursively until we have sub-volumes for each TreePiece of approximately equal number of particles. The decomposition

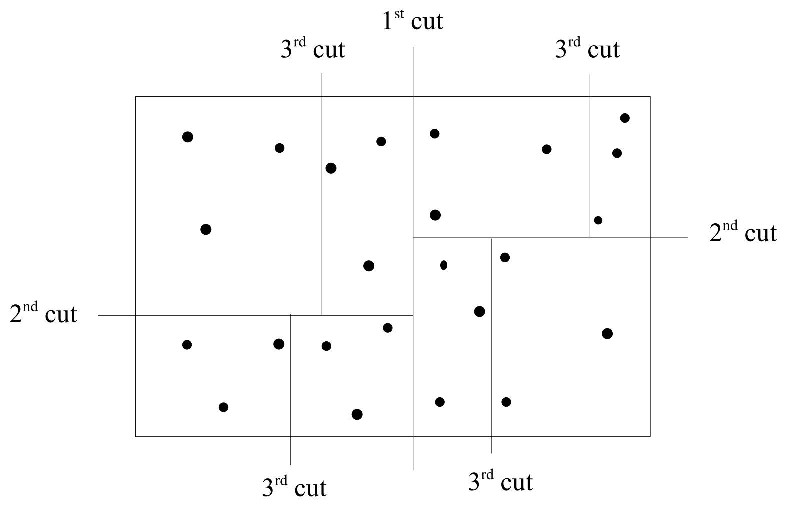


Figure 4.5: Distribution of particles according to ORB Decomposition

is always carried out in the longest dimension of the current simulation sub-volume. Though the number of particles are similar in each of the sub-boxes, the sizes of the boxes can vary significantly since the particle densities are variable in real datasets. The strategy of always decomposing the longest dimension makes the sizes of the boxes more even, as compared to strategy of decomposing the box along the three dimensions in serial circular order.

Figure 4.5 shows a simple example of 2-dimension ORB decomposition. We see that the number of particles in each of the 8 sub-boxes are equal whereas the shapes of the boxes are very different.

The gravity tree structure which is most suitable for ORB decomposition is an *ORB tree.* An ORB tree is constructed by bisecting the number of particles for each sub-volume of particles, which is similar to what ORB decomposition does. The top-level distribution of the particles for ORB tree built out of ORB decomposition is same as the one for Oct-tree decomposition. Figure 4.4 shows this distribution. The entire subtree below an internal node belongs to only one TreePiece, so, internal nodes are not replicated in more than one TreePiece.

The particle keys that we used in 4.1 and 4.2 are no longer valid for ORB decomposition. It is because the keys had been constructed using spatial position of the particles and they traced the path of the particle from the global root to the bucket in the spatial binary tree. Hence, they had acted as domain splitters to split the particles in spatial binary tree. Whereas in our case, the bisector of the box is not determined by the particle splitter keys and consequently, those keys don't give us any information about the path of the bucket from the root. Hence, they don't identify which box our particle belongs to, which was the main purpose a particle key served for SFC and Oct-tree decomposition.

For ORB decomposition, the particles contained in the box to be split, have to be sorted in the longest dimension before decomposition. There is no scheme which can assign keys to particles and meet this sorting requirement of ORB decomposition. Though ORB decomposition can be carried out easily without any requirement of particle keys, we designed a new methodology to assign keys to particles in ORB decomposition to help us in identify particles during the gravity force calculation phase. To assign keys to particles in ORB decomposition, we first have to assign keys to nodes as and when they are constructed in the tree-building phase. When we reach buckets, we calculate the number of particles in each bucket and assign keys to them based on the number. The prefix of the particle key is the key of the bucket node with the trailing part being the particle number of the particle in the bucket.

We now describe the ORB decomposition algorithm that we have implemented as part of ParallelGravity. We follow it up by the description of the tree-building phase. At every step in the algorithm, we always bisect the longest dimension of the simulation box. Also, when each TreePiece builds its local part of the ORB tree (in the tree-building phase), bisection takes place at the longest dimension only. The ORB decomposition algorithm has been outlined in algorithm 4. The main controller is a Charm++ chare object which initiates the decomposition. The algorithm does $log(n)$ iterations where $n$ is the number of boxes which the algorithm produces. This is also made equal to the number of TreePieces. In each iteration, the number of boxes increase by a factor of 2.

At the start of each iteration, the chare calculates the potential splitting positions $p_i$ in the longest dimensions $d_i$ for all the current boxes, for $i$ varying from 1 to the current number of boxes $n_c$. It broadcasts these $p_i$ to all the TreePieces. The TreePieces evaluate the counts of particles in each box based on these $p_i$ values. This is followed by a reduction of these counts to the main chare. On receiving these global counts, the chare determines if the counts of both the sibling boxes for each $p_i$ are equal within a tolerance value. If the counts are equal for all the $p_i$ values, then we are done with this iteration and we proceed to the next iteration. Before proceeding, we update the data structures to increase the number of current boxes to $2 * n_c$. In case the counts are unequal for at least one $p_i$, the $p_i$ values for which the counts are unequal, are updated and the new $p_i$ values are again broadcasted to the TreePieces which again evaluate the new positions and reduce the new found particle counts. This goes on till we are able to find a good estimate of all $p_i$ values for which counts of sibling boxes are almost equal. After $log(n)$ iterations, we have the required number of boxes and then, we initiate the data movement phase. Each TreePiece has the knowledge of the splitter positions and knows exactly what particles it has to send to each TreePiece. We could have carried out the data movement phase after every iteration but it would have resulted in much larger volumes of data being exchanged among processors leading to higher communication latencies. Having data movement at the end does make the implementation more complex but the gains in communication cost which we get, are too lucrative to ignore.

**Algorithm 4:** ORBDecompose(initialBox,numTreePieces)

```
begin
    $boxes[1] = initialBox$;
    $n_c = 1$;
    for $iteration \leftarrow 1$ to $\log numTreePieces$ do
        for $i \leftarrow 1$ to $n_c$ do
            $< p_i, d_i >= getSplittersInLongestDim(boxes[i])$;
        end
        //particleCounts is twice the size of p and d;
        //Evaluate the particle counts in all the TreePieces and do the reduction;
        $particleCounts = treeProxy.evaluateParticleCounts(p, d)$;
        while $!nextIteraion$ do
            if $countsEqualWithinTolerance(particleCounts)$ then
                $boxes = getNewBoxes(p, d)$;
                $n_c = n_c * 2$;
                $nextIteration = true$;
            end
            else
                for $i \leftarrow 1$ to $n_c$ do
                    $p_i = updateSplitters(p_i, boxes[i])$;
                end
                $particleCounts = treeProxy.evaluateParticleCounts(p)$;
            end
        end
    $treeProxy.broadcastFinalSplitters()$;
    //Carry out the data movement phase here;
end
```

We start building the ORB tree for each TreePiece from the global root. The global root has the whole space as its bounding box. We go on building the tree till we reach the root of the subtree which is completely local. As shown in 4.4, the nodes above the root of the subtree are either remote nodes (black) or nodes that are shared among multiple TreePieces (gray). Starting from the root of the subtree which is local, we start decomposing the particles in a similar fashion as we did in ORB decomposition. Only difference is that since all the particles being decomposed are local, we don't have to broadcast and reduce data. Once we split a box, we *squeeze* both the boxes based on the particles contained in them. Squeezing produces a significantly better tree structure.

## 4.4 Comparison Results

In this section, we present the comparison results of the three types of domain decomposition we described in the previous sections.

The particle decomposition phase is more complex for ORB as compared to SFC and Oct-tree. In SFC and Oct-tree, simple histogramming over all the TreePieces is performed to arrive at the correct splitter keys. Oct-tree, in addition, has the weightBalancer which tries to balance the count of particles in all the boxes. The weightBalancer performs pretty well in practice and each call to weightBalancer usually takes $O(n)$ time. Also, for SFC and Oct-tree, the particles are sorted based on their key values within each TreePiece before the decomposition phase begins. In ORB decomposition, we increase the number of boxes by a factor of 2 in each iteration. This bisection is quite complex to implement, since we have to find the correct bisector in each TreePiece. To make things easier, we do a $O(n \log n)$ implementation where particles are sorted according to the longest dimension and then, the bisector is found. In theory, it is possible to implement the bisection in $O(n)$ time using a median-finding algorithm but given the complexity of implementing such an algorithm, we do a $O(n \log n)$ implementation. We intend to implement a $O(n)$ algorithm sometime in the future.

The tree-building phase is really simple for a spatial binary tree. At each level of the tree, we just need to find the splitting particle which splits the box into two spatial equal boxes. This is a $O(\log n)$ operation. So, the whole tree-building process is a $O((\log n)^2)$ process. Building ORB tree is a little more complex. At each level of the tree, we need to find the particle at $n/2$ or $n/2+1$ position in the longest dimension. This is a $O(n)$ algorithm in practice. Hence, the whole tree takes $O(n \log n)$ to build. The ORB tree is a more balanced tree as compared to the spatial binary tree. It usually has an equal number of particles in all its buckets and an equal number of buckets for each TreePiece. The ORB tree is more memory-efficient since the number of tree nodes are less and tree is mostly balanced. ORB tree has been considered *optimal* for nearest neighbor finding, multi-dimensional key searches and some other applications, though it can perform really badly in gravity calculations for certain particle distributions as described in [1]. On the other hand, spatial binary trees result in more accurate forces due to a reduction in the higher order multipole moments [1]. Also, they don't suffer from bad performance for certain particle distributions. So, spatial binary trees are much better for gravity force calculation.

### 4.4.1 Improvement in Data Prefetching

The data prefetching phase described in section 2.3.2 improves performance for all types of decomposition. This is due to the increased hit rate of the cache. While executions without the prefetching phase generate a cache hit rate of about 90%, with the prefetching active the hit rate rises to 95-97% for SFC decomposition, and 100% for Oct-tree and ORB decomposition. The greater accuracy in prefetching for Oct-tree and ORB decomposition is due to the better prefetching algorithm we developed, given the constraint that prefetching must be lightweight. Better prefetching algorithm can be owed to the fact that we know the exact box in which the particles are contained for a TreePiece in case of Oct-tree and ORB decomposition. Although Oct-tree and ORB decomposition provides a clear benefit in terms of cache hit rate over SFC, the full effects on the entire execution time are more complex and will require more detailed studies to be fully characterized.

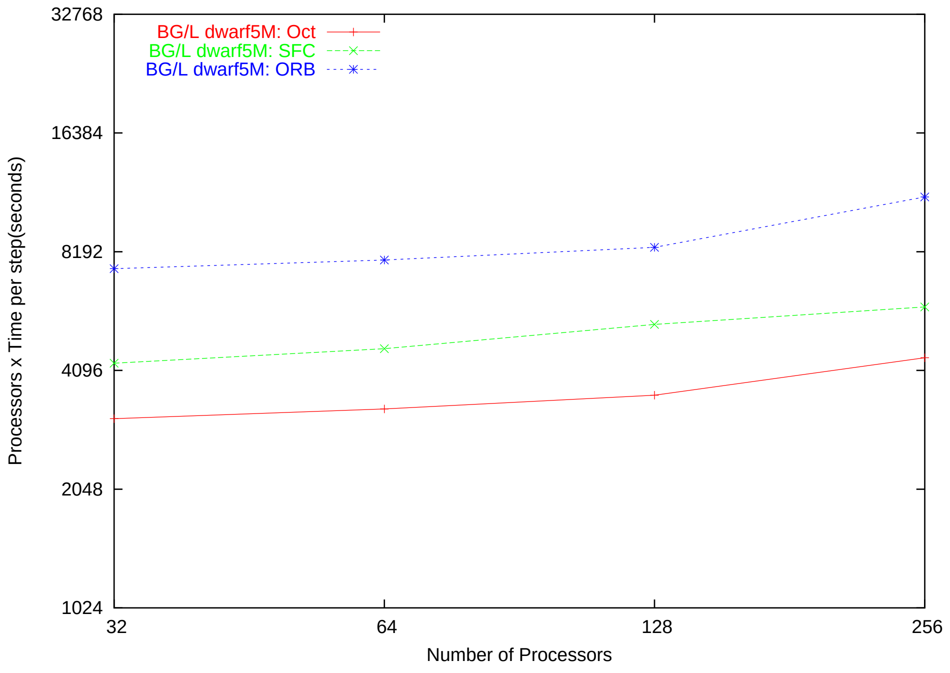


Figure 4.6: Comparison between SFC, Oct-tree, and ORB decomposition on BlueGene/L for the dwarf dataset

### 4.4.2 Scaling and Performance comparisons

We now present the scaling comparisons between the three types of particle space decompositions. We time the gravity calculation phase of ParallelGravity for all of these domain decompositions. The simulator has been run on large parallel machines like BlueGene/L and HPCx and it shows pretty good scaling till hundreds of processors for almost all the decompositions. We use both our datasets of 1 million and 5 million particles. As earlier, vertical axis is the product of the number of processors and the time per step. The horizontal axis represents the number of processors.

Figure 4.6 shows the comparison of the three techniques on BlueGene/L for the dwarf5M dataset. A horizontal line means perfect scaling. We see that all of them scale pretty well till 256 processors. The scaling worsens after 256 processors and is not shown. For ORB decomposition, scaling worsens a little for 256 processors as compared to the others. We

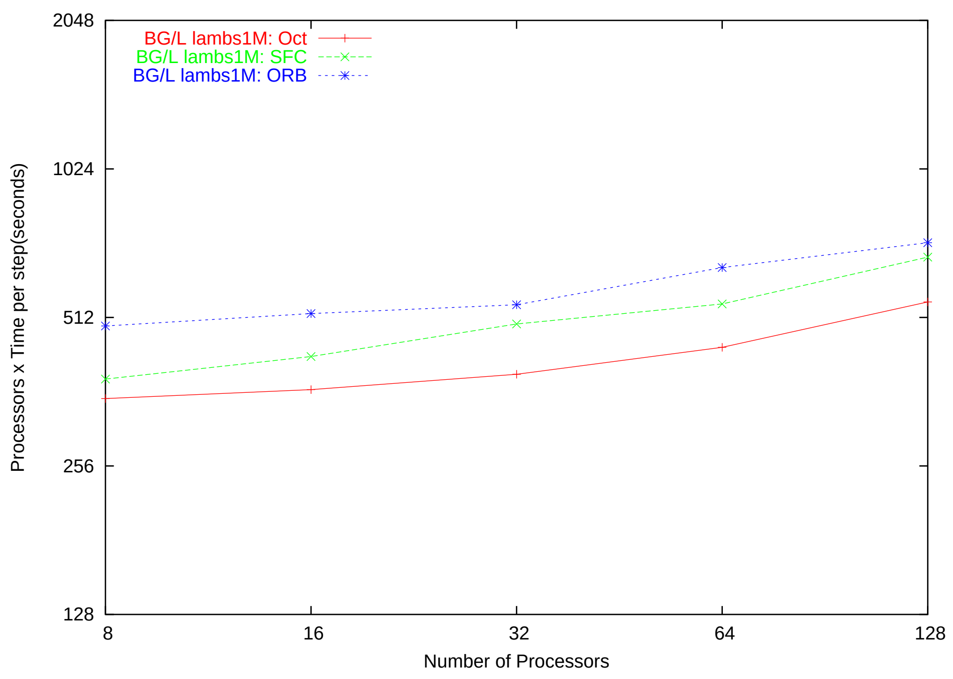


Figure 4.7: Comparison between SFC, Oct-tree, and ORB decomposition on BlueGene/L for the lambs dataset

observe that Oct-tree decomposition is the best among the three decomposition techniques. Since Oct-tree decomposition results in TreePieces with most uneven amount of computation, it makes use of the Charm++ measurement-based load balancers to a great extent to achieve the best performance. This is discussed further in section 4.4.3. During the whole range, Oct-tree decomposition performs better than SFC decomposition by about 25% to 35% and Oct-tree is better than the ORB decomposition by around 60%. ORB decomposition doesn't perform as well as we had expected. We did some initial investigations to ascertain the reason for this. We found that ORB does much more computation as compared to Oct-tree and SFC for the same dataset. Oct-tree and SFC do almost similar computation since the underlying tree is same for them. More computation in case of ORB can be attributed to the ORB tree. ORB tree has a higher number of particles per bucket on average as compared to Oct-tree and SFC. The bounding boxes of the nodes of the tree are more uneven in case of ORB which leads to higher opening radii and hence, more computation. These are some possible reasons which we thought might be responsible for the poor performance of ORB.

Table 4.1: Number of interactions, in millions

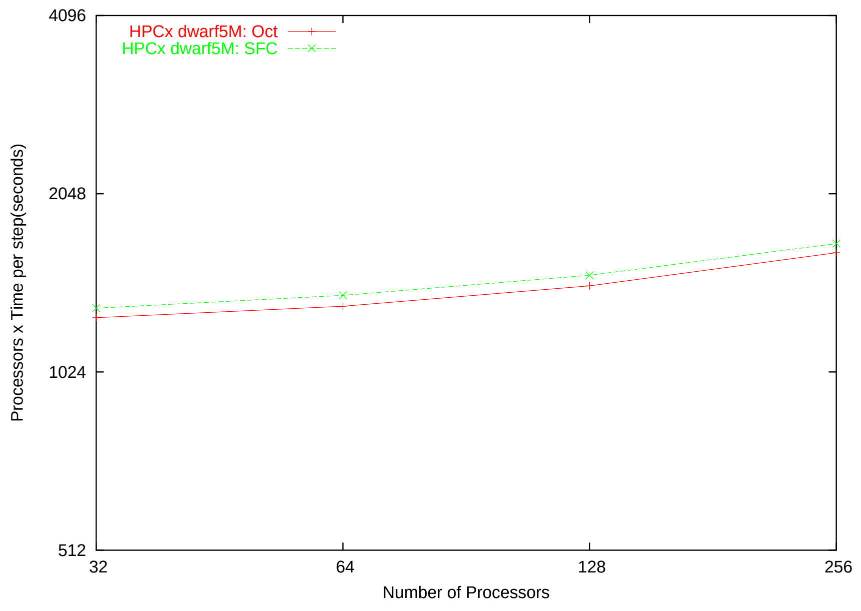
| | Particle-Node | Particle-Particle |
|---|---|---|
| SFC or Oct-tree Decomposition | 387 | 205 |
| ORB Decomposition | 503 | 320 |

The exact contribution of all the factors which cause this bad performance for ORB needs further investigation.

Table 4.1 illustrates the fact that ParallelGravity with ORB decomposition does more computation than the one with SFC (or Oct-tree) decomposition. The results presented are from executions of ParallelGravity on 8 processors of BlueGene/L with lambs1M dataset. We see that the number of particle-particle interactions and particle-node interactions are more in case of ORB than SFC or Oct-tree. One particle-particle interaction stands for the interaction of a particle in the bucket (being considered) with a particle in space. One particle-node interaction stands for the interaction of a particle in the bucket with a node in space. ORB decomposition does 23% more particle-node and 36% more particle-particle interactions. This rise in the number of interactions for ORB gets translated to a greater gravity calculation time.

Figure 4.7 shows the comparison on BlueGene/L for the lambs1M dataset. We see pretty decent scaling for all of them till 128 processors. Oct-tree performs better than the other two. ORB has the worst performance. Oct-tree is better than SFC by around 10% to 20% during the whole range. Oct-tree is better than ORB by about 30% during the entire range.

Figures 4.8(a) and 4.8(b) show the performance comparisons between Oct-tree and SFC decompositions on HPCx machine for dwarf5M and lambs1M datasets, respectively. We don't show the performance results for ORB in this case. Both, Oct-tree and SFC, show similar scalability. Scalability with the dwarf5M dataset is better than lambs1M. Oct-tree performs better than SFC by a margin of about 5% in both the cases. This is lower than what we observed for BlueGene/L.



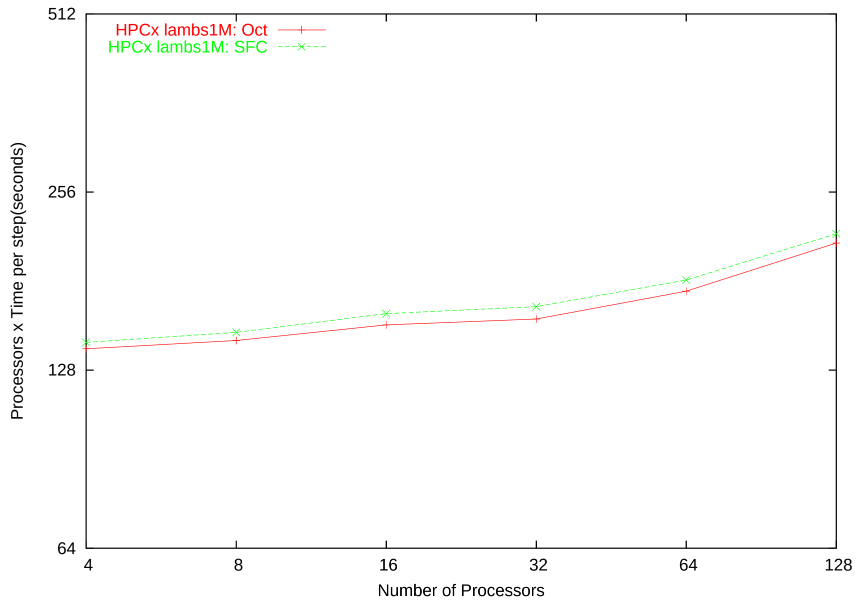


Figure 4.8: Comparison between SFC and Oct-tree decomposition on HPCx for the (a) dwarf dataset and (b) lambs dataset

### 4.4.3 Effect of Load Balancers

In this section, we discuss the effect of the Charm++ automatic load balancing framework on our decomposition schemes. The code instrumentation and the migration of chares in the system are totally automated, and do not require any programmer intervention. Oct-tree decomposition gains the most from load balancing because of its highly unbalanced computation as compared to the others. ORB and SFC decomposition show smaller gains from load balancing. Computation for SFC is more or less balanced, so gain is very little. ORB has quite unbalanced computation in certain cases but gains observed from load balancing have been small. We need to further investigate the reason of these small gains.

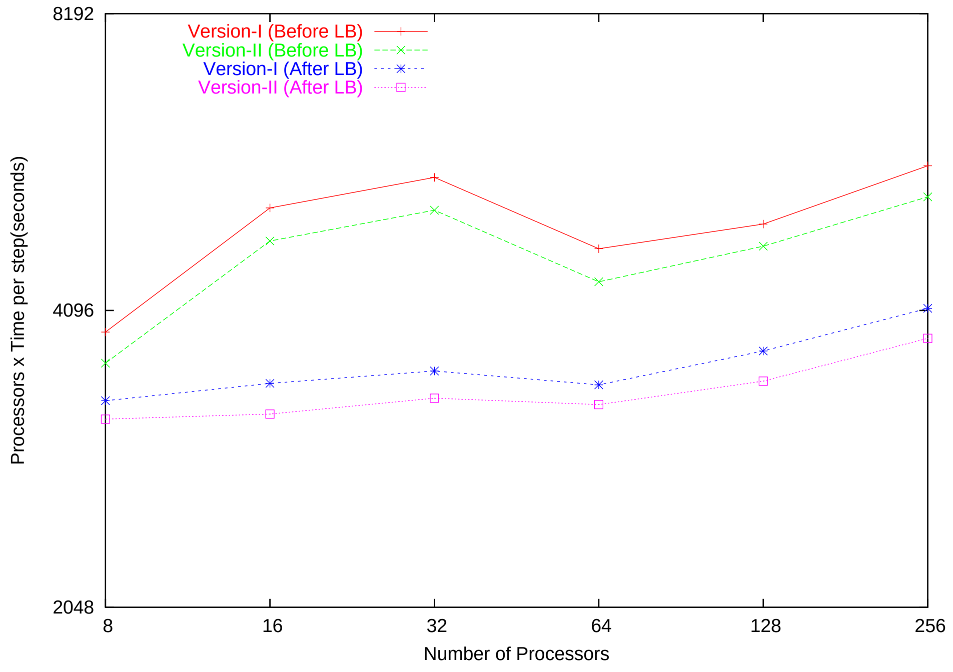


Figure 4.9: Comparison between regular ParallelGravity version and ParallelGravity with interaction lists with Oct-tree decomposition before and after load balancing on BlueGene for the dwarf dataset

Figure 4.9 shows the effect of load balancing on both the versions of the code for Oct-tree decomposition. The improvement from load balancing is similar in both the cases. We see that there is a super-linear speedup at certain points before load balancing. It is due to the fact that before load balancer kicks in, the decomposition of work on each processor is not balanced due to Oct-tree decomposition which results in varying amounts of particles and computation for each TreePiece. The decomposition is more balanced for some number of processors than others which appears as super-linear speedup. Once the load balancer kicks in, it balances the work more or less in all the processors resulting in a performance improvement of about 15% to 35% for both the versions of the code. Both the versions also show pretty good scaling.

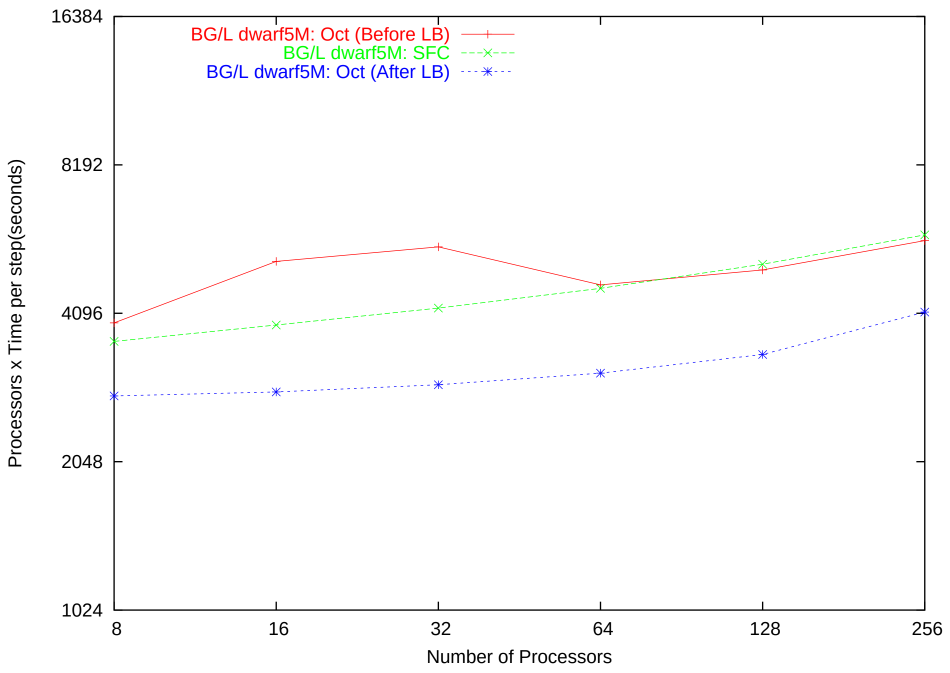


Figure 4.10: Comparison between SFC and Oct-tree decomposition before and after load balancing on BlueGene for the dwarf dataset

We compared the performance scaling in the case of Oct-tree decomposition (before and after load balancing) with that in SFC. Figure 4.10 shows the results. The performance improvement seen in Oct-tree decomposition as a result of load balancing makes it perform much better than SFC decomposition. Load balancers make very little improvement to SFC decomposition. So, we don't plot it here. SFC decomposition is inherently load balanced and doesn't need any runtime load balancing. That is why SFC has been used extensively in literature in the past. State of the art cosmological simulators like PKDGRAV [4] also use SFC decomposition. Better performance for Oct-tree decomposition using Charm++

powerful runtime load-balancing shows us a way of improving the current state-of-the-art cosmological simulators.

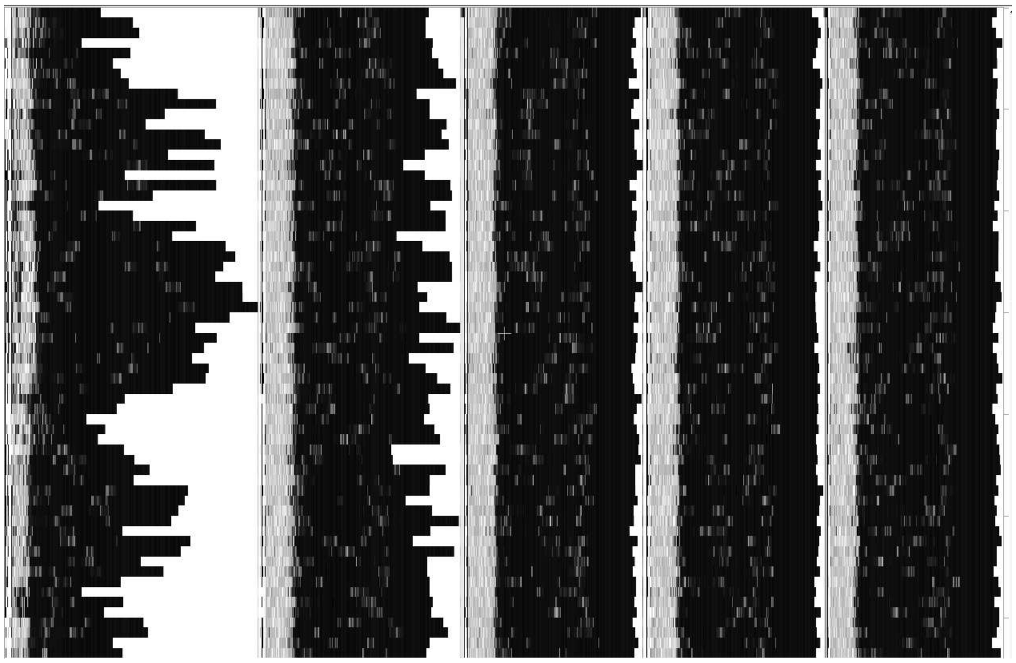


Figure 4.11: Effect of Load Balancer for the dwarf dataset on 64 BlueGene processors

To further analyze the improvements from the load balancers, figure 4.11 displays a view from our *Projections* performance analysis tool, a component of Charm++. The view corresponds to five timesteps of a simulation with Oct-tree decomposition on 64 BlueGene/L processors. The horizontal axis represents time, while each horizontal bar represents a processor. Darker colors represents higher utilization, with black as full utilization and white as idleness. One can see that even starting from a very unbalanced situation on the first time-step, after two timesteps the load balancer improves performance quite significantly, approaching almost perfect balance. The gray region at the beginning of each time-step, where utilization is lower, corresponds to the communication overhead due to data prefetching. The time spent by the application in load balancing and in domain decomposition is hardly visible in the figure. It corresponds to the period between the end of the longest bar in one time-step and the beginning of the gray region of the next time-step. That time in negligible.

It is relevant to notice that the dwarf dataset is highly clustered at the center of the simulation space, and its spatial distribution of particles is very uneven. This non-uniform particle distribution is reflected by the varying processor utilization in the first time-step of the simulation. Situations like this present the biggest challenge to obtain load balance across processors. Nevertheless, the Charm++ load balancers achieved very good balance.

# Chapter 5

# Conclusions and future work

In this thesis, we have presented some optimization techniques for a new parallel cosmological simulator named ParallelGravity. Our design was guided by the goal of achieving good scalability on modern parallel machines, with thousands of processors. The *interaction list* scheme for force computation was implemented using Charm++ virtualization. This virtualized implementation shows similar scalability as the original algorithm and leads to a performance improvement of about 10% in the force computation time.

Our experimental results for the comparison of various particle decomposition techniques show that Oct-tree decomposition is the best for gravity force computation however, all the techniques achieve good scalability. Oct-tree achieves this best performance by exploiting the Charm++ automatic load balancing framework to a greater extent than other decomposition techniques. This shows us that Oct-tree decomposition, implemented along with runtime load balancing, might be a way to improve the current state-of-the-art cosmological simulators.

In essence, by employing various optimizations, including the ones presented in the thesis, the gravity calculation phase in ParallelGravity was shown to scale very well up to large number of processors with real astronomical datasets. This level of scalability places ParallelGravity as a potentially powerful resource for the astronomy community.

Despite ParallelGravity's good observed scalability, we intend to study other load balancing schemes and parallelization techniques that may provide even further benefits. The implementation of *oct-trees* is being considered. Researchers have faced problems with oct-trees in the past because of the difficulty of load balancing the computation with them. However, we expect our powerful Charm++ load balancing framework to achieve a good load balance with oct-trees too. We also need to further investigate the reasons for not-so-good performance of certain optimizations in the current code. In particular, reasons for the bad performance of the simulator with ORB decomposition have to be found out. The effect of the load balancers on ORB decomposition also needs to be further investigated. Moreover, ParallelGravity still needs to incorporate several additional features to become a production-level simulator. We are working on adding support for more physics, such as fluid-dynamics and periodic boundaries, as well as providing multiple time-stepping. In addition, as we start our tests on thousands of processors, we are also analyzing the performance of other phases of the simulation, such as the construction of the particle tree.

# References

[1] R. Anderson. Tree data structures for n-body simulations. *SIAM Journal on Computing*, 28(6):1923–1940, 1999.

[2] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[3] Walter Dehnen. A hierarchical $O(N)$ force calculation algorithm. *Journal of Computational Physics*, 179:27–42, 2002.

[4] Marios D. Dikaiakos and Joachim Stadel. A performance study of cosmological simulations on message-passing and shared-memory multiprocessors. In *Proceedings of the International Conference on Supercomputing - ICS'96*, pages 94–101, Philadelphia, PA, December 1996.

[5] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[6] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[7] George Lake, Neal Katz, and Thomas Quinn. Cosmological N-body simulation. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 307–312, Philadelphia, PA, February 1995.

[8] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[9] H Sagan. *Space-Filling Curves.* Springer-Verlag New York, 1994.

[10] V. Springel, N. Yoshida, and S.D.M. White. GADGET: A code for collisionless and gasdynamical simulations. *New Astronomy*, 6:79–117, 2001.

[11] Volker Springel. The cosmological simulation code GADGET-2. *MNRAS*, 364:1105–1134, 2005.

[12] Joachim Gerhard Stadel. *Cosmological N-body Simulations and their Analysis.* PhD thesis, Department of Astronomy, University of Washington, 2001.

[13] Gengbin Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.