LOAD BALANCING SCIENTIFIC APPLICATIONS

A Dissertation

by

OLGA TKACHYSHYN PEARCE

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Nancy M. Amato
Committee Members,	Marvin L. Adams
	Bronis R. de Supinski
	Lawrence Rauchwerger
	Valerie E. Taylor
Head of Department,	Dilma Da Silva

December 2014

Major Subject: Computer Science

ABSTRACT

The largest supercomputers have millions of independent processors, and concurrency levels are rapidly increasing. For ideal efficiency, developers of the simulations that run on these machines must ensure that computational work is evenly balanced among processors. Assigning work evenly is challenging because many large modern parallel codes simulate behavior of physical systems that evolve over time, and their workloads change over time. Furthermore, the cost of imbalanced load increases with scale because most large-scale scientific simulations today use a Single Program Multiple Data (SPMD) parallel programming model, and an increasing number of processors will wait for the slowest one at the synchronization points.

To address load imbalance, many large-scale parallel applications use dynamic load balance algorithms to redistribute work evenly. The research objective of this dissertation is to develop methods to decide when and how to load balance the application, and to balance it effectively and affordably. We measure and evaluate the computational load of the application, and develop strategies to decide when and how to correct the imbalance. Depending on the simulation, a fast, local load balance algorithm may be suitable, or a more sophisticated and expensive algorithm may be required. We developed a model for comparison of load balance algorithms for a specific state of the simulation that enables the selection of a balancing algorithm that will minimize overall runtime.

Dynamic load balancing of parallel applications becomes more critical at scale, while also being expensive. To make the load balance correction affordable at scale, we propose a lazy load balancing strategy that evaluates the imbalance and computes the new assignment of work to processes asynchronously to the main application computation. We decouple the load balance algorithm from the application and run it on potentially fewer, separate processors. In this Multiple Program Multiple Data (MPMD) configuration, the load balance algorithm can execute concurrently with the application and with higher parallel efficiency than if it were run on the same processors as the simulation. Work is reassigned lazily as directions become available, and the application need not wait for the load balance algorithm to complete. We show that we can save resources by running a load balance algorithm at higher parallel efficiency on a smaller number of processors. Using our framework, we explore the trade-offs of load balancing configurations and demonstrate a performance improvement of up to 46%.

DEDICATION

To those who raised me, for instilling in me the value of purposeful search for knowledge.

To Roger, for your unwavering support.

To Zhanna, for showing me the joys of a child’s curiosity.

ACKNOWLEDGEMENTS

I am fortunate to have many supportive people who have helped me throughout my graduate school years.

I would like to thank my advisor, Dr. Nancy M. Amato, for her continual guidance during my graduate and undergraduate studies. She provided an environment where I could explore many different research interests. I would like to thank my committee members, Dr. Lawrence Rauchwerger, Dr. Marvin Adams, Dr. Bjarne Stroustrup, Dr. Valerie Taylor, and Dr. Bronis de Supinski, for their guidance and suggestions throughout this work.

I would like to thank many people at Lawrence Livermore National Laboratory, where I was a student intern and Lawrence Scholar. Dr. Bronis de Supinski provided significant guidance on my research and future career paths. Dr. Todd Gamblin and Dr. Martin Schulz invested many hours into mentoring and research discussions. The Kull, ParaDiS, and ddcMD teams helped me apply my research to their applications, the DEG team formed a great working environment, Dr. Kathryn Mohror and Dr. John Gyllenhaal provided moral support, and Tony Baylis taught me how to network.

I would like to thank the members of the Parasol Lab. Dr. Gabriel Tanase, Dr. Nathan Thomas, and Dr. Timmie Smith were my graduate student mentors as I was starting my research as an undergraduate and subsequently graduate student. Dr. Shawna Thomas, Dr. Roger Pearce, Dr. Yuriy Solodkyy, Dr. Damian Dechev, Dr. Bonnie Kirkpatrick, and Dr. Mauro Bianco formed the backbone of a strong research environment, and encouraged open collaboration.

I would like to thank the students and faculty of Western Oregon University. Dr. Bob Broeg, Dr. Scott Morse, Dr. Mike Ward, and Dr. Hamid Behmard provided guidance and encouraged my thirst for knowledge. Luke Friedrichsen, Justin Hoeckle, Josh James, Becci

Randall Buenau, and Ron Wessels encouraged camaraderie, Rick Knechtel helped secure funding for my undergraduate studies, and the Callen family helped me integrate in the U.S.

Also, many thanks to the vast (anonymous) women in science blogging community, for advice and moral support.

Finally, I would like to thank my family for their constant support.

This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-TH-663256).

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION

The largest supercomputers have millions of independent processors, and concurrency levels are rapidly increasing. Most large-scale scientific simulations today use a Single Program Multiple Data (SPMD) parallel programming model, computing simultaneously on multiple processors in order to obtain results faster. Scientific simulations often use mesh cells, particles, or other logical elements to represent their domains, and divide work between processes by assigning these logical elements to processes. When different processes are assigned different amounts of work, a subset of them will wait during a synchronizing operation for the slowest one to finish. This uneven assignment of work to processes is called *load imbalance*. The performance penalty of load imbalance increases with scale. Specifically, a simulation with more processes will waste more resources than a smaller-scale simulation when waiting on a single slow process. We must therefore fix even small imbalances at scale. Moreover, assigning computational work evenly becomes increasingly difficult in an application's strong-scaling limit, as the available parallelism becomes more and more coarse-grained with respect to the number of processes. Furthermore, many large modern parallel codes simulate behavior of physical systems that evolve over time. In such simulations, the computational work per logical element may change as the physical system evolves, causing a balanced assignment of work to processes to become imbalanced over time. Thus, the dynamic behavior of these simulations can lead to temporal imbalances in computational load among processes.

To address load imbalance, applications use dynamic *load balance algorithms* to reassign work to processes evenly at runtime. Current load balance mechanisms are often application-specific and make implicit assumptions about the computational load [24, 91]. Some strategies place the burden of providing accurate load information, including the decision on when to balance, on the application [36, 80, 97]. Existing application-independent mechanisms simply measure the application load without any knowledge of the migratable tasks in the application [52, 93], which limits them to identifying the imbalance without correcting it.

1.1 Contributions

The broad goal of this work is to develop general load balancing strategies that will optimize the performance of a diverse set of dynamic parallel applications with temporal imbalance by reassigning work to processes evenly at runtime throughout application execution. The research objective of this dissertation is to decide when and how to balance the applications, and to balance them effectively and affordably. We develop strategies to measure and evaluate the computational load of the application, and decide when and how to correct the imbalance. To make the load balance correction affordable at scale, we evaluate the imbalance and compute the new assignment of work to processes asynchronously to the main application computation. Research challenges of our work include describing the computation of a diverse set of applications in a uniform manner, allowing flexibility in instrumentation, and reusing load balancing algorithms while providing a way to add new ones. Efficiently evaluating load imbalance, regulating the frequency of load balance correction, and computing assignment of work to processes is a challenging problem as researchers seek to leverage modern supercomputers to answer scientific questions through simulation. We propose strategies to guide the decisions on when and how to rebalance an application, and to make load balancing affordable at scale.

This work makes the following contributions:

- A model enabling the comparison of and selection among balancing algorithms for a specific state of a simulation;
- A framework for decoupling and offloading the load balance computation, enabling

lazy load balancing;

- An accurate and fast method to evaluate and balance the load in N-Body simulations with highly non-uniform density, based on adaptive sampling.

1.1.1 Model for Comparison of Load Balance Algorithms

In large-scale physics simulations process loads reflect the underlying physics phenomena, so groups of processes simulating neighboring domains are likely to be similarly overloaded or underloaded. Evaluative studies in this dissertation show that performance of load balancing algorithms can be severely affected under such clustering. Our studies confirm that while *global* load balance algorithms may correct the imbalance in a single step at a cost of high overhead, *diffusive* algorithms correct the imbalance gradually but pay a penalty for taking many steps to converge. Diffusive algorithms may be sufficient for localized imbalances, but drastic and expensive imbalances across a large system may require a drastic correction.

Informed by our study, we developed a model for comparison of load balance algorithms in the context of a specific application imbalance scenario, enabling the selection of a balancing algorithm that will minimize overall runtime. In the applications studied, our model enables runtime evaluation of the imbalance in the application in terms of the effectiveness of the available load balancing algorithms. Our model correctly selects the algorithm that achieves the lowest runtime in up to 96% of the cases, and can achieve a 19% gain over selecting a single balancing algorithm for all cases.

1.1.2 Lazy Load Balancing

Several fundamental issues with the traditional approach to load balancing scientific simulations hamper the effectiveness of load balancing. First, SPMD simulations typically pause while a load balance algorithm runs, but this can cause the load balance algorithm *itself* to become a bottleneck. Second, the load balance algorithms often do not scale up as well as the simulation itself. For example, graph partitioners are widely used for balancing the work in parallel scientific simulations, but they do not strong-scale well.

We propose an approach that relies on two key observations. First, application state typically changes slowly in SPMD physics simulations, so work assignments computed in the past will still produce good load balance in the future. Second, we can decouple the load balance algorithm so that it runs concurrently with the application and more efficiently on a smaller number of processes. Our approach allows the application to proceed while the new assignment is computed; the application applies this work assignment once it has been computed. We call this *lazy load balancing*. We show that the rate of change in work distribution is slow for two parallel applications, and we implement a lazy load balancing infrastructure to exploit this property. We show that we can save resources by running a load balancing algorithm at higher parallel efficiency on a smaller number of processes. Using our framework, we explore the trade-offs of lazy load balancing and demonstrate a performance improvement of up to 46% on the application under study.

1.1.3 Load Balancing N-body Simulations

Our load balancing framework provides a common interface for suites of load balancing algorithms to choose from, and adds a new accurate and fast method to address a particularly challenging case of N-body applications with highly non-uniform density. N-body methods simulate the evolution of systems of particles (or bodies). They are critical for scientific research in fields as diverse as molecular dynamics, astrophysics, and material science [59, 91, 100]. Most load balancing techniques for N-body methods use particle count to approximate computational work [24, 91]. We found that the assumption that particles represent work is inaccurate, especially for systems with high density variation, because work in an N-body simulation is proportional to the particle *density*, not the particle count. We have demonstrated that existing techniques do not perform well at scale when particle density is highly non-uniform, and developed a load balance technique that efficiently migrates interactions instead of particles. We use adaptive sampling to create a hypergraph of interactions and particles to represent the computation. Our aggressive sampling makes the partitioning affordable by reducing the size of the graph by several orders of magnitude. We implement and evaluate our approach on a Barnes-Hut algorithm [12] and a large-scale dislocation dynamics application, ParaDiS [24]. Our method achieves up to 26% improvement in overall performance of Barnes-Hut and 18% in ParaDiS.

1.1.4 Summary

Our contributions include an asynchronous infrastructure for load balancing high performance applications, models for deciding when to balance and how to allocate the resources in the system, and a novel algorithm for balancing N-body simulations with highly non-uniform density. We evaluated the framework on a variety of high performance applications.

Portions of this research were previously published or are currently under review. Our load model for attributing computation to application elements was published at the *International Conference on Supercomputing* (ICS) 2012 [74]. Our methodology for explicitly load balancing N-body simulations with highly non-uniform density was published at the *International Conference on Supercomputing* (ICS) 2014 [73]. Finally, our approach for decoupling the resources used by the load balance algorithm and running the load balance algorithm asynchronously to the application is under review, presented here in Section 5.

1.2 Outline

Section 2 starts with the formal definitions of load balance and load balance metrics, and details existing load balance algorithms and frameworks. Section 3 describes our model for comparison of load balance algorithms in the context of a specific application imbalance scenario, which enables the selection of a balancing algorithm that will minimize overall runtime. Section 4 describes a load balancing method for N-body applications with highly non-uniform density using an adaptive sampling method for selecting migratable tasks and hypergraph partitioning for assigning them to processes. Section 5 describes our framework for decoupling the load balance algorithm resources from application resources, and our lazy load balance algorithm. This dissertation has revealed new directions for research that are discussed in the concluding section.

2. PRELIMINARIES AND RELATED WORK

In this section, we cover background topics and related work that will be referred to throughout the remainder of this dissertation. We define work and its assignment to processes in Section 2.1. We introduce load balance terminology and metrics in Section 2.2. We present load balancing strategies in Section 2.3, including the distinction between global and diffusive strategies (Section 2.3.1), built-in application load balancing strategies (Section 2.3.2), graph partitioning libraries (Section 2.3.3), and related work on overdecomposition, scheduling, and work-stealing (Section 2.3.4). We present a comparison table of existing load balancing strategies and their differences in Section 2.3.5. We present related work on load balance measurement tools, and useful tool stacks in Section 2.4. We present related work on lazy evaluation and its use for optimizing parallel performance in Section 2.5. Finally, we describe the benchmarks and applications that we use in our experiments, along with their built-in load balance algorithm, in Section 2.6.

2.1 Work Assignment and Load Balancing

Most SPMD physics simulations divide the simulated domain into logical elements, which are assigned to processors in the parallel machine. Often, the computational work per element varies over time, so evenly dividing elements among processors does not guarantee an even distribution of work. Figure 2.1 shows an example application domain with elements in the simulation divided among three processes. The domain is represented as a graph with elements as nodes weighted by their computational work. Computation associated with an element may depend on other elements, and this relationship is shown as edges. Dotted edges represent inter-process communication.

In order to keep all processors occupied, applications use dynamic load balance algorithms to reassign work periodically to processors. Fundamentally, load balance algorithms

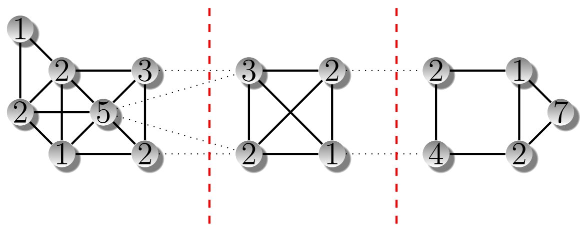


Figure 2.1: A Partitioned (Dashed Lines) Graph of Vertices (Circles) and Edges (Links)

solve a *work assignment problem*. That is, they attempt to distribute work evenly across all processes in the application by relocating application elements. Additionally, load balance algorithms may consider other constraints, such as preserving locality among application elements or minimizing communication.

More formally, a load balance algorithm computes an assignment, $A : V \to P$, where A is a function that maps a set of elements V onto a set of processors P. In a simulation that performs redundant computation to avoid communication, A may not be a function, but we restrict its definition for simplicity in this dissertation.

The work associated with any element v_i is denoted $W(v_i)$ and the set of all elements mapped to a processor $p_i \in P$ is denoted V_i:

$$V_i = \{v | A(v) = p_i\} \tag{2.1}$$

The total work assigned to any processor p_i is denoted $W(V_i)$:

$$W(V_i) = \sum_{v_j \in V_i} W(v_j) \tag{2.2}$$

An effective load balance algorithm attempts to ensure that the total work assigned to each processor is equal, that is:

$$W(V_i) \approx W(V_j) \quad \forall i, j \tag{2.3}$$

An optimal load balance algorithm will minimize the deviation of each processor's assigned work from the average:

$$\sum_i \left| W(V_i) - \frac{1}{|P|} \sum_j W(V_j) \right| \tag{2.4}$$

It has been shown in prior work that the load balancing problem is essentially a balanced graph partition problem. Unfortunately, this means that the load balanced assignment problem is NP complete [46, 55].

2.2 Load Balance Definition and Metrics

Load imbalance is an uneven distribution of computational *load* among tasks in a parallel system. In large-scale SPMD applications with synchronous time steps, imbalance can force all processes to wait for the most overloaded process. The performance penalty grows linearly as the number of processors increases, so regularly balancing large-scale synchronous simulations is particularly important as their load distribution evolves over time.

Load balance metrics characterize how unevenly work is distributed. The *percent imbalance* metric, λ, is most commonly used:

$$\lambda = \left(\frac{L_{max}}{\overline{L}} - 1 \right) \times 100\% \tag{2.5}$$

where L_{max} is the maximum load on any process and $\overline{L}$ is the mean load over all processes. This metric measures the performance lost to imbalanced load or, conversely, the performance that could be reclaimed by balancing the load. Percent imbalance measures the *severity* of load imbalance. However, it ignores statistical properties of the load distribution that can provide insight into how quickly a particular algorithm can correct an imbalance.

Statistical moments provide a detailed picture of load distribution that can indicate whether a distribution has a few highly loaded outliers or many slightly imbalanced pro-

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(L_i - \overline{L})^2} \tag{2.6}$$

$$g_1 = \frac{\frac{1}{n}\sum_{i=0}^{n}(L_i - \overline{L})^3}{\left(\frac{1}{n}\sum_{i=0}^{n}(L_i - \overline{L})^2\right)^{3/2}} \tag{2.7}$$

$$g_2 = \frac{\frac{1}{n}\sum_{i=0}^{n}(L_i - \overline{L})^4}{\left(\frac{1}{n}\sum_{i=0}^{n}(L_i - \overline{L})^2\right)^{2}} - 3 \tag{2.8}$$

Figure 2.2: Statistical Moments

cesses. These properties impact which balancing algorithm will most efficiently correct the imbalance. Diffusive algorithms [34] can quickly correct small imbalances while the presence of an outlier in the load distribution may require more drastic, global corrections. Figure 2.2 shows the three most common statistical moments, standard deviation σ, skewness g_1 and kurtosis g_2, where n is the number of processes and L_i is the load on the i^{th} process. Positive skewness means that relatively few processes have higher than average load, while negative skewness means that relatively few processes have lower than average load. A normal distribution of load implies skewness of 0. Higher kurtosis means that more of the variance arises from infrequent extreme deviations, while lower kurtosis corresponds to frequent modestly sized deviations. A normal distribution has kurtosis of 0. Statistical moments capture key information about load distribution but are insufficient to evaluate the speed with which we can correct imbalance because they do not include information about the proximity of application elements in the simulation space.

Table 2.1 uses several load distributions to show how the statistical moments fail

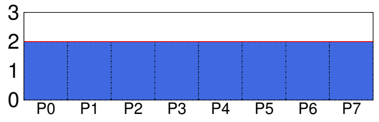
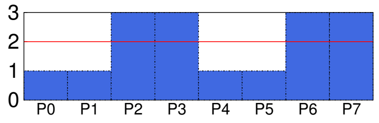
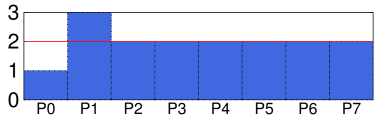
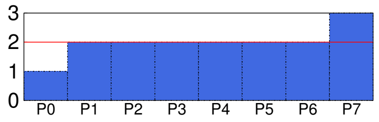
	Load on each Process	$\overline{L}$	λ	σ	g_1	g_2
(a)		2	0%	0	0	0
(b)		2	50%	1	1	-2
(c)		2	50%	.5	2	1
(d)		2	50%	.5	2	1

Table 2.1: Example Load Distributions and Their Moments

to distinguish key properties. For simplicity, we show a one-dimensional interaction pattern of processes $P_0 ... P_7$ in which P_i and P_{i+1} perform computation on neighboring domains. The figure shows that load metrics cannot distinguish cases (c) and (d) while the difficulty of correcting these load scenarios varies greatly if the computation is optimal when neighboring portions of the simulated space are assigned to the neighboring processes. In case (c), we could simply move the extra load on P_1 to P_0, while in (d) the extra load from P_7 must first displace work to P_6, then from P_6 to P_5, and so on through P_1 until the under-loaded P_0 receives enough work.

2.3 Load Balancing Strategies

2.3.1 Centralized and Distributed Strategies

Most dynamic load balancing strategies can be classified as centralized or fully distributed. In centralized strategies, a single (dedicated) processor gathers global load information and makes a decision about global balance. In fully distributed strategies, each processor exchanges information with its neighbors only.

Due to the smaller machine sizes of the past, many early load balancing methods were centralized. These include *state broadcast* algorithms, in which each of the processors broadcasts its new state with any change, a little less communication intensive *broadcast idle* algorithms, in which a processor only broadcasts its status message when it enters an idle state, and *poll when idle* algorithms [63]. Other solutions to the problem [27] include enumerative [83], graph theoretic [17, 88, 89], linear and dynamic programming [18, 38, 44, 64, 70], and queuing theory solutions [29, 57]. The main drawback of centralized approaches is that they do not scale well on modern architectures (petascale and beyond).

Several fully distributed load balancing methods have been implemented [103]. The Sender (Receiver) Initiated Diffusion strategies are asynchronous schemes that only use near-neighbor information. The Hierarchical Balancing Method organizes the system into a hierarchy of subsystems within which balancing is performed independently. The Gradient Model employs a gradient map of the proximities of underloaded processors in the system to guide the migration of tasks between overloaded and underloaded processors. The Dimension Exchange Method requires a synchronization phase prior to load balancing then balances iteratively. Diffusive approaches [32, 34] balance load based on local information by moving the work in the direction of decreasing load. While highly scalable, diffusive approaches perform more work and can take a long time to bring the application to a balanced state, which results in a higher imbalance in the meantime.

With scalability concerns of centralized approaches and poor global balance of distributed approaches in mind, our work explores a method that efficiently incorporates the global load information by working asynchronously to the main computation.

2.3.2 Application Load Balancing

Many applications that can suffer from load imbalance implement their own load balancing schemes. These schemes are usually implemented directly within an application, and, as a result, are tightly coupled with application data structures and cannot be used outside of the application. The drawback of the approach is that often only one algorithm is implemented, and the application developers are unable to compare it to other algorithms.

ParaDiS [24] computes plastic strength of materials. It used to rely heavily on geometric decomposition of the domain (i.e., hierarchical recursive bisection) for load balancing, but currently uses oct-trees to partition the problem domain into similar load subdomains.

SAMRAI [105] is a structured AMR application that load balances boxes at each level of mesh refinement. The processes are organized in a tree structure. An overloaded process sends the extra load to its parent in the tree; the parent then decides to send the load to one of its children or to send it further up the tree. This load balance strategy implicitly keeps neighboring boxes on neighboring processes, but results in a communication bottleneck at the root of the tree.

DistDLB [61] is a dynamic load balancing scheme for distributed cosmology simulations on heterogeneous distributed systems such as the TeraGrid. It targets structured AMR applications where the grid hierarchy is represented as a tree of grids. DistDLB employs a hierarchical approach where rebalancing is possible on a global level (between groups) and inside the groups themselves.

Chombo [31] is a collection of libraries for parallel block-structured AMR calculations. It can load balance the AMR calculations by taking user-specified weights for each box and assigning boxes to processors via the Kernighan-Lin algorithm [56] for solving the knapsack problems.

Network Weather Service (NWS) [106] is a system that takes periodic measurements

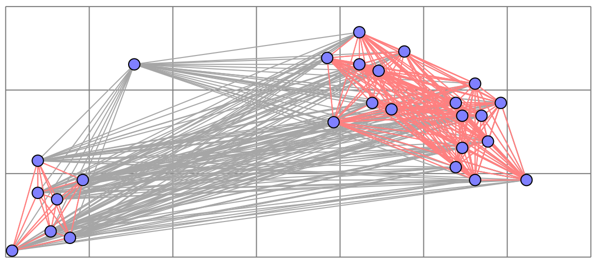


Figure 2.3: NBody Simulation Terminology: n Particles (Blue Circles), Short-Range and Long-Range Interactions (Red and Gray Edges)

of the currently deliverable performance (in the presence of contention) from each resource and uses numerical models to generate forecasts of future performance levels dynamically. Forecast data is continually updated and distributed so that resource allocation and scheduling decisions may be made at run time based on expected levels of deliverable performance. To forecast resource performance, NWS treats periodic measurements taken from a particular sensor as a time series, and then uses different statistical models to predict the next value in the series. Application level scheduling (AppLeS) [87] uses the NWS estimates to schedule gene sequence comparison in a master-slave model.

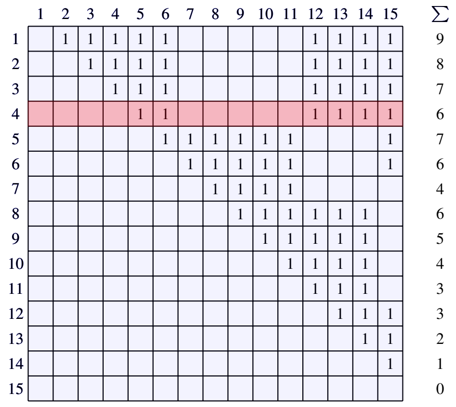
2.3.2.1 Existing N-Body Load Balancing Techniques

N-Body methods simulate the evolution of systems of particles (*bodies*) by computing force *interactions* on groups of particles. For forces like gravity and electromagnetism, interactions involve pairs of particles, but in other systems they may involve larger groups. Once the force is evaluated, particle positions are updated and the cycle repeats. Figure 2.3 shows a system of particles (circles) and the interactions between them (edges). In nearly all N-body simulations, force computation is the bulk of the work. Modern algorithms such as Barnes-Hut [12] and fast multipole [94] compute weak long-range forces (gray edges) less frequently than stronger near-range forces (red edges). These algorithms use a *cutoff radius* to determine whether an interaction is near- or long-range; the cutoff radius is determined based on the particular physics simulated. This optimization reduces the number of interactions computed. Still, the force computation dominates the runtime.

The largest N-body simulations are comprised of billions of particles and require a parallel computer in order to run. Implementing an efficient parallel N-body algorithm is difficult because the algorithm must evenly distribute work to all processes; this task of dividing work is called *domain decomposition*. Since N-body systems are dynamic, the interactions evaluated by each process change over time, and we must assess and *load balance* the work frequently as the simulation progresses. Finally, in addition to dividing work evenly, parallel N-body load balance algorithms must effectively manage *locality*. To compute a force interaction, the simulation needs information on all particles involved. If the particles are owned by different processes, then each process must gather information on remote particles. Local copies of remote particles are called *ghosts*. Ghost communication is expensive, so load balance algorithms must allocate particles so that such communication is minimized.

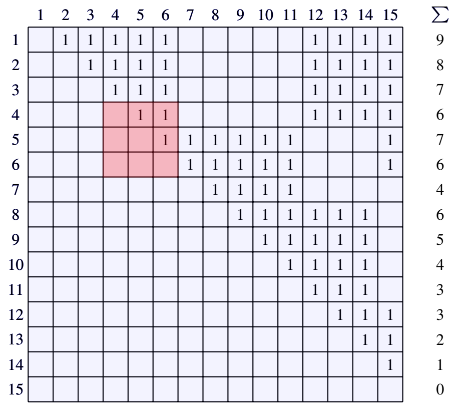
Plimpton [76] classifies N-Body load balancing algorithms into three categories: particle decomposition, force decomposition, and spatial decomposition. The remainder of this section discusses these methods and their limitations in terms of the above criteria.

Particle Decomposition. In a system of N particles running in parallel on P processes, particle decomposition assigns N/P particles to each process. Particle decomposition is also called row-wise decomposition, as each particle and its associated interactions comprise a single *row* in the force matrix, as shown in Figure 2.4. Each interaction is computed by the process that owns the particles involved. If particles in an interaction are owned by different processes, a tiebreaker function determines which process computes the interaction. Unless extra care is taken to preserve particle locality, particle decomposition does not minimize ghost communication. Work is balanced by moving the *particles* from



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$\sum$
1		1	1	1	1	1						1	1	1	1	9
2			1	1	1	1						1	1	1	1	8
3				1	1	1						1	1	1	1	7
4					1	1						1	1	1	1	6
5						1	1	1	1	1	1				1	7
6							1	1	1	1	1				1	6
7								1	1	1	1					4
8									1	1	1	1	1	1		6
9										1	1	1	1	1		5
10											1	1	1	1		4
11												1	1	1		3
12													1	1	1	3
13														1	1	2
14															1	1
15																0

Figure 2.4: Particle Decomposition in N-Body Applications (Assigns Particles or Rows to Processes)

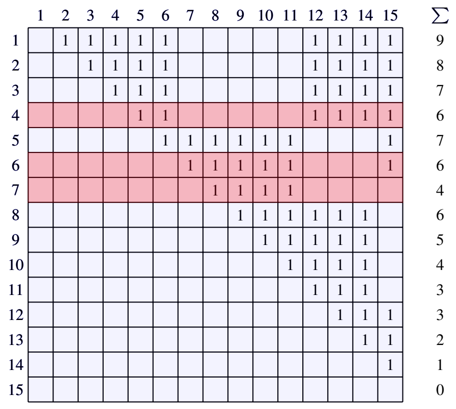


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$\sum$
1		1	1	1	1	1						1	1	1	1	9
2			1	1	1	1						1	1	1	1	8
3				1	1	1						1	1	1	1	7
4					1	1						1	1	1	1	6
5						1	1	1	1	1	1				1	7
6							1	1	1	1	1				1	6
7								1	1	1	1					4
8									1	1	1	1	1	1		6
9										1	1	1	1	1		5
10											1	1	1	1		4
11												1	1	1		3
12													1	1	1	3
13														1	1	2
14															1	1
15																0

Figure 2.5: Force Decomposition in N-Body Applications (Assigns Blocks to Processes)

process to process. However, with a cutoff radius, the number of interactions per particle varies with the local density of the system. Interactions are the bulk of the computation, so particle decompositions become increasingly imbalanced the more density varies.

Force Decomposition. Rather than assigning *rows* of the force matrix as in particle decomposition, force decomposition assigns individual *blocks* of the force matrix, as shown in Figure 2.5. Because particle ordering in the matrix has no geometric correspondence, force decomposition methods do not minimize ghost communication. This method works



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	$\sum$
1		1	1	1	1	1						1	1	1	1	9
2			1	1	1	1						1	1	1	1	8
3				1	1	1						1	1	1	1	7
4					1	1						1	1	1	1	6
5						1	1	1	1	1	1				1	7
6							1	1	1	1	1				1	6
7								1	1	1	1					4
8									1	1	1	1	1	1		6
9										1	1	1	1	1		5
10											1	1	1	1		4
11												1	1	1		3
12													1	1	1	3
13														1	1	2
14															1	1
15																0

Figure 2.6: Geometric Decomposition in N-Body Applications (Assigns Geometrically Defined Groups of Particles to Processes)

well for a naive N-body algorithm that does not use a cutoff radius, because the force matrix is densely populated. However, it does not work well for simulations that use a cutoff radius because each matrix block may contain different numbers of interactions. To balance work efficiently, blocks of the force matrix must be uniformly dense, which often does not hold. Thus, force decomposition is not widely used.

Spatial Decomposition. Most modern N-body simulations use spatial decomposition, in which the simulated physical domain is divided geometrically into subdomains, which are then assigned to processes. Examples include orthogonal recursive bisection [14, 100], octrees [85, 101], fractiling [9], and space-filling curves [26]. Orthogonal recursive bisection divides space recursively into cuboids until each cuboid contains approximately N/P particles. Octree methods similarly divide three-dimensional subspaces into octants until octants contain a similar number of particles. Other spatial decomposition methods include Voronoi cell decomposition [91], in which each process is assigned a centroid and owns the particles nearest its centroid (Figure 2.8a), and prismatic schemes [24], that are variations on recursive bisection that allow subspaces to be divided more than twice (Figure 2.8b). Spatial decomposition methods naturally preserve locality of particles and therefore reduce ghost communication.

In all of these methods, each process owns the computation associated with the particles in its subdomains. Figure 2.6 shows an example of particles 4, 6, and 7 assigned to a single process, along with their interactions, assuming that these particles are in a geometric subdomain. The work is balanced by adjusting the subdomain boundaries, and particles are moved among subdomains as part of balancing. Some of these methods use the number of particles per subdomain as an approximation of the actual workload, and interactions are assumed to be evenly distributed throughout the subdomain. This assumption can result in imbalanced computation. Other methods [104] weight each particle by the number of interactions in which it participates. This approach is accurate, but the interactions per particle may vary, necessitating the split of work on one particle between multiple processes. Many of these methods (e.g., octrees and recursive bisection) impose limits on *how* the space can be subdivided, which imposes further limits on assignment. Together, these approximations lead to less accurate load balancing, which limits application speedup.

Limitations. No existing method balances N-body interactions directly with high precision. Particle and spatial decomposition attempt to assign similar numbers of interactions to processes by assigning *particles*, introducing a high degree of approximation. The force decomposition *does* balance forces directly, but it assumes a dense force matrix, while many simulations have sparse force matrices. Spatial and particle decompositions reduce bookkeeping costs; tracking spatial boundaries or even individual particles is still no worse than $O(n)$ in the number of particles. However, because the number of interactions is much larger than the number of particles, tracking individual interactions quickly becomes intractable. Even the force decomposition assigns *blocks* rather than individual interactions, resulting in far less bookkeeping overhead. We need a new method that allows fine-grained *interaction* balancing without the memory and performance overheads of tracking each interaction directly.

2.3.3 Graph Partitioning

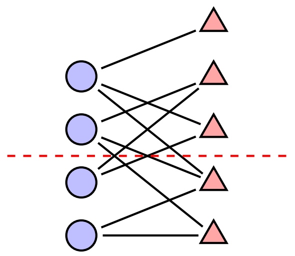


Figure 2.7: A Partitioned (Dashed Line) Hypergraph of Vertices (Triangles) and Hyperedges (Circles and Links)

A common approach to load balancing partitions computation associated with mesh or graph representations of the applications.

A *graph* is a representation of a set of objects (vertices) in which some pairs of objects are connected by links (edges), as shown in Figure 2.1. Both vertices and edges can be weighted. When modeling computation of a parallel application, graph vertices represent computation while graph edges represent communication.

A *hypergraph* is a generalization of a graph in which an edge (a hyperedge) can connect any number of vertices, as shown in Figure 2.7. When modeling computation of a parallel application, hypergraph vertices represent computation while hyperedges represent communication. Each hyperedge connects the computation that requires the same data.

The *graph partitioning* problem is defined on data represented in the form of a graph $G = (V, E)$, with V vertices and E edges. The graph G is partitioned into smaller components with specific properties. For instance, a k-way partition divides the vertex set into k smaller components. A good partition keeps the number of edges running between separated components small.

Several graph partitioners are available. ParMetis [80, 81] and Jostle [97, 98, 99] use multilevel techniques that construct a hierarchy of coarser graphs. These tools coarsen the graph by finding maximum independent matchings and collapsing them. At the coarsest level they (re)partition the graph and then refine it during the coarsening process. Both ParMetis and Jostle target unstructured graphs/meshes.

Zoltan [36, 37] includes a suite of geometric and graph partitioning algorithms, such as recursive coordinate bisection, recursive inertial bisection, refinement tree-based partitioning, oct-tree partitioning, ParMetis and Jostle. The application developer must supply weights for the units of computation. Zoltan also assists the application in data migration.

DRAMA [13] is a dynamic load balancing library for finite element methods that includes geometric and graph partitioning algorithms. Its repartitioning modules include iterative pairwise load balancing, recursive coordinate bisection (RCB), and using ParMetis and Jostle. Since DRAMA specializes on finite element methods, it includes cost functions to account for work and communication associated specifically with elements and nodes in the finite element mesh.

PLUM [16, 71, 72] is a load balancing framework for adaptive grid applications; it is capable of using any partitioning algorithm and assists in efficient processor assignment and remapping of the computation.

Users of these partitioners are left with the task of supplying the partitioners with information about the current state of the application and the system, as well as the decision of when to load balance.

2.3.4 Overdecomposition, Scheduling, Work-Stealing

Overdecomposition is a strategy to divide the work into many more pieces than processors and then map them onto real processors. Overdecomposition can be used to map multiple threads to each processor in an attempt to overlap computation and communication [20].

Charm++ [4, 15, 22, 58, 107] requires the programmer to express the decomposition (of data and work) into a large number of *objects* that have small, well-defined regions of memory on which they operate. Charm++ automates the mapping of data and work to processors and uses object migration to manage resources adaptively. The Charm++ runtime system (RTS) measures the work represented by objects and records object-to-object communication patterns. Based on the RTS measurements, the load balance algorithm may migrate the objects between processor queues. Migrations can only occur between method invocations which avoids having to save the stack. The application or an external timer can trigger the load balance algorithm. The load database records information about all objects (how much processor time they have consumed and the destinations and sizes of messages that each object sent). A global reduction distributes load information to all processors. Charm++ load balancing strategies include greedy strategies (assign heaviest object to least loaded processor; generates significant communication), refinement strategies (migrate heavy objects from the most overloaded processors to the least loaded ones; less communication), a Metis-based strategy (partitions the communication graph so that the edgecut is minimized), and a branch and bound strategy (runs until it attains specified improvement).

Work-stealing is a common approach for multi-threaded environments; as one thread completes its work, it ‘‘steals’’ some from a thread that still has work to do. CILK [30, 43] is a multithreaded language that implements work-stealing. COOL [28] is a concurrent object-oriented language designed to express task-level parallelism and uses a work-stealing runtime scheduler. An idempotent work-stealing algorithm [68] relaxes the constraints and guarantees that each task is executed *at least once* instead of *exactly once*. Dynamic scheduling in OpenMP [7, 39] uses an internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. A scheduler can also assign more processor internal resources to the most compute intensive tasks [19].

PREMA [10, 11] has a runtime model similar to Charm++, but in addition to explicit load balancing initiation, the runtime system may preempt the application execution at periodic intervals to perform load balancing functions. PREMA allows the user to plug in different load balancing algorithms and provides diffusion and work-stealing. It also employs a polling thread to keep a process up-to-date with current load information.

A hierarchical framework for irregular applications by Karamcheti and Chien [54] views the computation as being made up of different thread subsets, each of which is load balanced independently.

Overdecomposition, scheduling and work-stealing work only for the types of applications where the computation can be decomposed into independent objects; since many applications do not work this way, a more general approach is necessary.

2.3.5 Comparison Table

Table 2.2 presents a summary of the key features of frameworks designed for load balancing as their primary objective. While some of the frameworks rely exclusively on runtime instrumentation, others only consider the loads as specified by the application programmer. In this work, we consider both of those data acquisition schemes.

2.4 Tools for Measuring Load Imbalance

To understand what information would be useful in diagnosing and correcting load imbalance, one needs to know that some important SPMD applications proceed in synchronized time steps. During each synchronized time step, different computations (efforts) are performed one after another. Every synchronization barrier between time steps and various computations within them can lead to hundreds of thousands of processes waiting for the few overloaded processes. Additionally, process loads vary between efforts and change

	Framework	Information Collection	Imbalance Evaluation	Balancing Algorithms	Targeted Applications
Partitioners	Jostle [98], ParMetis [81]	programmer specified weights	no evaluation, application responsible	suite of geometric and graph partitioning algorithms	unstructured graphs/ meshes
	Zoltan [36]	programmer specified weights	no evaluation, application responsible	geometric and graph partitioning, ParMetis, Jostle; *customizable*	unstructured graphs/ meshes
	DRAMA [13]	provides cost functions for work and comm.	no evaluation, application responsible	iterative pairwise, RCB, ParMetis	finite element methods
	Chombo [31]	uniform weights	no evaluation, application responsible	Kernighan-Lin [56]	structured AMR
	PLUM [72]	uniform weights	no evaluation, application responsible	*customizable*	adaptive grid
Overdecomposition	Charm++ [4, 15, 22, 58, 107]	runtime instrumentation	task pool, application or timer triggered	greedy, refinement, Metis-based, branch-and-bound; *customizable*	decomposed into objects
	PREMA [11]	runtime instrumentation	task pool, application triggered or preemptive	work-stealing [30], diffusion [34]; *customizable*	decomposed into objects
	Cilk [43], COOL [28]	runtime	runtime driven	work-stealing	Languages for shared memory, multithreaded
Other	Network Weather Service [106]	runtime instrumentation	application triggered	application-driven master-slave scheduling	distributed
	Ours	runtime instrumentation and/or programmer specified weights; *customizable*	imbalance evaluation, asynchronous to main computation; application notified when rebalancing needs to happen and triggers it	libraries like ParMetis, *customizable*	variety

Table 2.2: Comparison with Other Load Balancing Frameworks

dynamically over time, making temporal observation necessary.

We can analyze application performance through profiling tools such as PAPI [21], TAU [84], VAMPIR [23] and Paradyn [65], that measure the total time spent on each processor in different methods of the application. Users can insert their own timers, which requires application-specific knowledge and can be time-consuming. Users can also look at all methods of the application, or the underlying MPI methods, e.g., one might be interested in how much time each processor spends at the synchronization barriers. Profiling can impact the execution of the underlying application, and recording all of the data can overwhelm the I/O system. Moreover, only total times for the entire application run are recorded, failing to reflect the dynamic nature of the changing loads, and the data is not available at runtime to adjust the loads dynamically.

Particularly suitable to our needs of load balance measurement is Libra [45], a scalable load balance measurement framework for SPMD codes. Libra classifies application loops into units of *progress* and *effort*. Progress loops represent steps towards some goal expressed in the application domain. Effort loops are nested within progress loops, and represent a unit of computation. Libra is a P^{N}MPI [82] tool that works with the MPI profiling interface [50] (Section 7.6). It collects load data per effort in each time step per process, providing the temporal load information necessary for load balancing. The tool uses lossy compression to make on-line measurement of production runs feasible. Compressed data is efficiently written out at the end of the application run. In this work, we extend this load balance measurement framework to make it useful for dynamic load balancing.

2.5 Lazy Evaluation and Its Use for Optimizing Parallel Performance

We build on the concurrent programming language notions of futures, promises, and lazy evaluation. A future can be defined as a return value of an asynchronous function, or a promise [8, 41]. Lazy evaluation [96] delays the evaluation of an expression until its value is needed (non-strict evaluation) and may also avoid repeated evaluations (sharing). In our work, we delay the rebalancing of the application until the directions to do so are known.

Overlapping computation and communication has long been considered an avenue for optimizing parallel performance [33]. Benefits of the overlap have been explored for different types of algorithms [47] and on different architectures [102]. The co-processor mode of operation of Blue Gene/L [3] paired an application processor with another processor dedicated to handling its communication tasks. Similarly, resources have been dedicated to collective operations [78], I/O [95], checkpointing [79], tool services [5], and visualization [77]. We take the idea of overlapping loosely coupled parts of the computation and apply it to load balancing by overlapping application computation with the load balance computation performed on dedicated resources.

2.6 Benchmarks and Applications

In this section, we describe the benchmarks and applications that we use in our experiments, along with their built-in load balance algorithms. We use a Barnes-Hut benchmark, and two large-scale scientific applications, ddcMD and ParaDiS, as well as a synthetic load balance benchmark.

2.6.1 Load Balance Benchmark

We use a benchmark to represent classes of load imbalance scenarios that occur in parallel scientific applications. Scientific applications can encounter varying initial load configurations, different patterns of element interactions, and different scenarios of how the imbalance evolves throughout the program. Our benchmark controls these variations to ensure a wide range of experiments.

The main input to our benchmark is a directed graph with vertices that represent application elements and edges that represent the communication/dependencies between them. We derive input graphs from a variety of meshes from actual simulations. We use

Algorithm 1 Benchmark

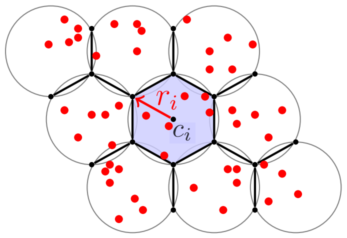
```
Input. $G \leftarrow$ graph of elements, where each process P is assigned a subgraph $G_P$ and remote edges
    represent interprocess communication
 1: for P ∈ processes in parallel do
 2:     for timesteps do
 3:         for remote edges of $G_P$ do
 4:             Irecv/Isend messages
 5:         end for
 6:         for v ∈ G assigned to process P do
 7:             do_work($weight_v$)
 8:         end for
 9:         MPI_Wait(all messages)
10:         if directive to rebalance then
11:             rebalance
12:         end if
13:         update info for load balancing framework
14:     end for
15: end for
```

a simple *do_work(time)* function that accesses an array in a random order. We tune this function for each architecture such that *do_work(1)* executes for one second. Algorithm 1 outlines our benchmark, which calls *do_work* for each graph vertex with the appropriate weights to represent the load scenario. We send an MPI message for each edge that connects vertices on different processes.

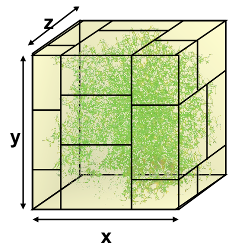
Our experiments vary the size of the graph, vertex and edge weights, and the initial distribution. We can reassign each vertex to any process, as determined by the load balance framework.

2.6.2 ddcMD

ddcMD [35, 48] is a highly optimized molecular dynamics application that has twice won the Gordon Bell prize for high performance computing [48, 90]. It is written in C and uses the MPI library for interprocess communication. In the ddcMD model, each process owns a subset of the simulated particles and maintains lists of other particles with which its



(a) ddcMD



(b) ParaDiS

Figure 2.8: Domain Decomposition in ddcMD and ParaDiS

particles interact.

To allocate particles to processes, ddcMD uses a *Voronoi* domain decomposition. Each process is assigned a point as its center. It ‘‘owns’’ the particles that are nearer to its center than any other. A *Voronoi cell* is the set of *all* points nearest to a particular center. Figure 2.8(a) shows a sample decomposition, with cells outlined in black and particles shown in red. Around each cell, ddcMD also maintains a *bounding sphere* that has a radius of the maximum distance of any atom in the domain to its center.

During execution, atoms that a process owns can move outside of their cell. When this happens, ddcMD uses a built-in diffusion load balance algorithm that uses a load particle density gradient calculation to reassign load. The balance algorithm moves the Voronoi centers so that the walls of the Voronoi cells shift towards regions of greater density. Voronoi centers can only move a limited amount closer to neighboring cells. Voronoi constraints on the shape of cells also limit the possible distributions.

For our global load balancing algorithm, we use a point-centered domain decomposition method developed by Koradi [60]. Each step, we calculate a bias b_i for each domain i. When the bias increases (decreases), the domain radius and volume increase (decrease). We assign each atom (with position vector x) to the domain that satisfies:

$$|x - c_i|^2 - b_i = minimal, \tag{2.9}$$

where c_i is the center of domain i, and we calculate the new centers as the center of gravity for the atoms in each cell.

Although the Koradi algorithm is diffusive, we can run its steps independently of the application execution until it converges. Thus, our implementation is a global method since it only applies the final center positions in the application. We further optimize the algorithm by parallelizing it and executing it on a sample of the atoms rather than the complete set.

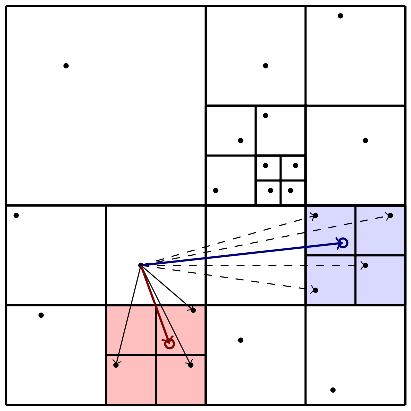


Figure 2.9: Octree in Barnes-Hut Benchmark

2.6.3 Barnes-Hut

Barnes-Hut [12] is a classic N-body algorithm that uses an octree to compute the approximate force that the n particles in the system exert on each other (e.g., through gravity). The n leaves of the octree are the individual particles, while the internal nodes summarize information about the particles contained in the subtree (i.e., combined mass and center of gravity). The octree is used to partition the volume hierarchically around the n particles into successively smaller cells. Algorithm 2 demonstrates the main Barnes-Hut algorithm, and Algorithm 3 details the force computation. While a precise computation

Algorithm 2 Pseudocode for Barnes-Hut

```
Input. particles ← /* read input */;
 1: for int step = 0; step < maxTimestep; step++ do
 2:     Octree octree = new Octree();
 3:     for ∀ particle p ∈ bodies do
 4:         octree.Insert(p);
 5:     end for
 6:     for ∀ Subtrees s ∈ octree do
 7:         s.ComputeCombinedMass();
 8:         s.ComputeCenterOfGravity();
 9:     end for
10:     for ∀ Particle p ∈ bodies do
11:         b.ComputeForce(octree);
12:     end for
13:     for ∀ Particle p ∈ bodies do
14:         b.Advance();
15:     end for
16: end for
```

would have to consider $O(n^2)$ interactions, the Barnes-Hut algorithm uses the summary information contained at each level of the hierarchy to approximate interactions for far away particles. Forces on particles that interact with other particles in nearby cells are computed directly, but for interactions with cells that are sufficiently far away, performing only one force computation with the cell is sufficient.

This algorithm has $O(n \log(n))$ complexity. For example, consider the two-dimensional hierarchical subdivision of space in Figure 2.9. The algorithm checks the distance to the red cell's center of gravity (red circle). Because the distance is not large (red arrow), interactions with all bodies in the red cell are computed (black arrows). Because the blue cell's center (blue circle) is far enough away, only the interaction with the center is computed (blue arrow, a single computation), instead of for each body (dashed arrows).

We created a distributed version of Barnes-Hut based on a shared memory implementation from the Lonestar suite in Galois [25, 75]. The code is written in C++ and uses MPI for communication.

Algorithm 3 Pseudocode for Barnes-Hut ComputeForce()

```
Input. Particle p;
 1: stack.PushBack(root);
 2: while !stack.empty() do
 3:    OctreeNode node = stack.PopBack();
 4:    if distance(p,node) > threshold /* node is far */ then
 5:       ComputeForce(p, node);
 6:    else
 7:       for ∀ child ∈ node.children() do
 8:          if child is leaf then
 9:             ComputeForce(p, child);
10:          else
11:             stack.PushBack(child);
12:          end if
13:       end for
14:    end if
15: end while
```

2.6.4 ParaDiS

ParaDiS [6, 24] is a large-scale dislocation dynamics simulation used to study the fundamental mechanisms of plasticity, written in C/C++ with MPI for interprocess communication. It computes the short-range forces directly and uses multipole expansion [49, 94] for long-range force computation. ParaDiS simulations grow in size as they progress, necessitating periodic rebalancing.

Currently, ParaDiS uses a spatial domain decomposition and has several methods for adjusting the decomposition at runtime. Recursive sectioning or recursive bisection can be used to decompose the domain into spatial prisms, and one prism is assigned to each process. The 3-dimensional recursive sectioning decomposition first segments the domain in the X direction, then in the Y direction within X slabs, and finally in the Z direction within XY slabs, as demonstrated in Figure 2.8(b). The recursive bisection algorithm bisects the space in the X, Y and/or Z dimensions into octants, quarters or halves (depending on the number of domains specified per dimension) such that the computational cost of each sub-partition is roughly the same; the decomposition is then recursively applied to each of the sub-partitions.

ParaDiS uses empirical measurements as an input to its load balancing algorithm. It estimates load by timing the computation that the developers consider most important for load balance. The load balancing algorithm adjusts work per process by shifting the boundaries of the sections. The size of neighboring domains constrains the magnitude of a shift since the algorithm does not move a boundary past the end of a neighboring section.

3. MODELING LOAD BALANCE [1]

Load balance is critical for performance in large parallel applications. Improving load balance requires a detailed understanding of the amount of computational load per process *and* an application's simulated domain, but no existing metrics sufficiently account for both factors. Current load balance mechanisms are often integrated into applications and make implicit assumptions about the load. Some strategies place the burden of providing accurate load information, including the decision on when to balance, on the application. Existing application-independent mechanisms simply measure the application load without any knowledge of application elements, which limits them to identifying imbalance without correcting it.

Applications need information on both *when* and *how* to rebalance; the three load balancing steps are:

1. Evaluate the imbalance;
2. Decide how to balance if needed;
3. Redistribute work to correct the imbalance.

We address the first two requirements and derive complete information on how to perform the third; the application must be able to redistribute its work units as instructed by our framework (a requirement also imposed by partitioners [36, 80]). Our load model couples abstract application information with scalable load measurements. We derive actionable load metrics to evaluate the accuracy of the information. Our load model evaluates the cost of correcting load imbalance with specific load balancing algorithms. We use it to select the method that most efficiently balances a particular scenario. We demonstrate this

[1]ACM, 2012. This is the authors version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *International Conference for Supercomputing (ICS), 2012* [74], http://dx.doi.org/10.1145/2304576.2304601.

methodology on two large-scale production applications that simulate molecular dynamics and dislocation dynamics. Overall, we make the following contributions:

- An application-independent load model that captures application load in terms of the application elements;
- Metrics to evaluate application-provided load models and to compare candidate application models;
- A methodology to evaluate load imbalance scenarios, and how efficiently particular load balance schemes correct it;
- A cost model to evaluate balancing mechanisms and to select the one most efficient for a particular imbalance scenario;
- An evaluation of load balance characteristics in the context of two large-scale production simulations.

We show that *ad hoc* application models can mispredict imbalance by up to 70% and the widely used ratio of maximum load to average load incompletely represents imbalance. Our models provide insight into the cost of algorithms such as diffusion [34] and partitioning [80]. Our model correctly selects the algorithm that achieves the lowest runtime in up to 96% of the cases, and can achieve a 19% gain over selecting a single balancing algorithm.

The remainder of this Section is organized as follows. We give an overview of our method in Section 3.1; it addresses the shortfalls of current load metrics described in Section 2.2. We define our application-independent load model in Section 3.2 and our cost model for load balancing algorithms in Section 3.3. We evaluate application models and demonstrate how to use our load model to select the appropriate load balancing algorithm in Section 3.4.

Algorithm 4 Using the Load Model *(Application code in Italics)*

```
Input. $G \leftarrow$ graph of work units and interactions
 1: for timesteps do
 2:     execute application iteration
 3:     send G to LB Framework
 4:     update Load Model based on iteration measurements and G
 5:     use Cost Model for cost-benefit analysis of available LB algorithms
 6:     if benefit of rebalancing > cost of rebalancing then
 7:         provide selected LB algorithm with accurate input
 8:         send instructions on how to rebalance to application
 9:     end if
10:     if instructed to rebalance then
11:         rebalance as directed by LB Framework
12:     end if
13: end for
```

3.1 Overview of Approach

The computational load in high-performance physical simulations can evolve over time. Our novel model, which represents load in terms of application elements, provides a cost-benefit analysis of imbalance correction mechanisms. Thus, it can guide the application developer on *when* and *how* to correct the imbalance.

Algorithm 4 summarizes the steps of our method. The core of our load model is a graph that abstractly represents application elements (vertices) and dependencies or communication between them (edges). The application elements are the entities that can be migrated to correct imbalance. A developer only needs to provide the work units and their interactions (the same input that they would provide to a partitioner) (Algorithm 4, line 3). Our framework then builds a graph to represent this abstract information.

Our load model combines the abstract application representation with existing tools' measurements of the *degree* of imbalance to evaluate the load accurately in terms of the application elements (Algorithm 4, line 4). We perform a cost-benefit analysis of available load balancing algorithms to determine if rebalancing the application would be beneficial at a given time, and, if so, which load balancing algorithm to use (Algorithm 4, line 5). We give accurate load information to the load balancing algorithm to determine how the application should be rebalanced (Algorithm 4, line 7). We instruct the application to rebalance (answering the *when* question) with that load balancing algorithm (answering the *how* question) (Algorithm 4, line 8).

We show that evaluation of the imbalance and correction mechanisms requires awareness of application information. Our general framework characterizes load imbalance and augments existing load metrics by facilitating the evaluation of developer-provided load estimation schemes. Thus, a developer can use it to refine *ad hoc* load models and to understand their limitations. We demonstrate this process for two large-scale applications in Section 3.4.1. The developer can then use our cost model to select from available load balancing algorithms, as we show in Section 3.4.2.

3.2 Element-Aware Load Model

Parallel scientific applications decompose their physical domain into work units, which, in different applications, can be elements that represent units of the simulated physical space, particles modeled, or random samples performed on the domain. Some application elements may involve more or less computation than others due to their physical properties or spatial proximity. Section 2.2 shows that a load model must be aware of the application elements and their interactions and placement in order to understand load imbalance and, more importantly, how to correct it. A model that does not include this information will fail to capture the effects of the proximity of elements in the simulation space and the mapping of the simulation space onto the process space.

Our investigation of large-scale scientific applications has shaped our novel application-element-aware, application-independent load model that represents application elements and interactions between them. Our API enables the application to provide our frame-

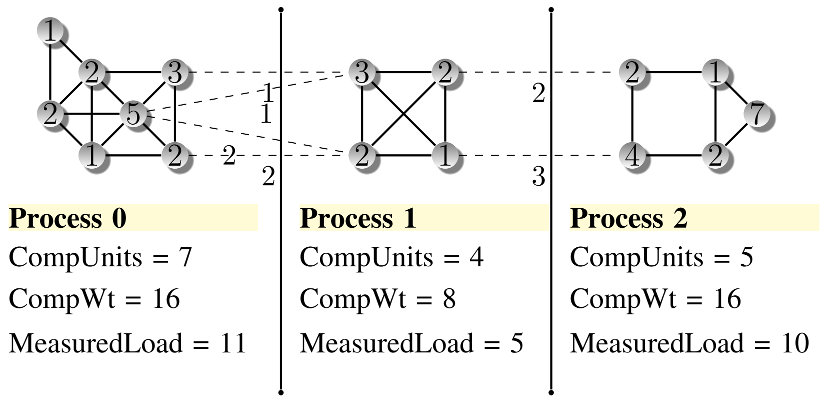


Figure 3.1: Application Element-Aware Load Model

work with abstract application information at the granularity of the application domain decomposition. This granularity allows our model to reflect application elements, their communication and dependencies, and their mapping to processes. Most load balancing algorithms analyze and redistribute work with the same granularity, which enables our framework to guide them. We provide a general methodology to map observed application performance accurately to the application elements at the appropriate granularity level.

Figure 3.1 illustrates our load model: the edges represent bidirectional interactions between application elements. Solid edges represent interactions within a process, while dashed edges represent interprocess communication. The relationships between application elements within the domain decomposition provide the communication structure and the relative weights of computation in the model. Node weights indicate the computation required for each element as anticipated by the application (i.e., the application load model). Importantly, we can correlate this information to wall-clock measurements of the load on each process. The example in Figure 3.1 shows that Process 0 has 7 work units with an application anticipated load or relative computation weight of 16, and its work units have 4 channels of communication with elements on Process 1 with a total relative communication cost of 6. For example, we measure the load on Process 0 to be 11.

We must carefully consider the difference in modeled and measured load. If the model is accurate, the two are linearly related. If they are not directly proportional, the application model is incomplete and could be improved. We discuss our methodology to evaluate abstract application information in Section 3.4.1. When we are satisfied with the model accuracy, we can use it to compute the load distribution metrics and to observe how the load is distributed throughout the process space in terms of application elements.

3.2.1 Applying Element-Aware Load Model to Scientific Applications

Table 3.1 illustrates the versatility of our model by showing work unit mappings for three major types of scientific applications.

3.2.1.1 Unstructured Mesh

In unstructured mesh applications, each cell in the mesh is an element. We represent the mesh connectivity with edges. In some unstructured mesh applications, the cells may require similar computation and we would anticipate unit computation per mesh cell. In others, the computation per cell may be proportional to the cell's volume, and we reflect this relationship in the weight of each node in our model. Table 3.1(a) shows an unstructured mesh application that performs a Monte Carlo algorithm on its mesh. In this case, the work is proportional to the number of samples in each mesh cell, so we use the sample count as the node weight. We show communication between neighboring grid cells as edges.

3.2.1.2 Molecular Dynamics

In classical molecular dynamics applications and other N-body simulations, each individual body is an element. Edges reflect the simulated neighborhood of the bodies: each body is connected to others within a cutoff radius (i.e., those with which it interacts), as Table 3.1(b) shows. As we discuss in Section 3.4.1, we can select from several models for computation per element. Simple models assume that the work per body is constant,

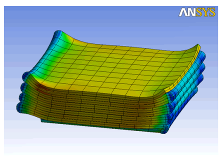
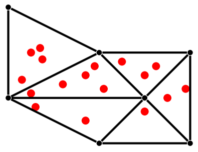
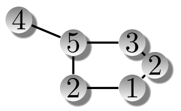
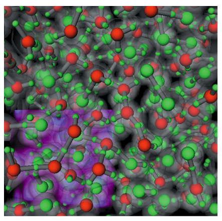
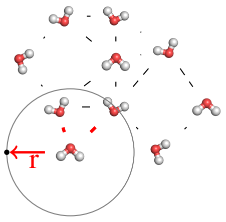
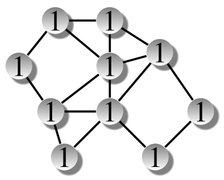
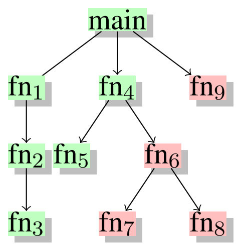
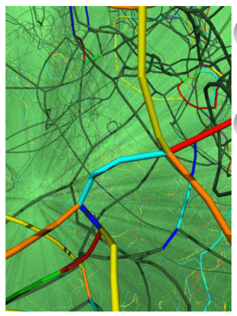
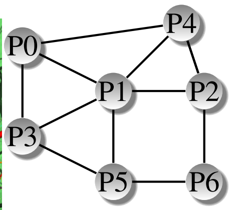
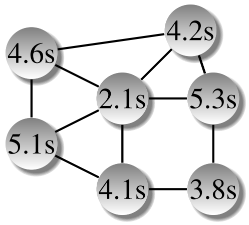
Type of Application, elements and interactions	Sample Application Image	Representation in App.	Our Representation
(a) Unstructured Mesh • e.g., particle transport or finite element applications • elements: cell volume or number of samples in each cell (Monte Carlo algorithms) • interactions: mesh connectivity			
(b) N-body • e.g., Molecular Dynamics Applications • elements: (sampled) molecules • interactions: molecules within range of interaction r (as defined by the application)			
(c) Other - Empirical Model • e.g., ParaDiS (Section 4.2.2) • interactions: graph of process communication • elements: time in developer-defined 'key' routines (green); incomplete coverage of application behavior (red)  main fn$_1$ fn$_4$ fn$_9$ fn$_2$ fn$_5$ fn$_6$ fn$_3$ fn$_7$ fn$_8$			

Table 3.1: Applications and Their Representation in Our Load Model

while others reflect the density of the body's neighborhood.

3.2.1.3 Empirical Model

Some applications, such as ParaDiS [24], use empirical models to anticipate computation per element. An application developer can construct this type of model by placing timers around important computation regions. Table 3.1(c) shows how *ad hoc* placement of timers may omit important load constituents.

3.3 Load Balance Cost Model

In this section, we use our load model to evaluate the cost of two types of load balancing algorithm. Our cost model can guide selection of the best algorithm for specific imbalance scenarios.

3.3.1 Types of Load Balancing Algorithms

3.3.1.1 Global Algorithms

A global balancing algorithm [36, 80, 97] takes information about the load on all tasks and decides how to redistribute load evenly in a single step. Global decisions can be costly. Sequential implementations must process data for an entire parallel system. Parallel implementations can communicate excessively. Global algorithms also can require substantial element movement. However, if the cost of balancing is low, global algorithms balance load in a single step and correctly handle local minima and maxima.

3.3.1.2 Diffusive Algorithms

A diffusive balancing algorithm [34] performs local corrections at each step and only moves elements within a local neighborhood in the logical simulation domain. Diffusive algorithms can take many steps to rectify a large imbalance because load can only move a limited distance. However, diffusive algorithms are scalable because they only require local information, and element movements can be mapped to perform well on high diameter

Variable	Definition	How Determined
L_{ave}	Average process load	$\frac{1}{procs}\sum_{i=0}^{procs} L_i$
L_i	Load of process i	Measured or estimated by Algorithm 5
D_i	Set of processes with elements that can be moved to process i	Derived from edges in load model
L_{ij}	Load of process $j \in D_i$	Measured or estimated by Algorithm 5
γ	Load shifting coefficient in Algorithm 5	Provided by application, $\gamma \leq 1$
convergence_steps	Number of steps for diffusive algorithm to converge	Derived by simulating diffusion, Algorithm 5
L_{max_i}	Maximum process load at step *i*	Simulated (diffusion); $\approx L_{ave}$ (global)
$ElementsMoved_i$	Largest number of elements moved to a process at step *i*	Simulated (diffusion); $\approx Elements_{max} \frac{L_{ave}}{L_{max}-L_{ave}}$ (global)
$C_{DataMvmt}$	Time required to send $ElementsMoved_i$	Modeled empirically, $\alpha + \beta ElementsMoved$
$C_{LbDecision}$	Runtime of load balancing algorithm, e.g., C_{global} and C_{diff}	Measured or modeled empirically
$C_{BalAlgo}$	Balancing algorithm cost: algorithm time plus redistribution time	$C_{LbDecision} + C_{DataMvmt}$
$AppTime_{BalAlg}$	Total application runtime under *BalAlg*, e.g., $AppTime_{diff}$	Modeled by cost model
steps	Number of time steps that the application takes	Arbitrary, same for global and diffusion algorithms

Table 3.2: Cost Model Variables

mesh and torus networks used in the largest machines.

3.3.2 Cost Model for Balancing Algorithms

Our cost model captures the rebalancing characteristics of diffusive and global algorithms. Developers typically choose load balance algorithms based on their intuition about the scalability of particular algorithms. For example, one might expect the cost of a global balancing scheme to be higher than that of a diffusive algorithm at scale because the time required for an immediate rebalance outweighs the amortized cost of local diffusive balancing. The intuition is approximate and sometimes inaccurate. Our cost model provides a quantitative basis for selecting among algorithms. Table 3.2 summarizes the variables that we use to define this cost model.

Our cost model only considers the current imbalance. Future imbalances are highly dependent on how the simulation evolves. Predicting them is generally infeasible (otherwise we could predict the result of the simulation). We assume a continuous evaluation of the imbalance in an application's load leading to new balancing decisions when necessary. These decisions can consider the (observed) rate at which the application becomes imbalanced, and apply a global load balance algorithm when drastic changes are necessary or a diffusion scheme to handle more modest imbalances.

A load balancing algorithm's cost is the time to decide which elements to move plus the time required to move the elements:

$$C_{BalAlg} = C_{LbDecision} + C_{DataMvmt} \tag{3.1}$$

where *BalAlg* can be *global* or *diffusion* (i.e., C_{global} or $C_{diffusion}$).

$C_{LbDecision}$, the time to run the balancing algorithm, can be known *a priori* or derived using a performance model, such as a regression model over timings that vary the algorithm's input parameters [62]. Typical parameters for the modeling approach include the input size (e.g., the number of vertices in the load model graph and the average number of edges per vertex) and the number of processes that a parallel balancing algorithm uses.

We define $C_{DataMvmt}$ as:

$$C_{DataMvmt} = \alpha + \beta ElementsMoved_{max} \times elementsize \tag{3.2}$$

where the application provides $elementsize$, α is the start-up cost of communication (latency), and β is the per-element send time (bandwidth), determined empirically per platform. A more detailed model could capture network contention. For global balancing algorithms, we approximate the number of elements moved as:

$$ElementsMoved_{global} \approx Elements_{max} \frac{L_{ave}}{L_{max} - L_{ave}} \tag{3.3}$$

where $Elements_{max}$ is the number of elements on the process with L_{max}. We approximate the portion of the elements that we must move from the most loaded process as proportional to the load imbalance, which assumes that the load per element is approximately constant. Although this assumption is coarse (load balance would be trivial), we find this simplification works well in practice.

The total cost of a load balancing algorithm is the application runtime when using the algorithm, which is the time to perform each computation step plus the cost of the algorithm at each step:

$$AppTime_{BalAlg} = \sum_{i=0}^{steps} (C_{BalAlg_i} + L_{max_i}) \tag{3.4}$$

where *steps* is the number of timesteps that the application takes, C_{BalAlg_i} is load balancing algorithm's cost at step *i*, which is zero for a global algorithm in all steps other than the one in which load balancing is performed. The time for each step of the computation is the time taken by the most heavily loaded process, L_{max}.

For a global scheme, the total cost reduces to:

$$AppTime_{global} = C_{global_1} + steps \times L_{ave} \tag{3.5}$$

since we assume that the global load balancing algorithm is only invoked in the first step. We estimate the time per computation step as L_{ave} under the assumption that imbalance is corrected.

Algorithm 5 Diffusion Simulation [34]

```
Input. $L_i \leftarrow$ load of process $i$
    $D_i \leftarrow$ neighborhood of process $i$, defined in Load Model graph
    $L_{ij} \leftarrow$ load of process $j \in D_i$
    $\gamma \leftarrow$ coefficient for how much load can be moved in one timestep
    *threshold* $\leftarrow$ lowest attainable level of imbalance for the application *convergence_steps* $\leftarrow 0$
 1: All processes in **parallel do**
 2: **for** timesteps **do**
 3:     **if** imbalance $>$ threshold **then**
 4:         *convergence_steps*++
 5:     **end if**
 6:     $L_i = L_i + \sum_{j \in D_i} \gamma(L_i - L_{ij})$
 7:     $\mathit{ElementsMoved}_i = \mathit{NumElements}_i \sum_{j \in D_i} \gamma(L_i - L_{ij})$
 8:     $\mathit{NumElements}_i = \mathit{NumElements}_i + \mathit{ElementsMoved}_i$
 9:     $L_{max_i} = max(L_i)$ $\forall$ processes at timestep *i*
10:     $\mathit{ElementsMoved}_{max_i} = max(\mathit{ElsMoved}_i)\forall$ procs at step *i*
11: **end for**
```

For diffusion, we compute the total application time as:

$$\mathit{AppTime}_{\mathit{diff}} = \sum_{i=0}^{steps} (C_{\mathit{diff}_i} + L_{max_i}) \tag{3.6}$$

To compute the total application time for diffusion, we have developed a diffusion simulator that mimics the behavior of diffusive load balancing algorithms. Algorithm 5 gives a high level overview of our diffusion simulator. We apply Algorithm 5 to our load model to simulate the movement of load at each iteration. At each step, process i moves a portion of its load to its neighboring processes. We define a coefficient, γ, to model the amount of load that can be moved in one time step to reflect any application limitations (e.g., the maximum amount that domain boundaries can move in one time step). If the application does not limit element movement, $\gamma = 1$. Our algorithm accounts for local minima and maxima because it moves the simulated load through our graph-based model similarly to data motion under an actual diffusive algorithm.

Algorithm 5 records L_{max_i} and $ElementsMoved_{max_i}$ at each simulated step. We use those values in Equation 3.6. Additionally, Algorithm 5 defines an important metric, *convergence steps*, or the number of steps a diffusion algorithm takes to balance the load. This metric differentiates scenarios that a diffusion algorithm can correct quickly from those for which diffusion performs poorly.

Algorithm 5 determines the costs required for Equation 3.6 much faster than the actual diffusion can be performed. We can evaluate its cost without perturbing the application. If the simulation predicts that diffusion will take too long, we can use a different load balancing algorithm such as a global load balancing scheme.

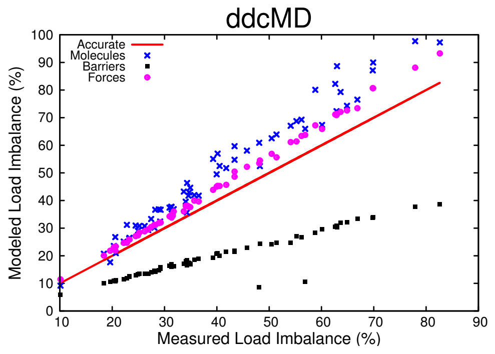
We compare $AppTime_{diff}$, $AppTime_{global}$, and $AppTime_{none}$ to determine which load balance algorithm to use, where:

$$AppTime_{none} = steps \times L_{max_1} \tag{3.7}$$

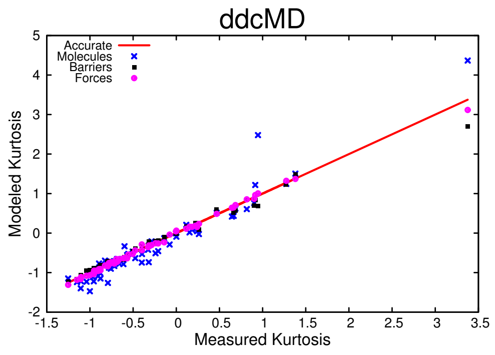
We report the effectiveness of this decision in Section 3.4.2.

3.4 Evaluation

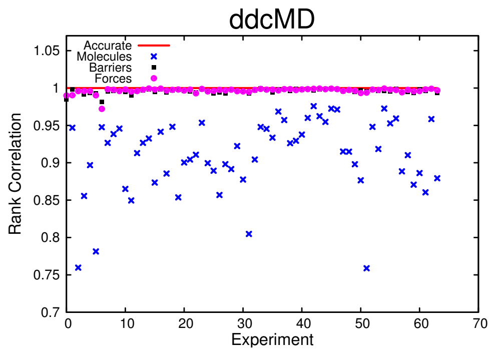
For all ParaDiS experiments, we use a Linux cluster that has 800 compute nodes, each with four quad-core 2.3 GHz AMD Opteron processors, connected by Infiniband. We use a similar cluster that has 1,072 compute nodes, each with four dual-core 2.4 GHz AMD Opteron processors connected by Infiniband for all ddcMD runs in Section 3.4.1. On both Linux systems, we use gcc 4.1.2 and MVAPICH v0.99 for the MPI implementation. We use a Blue Gene/P system with 1,024 compute nodes with 4 32-bit PPC450d (850MHz) cores each and 64 32-bit PPC450d I/O nodes for all ddcMD experiments in Section 3.4.2. On this system, we use gcc 4.1.2 for our measurement framework and compile ddcMD with xlC 9.



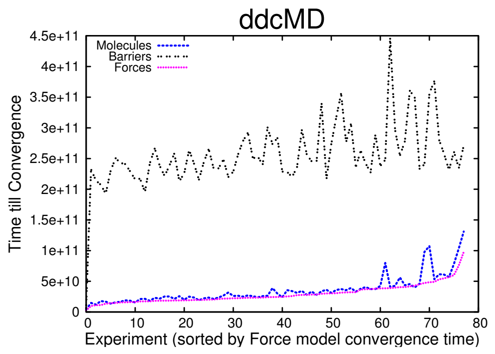
(a) Percent Imbalance in Runs



(b) Kurtosis of Load Distributions

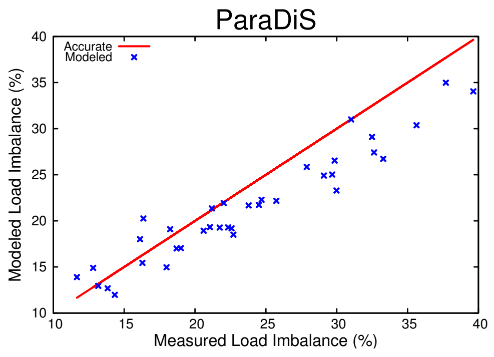


(c) Rank correlation of load distributions

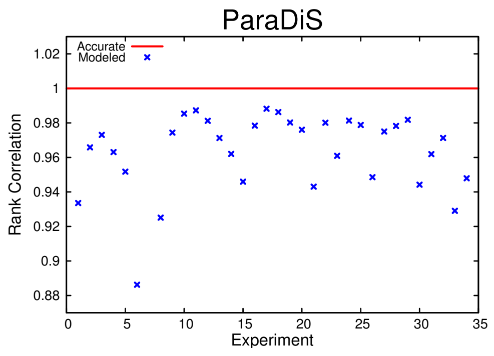


(d) Steps for Diffusion Algorithm to Converge

Figure 3.2: Evaluation of Three ddcMD Models



(a) Percent Imbalance in Runs



(b) Rank Correlation of Load Distributions

Figure 3.3: ParaDiS Model Evaluation

Our experiments use a range of decompositions that exhibit different load balance properties by varying the placement of Voronoi cell centers. We evaluate all three models in Section 3.4.1, and vary load distributions in Section 3.4.2. We used two problem sets for ddcMD, a nanowire simulation and a condensation simulation. The *nanowire* simulation is a finite system of 133,280 iron (Fe) atoms that incurs imbalance due to uneven partitioning of the densely populated cylindrical body surrounded by vacuum. Atom interaction is modeled with EAM potentials. We ran the nanowire problem on 64 processes. The *condensation* simulation is a Lennard-Jones condensation problem with 2.5e+6 particles and the interactions modeled with Lennard-Jones potentials [53]. It incurs imbalance due to condensation droplets forming in some of the simulated domains. We ran the condensation simulation on 512 processes.

To validate application models, we measure the work per process using Libra [45], a scalable load balance measurement framework for SPMD codes. Libra measures the time spent in specific regions of an application per time step using the *effort model*. In this model, time steps, or *progress steps*, model each step of the synchronous parallel computation, and fine-grained *effort regions* within these steps model different phases of computation.

We extend Libra's effort model to serve as input to our load model and add an interface to query load information during execution. We measure the computational load on each process by summing the time spent in all effort regions. Our load model couples these measurements with the application abstractions, allowing us to validate the abstractions against empirical measurements.

We use P^{N}MPI [82] to integrate Libra with our load model infrastructure. P^{N}MPI stacks independent tools that use the MPI profiling interface [50], which we apply to combine Libra with our new load model component. P^{N}MPI also supports direct communication among tools, which we use to exchange element interaction and performance information. Our tool stack imposes 3% overhead on average, an insignificant perturbation of application behavior.

3.4.1 Evaluating Application Abstractions

In this section, we evaluate the quality of *ad hoc*, developer-provided application load abstractions by comparing them with empirical measurements obtained from Libra. To quantify the quality of the abstractions, we report the accuracy with which they capture the measured load imbalance and the statistical moments of the measured load distribution, as defined in Section 2.2.

For additional analysis, we validate application abstractions with a *rank correlation* metric. Rank correlation measures how accurately the abstraction ranks each process's load relative to that of other processes. To calculate the rank correlation r of process loads between the application abstraction M and measured load L, we first order all ranks based on the values of the developer-provided load abstraction and the measured load data. The resulting rank number for process i is stored in l_i and m_i respectively, as is the mean rank in $\bar{l}$ and $\bar{m}$. We then calculate the correlation with:

$$r = \frac{\displaystyle\sum_{i=0}^{n}(m_i - \bar{m})(l_i - \bar{l})}{\sqrt{\displaystyle\sum_{i=0}^{n}(m_i - \bar{m})^2 \sum_{i=0}^{n}(l_i - \bar{l})^2}} \tag{3.8}$$

To accommodate tied ranks correctly, we use Pearson's correlation coefficient [69]. We now apply our abstraction evaluation methodology to ddcMD and ParaDiS.

To represent elements and the associated load, ddcMD uses three application-specific models:

1. *Molecules*: number of particles (molecules) per process;
2. *Barriers*: time each process spends outside of barriers;
3. *Forces*: time spent calculating interactions on each process.

	Molecules	Barriers	Forces
imbalance	15.917	25.769	16.095
kurtosis	0.444	0.079	0.057
rank corr.	0.138	0.008	0.007

Table 3.3: RMSE for Plots in Figure 3.2

Figure 3.2(a) demonstrates how well the ddcMD abstractions capture the imbalance in the problem. Table 3.3 shows root mean squared error (RMSE) of load imbalance and the statistical moments calculated over our experiments. Abstractions based on the number of molecules and force computation overestimate the imbalance in the system, while the abstraction based on execution time excluding time spent at barriers underestimates the imbalance. Underestimating the imbalance leads to slower imbalance correction with a diffusion scheme because it is less aggressive than necessary. Alternately, overestimation pushes the limits of how much the load can be redistributed at each time step, and thus converges faster, as long as the overestimation correctly captures the *relative* loads.

Figure 3.2(b) shows how the three ddcMD abstractions capture the kurtosis of the load distributions for each run; Table 3.3 shows the RMSE. The abstraction based on the number of molecules does most poorly, because much of the imbalance arise from imbalanced neighbor communication, which that abstraction omits.

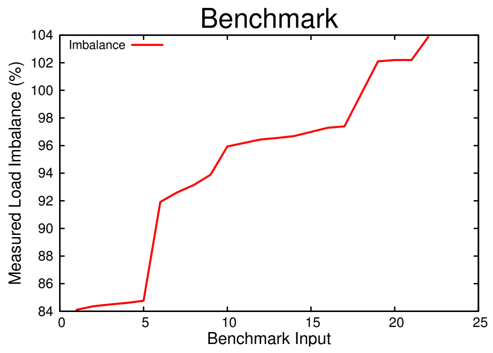
Figure 3.2(c) shows the rank correlation between the modeled and measured distributions for each of our test cases; Table 3.3 shows the corresponding RMSE. Again, abstractions based on time and force computation detect any outliers fairly well, while the abstraction based solely on the number of particles does worse.

Overall, our analysis indicates that the force computation abstraction is the most accurate and, thus, the most suitable for use as input to the diffusion mechanism. To validate our conclusions, Figure 3.2(d) shows the number of *steps to converge* when the diffusion algorithm uses the three abstractions as input. We use a threshold of 12% imbalance because ddcMD's best achievable balance is limited by constraints on the shape of Voronoi cells in its domain decomposition. As predicted, the abstraction based on the force calculation is the most accurate and thus corrects the load most quickly. The abstraction based on the number of molecules outperforms the Barrier abstraction, partly because the former overestimates the imbalance making the diffusion scheme take more drastic measures and arrive at a balanced state sooner.

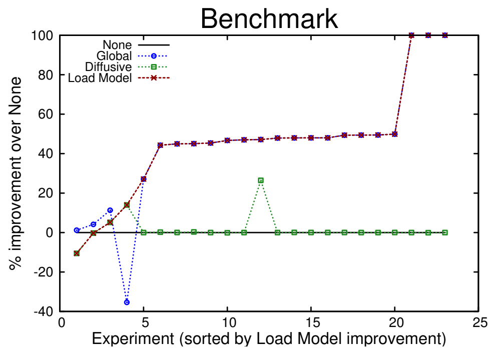
ParaDiS uses empirical measurements as an input to its load balancing algorithm. It estimates load by timing the computation that the developers consider most important for load balance. Figure 3.3(a) shows the accuracy with which the ParaDiS application abstraction represents its load imbalance. Figure 3.3(b) shows the rank correlation of actual and modeled load distributions. The figures show that the abstraction is somewhat inaccurate, which we suspected because it does not include certain major phases of the computation that are captured by the measurements; the developers only measure the main force computation. Our load model in conjunction with Libra's data shows that this fails to capture the behavior of communication, collision detection, and remesh phases. When we compare ParaDiS's calipers to Libra's measurements of only the force computation, the model is quite accurate. Depending on the problem, these omitted regions comprise up to 15% of the execution time.

3.4.2 Cost Model Case Study

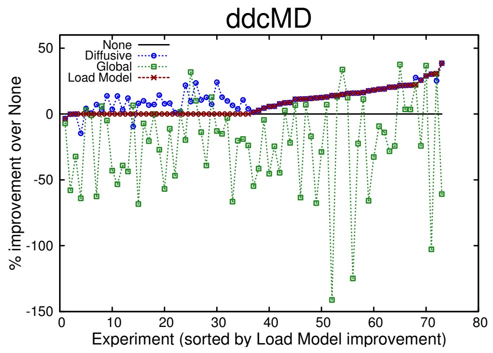
In this section, we evaluate how well our model selects the most effective load balance algorithm for particular imbalance scenarios, and we further evaluate the net performance improvement achieved using our model. We use the cost model defined in Section 3.3 to select the load balancing algorithm that would lead to the shortest runtime of our benchmark. We then apply our cost model to the global and diffusive load balancing schemes in ddcMD.



(a) Starting Imbalance of Benchmark Runs



(b) Model Performance on Benchmark Runs



(c) Model Performance on ddcMD Runs

Figure 3.4: Evaluation of Our Load Model on Benchmark and ddcMD

For our benchmark, we compare total application runtime when using the following load balancing algorithms:

1. **Global:** Correcting imbalance during the first time step using Zoltan's graph partitioner [36]; modeled by Equation 3.5;
2. **Diffusive:** Correcting imbalance at every time step using the Koradi method [60]; modeled by Equation 3.6;
3. **None:** No correction; modeled by Equation 3.7.

We conduct runs spanning 2 to 64 processes with graphs with between 8,000 and 512,000 vertices and varying weights and initial decompositions. Figure 3.4(a) shows initial imbalance in the benchmark runs; we chose these initial imbalance scenarios because they are representative of some of the application runs that we observed.

Figure 3.4(b) shows that our load model correctly selects the algorithm that achieves the lowest runtime in 87% of the cases, tracing the curve with highest performance improvement for most of the experiments. In most cases, our model chooses the global algorithm. This algorithm performs well in 96% of the cases, but 4% of the time, its high algorithmic and redistribution cost (as modeled by Equation 3.1) outweighs the performance benefit so it incurs a 35% performance penalty. In these cases, the diffusive algorithm outperforms the global algorithm, and our model correctly chooses it instead. In the only cases where our model does not choose the correct algorithm, it only suffers a penalty of 5.43% because these were scenarios in which the global and diffusive algorithm performed within 6% of each other.

On average, using our model can achieve a 49% performance gain while the next best alternative, the global algorithm, achieves 48% overall improvement in runtime. While the diffusive algorithm performed much worse than either of these overall (averaging net gains of only 3% over doing nothing), our model is still able to exploit it in the rare cases where

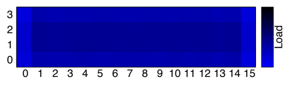
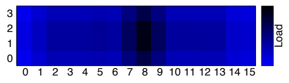
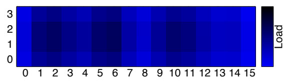
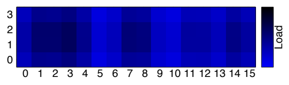
	Load on each Process	Original	Diffusive	Global	Scheme Selected by Model
(a)		226	212	248	diffusive
(b)		344	459	269	global
(c)		286	355	239	global
(d)		270	267	235	global

Table 3.4: Sample ddcMD Imbalance Scenarios

it *did* outperform the global algorithm, leading to significant gains in these scenarios and more reliable performance. For these cases, the diffusive algorithm performs significantly better than Zoltan, and using our model can prevent performance loss for workloads that contain many such pathological runs.

For evaluating the performance of our model for ddcMD, we applied it to the input sets also used in Section 3.4.1; their initial load properties are demonstrated in Figures 3.2 (a-c). Our model selected among the following load balancing algorithms:

1. **Global:** Correcting imbalance during the first time step using the Koradi method [60] several times to mimic a method that corrects the imbalance in one step; modeled by Equation 3.5;
2. **Diffusive:** Correcting imbalance at every time step using the *ad hoc* Voronoi decomposition method with the *Forces* abstraction evaluated in Sec. 3.4.1 as input; modeled by Equation 3.6;
3. **None:** no correction; modeled by Equation 3.7.

Table 3.4 shows runtimes of the load balancing algorithms for several imbalance scenarios in the nanowire simulation. These cases ran on 64 processors organized as a 4x16 process grid. Table 3.4 shows the relative load in the beginning of the simulation, with darker sections representing higher load and lighter blue representing lower load for the particular process. For each of these, we show the execution time without load balancing, the execution time with the diffusion algorithm, and the execution time using the global algorithm. We run the nanowire simulation for 200 time steps. The diffusion algorithm has a cost (as defined in Equation 3.1) of 0.01 seconds per simulation step. The global load balancing method incurs a one-time cost of 9 seconds. In all cases, our cost model guides the selection of the appropriate balancing algorithm.

Figure 3.4(c) shows that our load model correctly selects the best algorithm in 96% of the cases, tracing the curve of the best performing algorithm. Because the *ad hoc* Voronoi algorithm improves performance in 82% of the cases with an average performance gain of 14%, our model correctly selects it in most cases. The Koradi algorithm consistently performs worse and is only selected by the model in a few cases where its performance improvement outweighs the high cost of Koradi; overall, the model achieves a 19% gain over the Koradi algorithm.

While our experiments are designed to explore a range of values, a suite of production runs might contain a variety of pathological cases, and our model will allow a code to perform well even in the cases where the otherwise preferred balancing algorithm will perform poorly. Our model provides a means to select the appropriate load balancing algorithm at runtime without developer intervention, correctly selecting the algorithm that achieves the lowest runtime in up to 96% of the cases, achieving a 19% gain over selecting a single balancing algorithm for all cases.

3.5 Summary

We have presented a novel load model based on application elements and their interactions. Our load model establishes a mapping between application elements and computation costs while maintaining information on dependencies between application elements. Our load model enables an application-independent representation of load distribution and can form the basis for a new generation of generic, yet element-aware load balance tools. We have shown that our element-aware approach overcomes deficiencies of conventional statistical load metrics, which fail to represent element interaction information. Using our element-aware load model, we developed a new set of actionable metrics that accurately characterize load distribution.

We demonstrated the effectiveness and versatility of our load model on several case studies. We provided a mechanism to evaluate and to contrast several application-provided abstractions. We have used our load model to analyze the load imbalance in two production applications. Finally, we evaluated the ability of available load balance schemes to correct imbalance. In all experiments, adding the application element interaction information to the load data was critical to understanding and analyzing the application's load behavior.

4. LOAD BALANCING N-BODY SIMULATIONS[1]

N-body methods simulate the dynamic evolution of a system of particles (*bodies*) under the influence of physical forces. These algorithms are critical to many scientific fields, including astrophysics, computational biology, chemistry, and material science [59, 86, 91, 100]. In an N-body simulation, each particle may exert a force on any other. The simulation progresses by repeatedly computing force *interactions* between pairs of particles, then updating the particles to reflect the force's effect. These forces typically comprise the bulk of the simulation's execution time. If all forces are considered, a naive N-body algorithm runs in $O(n^2)$ time with respect to the number of particles. Modern algorithms such as Barnes-Hut [12] and the fast multipole method [94] use more sophisticated algorithms to reduce the number of interactions that need to be computed, resulting in $O(n \log n)$ or $O(n)$ runtime, but even with these algorithms, the interaction computation dominates the runtime. Large N-body simulations may involve billions of particles, and they need to be run on parallel computers.

Load balance is a major performance problem for N-body methods at scale. For the best parallel performance, computational work must be evenly decomposed over all processing elements of the machine. Currently, many N-body simulations use a geometric domain decomposition to assign groups of *particles* to processes, and each process computes the interactions involving the particles assigned to it. However, the work in N-body simulations is proportional to the number of *interactions* that each process computes, i.e., the local density of particles in the simulated domain. Particle decompositions can therefore distribute work unevenly when particle density varies widely. This type of load imbalance

[1]ACM, 2014. This is the authors version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *International Conference for Supercomputing (ICS), 2014* [73], http://dx.doi.org/10.1145/2597652.2597659.

is particularly expensive at scale, because hundreds of thousands of idle processors may wait on a single overloaded processor.

We show that particle-based decompositions are prohibitively imprecise at scale, particularly when interaction density is highly non-uniform, and we present a load balancing method that explicitly balances the real work: interactions. Current approaches do not balance interactions explicitly because of memory and performance concerns: interactions greatly outnumber particles. Our approach makes balancing interactions affordable by using adaptive sampling to select uniformly sized groups of interactions, which we call *work units*. We then apply a hypergraph partitioner to the work units to assign them to processes. The overhead of this approach is low because the coarse granularity of the work units and their uniform size make the hypergraph partitioner run efficiently.

We apply our load balancing technique to a Barnes-Hut benchmark and a large scale dislocation dynamics application, ParaDiS. This chapter makes the following contributions:

1. An algorithm for load balancing interactions in N-body simulations, using work unit selection and hypergraph partitioning to assign interactions to processes explicitly;
2. An adaptive sampling approach to select work units with uniform sizes for good load balance, and coarse granularity for good partitioning performance;
3. Demonstration of significantly improved load balance, low overhead, and overall performance improvements of up to 26% on Barnes-Hut and 18% on ParaDiS.

To our knowledge, we present the first approach to load balancing interactions explicitly with low overhead. Section 2.3.2.1 summarized traditional load balance methods for N-body applications. Section 4.1 describes our algorithm for load balancing interactions, which uses an adaptive sampling approach for selecting work units and hypergraph partitioning for assignment to processes, as detailed in Sections 4.1.1 and 4.1.2. Section 4.2 outlines our implementation and its application to Barnes-Hut and ParaDiS. We evaluate the performance of our approach in Section 4.3.

4.1 An Interaction-Based Load Balance Algorithm

The load balance algorithm must be precise to achieve the evenly balanced load required for performance at scale. In particular, to address the limitations discussed in Section 2.3.2.1, the load balance algorithm must:

1. Balance interactions directly with fine granularity;
2. Preserve locality to reduce ghost communication;
3. Run fast and not incur excessive bookkeeping overhead.

In this section, we present a load balancing algorithm that satisfies all three criteria. It consists of the following steps:

1. **Select work units with sampling.** Sample interactions; use samples to divide interactions into subsets, or *work units*.
2. **Construct model.** Use work units, proximity information.
3. **Partition model.** Assign work units to p processes by partitioning the work units into p groups.

Optimal partitioning is NP-hard [66] with many heuristic partitioning algorithms. However, while existing partitioning algorithms are sufficient for off-line use, our challenge is to use them in a dynamic, on-line load balance algorithm. Even with an efficient heuristic algorithm, the number of interactions on each process is $O((\frac{N}{P})^2)$. Repeatedly partitioning a system this large at runtime is too slow.

The crux of our approach is to reduce the number of work units under consideration by several orders of magnitude using *sampling*. Further, we exploit two key aspects of any partitioner, namely that the partitioner has a higher likelihood of finding an optimal solution [66] and will therefore run faster if: 1) work unit sizes are small compared to process load, and 2) the sizes are fairly uniform. We have developed adaptive techniques to split large sample groups and to narrow distribution of work unit sizes. We have also experimented with different sample granularities to find a sufficiently fine granularity without excessive overhead.

To our knowledge, our load balancing algorithm is the first on-line algorithm that directly partitions interactions instead of particles. Our technique is also the first algorithm to sample *interactions* in a large-scale N-body problem. We discuss our adaptive sampling techniques in detail in Section 4.1.1, and we present our techniques for model construction and partitioning in Section 4.1.2.

4.1.1 Selecting Work Units

As discussed, using a hypergraph partitioner on the full set of interactions in an N-body simulation is infeasible. Thus, we have developed an *adaptive sampling strategy* that works in two ways. First, sampling *coarsens* the data set by several orders of magnitude, which allows us to solve a much smaller partitioning problem. Second, our sampling strategy is adaptive: it samples denser regions of the problem space more finely so that work units are relatively uniform in size, avoiding many of the pitfalls of the decompositions discussed in Section 2.3.2.1. Our strategy ensures that the partitioning is both fast and accurate.

Algorithm 6 outlines the steps of our approach. Our algorithm takes as input the set of particles P, a set of interactions I, and an *adaptive sampling threshold* s. Our algorithm's output is a set of *work units*. A work unit is a sampled interaction and an associated neighborhood. Each work unit represents all samples in a particular neighborhood, and it consists of the sampled interaction, an associated *centroid*, and a number of non-sampled interactions.

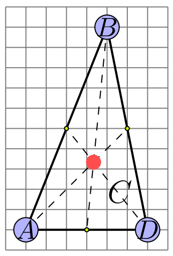
On line 2, Algorithm 6 starts by iterating over all particles. For each particle, on lines 4 and 5, the algorithm samples at least one interaction. If a particle is involved in more than the average number of interactions, then we take more samples. The adaptive sampling

Algorithm 6 Adaptive Interaction Sampling

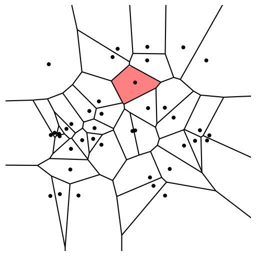
```
Input. $P \leftarrow$ particles, $I \leftarrow$ interactions, $s \leftarrow$ adaptive sampling threshold
 1: $count_{avg} = |I|/|P|$
 2: for all $p_j \in$ P do
 3:     $i_{p_j}$ = set of interactions of $p_j$
 4:     nSubsets$_j$ = max$(1, s \times |i_{p_j}|/count_{avg})$
 5:     take nSubsets$_j$ samples from $i_{p_j}$
 6:     if $nSubsets_j > 1$ then
 7:         build $k$-d tree from samples taken
 8:         for all interactions of $p_j$ do
 9:             select the subset $w_{jk}$ to which interaction belongs
10:             $|w_{jk}|$++
11:         end for
12:         $w_{j_{avg}} = |i_{p_j}|/$nSubsets$_j$
13:         for all subset$_{jk} \in$ subsets$_j$ do
14:             if $w_{jk} > s \times w_{j_{avg}}$ then
15:                 adaptively sample within subset$_{jk}$, calculate weights
16:             end if
17:         end for
18:     end if
19: end for
Output. $W \leftarrow$ work units with desired size and $\sim$uniform size distribution
```

threshold, s, determines the number of additional samples to take, and the caller can use s to adjust the aggressiveness of sampling in dense regions of the domain. The number of interactions sampled from particle p_j is stored in $nSubsets_j$.

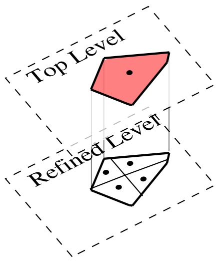
Once we have a sampled interaction, we assign it a coordinate in space based on the *centroid* of the particles that it involves. For a pairwise force, like gravity, the centroid is the midpoint between two particles. For more complex forces, it is the center of mass of the polygon defined by the member particles (Figure 4.1(a)). To define neighborhoods for work units, each sampled interaction's centroid is used as the center of a *Voronoi cell* that defines the neighborhood. A centroid's Voronoi cell is the set of points closest to that centroid. Figure 4.1(b) shows a set of points and their enclosing Voronoi cells. Any interactions in a sampled interaction's Voronoi cell are considered part of its work unit.



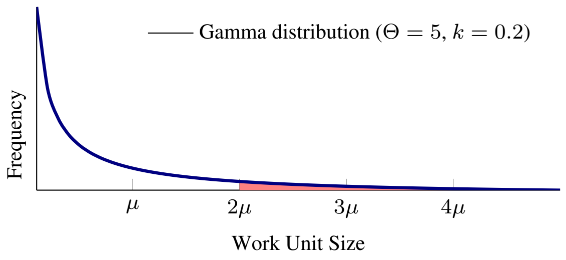
(a) Centroid C of Particles A, B, D



(b) Sampled Interactions Define Voronoi Cells



(c) Multi-Layer Voronoi Cells



(d) Gamma Distribution Probability Density Function

Figure 4.1: Defining Interaction Subsets

Our adaptive sampling technique ensures that each Voronoi cell contains approximately the same number of interactions. If a cell contains too many interactions, e.g., the cell in Figure 4.1(c), then we increase the number of samples in its neighborhood, effectively splitting it into subcells. Thus, our work units have nearly uniform granularity and are easy to partition. However, the splitting is potentially expensive. With one sample per particle, we can easily track which interactions belong to a particular work unit by associating the interactions with their owning particle. With multiple samples for a particle, we need another ownership mechanism. For particles with multiple samples, we use a k-d tree [42] to determine which interactions are closest to each sample.

4.1.1.1 Right-Tailed Distribution

Using a k-d tree ensures that each work unit has high locality and the accuracy of our sampling scheme. However, constructing it is expensive. We must therefore be careful to set the adaptive sampling threshold to a value that balances granularity with range query cost.

Fortunately, an obvious way to set s exists for nearly all N-body systems, In the natural sciences, the density of samples of objects in physical and natural processes (such as particles in a dynamical system) obey a power law distribution [40]. This phenomenon is known as *Taylor's Law* in ecology and the *fluctuation scaling law* in physics. For our purposes, it implies that random sampling leads to work units with sizes that can be fit to a gamma distribution:

$$\mathbb{P}(x) = \frac{1}{\Theta^k}\frac{1}{\Gamma(k)}x^{k-1}e^{-\frac{x}{\Theta}} \tag{4.1}$$

where k is a *shape* parameter, Θ is a *scale* parameter, and $\Gamma(k)$ is the *gamma function* evaluated at k. While the parameters vary, all examples that we have considered exhibit a long right tail as shown in Figure 4.1(d), which implies that relatively few samples have a larger than average number of interactions. Thus, we can achieve an even distribution of work unit weights by splitting relatively few work units into smaller pieces. The (red) shaded area under the tail of the gamma distribution in Figure 4.1d depicts the number of work units that are larger than 2μ, or $2\times$ the mean. Thus a domain scientist can easily pick a good value for s for a particular problem: s should generally be chosen to ''chop off'' all or part of the tail of the gamma distribution. In Section 4.3, we show empirically that our method produces work units with relatively uniform sizes, and we demonstrate the positive impact on the resulting load balance.

4.1.2 Assigning Work Units to Processes

Section 4.1.1 described how we select uniformly sized work units for load balancing; next, we construct a model from these work units to represent both parallel computation and communication. A hypergraph is a well suited model for this problem because hypergraphs have been used extensively to represent the behavior of parallel applications [51]. Further,

Algorithm 7 Hypergraph Construction

```
Input. $P$ (set of particles), $SI$ (set of sampled interactions)
Output. $H = (V, E^H)$ (graph of particles and interactions)
1: for $p_j \in$ P do
2:     $H$.insert($e_i$) to represent the particle
3:     for $i \in$ subsets$_j$ do
4:         $H$.insert($v_{ij}$)
5:         add edges from $v_{ij}$ to all $e_i$, hyperedges needed to compute $v_{ij}$
6:     end for
7: end for
```

we can use well established hypergraph partitioning algorithms to guide load balancing.

Hypergraphs are a generalization of graphs. Where a graph contains vertices, and pairs of vertices are connected by edges, in a hypergraph, each *hyperedge* may connect *one or more* vertices. Thus, if we represent interactions with vertices, particles are a natural fit for hyperedges, because a particle may be involved in many interactions. Hyperedges also accurately represent ghost communication in N-body simulations, because if two interactions in the same partition share a hyperedge that represents a remote particle, a single ghost node will need to be fetched. Partitioning a hypergraph tries to minimize the number of hyperedges cut by partition boundaries, and thus minimizes inter-process communication.

Formally, given a weighted *hypergraph* $H = (V, E^H)$ where V is a set of vertices and E^H is a set of hyperedges, hypergraph partitioning divides V into k sets based on the following two objectives:

1. **Equal partitions:** Vertices are assigned to processes so that the total vertex weight on each process is approximately equal.
2. **Minimized hyperedge cut:** Minimize the number of shared particles cut by the partitions.

In our hypergraph, the *vertices* are the work units selected in Section 4.1.1 (to represent

Algorithm 8 Sampling-based Interaction Load Balance Algorithm

$n \leftarrow$ number of particles, $p \leftarrow$ number of processes,
$m \leftarrow$ number of interactions, $s \leftarrow$ adaptive sampling threshold

	Step	Cost
1:	Build list of interactions per particle	*incurred*
2:	Adaptively sample interactions (Alg. 6)	$O(s\frac{n}{p} + \frac{m}{p}\log(s))$
3:	Construct hypergraph (Alg. 7)	$O(s\frac{n}{p})$
4:	Partition hypergraph	$O(s\frac{n}{p}\log(s\frac{n}{p}))$
5:	Redistribute particles, samples, setup ghosts	*incurred*
6:	Build list of interactions per particle	*incurred*
7:	Interaction $\rightarrow$ particle $\rightarrow$ subset $\rightarrow$ process	$O(\frac{m}{p}\log(s))$

interactions), and *hyperedges* represent particles (storage units). Algorithm 7 shows our procedure. We first add all particles as hyperedges to the sampled interaction hypergraph H to ensure the graph is connected (line 2). We add the work units from Section 4.1.1 as vertices (line 4) that will be partitioned into equal partitions. We add edges between the vertices (work units) and the needed hyperedges (particles) to preserve the *spatial proximity* information in the graph (line 5). We use a hypergraph partitioner to partition the resulting hypergraph.

4.1.3 Interaction-Based Load Balance Algorithm Using Sampling and Hypergraph Partitioning

Algorithm 8 shows all steps of our approach together with phases of a host N-body application. To quantify the asymptotic overhead of our algorithm, we list the computational complexity of each phase. Again, since we have chosen to use a hypergraph as our model, the complexities reflect those of hypergraph partitioning. For all complexities, p is the number of processes, n is the number of particles, m is the number of interactions, and s is our sampling threshold. The complexities of some phases are listed as *incurred*. These are phases that an N-body application would perform regardless of whether it uses our load balancing approach, so we do not count the runtime of these phases as overhead.

Our load balance algorithm starts by building interaction lists for each particle. The application would need this step (or at least a loop, if not a list) to compute interactions, so the cost is not part of overhead. The list of interactions is then passed to Algorithm 6, which samples interactions and constructs work units. We then construct a model, in this case a hypergraph, from the work units in Algorithm 7. We pass this model to partitioning. Assuming a power law distribution as mentioned in Section 4.1.1, the number of work units added to the hypergraph is $O(n)$. The work units in the tail of the hypergraph do not increase this upper bound. Because the graph is constructed in a distributed manner across all processes, the cost is $O(s\frac{n}{p})$. This cost can vary based on the load balance of the input graph, but for this analysis, we assume that the input is not highly imbalanced initially, which is true for all but the first invocation, assuming our algorithm is run frequently.

Hypergraph partitioning is $O(|V|log(|V|))$, in the size of the input graph, and for our graph, $|V| = s\frac{n}{p}$, which gives $O(s\frac{n}{p}log(s\frac{n}{p}))$ for phase 4. Thus, the complexity is in terms of n and our algorithm partition n objects instead of $m = p(\frac{n}{p})^2$ interactions.

After partitioning, we rely on the application to distribute work according to the outcome of partitioning. These costs are incurred. Last, during the force computation, we must add logic to check each interaction computation against our computed assignment, which is $O(\frac{m}{p}log(s))$. The extra $log(s)$ factor reflects the range lookup required for the small number of particles with split interactions.

4.2 Applications and Implementation

Our load balance algorithm implementation requires support libraries for partitioning and for geometric range queries. Several hypergraph partitioning libraries are freely available [36, 80]. In this work, we use the hypergraph partitioner from Zoltan [37], which is developed by Sandia National Laboratories. We use the k-d tree implementation from

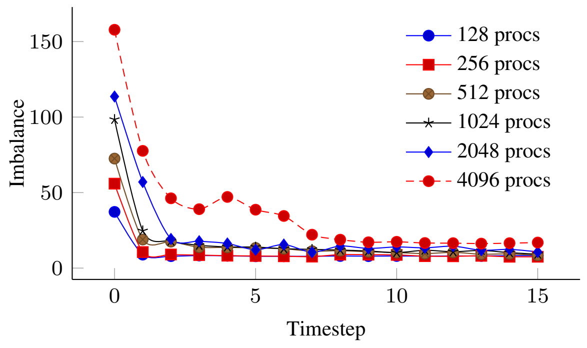
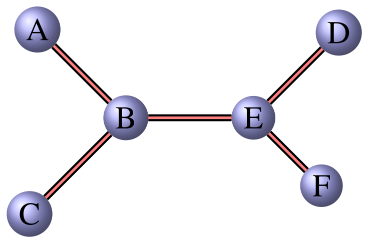
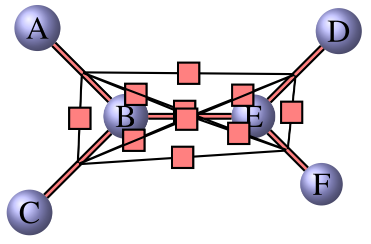


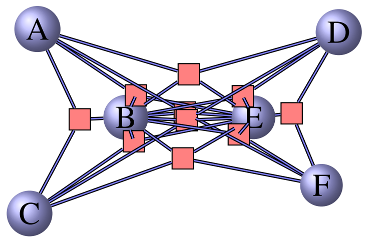
Figure 4.2: Imbalance over Time in ParaDiS with Built-in Recursive Bisection Load Balancer



(a) Particles (blue spheres) and Arms (red lines)

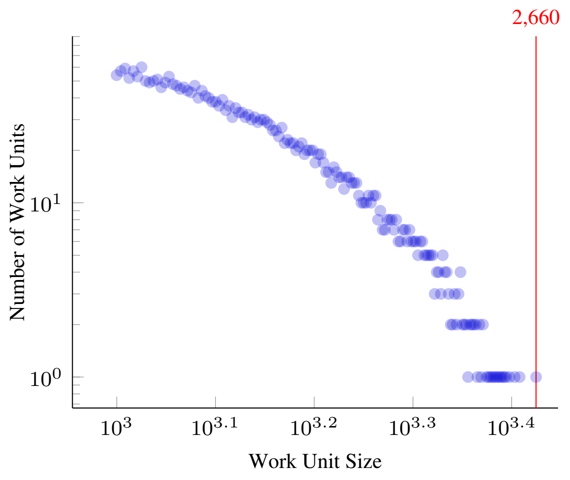


(b) Interactions (red squares) between Arms (red edges)

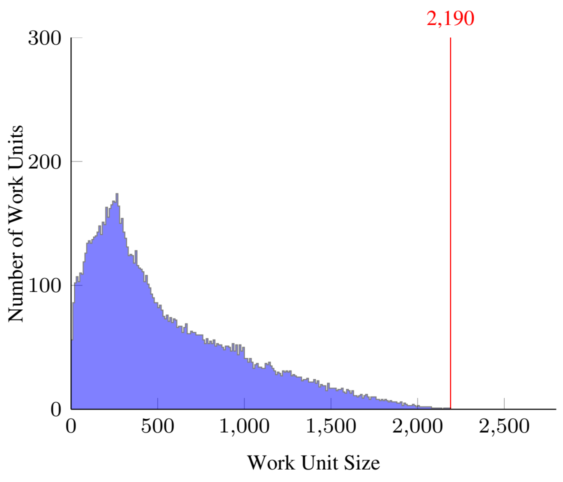


(c) Hypergraph: Vertices (red), Hyperedges (blue circles + adj. edges)

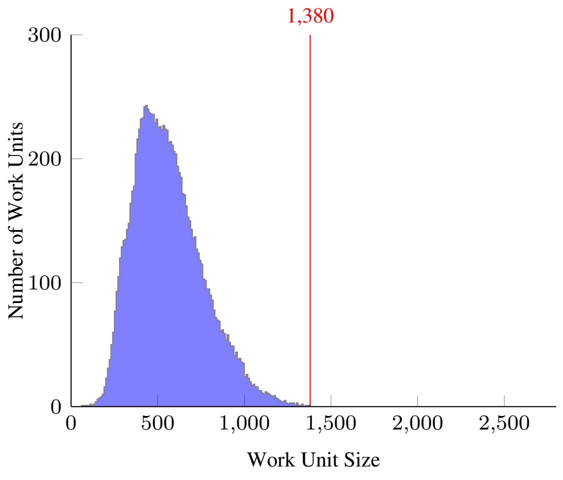
Figure 4.3: ParaDiS Computation as a Hypergraph



(a) Random Sampling



(b) Per-Particle Sampling



(c) Adaptive Sampling

Figure 4.4: Effect of Sample Size and Sampling Strategy on Work Unit Size Variability in Barnes-Hut

the CGAL [2] Computational Geometry Library for the nearest neighbor computation.

4.2.1 Barnes-Hut

The first application to which we have applied our interaction-based load balance algorithm is our own implementation of the classic Barnes-Hut algorithm, described in Section 2.6.3. In this implementation, we run our load balance algorithm at the end of each timestep. We generate our hypergraph by extracting the particle interactions from the octree data structure. Once partitioned, we redistribute the particles and assign interactions. As a baseline comparison for our results, we use a decomposition that allows assignment of any particle (along with its interactions) to any process, which is more flexible than many implementations of spatial decomposition. To preserve locality, we ordered the atoms by a space filling curve as done by Winkel, et al. [104], Warren and Salmon [101], and used with modifications by Sundar, et al. [92]. The related work shows speedup for 'homogeneous' and 'non-homogeneous' particle distributions; the drop in scalability for 'non-homogeneous' particle distributions reveals that this load balancing scheme is insufficient for this case, which our work targets. Unfortunately, the prior work does not explicitly quantify the load imbalance.

4.2.2 ParaDiS

The second application we use in our experiments is ParaDiS [6, 24]. Figure 4.2 shows the effectiveness of the recursive bisection load balance algorithm. This fully distributed approach improves load balance over time. However, beyond some point, it cannot improve load balance further due to its approximate assignment of interactions. We use the lowest load imbalance values achievable by the built-in load balance algorithm as the baseline for our comparisons in Section 4.3.3.

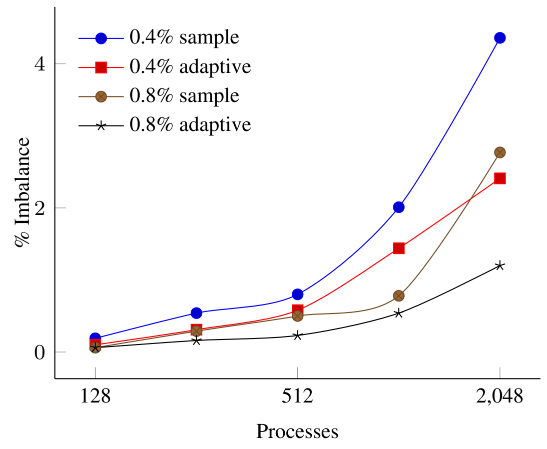
Figure 4.3 demonstrates how we describe the ParaDiS computation as a hypergraph of dislocation nodes (particles) and interactions, where a *dislocation node* is a degree of freedom in the problem. Dislocation nodes are the units of data stored in application data structures, and *arms* or *segments* are the connecting edges, as shown in Figure 4.3(a). ParaDiS imposes a regular grid of cells to discretize the space, which is used to determine proximity and divide the interactions into short and long range. A segment interacts with all other segments in its own cell and the cells surrounding it, 27 cells in total (assuming periodic boundaries). A *segment interaction* is a unit of work in ParaDiS, as illustrated by red squares in Figure 4.3(b). Each interaction involves three or four dislocation nodes (particles), unlike many n-body applications which define interactions between pairs of particles. Figure 4.3(c) demonstrates the hypergraph that we use for ParaDiS. The *dislocation nodes* and *segment interactions* are the same as in Figure 4.3(b) (shown as blue spheres and red squares). The *dislocation nodes* become the hyperedges, connected to all interactions that they support.

4.3 Performance Evaluation

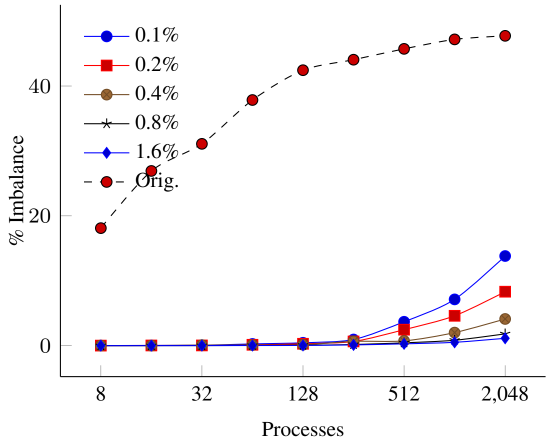
For our experiments, we use a Linux cluster with nodes consisting of two Hex-core Intel Xeon EP X5660 processors running at 2.8 GHz, with twelve cores per node and 22,272 cores total. All nodes are connected by QDR Infiniband. We use GCC 4.4.7 and MVAPICH v0.99 on top of CHAOS [1], an HPC variant of RedHat Enterprise Linux (RHEL), running at Linux kernel version 2.6.32.

4.3.1 Distribution of Work Unit Sizes and Impact on Performance

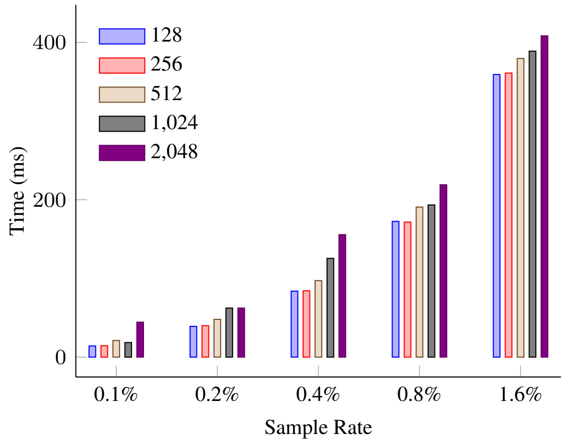
This section examines the distribution of work unit sizes under the sampling strategies described in Section 4.1.1, and how more uniform distribution leads to more evenly distributed load. These experiments use a Barnes-Hut problem with 32K particles, which we strong scale from 8 to 2,048 processes. We chose this problem since it is the largest problem that can fit into memory for 8 processes. We chose strong scaling since reproducing density variations for weak scaling is difficult. Strong scaling allows us to use the same



(a) Per-Particle vs. Adaptive Sampling

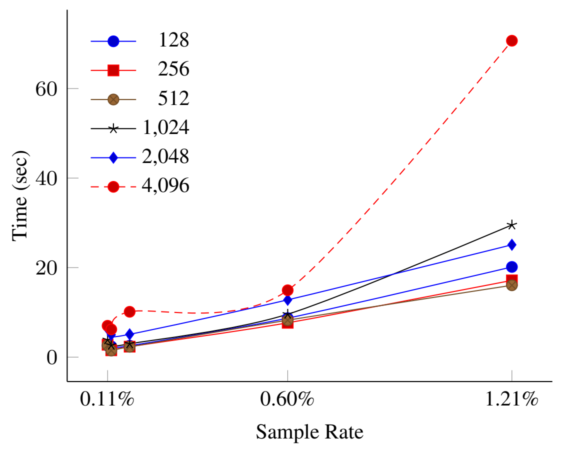


(b) Sample Rate and Resulting Imbalance



(c) Sample Rate & Aggregate Algorithm Cost

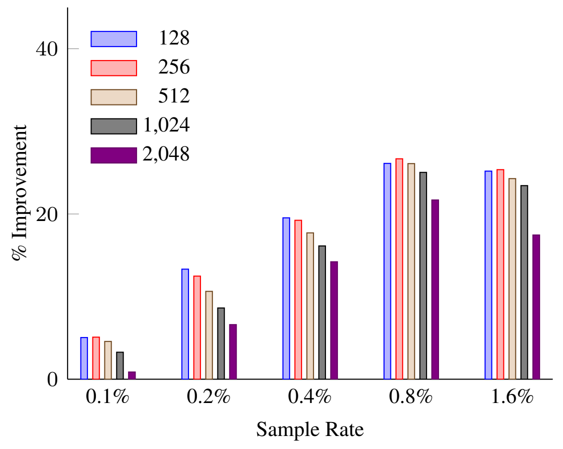
Figure 4.5: Impact of Sampling Strategy on Resulting Imbalance and Cost of Load Balancing Algorithm



(a) Partitioner Time

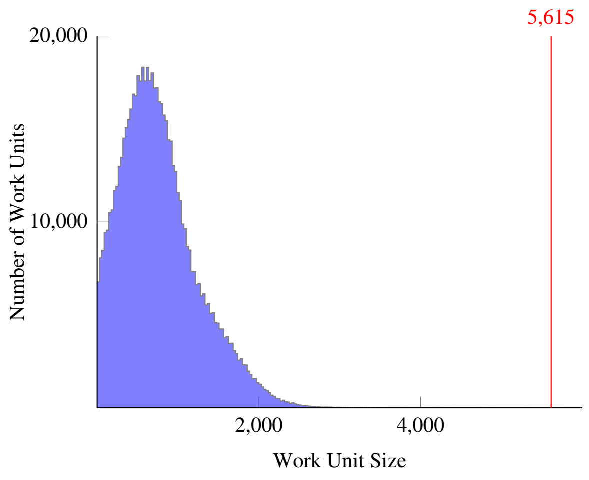
LB	Number of Processes				
	128	256	512	1,024	2,048
0.1%	101.64	51.26	26.29	13.21	7.95
0.2%	92.78	47.27	24.62	13.14	7.49
0.4%	86.13	43.62	22.67	12.06	6.88
0.8%	79.08	39.60	20.36	10.78	6.28
1.6%	80.08	40.31	20.86	11.01	6.62
Original	107.04	54.01	27.55	14.38	8.02

(b) Total Computation Time (sec)

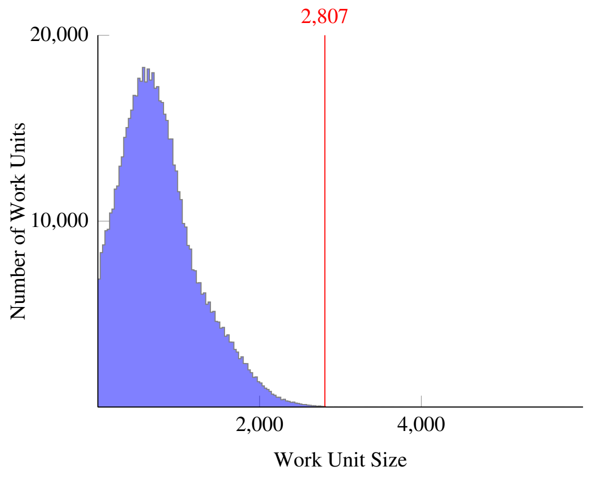


(c) Improvement over Original

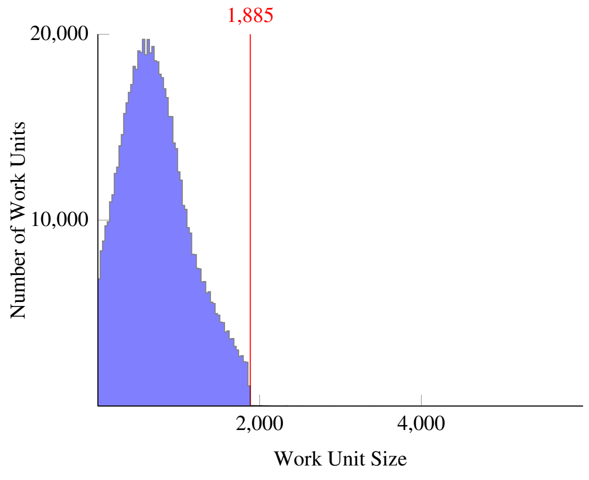
Figure 4.6: Impact of Sampling Rate on Performance of Graph Partitioner and Barnes-Hut Application



(a) Per-Particle Sampling (0.11% sample)

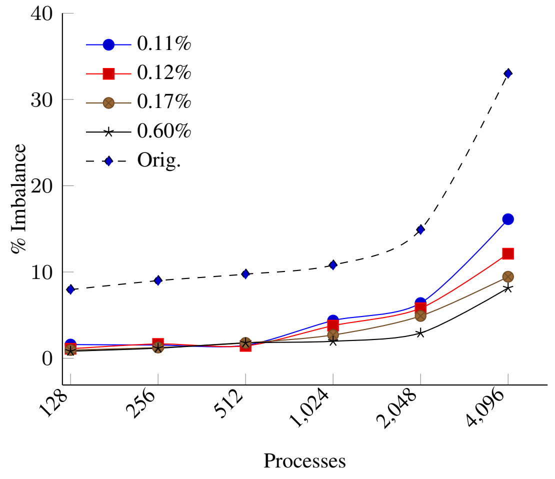


(b) Adaptive Sampling (0.12% sample)

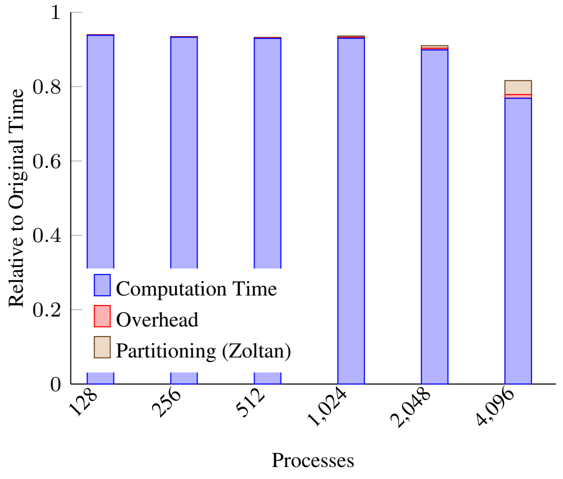


(c) Adaptive Sampling (0.17% sample)

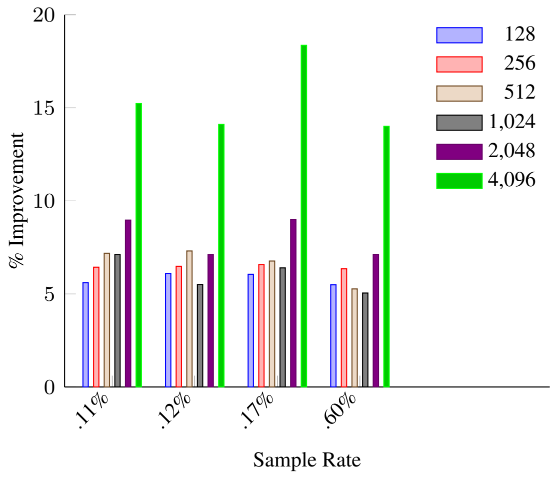
Figure 4.7: Effect of Sample Size and Technique on Work Unit Size Variability in ParaDiS



(a) Resulting Imbalance



(b) Costs for 0.17% Sample Rate



(c) Improvement over Original

Figure 4.8: Impact of Sampling on Load Balance and Application Performance

problem at all scales so data points are comparable.

Figure 4.4 shows the effect of sample size and sampling strategy on the variability of work unit size in Barnes-Hut. As discussed in Section 4.1.1, sampling a simulated domain with high density variability results in a power law distribution of work units, as Figure 4.4(a) shows (note that the horizontal axis is log-scale). The maximum work unit size (2,660) is indicated at the top of the figure. We can partially mitigate the density properties by proportionally sampling interactions per-particle (Figure 4.4(b)). By running Algorithm 6 *without* any adaptation, we achieve a maximum work unit size of 2,190. We achieve a tighter, nearly normal distribution of the interactions that each work unit represents by using the full adaptive sampling approach from Algorithm 6, as Figure 4.4(c) shows. Here, the maximum work unit size is 1,380.

Figure 4.5 demonstrates the impact of the sample size and strategy. Figure 4.5(a) compares how the two approaches to sampling impact the ability of the load balance algorithm to assign equal partitions to processes, where imbalance is:

$$\text{imbalance} = \frac{\text{maximum load - average load}}{\text{average load}}$$

Because adaptive sampling makes the distribution of sample weights more uniform, it results in better partition quality (lower imbalance), as Figure 4.5(a) shows. Roughly, adaptive sampling can achieve the same partitioning quality as doubling the sample size, and keeps the number of resulting work units roughly the same as its non-adaptive version. Since the number of work units directly impacts the cost of our load balancing method, using a sampling approach that more uniformly distributes work unit sizes is of increasing importance as process count increases.

We also evaluate the quality of the load balance achieved by our load balancing approach, measured by the percent imbalance. We compare the different sampling rates and the traditional particle-based approach. Figure 4.5(b) shows that, while imbalance of the application using a particle-based method grows quickly as the number of processes increases, our direct interaction assignment scheme is able to achieve much lower levels of imbalance. Because our method is sampling based, quality is, to a large extent, a function of the number of samples, or the work units assigned to processes. When the number of work units is too small, quality partitioning is difficult to achieve. However, even modest sample sizes of under 1% of all interactions allows for quality partitions. Samples above 1% show diminishing returns on partition quality.

The imbalance increases with the number of processes in this strongly scaled example since we have fewer individual work units to assign to each process. Thus the job of the partitioner is more difficult. One of the strengths of our method is that we can choose the number of work units that we select, which allows us to trade off between cost and accuracy.

Figure 4.5(c) shows the aggregate overhead of our load balance algorithm. As mentioned earlier, sample count directly impacts the cost of our algorithm because it determines the cost of partitioning, sampling and nearest-neighbor assignment. The figure clearly demonstrates the linear relationship between the sample size and the cost of our algorithm. The same size bars within a sample size group would indicate consistent aggregate compute time across all processes, i.e., perfect scaling. While the sampling and nearest-neighbor assignment scale well, the overhead numbers show some degradation in scalability due to the limited scalability of the partitioner. The latter is a well known problem, but can be remedied. Since our method is sampling based and the resulting graph of work units is small, we could gather this graph on a smaller number of processes for partitioning, and then scatter the results. This optimization would allow us to pick the optimal scale for the partitioner independent of the scale at which the application is run, and thus reduce overall runtime. We explore this optimization in Section 5.

Sampling enables us to use the graph partitioning when partitioning the entire graph would be prohibitively expensive. Figure 4.6(a) shows the hypergraph partitioner time (Zoltan) for different sample rates using our ParaDiS data set. The complete graph for this data set has 1M particles and 547M interactions. It takes seconds to partition the graph with just 0.11% of the interactions sampled; partitioning the complete graph would be extremely expensive.

Overall, our sampling approach is an effective way to reduce the size of the graph to be partitioned while preserving the quality of the resulting partitions. The savings in partitioner time make the approach of explicitly load balancing the interactions affordable. As mentioned in Algorithm 4.1.3, we have effectively reduced the complexity of balancing interactions to that of balancing particles.

4.3.2 Impact on Barnes-Hut Performance

We show the impact of our load balance algorithm on performance of the Barnes-Hut benchmark in Figure 4.6. The total times for our 32K particle simulation with different sampling rates are listed in Figure 4.6(b); Figure 4.6(c) illustrates these total times relative to the original load balance algorithm. With 0.1% sample, our method shows marginal performance improvement over the original load balance algorithm; poor performance is due to undersampling and the partitioner's inability to form equal partitions from the work units provided. With more sampling, our method outperforms the original method; in this example, the point of diminishing returns is apparent for the sampling ratio of 1.6%. Although our method gains more from its accuracy at scale, two reasons inhibit its performance when the sampling ratio is held constant while increasing the process count. First, the partitioner has fewer work units to divide between a larger number of partitions, resulting in slight increase in load imbalance. In a more realistic scenario, one would chose the sampling ratio relative to both the problem size and the number of partitions

LB	Number of Processes					
	128	256	512	1,024	2,048	4,096
0.11%	6118.71	3061.16	1529.36	774.96	405.14	228.51
0.12%	6086.44	3059.56	1527.47	788.39	408.94	231.51
0.17%	6089.01	3056.97	1536.31	780.96	405.07	220.05
0.60%	6126.13	3064.07	1560.99	792.21	413.34	231.80
Original	6482.01	3271.84	1647.85	834.33	445.09	269.57

Figure 4.9: Total Computation Time of ParaDiS (seconds)

needed, thus not observing this performance degradation. Second, the partitioner scales poorly when partitioning the same small graph on more processes (as discussed previously), necessitating the decoupling of the partitioner scale from the problem scale.

Our interaction-based load balance algorithm with sufficient sampling performs well and clearly outperforms the particle-based load balance algorithm. Overall, we observe 23-26% improvement for the optimal sampling rate.

4.3.3 Impact on ParaDiS Performance

We evaluate the impact of our load balance algorithm on the performance of ParaDiS. We use a highly dynamic crystal simulation input set for ParaDiS, with 1M degrees of freedom at the beginning of the simulation growing to 1.1M degrees of freedom by the end of the run. We strongly scaled this simulation up to 4,096 processes.

Figure 4.7 shows the effect of the sample size and strategy on work unit size variability. Per-particle sampling results in a distribution with a long right tail, as demonstrated in Figure 4.7(a). The maximum work unit represents 5,615 interactions, and the sample size is 0.11% of the interactions in the problem. As shown in Figure 4.7(b), with an only 0.01% increase in sample size (for a total sample rate of 0.12%), we decrease the maximum work unit size to 2,807. With an additional 0.05% increase in sample size (for a total sample rate of 0.17%), we decrease the maximum work unit size to 1,885, as Figure 4.7(c) shows.

Figure 4.8 details performance of the load balance algorithm and the application with different sampling strategies. Figure 4.8(a) demonstrates the impact that the different sampling strategies have on load imbalance, along with the lowest observed load balance achieved by the built-in load balance algorithm. With an addition of more work units to partition and more uniform work unit size distribution, the hypergraph partitioner achieves lower levels of load imbalance. As the number of processes grows, the imbalance increases in this strong scaling problem due to the partitioner having to divide the same number of work units into more partitions. The partitioner needs more work units to work with at scale; a higher sampling rate or a bigger problem with the same sampling rate would allow the partitioner to accomplish similar levels of imbalance at scale.

Figure 4.8(b) shows the cost break down for a sample rate of 0.17% relative to the performance using the existing load balance algorithm. The computation time required with our load balance algorithm is lower, especially at larger process counts when the existing load balance algorithm performs significantly worse. The time spent in the hypergraph partitioner increases as the process count increases because the partitioner does not scale optimally and must partition the same number of work units into more partitions, a more difficult problem to solve.

Figure 4.8(c) shows the runtime of the problem with different sampling levels, relative to the runtime of the problem with the existing algorithm. For all sampling levels, our method shows greater improvement as the number of processes grows, due to larger improvement in load balance. Further improvement is possible over the per-particle sampling (0.11% sample) because our adaptive sampling improves the distribution of work unit sizes. Because the cost of our algorithm is dependent on the sample size, a sample rate of 0.17% only slightly outperforms one of 0.12%. Performance degrades with the 0.60% sample due to the costs outweighing the benefit of sampling more. Figure 4.9 lists the total runtimes.

We compare to the *second*, optimized load balance scheme that the ParaDiS developers implemented. Overall, we achieve improvement in performance of 6-18% over this already highly optimized and dynamically load balanced production application.

4.4 Summary

Traditional parallel N-body load balance algorithms use approximate methods to assign computational work, or *interactions* to processes. Those that do balance interactions directly, such as force decomposition, do so with coarse granularity because the interaction graph is large and costly to partition directly. We have developed the first approach for explicitly balancing interactions in N-body applications at runtime. Our approach uses sampling to reduce the size of the interaction hypergraph by several orders of magnitude, and aggressive *adaptive* sampling to make the size of sampled work units more uniform. The combination of these two techniques enables extremely efficient partitioning. Using these techniques in conjunction with a hypergraph partitioner to minimize inter-process communication, we have shown for two optimized parallel applications, Barnes-Hut and the ParaDiS dislocation dynamics code, that our method achieves 23-26% and 6-18% improvement in overall performance. To our knowledge, our approach is the first to balance interactions directly with such fine granularity.

5. LAZY LOAD BALANCING

The largest supercomputers have millions of independent processors, and concurrency levels are rapidly increasing. For ideal efficiency, developers of the simulations that run on these machines must ensure that computational work is evenly balanced among processing elements. Rebalancing throughout application execution is necessary because many simulations have workloads that evolve dynamically over time. For example, physical simulations may use mesh cells, particles, or other logical elements to represent their domains. However, the computational work per element may change as the physical system evolves, causing an initially balanced assignment of work to processors to become imbalanced over time.

The cost of imbalanced load increases with scale. Most large-scale scientific simulations today use an SPMD parallel programming model, and underloaded processes will wait on the overloaded ones during frequent synchronizing operations. Specifically, a simulation with more processes will waste more resources than a smaller-scale simulation when waiting on a single slow process. Moreover, in an application's strong-scaling limit, it becomes increasingly difficult to balance computational work evenly as the available parallelism becomes more and more coarse-grained with respect to the number of processes. We must therefore fix even small imbalances at scale.

For this reason, many large-scale parallel applications use load balance algorithms to redistribute work evenly. Depending on the application, a fast, local load balance algorithm may be suitable. However, graph partitioners are typically employed for the best balance, efficient communication optimizations, and for work assignment to be aware of locality within the simulated physical domain [36, 80]. Graph partitioners are computationally intensive and can be themselves a performance bottleneck that requires sophisticated parallelization. Further, they typically exhibit worse strong scaling performance than the simulation itself. The poor scaling causes graph partitioners to be too expensive for use at large scale.

In this Section, we describe *lazy load balancing*, a new approach we propose for load balancing, that allows even a poorly scaling, high latency load balance algorithm (such as graph partitioning) to be used efficiently in large-scale applications. In lazy load balancing, we decouple the load balance algorithm from the application and run it on potentially fewer, separate processors. In this MPMD configuration, the algorithm can execute concurrently with the application and with higher parallel efficiency than if it were run on the same processors as the simulation. Work is reassigned *lazily* as assignment directions become available, and the application need not wait for the load balance algorithm to complete.

One challenge of lazy work assignment is that application state can change while the load balance algorithm computes a new assignment, resulting in a change in how work is assigned to processes. We call this change in work distribution *drift*. Our lazy load balancing approach exploits the fact that application state changes slowly in most applications, and a work assignment computed from an application state in the past typically continues to be a good assignment for many time steps. As part of this work, we have developed techniques that guarantee a correct application state after lazy load balancing, even after significant drift.

Our lazy load balancing approach allows an application to remove load balance computation from its critical path, and to decouple, offload, and right-size a partition of processors for the load balance algorithm. The contributions described in this Section are:

- An empirical evaluation of drift metrics for two applications, Barnes-Hut and ParaDiS;
- Techniques that map a lazily computed work assignment to the the current state of a simulation after drift;

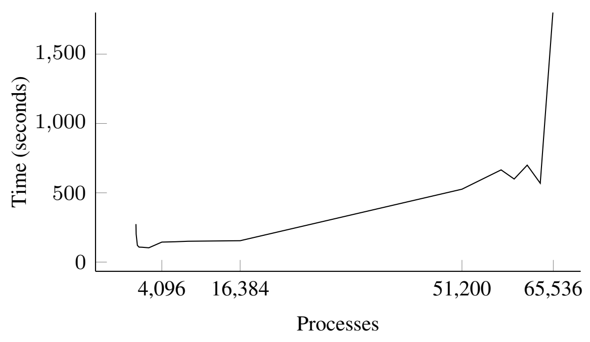


Figure 5.1: Graph Partitioner Runtime (Strong Scaling). BGQ, 265K Vertices, 65K Partitions.

- A prototype framework for decoupling and offloading load balance computation;
- An analytical model that predicts the right size for a lazy load balance algorithm partition from a set of application characteristics.

Combined, these techniques enable lazy load balancing. We show that our approach can improve performance by up to 46%, even for applications with substantial drift.

Section 5.1 discusses scalability of a graph partitioner, a computationally expensive load balance algorithm that requires sophisticated parallelization. Section 5.2 outlines the application properties and drift metrics that make lazy load balancing feasible. Section 5.3 is an overview of the differences between inline and lazy load balancing. Section 5.3.1 introduces the concept of decoupling load balance algorithm resources to optimize execution. Section 5.3.2 describes how offloading the load balance computation and overlapping it with the application execution impacts the application. Section 5.5 describes our framework and its implementation. Section 5.6 shows our results.

5.1 Graph Partitioning is Useful but Scales Poorly

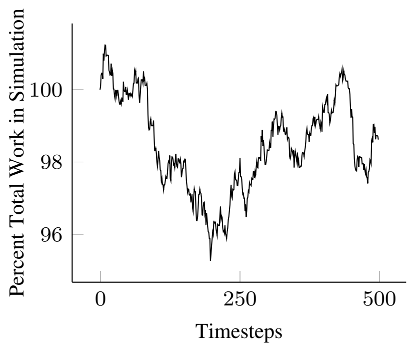
Prior work has shown that the load balancing problem is essentially a balanced graph partition problem, and therefore NP complete [46, 55]. In practice, many applications employ local load balancing to balance work among neighboring processes. However, these methods do not balance or minimize communication as well as a graph partitioner can. For this reason, many applications still employ a graph partitioner for load balancing. While graph partitioning is well studied and many heuristic solutions exist [36, 80], *parallel* graph partitioning algorithms are known to scale poorly. Figure 5.1 shows strong scaling performance of a parallel graph partitioner running on an IBM Blue Gene/Q supercomputer for 32 to 65,536 processors. Peak efficiency is achieved with 2,048 processors, and from this point on, runtime increases. On 65K processes, the algorithm spends all of its time in communication, and the runtime skyrockets. At scale, we could do better by running the graph partitioner on fewer processes and avoiding these excessive costs.

5.2 Application Drift

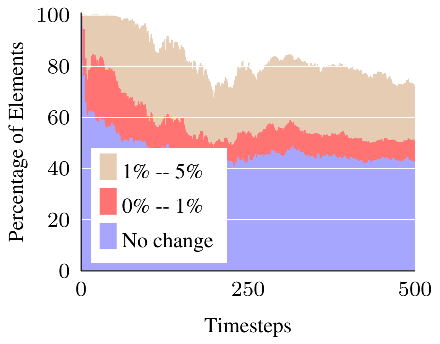
Even if we run the graph partitioner on fewer processes, it can take seconds to run. This is well above the time step length of many applications. However, if we could run a load balance algorithm concurrently with the simulation, thus amortizing the load balance algorithm execution time over many time steps, we could eliminate the excessive overhead.

A potential problem with lazy load balancing is that application state changes over time. Work per element may change, as may the number of elements. We call this change *drift*. However, in this Section we observe that the work distribution in most parallel SPMD applications changes slowly. In many cases, a balanced assignment computed from a past application state is a good approximation of a balanced assignment for the current state. That is, we *can* compute the assignment asynchronously and apply it *lazily* when the result is available.

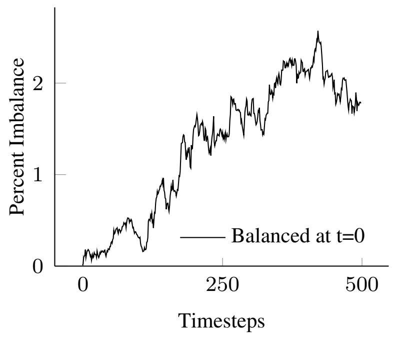
Given an assignment $A : V \to P$ for a past time step's elements V, we can compute a *drifted* assignment $A' : V' \to P$ for the current time step's elements V' by determining the



(a) Total Work Over Simulated Time

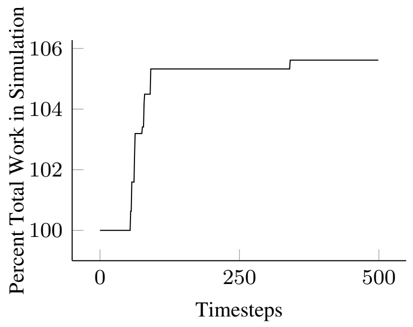


(b) Work per Element Over Simulated Time

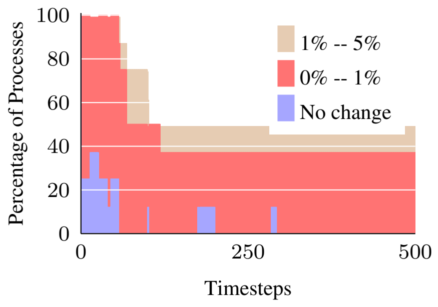


(c) Imbalance Over Simulated Time

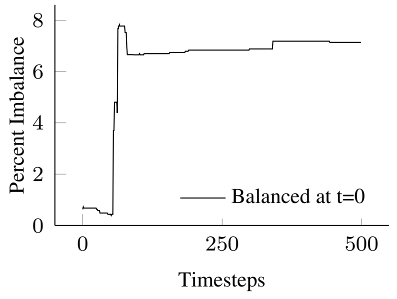
Figure 5.2: Application Drift in Barnes-Hut, 26M Interactions, 8192 Processes



(a) Total Work Over Simulated Time



(b) Work per Process Over Simulated Time



(c) Imbalance Over Simulated Time

Figure 5.3: Application Drift in ParaDiS

relationship between V and V'. Depending on the simulation, there are three possibilities:

$$A'(v) = \begin{cases} A(v) & v \in V \cap V' \\ C(v) & v \in V' \setminus V \\ undefined & v \in V \setminus V' \end{cases}$$

In the first case, v is in both the old and the new application state, and we simply reuse its assignment from A. This case will typically cover most of the elements, and it is the only case for most N-body simulations and for unstructured mesh applications with a static number of cells in the mesh. In both of these applications, $V = V'$. In the second case, v represents an application element or task that has been created since the past state, and we must construct a new function $C : V' \setminus V \rightarrow P$ to assign it. In the third case, v represents an application element or task that no longer exists in the current state and we can ignore its prior assignment. The second and third cases are typical of adaptive mesh refinement (AMR) applications, and other applications in which elements may be created based on the physics. Computing a good function for C is application-dependent and we leave the description of this function to later sections.

5.2.1 Assignment Validity

For A' to be *valid* for the new application state it must be defined for the entire set V'. Trivially, we observe that A' is defined for $(V \cap V') \cup (V' \setminus V)$, which, by inspection, equals V'.

5.2.2 Assignment Efficiency

An assignment is *efficient* if it minimizes the deviation in Equation 2.4, as minimizing the deviation effectively minimizes load imbalance. We define the *imbalance* of an assignment $I(A)$ as the scaled maximum load on any processor minus the average:

$$I(A : V \to P) = \frac{\max_i(W(V_i)) - \frac{1}{|P|}\sum_i W(V_i)}{\frac{1}{|P|}\sum_i W(V_i)} \tag{5.1}$$

In a bulk synchronous application, the extra load corresponds to the performance degradation of any overloaded processors.

5.2.3 Empirical Evaluation of Drift Metrics

We have empirically evaluated the imbalance of drifted assignments for two applications. The first, shown in Figure 5.2, shows the drift metrics for a Barnes-Hut N-body simulation [12]. Barnes-Hut is a gravitational force simulation where the number of particles remains the same throughout the simulation, but the interactions computed per particle can change as particles move because the simulation only computes gravitational interactions within a cutoff radius. Figure 5.2(a) shows the total work, summed over all particles in the simulation, over time. We can see that this fluctuates over a range of about 5% of the total initial work. Figure 5.2(b) shows how the work per particle in the simulation changes over time. Approximately half the particles have the same amount of work throughout the execution, but the other half's work changes up to 5% per time step. Figure 5.2(c) shows the imbalance that results from this change on each successive step, assuming that we use a completely balanced assignment from time step zero. The imbalance grows only slightly, despite the larger changes in work per element in the simulation. Thus, the assignment computed at step 0 is still an efficient assignment for subsequent steps.

In Figure 5.3 we show similar drift metrics for ParaDiS [24], a material strength application that uses crystal dislocations as its elements. In ParaDiS, elements can be created or destroyed as the simulation continues, but interactions between them are computed much as they are in the Barnes-Hut N-body simulation. In ParaDiS, the total work in the simulation can grow much more rapidly than in Barnes-Hut (Figure 5.3(a)). The more rapid growth in total work is due to the creation and deletion of elements. Figure 5.3(b) shows the number of element interactions per processor in the simulation: we can see that in ParaDiS nearly all processors experience a change in workload on every time step, and that the changes grow as the simulation continues. The change in imbalance, shown in Figure 5.3(c), shows that the largest change in imbalance comes from the creation of elements, but that the use of a drifted assignment still results in an imbalance of at most 7%. These simulations and their performance with our load balancing technique were described in detail in Section 4.2.

5.3 The Lazy Load Balancing Approach

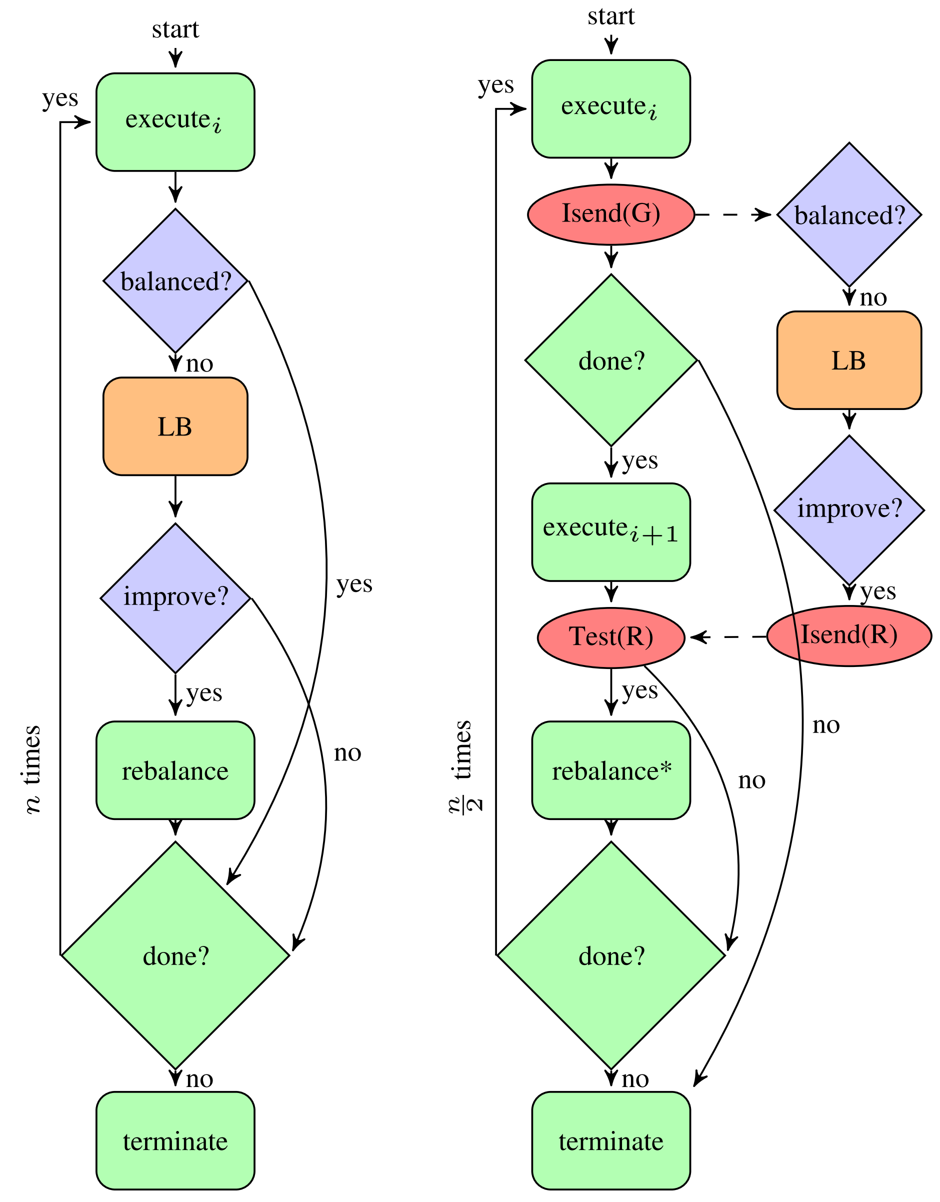
Figure 5.4(a) shows the main components of traditional approach to load balancing an application. The main steps are:

1. *When to Balance*: evaluate the imbalance, decide whether to correct load imbalance at this point in execution;
2. *How to Assign Work*: use a load balance method to compute directions on how to rebalance;
3. *Rebalance* the application if needed.

As Figure 5.4(a) shows, these steps are typically performed sequentially (in the SPMD sense) with the application's computation. Application execution pauses while load balance decisions are made. As discussed, this approach is not well suited to using a graph partitioner load balance algorithm, as these algorithms do not scale to the process counts that the application does. Potentially thousands of application processes may have to wait for the load balancing algorithm while it runs at sub-optimal efficiency.

5.3.1 Decoupling the Load Balance Algorithm

The load balance algorithm is distinct from the application calculation, and the amount of computation performed by the load balance algorithm is usually smaller than the actual



(a) Balancing Inline (b) Balancing Asynchronously

Figure 5.4: Inline vs. Asynchronous Load Balancing. Application components are shown in green, load balancing decisions are shown in blue, external libraries are shown in orange, asynchronous communication between application and load balancing processes is shown in red.

Algorithm 9 Steps of Decoupled LB ($P_{App} \cap P_{LB} \neq \emptyset$)

```
1:  for all $p_i \in P_{App}$ in parallel do
2:      Pause computation
3:      Send input to corresponding LB process $P_{LB_j}$
4:  end for
5:  for all $p_j \in P_{LB}$ in parallel do
6:      Receive input from all corresponding App process(es) $P_{App_i}$
7:      Apply LBAlgorithm
8:      Send output to all corresponding App process(es) $P_{App_i}$
9:  end for
10: for all $p_i \in P_{App}$ in parallel do
11:     Receive LBAlgo output from corresponding $P_{LB_j}$
12:     Rebalance
13:     Proceed with balanced computation
14: end for
```

computational work in the application. Using the same number of processes as the application, the load balance algorithm has much less available parallelism because the granularity of assignment for load balancing is typically much coarser than the finest granularity of elements in the simulation. *Decoupling* moves the data to be partitioned onto a different set of processes from those used by the application. The decoupling allows an algorithm such as graph partitioning to run more efficiently on a smaller number of processes.

We outline the steps of a decoupled balancing configuration in Algorithm 9 and illustrate them pictorially in Figure 5.4(b). First, the application sends its state to the corresponding load balance processes (line 3 in Algorithm 9, red fan-in in Figure 5.5(b)). The load balance processes receive the application information (line 6), run the load balance algorithm in parallel (line 7, shown in blue), and send the instructions to the corresponding application processes (line 8, red fan-out). The application then receives the instructions (line 11), rebalances (line 12), and proceeds with the computation in a balanced state (line 13, shown in green).

p	Total number of processes	t	Number of timesteps	$f(n, w)$	Runtime of a balanced timestep, run on n processes
n	Processes used by Application	u	Number of unbal. timesteps	$g(m, w')$	LB runtime on m processes
m	Processes used by Decoupled LB	w	Work in the Application	$h(m, p, w')$	Comm. bt/w m Decoupled LB and p App. procs
m'	Processes used by Asynchronous LB	w'	Work in the LB algorithm	$h'(m', n, w')$	Comm. bt/w m' Asynchronous LB and n App.procs
α	Initial Application imbalance	i	Rate of change in imbalance		

	Standard	Decoupled	Asynchronous
Processes	$p = m = n$	$m \leq n = p$	$m' + n = p$
Timesteps	$u = 0$	$u = 0$	$u > 0$
LBTime	$g(p, w)$	$g(m, w') + h(m, p, w')$	$g(m', w') + h'(m', n, w')$
AppTime	$tf(p, w)$	$tf(p, w)$	$tf(n, w) + u\alpha f(n, w)$
TotalTime	$g(p, w') + tf(p, w)$	$g(m, w') + h(m, p, w') + tf(p, w)$	$tf(n, w) + u\alpha f(n, w)$

Table 5.1: Break Down of Execution Time for Standard, Decoupled, and Asynchronous Approaches to Load Balancing

5.3.2 *Asynchronous, Concurrent Load Balance Algorithm*

To avoid pausing the application computation while computing a load balance assignment, we can offload the load balance algorithm computation to a separate *balancing partition*. Separation allows our load balance algorithm to run concurrently with the application, overlapping application computation and load balance algorithm computation. Assignments are applied by the application *lazily* as they become available.

We outline the steps of an asynchronous balancing configuration in Algorithm 10 and show it pictorially in Figure 5.5(c). First, the application sends its state to the corresponding load balance processes (line 2 in Algorithm 10, red fan-in in Figure 5.5(c)). The application proceeds with the computation in the imbalanced state until further notice (line 3, first green block). The load balance processes receive the application information (line 6), run the load balance algorithm in parallel (line 7, shown in blue), and send the instructions to

Algorithm 10 Steps of Asynchronous LB ($P_{App} \cup P_{LB} = \emptyset$.)

```
 1: for all $p_i \in P_{App}$ in parallel do
 2:     Send input to corresponding LB process $P_{LB_j}$
 3:     Proceed with imbalanced computation until further notice
 4: end for
 5: for all $p_j \in P_{LB}$ in parallel do
 6:     Receive input from all corresponding App process(es) $P_{App_i}$
 7:     Apply Load Balancing Algorithm
 8:     Sent output to all corresponding App process(es) $P_{App_i}$
 9: end for
10: for all $p_j \in P_{App}$ in parallel do
11:     Receive LBAlgo output from corresponding $P_{LB_j}$
12:     Pause computation, Rebalance
13:     Proceed with balanced computation
14: end for
```

the corresponding application processes (line 8, red fan-out). The application then receives the instructions (line 11), rebalances (line 12), and proceeds with the computation in a balanced state (line 13, second green block).

One of the differences with Algorithm 9 is that by the time that the application receives the load balance directions (line 11), the application has moved forward from its state when it sent the instructions. We assume that the application can only be rebalanced at timesteps. The load balance algorithm uses a snapshot s_0 of the application at timestep t_0 to compute how to rebalance the application; we refer to this decision as d_0. Assume that d_0 results in a balanced state of the application. However, while the load balance processes compute, the application advances k timesteps. Applied to s_k, d_0 may result in an imbalanced state.

The asynchronous approach can optimize the resources for the load balance algorithm, and overlap the load balance algorithm with the main computation to avoid wasting resources. Running the load balance algorithm asynchronously also means we can run the load balance algorithm continuously, correcting the imbalance with a higher frequency than otherwise would be affordable. Figure 5.5(d) demonstrates how the application can continue sending its state to the load balance processes, and the load balance algorithm can continually compute the rebalancing directions. The frequency of the rebalancing should be dictated by the amount of *drift* that an application can tolerate, as discussed it in Section 5.2 vs. the number of resources a user is willing to dedicate to load balancing and with that the frequency in which new work assignments can be produced.

5.4 Performance Model for Allocating Resources

To understand the performance of different load balancing configurations and to minimize the overall runtime, we develop a performance model that provides a quantitative basis for deciding how to allocate the available resources in the system. The cost model captures the performance characteristics of the standard, decoupled, and asynchronous load balancing configurations. To simplify the explanation, we assume that the load balance algorithm is able to balance the application fully.

Figure 5.5 illustrates how the different load balancing configurations result in different resource usage. The height of the block indicates the number of processes (resources); the length of the block indicates the runtime. We consider application computation, load balance algorithm computation, and communication overhead. Table 5.1 summarizes the variables that we use in the performance model.

Model Input:

- p -- Total number of processes;
- α -- Initial application imbalance;
- t -- The smaller of the number of time steps that the simulation takes, or the number of steps the simulation takes prior to *drift* becoming too large;
- Runtime of the application, load balance algorithm, and communication overhead, as defined in the subsections to follow.

Model Output:

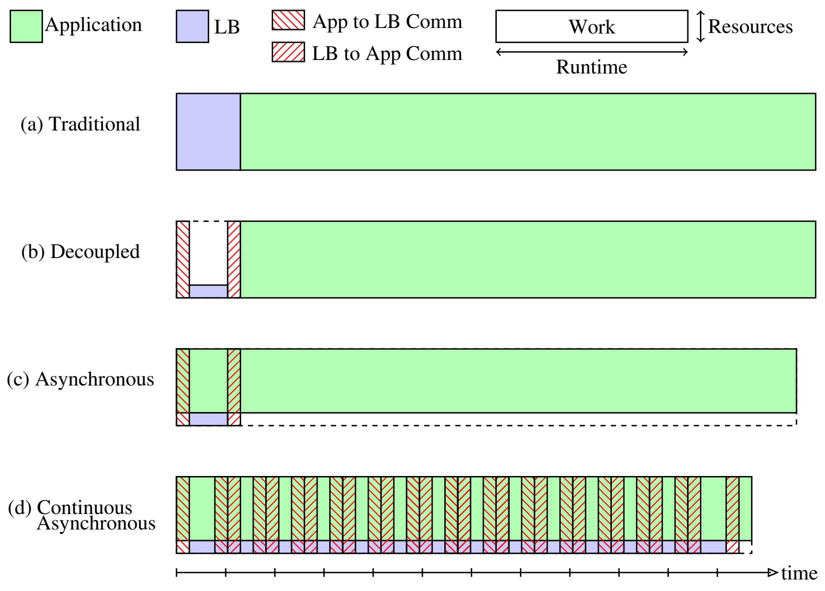


Figure 5.5: Resource Diagram of Load Balancing Configurations

- Choice of standard, decoupled, or asynchronous configuration;
- m -- Number of processes for the load balance algorithm;
- n -- Number of processes for the application.

Predictive modeling of the performance of specific load balance algorithms (such as graph partitioners) and physical simulation algorithms are research topics in their own right and are outside the scope of this dissertation. Here, we use curve fits of the load balance algorithm and application runtime, from which our model determines how to allocate the resources in the system.

5.4.1 Modeling the Application

The computation time of an application, $f(n, w)$, is a function of the number of processes and the amount of work in the application, and we model it with a curve fit. The application uses p processes in the standard and decoupled configurations, and $n < p$ processes in the asynchronous configuration.

At the beginning of the execution, the application has an initial imbalance α. Because the standard and decoupled configurations pause the application execution while the load balance algorithm computes, the application is redistributed immediately and computes all t timesteps in a balanced state. In the asynchronous configuration, the application continues running imbalanced for u steps, while the offloaded load balance algorithm computes the directions; we discuss how we determine u in the next subsection. Once the rebalancing directions are received, the application is redistributed and completes the remaining $t - u$ steps in a balanced state. The application execution time therefore depends on whether the timesteps are executed prior to or after redistribution, and how many processes the application uses.

The application provides t', the number of time steps that it intends to run. For simplicity, we assume that the *drift* metric i, rate of change in imbalance, is constant. We estimate i at runtime or accept it as input from the application. Using the rate of change, we compute how many steps t' the application can execute before it needs to be rebalanced again because it reaches an imbalance threshold tol:

$$t'' = \frac{log(tol \times f(n,w)) - log(f(n,w))}{log(1+i)} \tag{5.2}$$

We then choose the smaller of those values and use it as the number of timesteps in our model:

$$t = min(t', t'') \tag{5.3}$$

5.4.2 Modeling the Load Balance Algorithm

The computation time of a load balance algorithm, $g(m, w')$, is a function of the amount of work and the number of processes used. The amount of work in the load balance algorithm, w', is usually smaller than the amount of work in the application itself,

w. The performance of each load balance algorithm is modeled with curve fitting, and the algorithm will use p processes in the standard configuration, $m \leq p$ processes in the decoupled configuration, and $m' < p$ processes in the asynchronous configuration.

In the decoupled and asynchronous configurations, the application processes first send relevant information to the load balance processes. Once the load balance directions are computed, they are sent back to the application. We refer to this overhead, $h(m, p, w')$ and $h'(m', n, w')$, as communication overhead for decoupled and asynchronous configurations. In our implementation, we use asynchronous communication between the application processes and the load balance processes. However, for simplicity we model this communication as if it cannot be overlapped with the computation. The runtime of gather/scatter operations depends on the number of the application processes and load balance processes, and the amount of data sent.

In the asynchronous configuration, the application proceeds in an imbalanced state for u steps, while the load balance information is gathered, the load balance algorithm executes, and the directions are scattered back to the application. We calculate u as a fraction of the time until the directions are available, and the length of the imbalanced timestep:

$$u = \frac{g(m, w') + h'(m', n, w')}{\alpha\, f(n, w)} \tag{5.4}$$

The extra time that the application runs because of the delay in rebalancing becomes part of the overhead of the asynchronous load balancing configuration.

5.4.3 Modeling Overall Runtime

The total time for the standard and decoupled configurations is a sum of load balance algorithm time and application time, including the gather/scatter overhead in the decoupled configuration. The total time for the asynchronous configuration only includes the application time since the load balance algorithm time is overlapped; however, the application runs

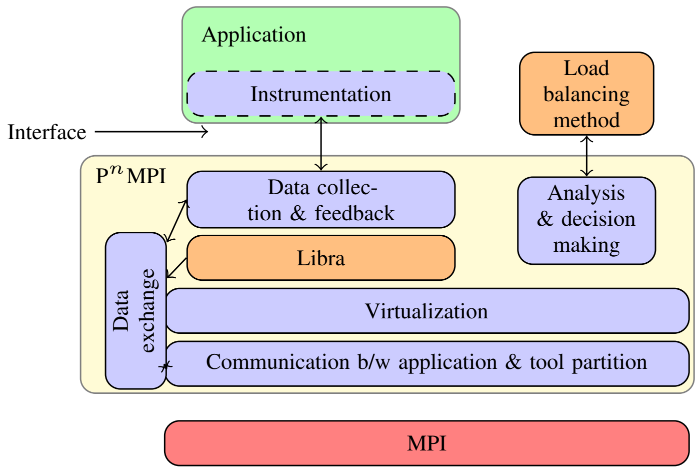


Figure 5.6: Asynchronous Load Balancing Infrastructure and Interaction Between the Components. Framework Modules are in Blue.

using fewer processes because some of the resources were reserved for the load balance algorithm, so that the application runtime may be longer. Additionally, as discussed above, the application runs in an imbalanced state for u steps, which increases the total time as well.

The decoupled configuration is a generalization of the standard configuration when $m = n = p$, $h(m, p, w') = 0$, so we only discuss how to choose between the decoupled and asynchronous configurations. Given p, the model attempts to choose m and n such that:

$$min \begin{cases} g(m, w') + h(m, p, w') + t\, f(p, w) \\ t\, f(n, w) + u\alpha\, f(n, w) \end{cases} \tag{5.5}$$

We discuss an instantiation of our model for our test applications and load balance algorithm in Section 5.6.2.

G	Send Graph
R	Rebalance

G	R
0	0
1	0
1	1

Figure 5.7: Valid State Transitions

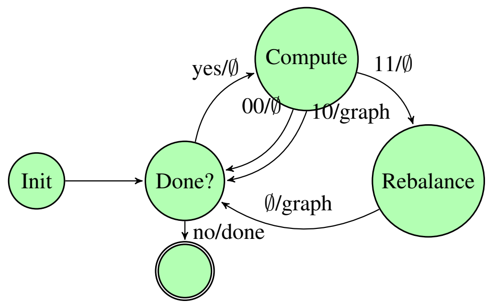


Figure 5.8: Application State Machine. Transitions Show Input (from Load Balance Algorithm) and Output (from Application to Load Balance Algorithm).

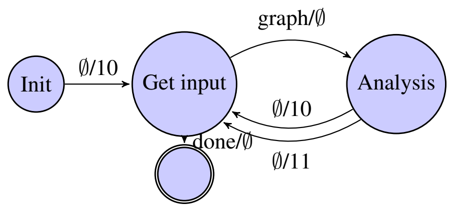


Figure 5.9: Load Balance Algorithm State Machine. Transitions Show Input (from Application) and Output (from Load Balance Algorithm to Application).

5.5 Lazy Load Balancing Implementation

To enable applications to exploit the lazy load balancing approach, we develop a framework for transparently, i.e., without changes to the actual computation in the application, decoupling the resources used by the load balance algorithm from the application resources. Our framework enables offloading the load balance algorithm to separate processes, allowing the load balance algorithm to execute asynchronously to the application, without pausing the application.

We achieve this separation by *virtualizing* the application to execute on a different, potentially smaller, set of processes than the job allocation, as described in Section 5.5.1. We describe our *asynchronous* communication protocol between the application and balancing processes in Section 5.5.2.

5.5.1 Transparently Splitting Existing Application Partitions

Without a loss in generality of the concepts, we focus our implementation on applications built on top of the Message Passing Interface (MPI) [67], currently the dominant standard for large scale scientific applications. We implement a portable mechanism using standard MPI to allocate the processes in a parallel machine in an MPMD fashion, i.e., the ability to execute multiple independent programs concurrently (in our case the application and the load balance algorithm) within the same job partition, without modifying the applications.

We use P^NMPI [82] to integrate modules in our framework. P^NMPI is a tools framework that stacks independent tools built using the MPI profiling interface [50]. We use this framework to combine the necessary components or tools implementing the virtualization of the underlying MPI process into separate partitions, the measurement tools, the interaction with the application, and our new asynchronous interaction component. P^NMPI also supports direct communication among tools, which we use to exchange element interaction and performance information. Our tool stack imposes negligible overhead, so perturbation of application behavior is insignificant.

The virtualization module intercepts all application communication, and transparently replaces MPI_COMM_WORLD with a smaller communicator, APP_COMM and with that provides the illusion for the application that it is executed on a smaller set of resources. With a predetermined number of processes reserved for the load balance algorithm, the application proceeds as before, using MPI_COMM_WORLD and other MPI communicators that now simply use fewer processes. The load balance algorithm executes asynchronously in a separate communicator, LB_COMM, where:

$$\text{MPI\_COMM\_WORLD} = \text{APP\_COMM} \cup \text{LB\_COMM} \tag{5.6}$$

Figure 5.6 illustrates the P^{N}MPI modules in our framework and their interaction with the application and libraries. The virtualization module splits the processes between the two communicators. Data collection is performed on APP_COMM, which allows the use of additional data collection modules like Libra [45]. All load balance calculations are run in LB_COMM, including external load balance libraries like Zoltan [36]. LB_COMM processes also determine whether to send the application rebalance directions. We next describe the communication between the two communicators.

5.5.2 Asynchronous Interaction Protocol Between Application and Load Balancing Processes

The communication in Figure 5.6 enables the asynchronous execution and coordination between the application and the load balancing processes. This communication is performed completely outside the application, preserving the original execution model for the application.

Figures 5.8 and 5.9 demonstrate the asynchronous interaction protocol between the application and the load balance processes. Figure 5.7 shows the shorthand for the valid state transitions, which indicate when the load representation (graph) needs to be sent and when the application should rebalance. The application sends information about its state and continues to run while the load balance algorithm computes. The execution within the load balancing processes can be synchronous, allowing us to call MPI libraries (i.e., partitioners), directly, without impacting application execution. The application is sent the rebalancing instructions only when they have been computed. The application can then apply this decision at the next stopping point (i.e., between time steps).

If the application is at a stopping point and has not received rebalancing instructions, it can:

- always wait for load balancing directions (inline);
- never wait for load balancing directions (fully asynchronous);
- continue for up to a given number of time steps and then wait (middle ground).

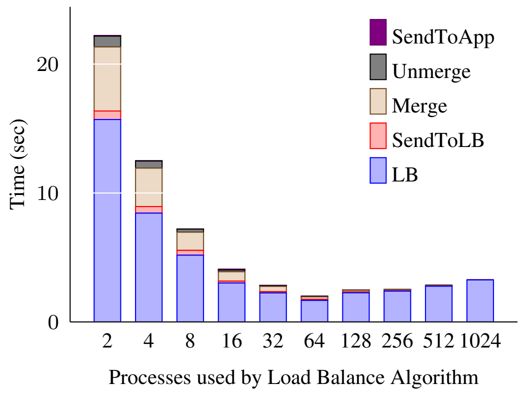
The above decision can be made dynamically at runtime for each stopping point and should be based on a function of how quickly the application load balance deteriorates. For the purpose of the remainder of this section, we run in a fully asynchronous mode, since this choice stresses the concept of lazy load balancing the most and therefore allows the most thorough evaluation.

5.6 Performance Evaluation

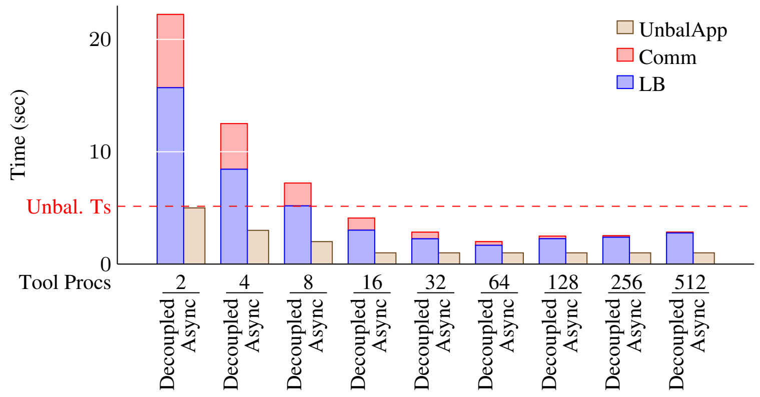
5.6.1 Evaluating Overhead of Lazy Load Balancing

For our experimental studies, we show strong scaling for all comparisons. Weak scaling studies of N-body computations are difficult to construct accurately due to variability in the particle density during scaling. Showing strong scaling ensures a fair comparison.

Figure 5.10(a) shows the break down of the costs of decoupling the load balance algorithm as a function of the resources provided to the load balance algorithm. These costs

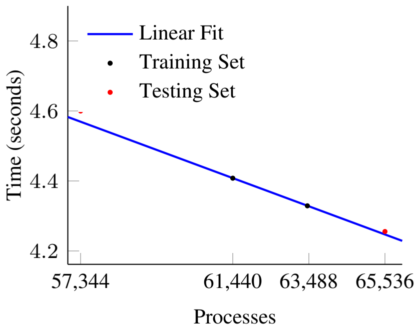


(a) Costs of Decoupling the Load Balance Algorithm

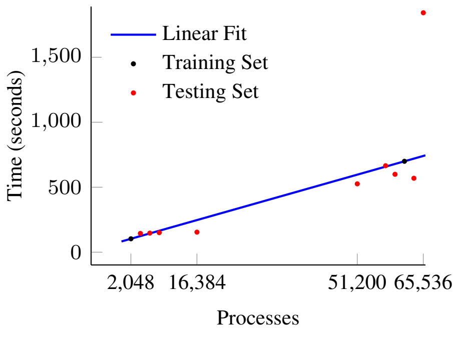


(b) Overhead of Balancing Configurations (Standard, Decoupled, and Asynchronous)

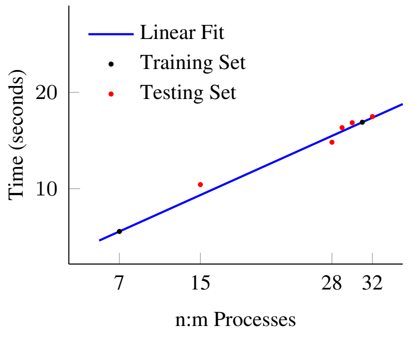
Figure 5.10: Lazy Load Balance Algorithm Overhead. Linux Cluster, Barnes-Hut, 64.6M Interactions, 1024 Processes



(a) Application Timestep Length



(b) Load Balance Algorithm



(c) Communication Parameters

Figure 5.11: Model Parameters Obtained via Curve Fitting. BGQ, Barnes-Hut, 306.5M Interactions

include sending the data to the load balancing processes, merging the data from several application processes on a single load balancing process, running the load balance algorithm in parallel on load balancing processes, unmerging, and sending the load balancing instructions back to corresponding application processes. The communication overhead imposed by the decoupling has a strong relationship with the $n : m$ ratio of application processes to load balancing processes, consistent with fan in/fan out costs that one would expect. We discuss the scaling of the load balance algorithm in the next subsection.

Figure 5.10(b) shows the overhead of the decoupled and asynchronous load balancing configurations as a function of the resources provided to the load balance algorithm. In the decoupled configuration, the overhead includes the load balance algorithm time and the communication overhead. In the asynchronous configuration, the overhead includes the time lost due to running the application in an imbalanced state while the load balancing directions are computed. Unbalanced timestep length is shown for scale.

5.6.2 Model Validation

We use least squares fit to model the parameters in our model. Figure 5.11 shows the performance of the load balance algorithm, application timestep, and communication overhead for a Barnes-Hut problem with 306.5M interactions. The training set measurements are shown as black dots, along with the curve that we fit to them. We show the measurements that we use for validation as red dots. We model the load balance algorithm and the application timestep as a function of processes. The communication overhead, however, is closely related to the ratio between the number of application processes and the number of load balancing processes, so we model it as a function of the ratio. Linear fits are sufficient in all cases. Poor performance of the graph partitioner (it becomes slower with additional resources) is due to the small size of the partitioned graph (265K vertices), which results in little work per process and high serialized overhead that increases with

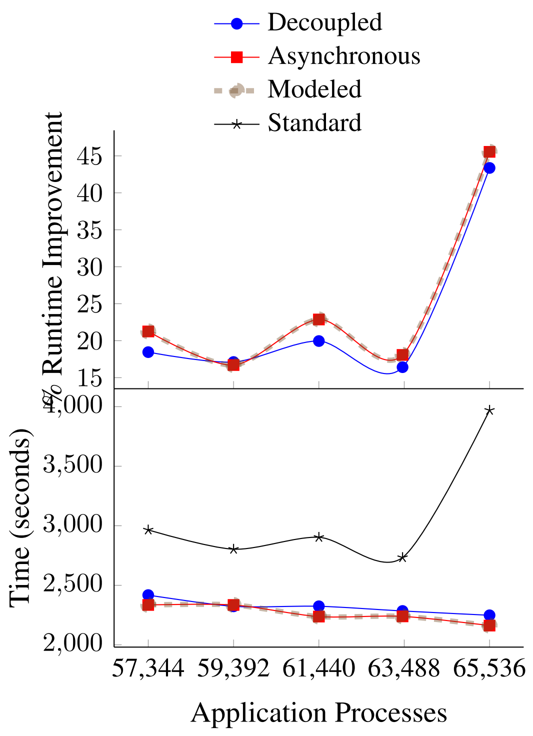


Figure 5.12: Runtime of Load Balance Configurations and Model Selection. Percent Runtime Improvement of Best Decoupled and Best Asynchronous Load Balancing Configurations over Standard

the number of processes. This inefficiency is one of the motivations for our work: by decoupling the load balance algorithm we can use a graph partitioner when it would be too expensive to run at the full scale of the application.

Figure 5.12 shows the runtime and percent improvement over the standard approach. For *asynchronous* and *decoupled*, only the runtimes with the best performing parameters (out of 7-11 parameters) are shown. The runtime for the configuration selected by the model (out of 15-23 valid options) is shown. In all but one case, the model accurately selects the best performing configuration. In the case that the model selects the second best configuration, the difference in runtime between the first and second best configuration is small, so the error results in little performance loss.

Overall, the decoupled and asynchronous configurations are able to reclaim the performance lost due to the poor strong scalability of the graph partitioner, resulting in 15-46% runtime improvement. The asynchronous configuration results in the shortest runtime in most cases due to overlapping the application and graph partitioning computations. Our model generally correctly selects the best performing configuration along with the parameters, resulting in 17-46% runtime improvement.

5.7 Conclusions

Our novel lazy approach to load balancing is based on decoupling the load balance algorithm from the application and offloading the load balance computation to overlap it with the application execution. We implemented a framework that performs the decoupling and offloading of the load balance algorithm transparently to the application, requiring no modifications to the application. We characterized the application properties and drift metrics that determine suitability of lazy load balancing. We provided a model for allocating the resources in the system based on the performance of the application and the load balance algorithm.

We have evaluated the usefulness of our proposed drift metrics on a Barnes-Hut benchmark and a production application, ParaDiS. We studied the performance properties of lazy load balancing and have demonstrated runtime improvements of up to 46%. Finally, we have shown that our resource allocation model can accurately predict the load balance configuration and the resource allocation that result in the lowest execution time of the application. Our MPMD approach to allocating the supercomputer removes the load balance algorithm from the critical path of the application, shortening the overall runtime of the simulation.

6. CONCLUSIONS

The research conducted as a part of this dissertation has reestablished the importance of effective and affordable dynamic load balance correction at scale. Our work shows that affordable distributed load balancing techniques are often too inaccurate to use at scale when even small imbalances can result in severe performance degradation. Yet the more accurate methods are frequently too expensive to be affordable at runtime.

This dissertation makes several important contributions towards effective and efficient load balancing of dynamic parallel applications. We developed strategies to measure and evaluate the computational load of the application, and decide when and how to correct the imbalance. We developed a model for comparison of load balance algorithms for a specific state of the simulation that enables the selection of a balancing algorithm that will minimize overall runtime.

To make the load balance correction affordable at scale, we propose a lazy load balancing strategy that evaluates the imbalance and computes the new assignment of work to processes asynchronously to the main application computation. We decouple the load balance algorithm from the application and run it with higher parallel efficiency on potentially fewer, separate processors. In this Multiple Program Multiple Data (MPMD) configuration, and application work is reassigned lazily as directions become available. We show that we can save resources by running a load balance algorithm at higher parallel efficiency on a smaller number of processors, and provide a model for allocating the resources in the system based on the performance of the application and the load balance algorithm. We studied the performance properties of lazy load balancing and have demonstrated runtime improvement of up to 46%. We have shown that our resource allocation model accurately predicts the load balance configuration and the resource allocation that result in the lowest execution time of the application.

Based on this work, several directions for future research are possible. The load balancing framework can be enhanced with more sophisticated algorithms for load balancing, including customized algorithms for applications with unusual resource requirements. Extending the generality of the model for evaluating load balance algorithm performance to include more types of load balancing algorithms is another direction for research.

The work presented on deciding how to allocate the resources between the application and the load balance algorithm is specific to a particular load balance algorithm and two applications described. For this work to be generally useful, other load balance algorithms would have to be modeled, along with applications to which they are applied.

REFERENCES

[1] chaos-release: Linux Distribution for High Performance Computing, 2010. http://code.google.com/p/chaos-release/wiki/CHAOS_Description.

[2] CGAL, Computational Geometry Algorithms Library, 2010. http://www.cgal.org.

[3] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, J. Brunheroto, C. Caşcaval, J. Casta nos, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. Cipolla, P. Crumley, K. Desai, A. Deutsch, T. Domany, M. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. Haring, D. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. Jackson, T. Jamal-Eddine, G. Kopcsay, E. Krevat, M. Kurhekar, A. Lanzetta, D. Lieber, L. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. Moreira, B. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. Tremaine, M. Tsao, A. Umamaheshwaran, P. Verma, P. Vranas, T. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. Seager, J. Vetter, and K. Yates. An Overview of the BlueGene/L Supercomputer. In *SC'02*, November 2002.

[4] T. Agarwal, A. Sharma, and L. V. Kalé. Topology-Aware Task Mapping for Reducing Communication Contention on Large Parallel Machines. In *International*

Parallel and Distributed Processing Symposium (IPDPS), April 2006.

[5] D. Arnold, D. Ahn, B. de Supinski, G. Lee, B. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *Parallel and Distributed Processing Symposium (IPDPS)*, March 2007.

[6] A. Arsenlis, W. Cai, M. Tang, M. Rhee, T. Oppelstrup, G. Hommes, T. G. Pierce, and V. V. Bulatov. Enabling Strain Hardening Simulations with Dislocation Dynamics. *Modelling and Simulation in Materials Science and Engineering*, 15(6):553, 2007.

[7] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martinez, X. Martorell, and R. Silvera. Is the Schedule Clause Really Necessary in OpenMP? In *International Workshop of OpenMP Applications and Tools, Lecture Notes in Computer Science*, June 2003.

[8] H. C. Baker, Jr. and C. Hewitt. The Incremental Garbage Collection of Processes. In *ACM Symposium on Artificial Intelligence and Programming Languages*, 1977.

[9] I. Banicescu and S. Flynn Hummel. Balancing Processor Loads and Exploiting Data Locality in N-Body Simulations. In *SC'95*, November 1995.

[10] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali. A Load Balancing Framework for Adaptive and Asynchronous Applications. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):183--192, 2004.

[11] K. Barker and N. Chrisochoides. An Evaluation of a Framework for the Dynamic Load Balancing of Highly Adaptive and Irregular Parallel Applications. In *SC'03*, November 2003.

[12] J. Barnes and P. Hut. A Hierarchical O(N log N) Force-Calculation Algorithm. *Nature*, 324:446--449, December 1986.

[13] A. Basermann, J. Clinckemaillie, T. Coupez, J. Fingberg, H. Digonnet, R. Ducloux, J. M. Gratien, U. Hartmann, G. Lonsdale, B. Maerten, D. Roose, and C. Walshaw. Dynamic Load-Balancing of Finite Element Applications with the DRAMA Library. *Applied Mathematical Modelling*, 25(2):83--98, 2000.

[14] M. J. Berger and S. H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Transactions on Computers*, 36(5):570--580, May 1987.

[15] A. Bhatelé, L. V. Kalé, and S. Kumar. Dynamic Topology Aware Load Balancing Algorithms for MD Applications. In *SC'09*, November 2009.

[16] R. Biswas, L. Oliker, S. K. Das, and D. Harvey. Portable Parallel Programming for the Dynamic Load Balancing of Unstructured Grid Applications. In *IEEE International Parallel Processing Symposium (IPPS)*, April 1999.

[17] S. H. Bokhari. Dual Processor Scheduling with Dynamic Reassignment. *IEEE Transactions on Software Engineering*, SE-5(4):326--334, July 1979.

[18] S. H. Bokhari. A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System. *IEEE Transactions on Software Engineering*, SE-7(6):335--341, November 1981.

[19] C. Boneti, R. Gioiosa, F. J. Cazorla, and M. Valero. A Dynamic Scheduler for Balancing HPC Applications. In *SC'08*, November 2008.

[20] L. A. Bongo, B. Vinter, O. J. Anshus, T. Larsen, and J. M. Bjorndalen. Using Overdecomposition to Overlap Communication Latencies with Computation and Take Advantage of SMT Processors. In *International Conference on Parallel Processing Workshops (ICPPW)*, August 2006.

[21] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Coun-

ters. In *SC'00*, November 2000.

[22] R. K. Brunner and L. V. Kalé. Handling Application-Induced Load Imbalance using Parallel Objects. *International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, 2000.

[23] H. Brunst, H.-C. Hoppe, W. E. Nagel, and M. Winkler. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach. In *International Conference on Computational Science-Part II*, 2001.

[24] V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis. Scalable Line Dynamics in ParaDiS. In *SC'04*, November 2004.

[25] M. Burtscher and K. Pingali. An Efficient CUDA Implementation of the Tree-Based Barnes-Hut N-Body Algorithm. In *GPU Computing Gems Emerald Edition*, 2011.

[26] A. R. Butz. Convergence with Hilbert's Space Filling Curve. *Journal of Computer & System Sciences*, 3(2):128--146, 1969.

[27] T. L. Casavant and J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141--154, 1988.

[28] R. Chandra, A. Gupta, and J. L. Hennessy. Data Locality and Load Balancing in COOL. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 249--259, 1993.

[29] T. C. K. Chou and J. A. Abraham. Load Balancing in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-8(4):401--412, July 1982.

[30] R. B. Christopher, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, pages 207--216, 1995.

[31] P. Colella, D.Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. V. Straalen. Chombo: Software Package for AMR Applications -- Design Document. 2003.

[32] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive Load-Balancing Policies for Dynamic Applications. *IEEE Concurrency*, 7(1):22--31, 1999.

[33] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, May 1993.

[34] G. Cybenko. Dynamic Load Balancing for Distributed Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279--301, 1989.

[35] B. R. de Supinski, M. Schulz, V. V. Bulatov, W. Cabot, B. Chan, A. W. Cook, E. W. Draeger, J. N. Glosli, J. A. Greenough, K. Henderson, A. Kubota, S. Louis, B. J. Miller, M. V. Patel, T. E. Spelce, F. H. Streitz, P. L. Williams, R. K. Yates, A. Yoo, G. Almasi, G. Bhanot, A. Gara, J. A. Gunnels, M. Gupta, J. Moreira, J. Sexton, B. Walkup, C. Archer, F. Gygi, T. C. Germann, K. Kadau, P. S. Lomdahl, C. Rendleman, M. L. Welcome, W. McLendon, B. Hendrickson, F. Franchetti, S. Kral, J. Lorenz, C. W. Uberhuber, E. Chow, and U. Catalyurek. BlueGene/L Applications: Parallelism on a Massive Scale. *Internationa Journal of High Performance Computing Applications (IJHPCA)*, 22(1), 2008.

[36] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, J. Teresco, J. Faik, J. Flaherty, and L. Gervasio. New Challanges in Dynamic Load Balancing. *Applied Numerical*

Mathematics, 52(2-3):133--152, 2005.

[37] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan. Design of Dynamic Load-Balancing Tools for Parallel Applications. In *International Conference on Supercomputing (ICS)*, May 2000.

[38] K. W. Doty, P. L. McEntire, and J. G. O'Reilly. Task Allocation in a Distributed Computer System. *IEEE InfoCom*, pages 33--38, 1982.

[39] A. Duran, M. Gonzàlez, and J. Corbalán. Automatic Thread Distribution for Nested Parallelism in OpenMP. In *International Conference on Supercomputing (ICS)*, June 2005.

[40] Z. Eisler, I. Bartos, and J. Kertesz. Fluctuation Scaling in Complex Systems: Taylor's Law and Beyond. *Advances in Physics*, 57(1):89--142, 2008.

[41] D. Friedman and D. Wise. The Impact of Applicative Programming on Multiprocessing. In *International Conference on Parallel Processing (ICPP)*, pages 263--272, September 1976.

[42] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software*, 3(3):209--226, Sept. 1977.

[43] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 33, 1998.

[44] A. Gabrielian and D. B. Tyler. Optimal Object Allocation in Distributed Computer Systems. In *International Conference on Distributed Computing Systems (ICDCS)*, May 1984.

[45] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed. Scalable Load-Balance Measurement for SPMD Codes. In *SC'08*, November 2008.

[46] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co., New York, NY, USA, 1979.

[47] E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Touriño, and K. Yelick. Communication Avoiding and Overlapping for Numerical Linear Algebra. In *SC'12*, November 2012.

[48] J. N. Glosli, K. J. Caspersen, J. A. Gunnels, D. F. Richards, R. E. Rudd, and F. H. Streitz. Extending Stability Beyond CPU Millennium: A Micron-Scale Atomistic Simulation of Kelvin-Helmholtz Instability. In *SC'07*, November 2007.

[49] L. Greengard and V. Rokhlin. A Fast Algorithm for Particle Simulations. *Journal of Computational Physics*, 135:280--292, 1987.

[50] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 1994.

[51] B. Hendrickson and T. G. Kolda. Graph Partitioning Models for Parallel Computing. *Parallel Computing*, 26(12):1519--1534, 2000.

[52] K. A. Huck and J. Labarta. Detailed Load Balance Analysis of Large Scale Parallel Applications. In *International Conference on Parallel Processing*, September 2010.

[53] J. E. Jones. On the Determination of Molecular Fields. II. From the Equation of State of a Gas. *Royal Society of London Proceedings Series A*, 106:463--477, Oct. 1924.

[54] V. Karamcheti and A. A. Chien. A Hierarchical Load-Balancing Framework for Dynamic Multithreaded Computations. In *SC'98*, November 1998.

[55] R. M. Karp. Reducibility Among Combinatorial Problems. *Complexity of Computer Computations*, pages 85--103, 1972.

[56] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49:291--307, February 1970.

[57] L. Kleinrock and A. Nilsson. On Optimal Scheduling Algorithms for Time-Shared Systems. *Journal of the ACM*, 28(3):477--486, 1981.

[58] G. A. Koenig and L. V. Kalé. Optimizing Distributed Application Performance Using Dynamic Grid Topology-Aware Load Balancing. In *IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS)*, March 2007.

[59] N. Komatsu, T. Kiwata, and S. Kimura. Thermodynamic Properties of an Evaporation Process in Self-Gravitating N-Body Systems. *Physics Review E*, 82, August 2010.

[60] R. Koradi, M. Billeter, and P. Gntert. Point-Centered Domain Decomposition for Parallel Molecular Dynamics Simulation. *Computer Physics Communications*, 124:139--147, 2000.

[61] Z. Lan, V. E. Taylor, and Y. Li. DistDLB: Improving Cosmology SAMR Simulations on Distributed Computing Systems Through Hierarchical Load Balancing. *Journal of Parallel and Distributed Computing*, 66(5):716--731, 2006.

[62] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of Inference and Learning for Performance Modeling of Parallel Applications. In *International Symposium on Principles and Practices of Parallel Programming (PPoPP)*, March 2007.

[63] M. Livny and M. Melman. Load Balancing in Homogeneous Broadcast Distributed Systems. In *Proceedings of the Computer Network Performance Symposium*, pages

47--55, 1982.

[64] P. Y. R. Ma, E. Y. S. Lee, and J. Tsuchiya. A Task Allocation Model for Distributed Computing Systems. *IEEE Transactions on Computers*, C-31(1), January 1982.

[65] B. M. Mark, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. Bruce, I. Karen, L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28:37--46, November 1995.

[66] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Chichester: John Wiley and Sons, New York, NY, USA, 1990.

[67] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, September 2012.

[68] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent Work Stealing. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, February 2008.

[69] J. L. Myers and A. D. Well. *Research Design and Statistical Analysis (2nd edition)*. Lawrence Erlbaum Associates Publishers, Mahwah, NJ, USA, 2003.

[70] L. M. Ni and K. Hwang. Optimal Load Balancing Strategies for a Multiple Processor System. In *International Conference on Parallel Processing (ICPP)*, August 1981.

[71] L. Oliker and R. Biswas. Efficient Load Balancing and Data Remapping for Adaptive Grid Calculations. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1997.

[72] L. Oliker and R. Biswas. PLUM: Parallel Load Balancing for Adaptive Unstructured Meshes. *Journal of Parallel and Distributed Computing*, 52(2):150--177, 1998.

[73] O. Pearce, T. Gamblin, B. R. de Supinski, T. Arsenlis, and N. M. Amato. Load Balancing N-Body Simulations with Highly Non-Uniform Density. In *International Conference on Supercomputing (ICS)*, June 2014.

[74] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato. Quantifying the Effectiveness of Load Balance Algorithms. In *International Conference on Supercomputing (ICS)*, June 2012.

[75] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The Tao of Parallelism in Algorithms. In *Conference on Programming Language Design and Implementation (PLDI)*, February 2011.

[76] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1 -- 19, 1995.

[77] R. B. Ross, T. Peterka, H.-W. Shen, Y. Hong, K.-L. Ma, H. Yu, and K. Moreland. Visualization and Parallel I/O at Extreme Scale. *Journal of Physics: Conference Series*, 125(1):12--99, 2008.

[78] J. Sancho, D. Kerbyson, and K. Barker. Efficient Offloading of Collective Communications in Large-Scale Systems. In *International Conference on Cluster Computing (ICCC)*, September 2007.

[79] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka. Design and Modeling of a Non-Blocking Checkpointing System. In *SC'12*, November 2012.

[80] K. Schloegel, G. Karypis, and V. Kumar. A Unified Algorithm for Load-Balancing Adaptive Scientific Simulations. In *SC'00*, November 2000.

[81] K. Schloegel, G. Karypis, and V. Kumar. Parallel Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *International Euro-Par Conference on Parallel Processing*, August 2000.

[82] M. Schulz and B. R. de Supinski. P^{N}MPI Tools: A Whole Lot Greater Than the Sum of Their Parts. In *SC'07*, November 2007.

[83] C. Shen and W. Tsai. A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion. *IEEE Transactions on Computers*, C-34(3):197--203, March 1985.

[84] S. S. Shende and A. D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications (IJHPCA)*, 20(2):287--311, 2006.

[85] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta. A Parallel Adaptive Fast Multipole Method. In *SC'93*, November 1993.

[86] C. D. Snow, E. J. Sorin, Y. M. Rhee, and V. S. Pande. How Well Can Simulation Predict Protein Folding Kinetics and Thermodynamics? *Annual Review of Biophysics and Biomolecular Structure*, 34(1):43--69, 2005.

[87] N. Spring and R. Wolski. Application Level Scheduling of Gene Sequence Comparison on Metacomputers. In *International Conference on Supercomputing (ICS)*, July 1998.

[88] H. S. Stone. Critical Load Factors in Two-Processor Distributed Systems. *IEEE Transactions on Software Engineering*, SE-4(3):254--258, May 1978.

[89] H. S. Stone and S. H. Bokhari. Control of Distributed Processes. *Computer*, 11:97--106, July 1978.

[90] F. Streitz, J. Glosli, M. Patel, B. Chan, R. Yates, B. de Supinski, J. Sexton, and J. Gunnels. 100+ TFlop Solidification Simulations on BlueGene/L. In *SC'05*, November 2005.

[91] F. Streitz, J. Glosli, M. Patel, B. Chan, R. Yates, B. de Supinski, J. Sexton, and J. Gunnels. Simulating Solidification in Metals at High Pressure: The Drive to Petascale Computing. *Journal of Physics: Conference Series*, 46:254--267, 2006.

[92] H. Sundar, R. S. Sampath, and G. Biros. Bottom-Up Construction and 2:1 Balance Refinement of Linear Octrees in Parallel. *SIAM Journal on Scientific Computing*, 30(5):2675--2708, 2008.

[93] N. R. Tallent, J. M. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto. Scalable Fine-Grained Call Path Tracing. In *International Conference on Supercomputing (ICS)*, June 2011.

[94] K. S. Thorne. Multipole Expansions of Gravitational Radiation. *Reviews of Modern Physics*, 52:299--340, Apr. 1980.

[95] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. B. Ross, and K. Yoshii. Accelerating I/O Forwarding in IBM Blue Gene/P Systems. In *SC'10*, November 2010.

[96] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

[97] C. Walshaw and M. Cross. Parallel Optimization Algorithms for Multilevel Mesh Partitioning. *Parallel Computing*, 26(12):1635--1660, 2000.

[98] C. Walshaw, M. Cross, and M. G. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *Journal of Parallel and Distributed Computing*, 47(2):102--108, 1997.

[99] C. Walshaw, M. Cross, and K. McManus. Multiphase Mesh Partitioning. *Applied Mathematical Modelling*, 25(2):123--140, 2000.

[100] M. S. Warren and J. K. Salmon. Astrophysical N-Body Simulations Using Hierarchical Tree Data Structures. In *SC'92*, November 1992.

[101] M. S. Warren and J. K. Salmon. A Parallel Hashed Oct-Tree N-Body Algorithm. In *SC'93*, November 1993.

[102] J. White and J. Dongarra. Overlapping Computation and Communication for Advection on Hybrid Parallel Computers. In *Parallel Distributed Processing Symposium (IPDPS)*, May 2011.

[103] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 4(9):979--993, 1993.

[104] M. Winkel, R. Speck, H. Hübner, L. Arnold, R. Krause, and P. Gibbon. A Massively Parallel, Multi-Disciplinary Barnes-Hut Tree Code for Extreme-Scale N-Body Simulations. *Computer Physics Communications*, 183(4):880--889, 2012.

[105] A. M. Wissink, D. Hysom, and R. D. Hornung. Enhancing Scalability of Parallel Structured AMR Calculations. In *International Conference on Supercomputing (ICS)*, June 2003.

[106] R. Wolski, N. Spring, and C. Peterson. Implementing a Performance Forecasting System for Metacomputing: the Network Weather Service. In *SC'97*, November 1997.

[107] G. 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.