Load Balancing N-Body Simulations with Highly Non-Uniform Density

Olga Pearce[⋆†], Todd Gamblin[†], Bronis R. de Supinski[†],
Tom Arsenlis[†], Nancy M. Amato[⋆]

[⋆]Department of Computer Science and Engineering, Texas A&M University, College Station, TX, USA
{olga,amato}@cse.tamu.edu
[†]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA
{olga,tgamblin,bronis,arsenlis1}@llnl.gov

ABSTRACT

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. Most load balancing techniques for N-body methods use particle count to approximate computational work. This approximation 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. In this paper, we demonstrate that existing techniques do not perform well at scale when particle density is highly non-uniform, and we propose a load balance technique that efficiently assigns load in terms of interactions instead of particles. We use adaptive sampling to create an even work distribution more amenable to partitioning, and to reduce partitioning overhead. We implement and evaluate our approach on a Barnes-Hut algorithm and a large-scale dislocation dynamics application, ParaDiS. Our method achieves up to 26% improvement in overall performance of Barnes-Hut and 18% in ParaDiS.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Hypergraphs*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Parallel*

Keywords

load balance; parallel algorithm; performance; simulation

1. INTRODUCTION

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 [17, 27, 28, 34]. 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 naïve N-body algorithm runs in $O(n^2)$ time with respect to the number of particles. Modern algorithms such as Barnes-Hut [5] and the fast multipole method [31] 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 it is assigned. However, the work in N-body simulations is proportional to the number of *interactions* each process computes, i.e., the local density of particles in the simulated domain. Particle decompositions can therefore distribute work unevenly when there is high particle density variation. This type of load imbalance 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 and 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 paper 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;

ICS'14, June 10–13 2014, Munich, Germany.
Copyright 2014 ACM 978-1-4503-2642-1/14/06 ...$15.00.
http://dx.doi.org/10.1145/2597652.2597659.

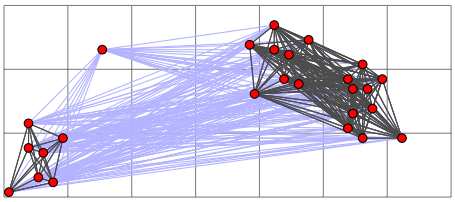
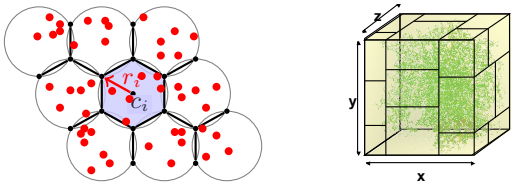


Figure 1: Particles Shown as Circles (n); Short-Range and Long-Range Interactions Shown as Black and Blue Edges



(a) ddcMD (b) ParaDiS

Figure 2: Geometric Domain Decomposition

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 summarizes traditional load balance methods for N-body applications. Section 3 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 3.1 and 3.2. Section 4 outlines our implementation and its application to Barnes-Hut and ParaDiS. We evaluate the performance of our approach in Section 5.

2. LIMITATIONS OF 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 1 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 [5] and fast multipole [31] compute weak long-range forces (blue edges) less frequently than stronger near-range forces (black 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 running time.

The largest N-body simulations comprise 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 that each process evaluates change over time, and we must load balance frequently. Finally, in addition to evenly dividing work, parallel N-body load balancers 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 balancers must allocate particles to minimize such communication.

Plimpton [23] 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 on P processes, *particle decomposition* (also called row-wise decomposition) assigns N/P particles to each process. Each particle and its associated interactions comprise a single *row* in the force matrix. Each interaction is computed by the process that owns the particles involved. If particles in an interaction are owned by different processes, a tiebreaker determines which process should compute 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 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 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 *blocks* of the force matrix. Because particle ordering in the matrix has no geometric correspondence, force decomposition methods do not minimize ghost communication. This method works 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 when there is a cutoff radius because each matrix block may contain very different numbers of interactions. To balance work efficiently, blocks of the force matrix must be uniformly dense.

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 [7, 34], octrees [26, 35], fractiling [4], and space-filling curves [10]. Orthogonal recursive bisection divides space recursively into cuboids until each 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 [28], where each process is assigned a centroid and owns the particles nearest its centroid (Figure 2(a)), and prismatic schemes [8], a variation on recursive bisection allowing subspaces to be divided more than twice (Figure 2(b)). Spatial decomposition methods 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 and the work is balanced by adjusting the subdomain boundaries. Particles are moved among subdomains as part of balancing. Some methods use the number of particles per subdomain as an approximation of the workload, while others [36] weight each particle by the number of interactions in which it participates. Our method can split the 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. 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 as-

sumes a dense force matrix, but many simulation force matrices are sparse. 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; tracking individual interactions would quickly become intractable. Even the force decomposition assigns *blocks* rather than individual interactions, thus reducing bookkeeping overhead. Our new method that allows fine-grained *interaction* balancing also avoids the memory and performance overheads of tracking individual interactions.

3. AN INTERACTION-BASED BALANCER

The load balancer must be precise to achieve the evenly balanced load required for performance at scal. In particular, to address the limitations discussed in Section 2, the load balancer 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 [18] 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 balancer. 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 [18] and will therefore run faster if: 1) work unit sizes are small compared to process load, and 2) the sizes are relatively uniform in size. 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 online 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 3.1, and we present our techniques for model construction and partitioning in Section 3.2.

3.1 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. Our strategy ensures that the partitioning is both fast and accurate.

Algorithm 1 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

Algorithm 1 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

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.

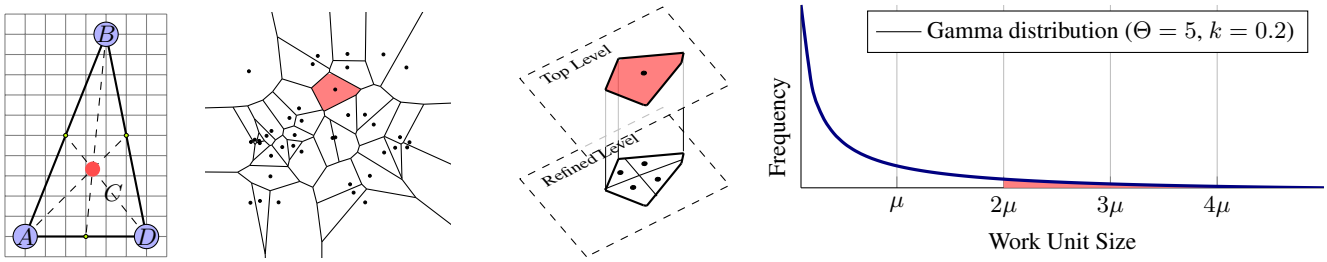
On line 2, Algorithm 1 starts by iterating over all particles. For each particle, on lines 4 and 5, the algorithm samples at least one interaction. If a particle is involved in more than the average number of interactions, then we take more samples. The adaptive sampling threshold, s, determines the 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 3(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 3(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.

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 3(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 [14] to determine which interactions are closest to each sample.

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



(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 3: Defining Interaction Subsets

Algorithm 2 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 ∈ P do
2:     H.insert(e_i) to represent the particle
3:     for i ∈ 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
```

in physical and natural processes (such as particles in a dynamical system) obey a power law distribution [13]. 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:

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

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 3(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 3(d) 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 off the gamma distribution. In Section 5, we show empirically that our method produces work units with relatively uniform sizes, and we demonstrate the positive impact on the resulting load balance.

3.2 Assigning Work Units to Processes

Section 3.1 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 [16]. Further, 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* of 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 interprocess 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 3.1 (to represent interactions), and *hyperedges* represent particles (storage units). Algorithm 2 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 3.1 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.

Algorithm 3 Sampling-based Interaction Load Balancer

$n \leftarrow$ 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. 1)	$O(s\frac{n}{p} + \frac{m}{p}\log(s))$
3:	Construct hypergraph (Alg. 2)	$O(s\frac{n}{p})$
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	$O(\frac{m}{p}\log(s))$

3.3 Interaction-Based Load Balancer Using Sampling and Hypergraph Partitioning

Algorithm 3 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.

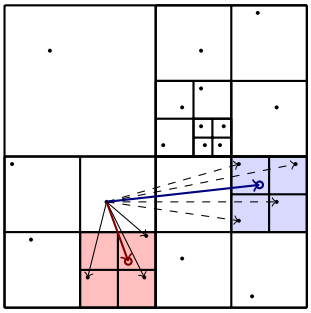


Figure 4: Octree in Barnes-Hut Benchmark

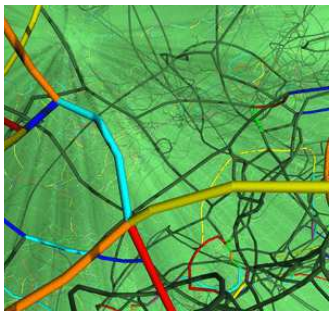


Figure 5: Dislocation Nodes and Arms in ParaDiS

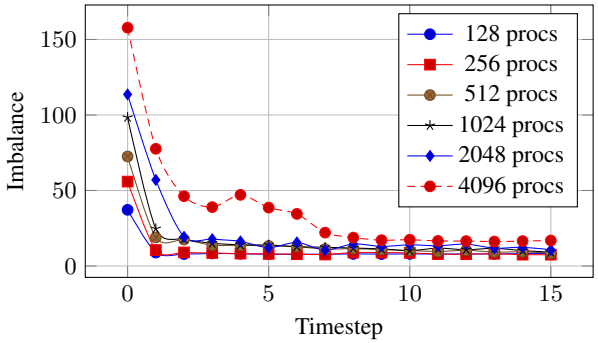


Figure 6: Imbalance over Time in ParaDiS with Built-in Recursive Bisection Load Balancer

Our load balancer 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 1, which samples interactions and constructs work units. We then construct a model, in this case a hypergraph, from the work units in Algorithm 2. We pass this model to partitioning. Assuming a power law distribution as mentioned in Section 3.1, 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. APPLICATIONS & IMPLEMENTATION

Our load balancer implementation requires support libraries for partitioning and for geometric range queries. Several hypergraph partitioning libraries are freely available [11, 24]. In this work, we use the hypergraph partitioner from Zoltan [12]. We use the k-d tree implementation from the CGAL [2] Computational Geometry Library for the nearest neighbor computation.

4.1 Barnes-Hut

The first application to which we have applied our framework is an implementation of the classic Barnes-Hut algorithm. We created a distributed version of Barnes-Hut [5] based on a shared memory implementation from the Lonestar suite in Galois [9, 22]. The code is written in C++ and uses MPI for communication. Its force calculation phase uses an octree to compute approximately the 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). This octree effectively partitions the volume hierarchically around the n particles into successively smaller cells. While a precise computation 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. Particles that interact with other particles in nearby cells are computed directly, but for interactions with cells that are sufficiently far away, one force computation with the cell is sufficient.

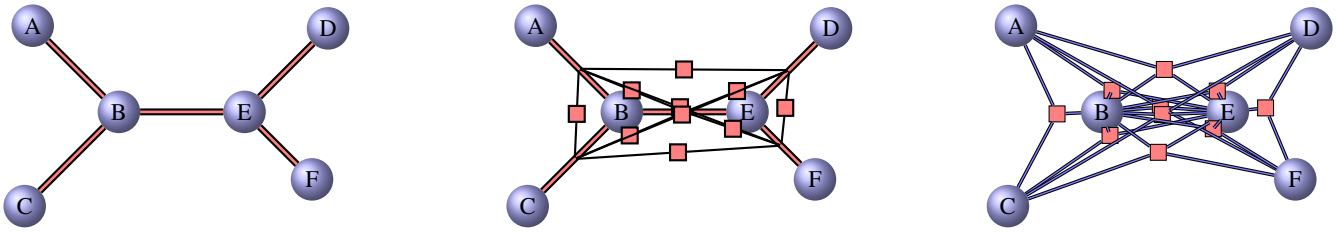
This algorithm has $O(n\log(n))$ complexity. For example, consider the two-dimensional hierarchical subdivision of space in Figure 4. The algorithm checks the distance to the red cell's center of gravity (red circle). Because the distance is not large (red arrow), interactions with all bodies in the 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 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. [36], Warren and Salmon [35], and used with modifications by Sundar, et al. [29]. 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 ParaDiS

ParaDiS (Figure 5) is the second application that we use in our experiments [3, 8]. This large-scale dislocation dynamics simulation, which is written in C/C++ with MPI for interprocess communication, is used to study the fundamental mechanisms of plasticity. It computes short-range forces directly and uses multipole expansion [15, 31] to compute long-range forces. 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(b). The recursive bisection algorithm bisects the space in the X, Y and/or Z dimensions into octants, quar-

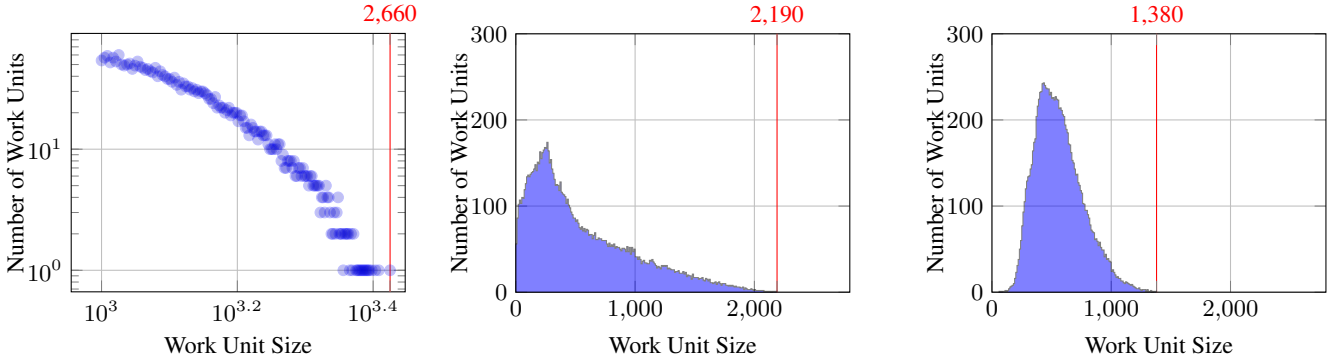


(a) Particles (blue spheres) and Arms (red lines)

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

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

Figure 7: ParaDiS Computation as a Hypergraph



(a) Random Sampling

(b) Per-Particle Sampling

(c) Adaptive Sampling

Figure 8: Effect of Sample Size and Technique on Work Unit Size Variability in Barnes-Hut

ters 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.

Figure 6 shows the effectiveness of the recursive bisection load balancer. 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 balancer as the baseline for our comparisons in Section 5.3.

Figure 7 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 7(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 7(b). Each interaction involves three or four dislocation nodes (particles), unlike many n-body applications which define interactions between pairs of particles. Figure 7(c) demonstrates the hypergraph that we use for ParaDiS. The *dislocation nodes* and *segment interactions* are the same as in Figure 7(b) (shown as blue spheres and red squares). The *dislocation nodes* become the hyperedges, connected to all interactions that they support.

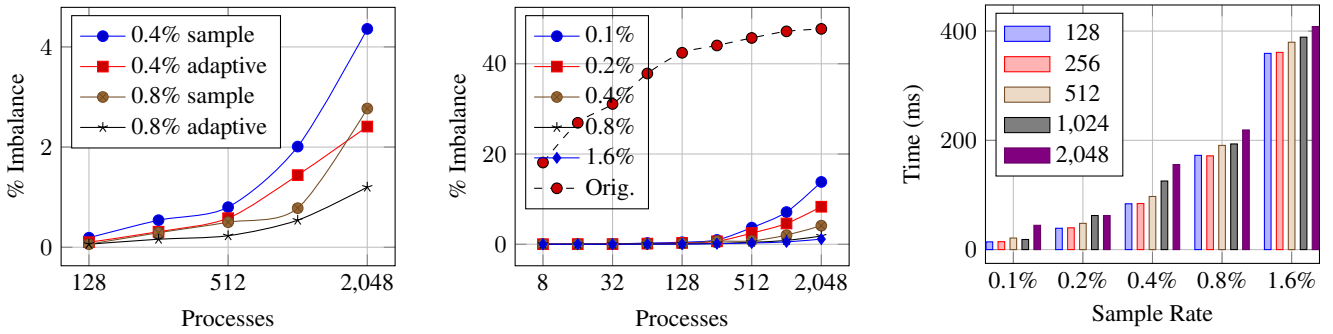
5. PERFORMANCE EVALUATION

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.

5.1 Distribution of Work Unit Sizes and Impact on Performance

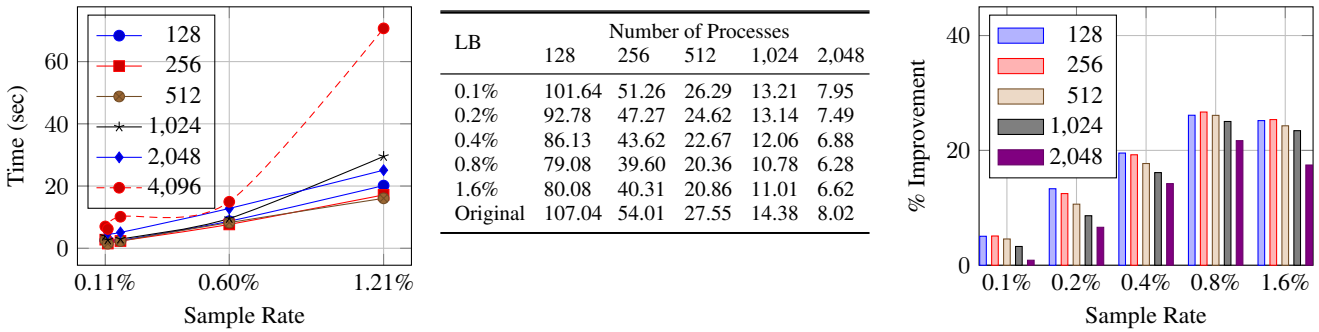
This section examines the distribution of work unit sizes under the sampling strategies described in Section 3.1, and how more uniform distribution leads to more evenly distributed load. These experiments use a Barnes-Hut problem with 32K particles, which we strong scale from 8 to 2,048 processes. We 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 problem at all scales so data points are comparable.

Figure 8 shows the effect of sample size and sampling strategy on the variability of work unit size in Barnes-Hut. As discussed in Section 3.1, sampling a simulated domain with high density variability results in a power law distribution of work units, as Figure 8(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



(a) Per-Particle vs. Adaptive Sampling (b) Sample Rate and Resulting Imbalance (c) Sample Rate & Aggregate Algorithm Cost

Figure 9: Impact of Sampling Strategy on Resulting Imbalance and Cost of Load Balancing Algorithm



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

(a) Partitioner Time (b) Total Computation Time (sec) (c) Improvement over Original

Figure 10: Impact of Sampling Rate on Performance of Graph Partitioner and Barnes-Hut Application

interactions per-particle (Figure 8(b)). By running Algorithm 1 *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 1, as Figure 8(c) shows. Here, the maximum work unit size is 1,380.

Figure 9 demonstrates the impact of the sample size and strategy. Figure 9(a) compares how the two approaches to sampling impact the ability of the load balancer to assign equal partitions to processes, where imbalance is:

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

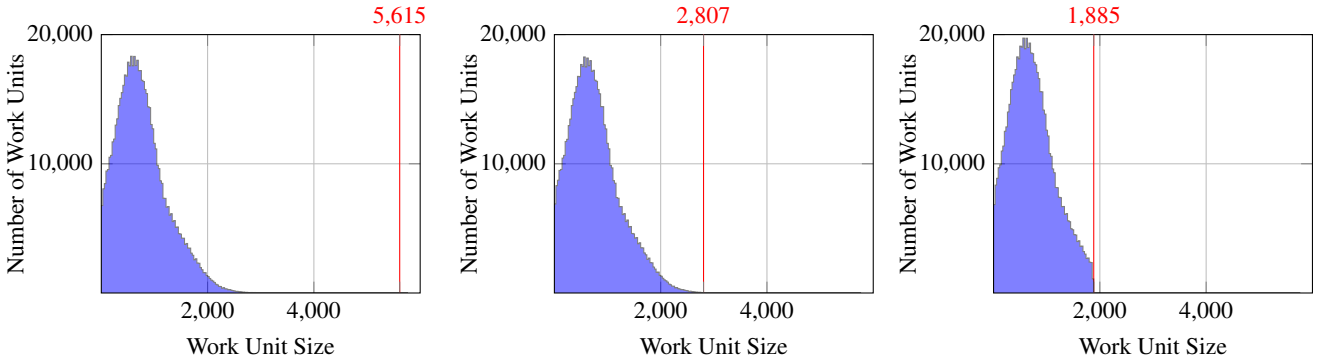
Because adaptive sampling makes the distribution of sample weights more uniform, it results in better partition quality (lower imbalance), as Figure 9(a) shows. Roughly, adaptive sampling can achieve the same partitioning quality as doubling the sample size, and keeps the number of resulting work units 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 9(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.

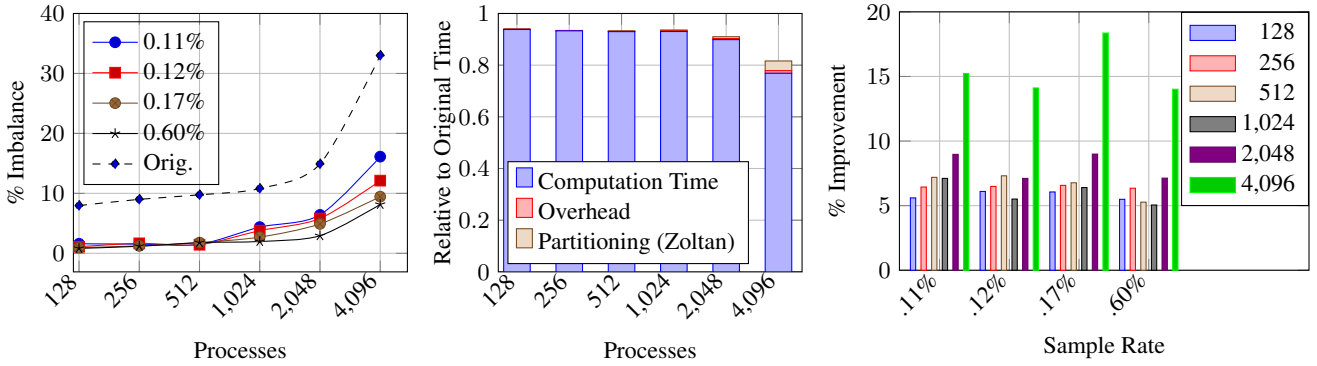
Figure 9(c) shows the aggregate overhead of our load balancer. As mentioned earlier, sample count directly impacts the cost of our load balancer because it determines the cost of partitioning, sampling and nearest-neighbor assignment. The figure clearly demonstrates the linear relationship between the sample size and the cost of our balancer. 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 leave this optimization for future work.

Sampling enables us to use the graph partitioning when partitioning the entire graph would be prohibitively expensive. Figure 10(a)



(a) Per-Particle Sampling (.11% sample) (b) Adaptive Sampling (.12% sample) (c) Adaptive Sampling (.17% sample)

Figure 11: 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 12: Effect of Sampling Technique on Load Balance and Application Performance

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 3.3, we have effectively reduced the complexity of balancing interactions to that of balancing particles.

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 13: Total Computation Time of ParaDiS (sec)

5.2 Impact on Barnes-Hut Performance

We show the impact of our load balancer on performance of the Barnes-Hut benchmark in Figure 10. The total times for our 32K particle simulation with different sampling rates are listed in Figure 10(b); Figure 10(c) illustrates these total times relative to the original load balancer. With 0.1% sample, our method shows marginal performance improvement over the original load balancer; 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 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 balancer with sufficient sampling performs well and clearly outperforms the particle-based load balancer. Overall, we observe 23-26% improvement for the optimal sampling rate.

5.3 Impact on ParaDiS Performance

We evaluate the impact of our load balancer 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 11 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 11(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 11(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 11(c) shows.

Figure 12 details performance of the load balancer and the application with different sampling strategies. Figure 12(a) demonstrates the impact that the different sampling strategies have on load imbalance, along with the lowest possible load balance achieved by the built-in balancer. 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 12(b) shows the cost break down for a sample rate of 0.17% relative to the performance using the existing load balancer. The computation time required with our load balancer is lower, especially at larger process counts when the existing load balancer 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. As already discussed, our future work will address this issue.

Figure 12(c) shows the runtime of the problem with different sampling levels, relative to the runtime of the problem with the built-in balancer. 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 13 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.

6. RELATED WORK

Section 2 discussed related work on N-body applications. Related work also includes applications and frameworks that use geometric and graph-based load balancing methods.

In theoretical work on load balancing hierarchical N-body applications, Teng partitions the communication graph [30] and proves certain sub-types of communication graphs can be partitioned. Teng builds a communication graph between the boxes in the hierarchy, and adds edges between the leaf boxes and the actual particles. Noting that edge density makes graph partitioning difficult, Teng then combines some of the edges and shows that the in- and out-degree of the communication graph is bounded by $O(\log n + \mu)$, where n is the number of boxes and μ is a measure of uniformity. Thus, the graph becomes more dense as the problem or non-uniformity grow. On the contrary, the density of our hypergraph is not tied to the size of the input but rather the sampling rate, or the granularity needed by the partitioner to create quality partitions. Teng's approach assigns weights to particles to meet the load balancing criteria. However, graph partitioners perform poorly with the large variation in weights. By design, our algorithm achieves relatively uniform vertex weights, improving partition quality.

Other types of scientific applications employ geometric load balancers. SAMRAI [37] is a structured AMR application that orders boxes by their spatial location by placing a Morton curve (or Z-curve) [19] through the box centroids to increase the likelihood that the neighboring patches will reside on the same processor after load balancing. PLUM [20, 21] is a load balancing framework for adaptive grid applications. It can use any partitioning algorithm and assists in efficient processor assignment and remapping of the computation. DRAMA [6] 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). Implicit treatment of work units and inherent approximations make these approaches fast but lead to higher imbalance at scale.

An alternative to geometric methods are partitioners that work with mesh, graph, or hypergraph representation of computation, i.e., ParMetis [24, 25], Jostle [32, 33], and Zoltan [11, 12]. Our work determines an appropriate representation for the problem, and leverages these powerful tools by providing them with inputs that enable quality solutions to the problem.

7. CONCLUSIONS

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 even the size of sampled work units. 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.

8. ACKNOWLEDGMENTS

The research of Amato and Pearce supported in part by NSF awards CNS-0551685, CCF-0833199, CCF-0830753, IIS-0916053, IIS-0917266, EFRI-1240483, RI-1217991, by NIH NCI R25 CA090301-11, by DOE awards DE-AC02-06CH11357, B575363, and by Samsung, Chevron, IBM, Intel, and Oracle/Sun. Pearce supported in part by an NSF Graduate Research Fellowship, a Department of Education Graduate Fellowship (GAANN), and the Lawrence Scholar Program Fellowship. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-648577).

9. REFERENCES

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

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

[3] 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.

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

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

[6] 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.

[7] M. J. Berger and S. H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Transactions on Computers*, 1987.

[8] 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*, 2004.

[9] 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.

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

[11] 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), 2005.

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

[13] 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.

[14] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Softw.*, 3(3):209–226, Sept. 1977.

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

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

[17] N. Komatsu, T. Kiwata, and S. Kimura. Thermodynamic Properties of an Evaporation Process in Self-Fravitating N-Body systems. *Phys. Rev. E*, 82, Aug 2010.

[18] S. Martello and P. Toth. Knapsack Problems: Algorithms and Computer Implementations. *Chichester: John Wiley and Sons*, 1990.

[19] G. Morton. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. *IBM tech report*, 1966.

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

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

[22] 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 *Programming Language Design and Implementation (PLDI)*, 2011.

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

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

[25] K. Schloegel, G. Karypis, and V. Kumar. Parallel Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *Intl. Euro-Par Conf. on Parallel Processing*, 2000.

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

[27] 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.

[28] 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.

[29] 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.

[30] S. Teng. Provably Good Partitioning and Load Balancing Algorithms for Parallel Adaptive N-Body Simulation. *SIAM Journal on Scientific Computing*, 19(2):635–656, 1998.

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

[32] C. Walshaw and M. Cross. Parallel Optimization Algorithms for Multilevel Mesh Partitioning. *Parallel Computing*, 2000.

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

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

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

[36] 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.

[37] A. M. Wissink, D. Hysom, and R. D. Hornung. Enhancing Scalability of Parallel Structured AMR Calculations. In *Intl. Conf. on Supercomputing (ICS)*, 2003.