Periodic Hierarchical Load Balancing for Large Supercomputers

Gengbin Zheng, Abhinav Bhatelé, Esteban Meneses and Laxmikant V. Kalé

Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{gzheng, bhatele, emenese2, kale}@illinois.edu

Abstract

Large parallel machines with hundreds of thousands of processors are being built. Ensuring good load balance is critical for scaling certain classes of parallel applications on even thousands of processors. Centralized load balancing algorithms suffer from scalability problems, especially on machines with relatively small amount of memory. Fully distributed load balancing algorithms, on the other hand, tend to yield poor load balance on very large machines. In this paper, we present an automatic dynamic hierarchical load balancing method that overcomes the scalability challenges of centralized schemes and poor solutions of traditional distributed schemes. This is done by creating multiple levels of load balancing domains which form a tree. This hierarchical method is demonstrated within a measurement-based load balancing framework in Charm++. We present techniques to deal with scalability challenges of load balancing at very large scale. We show performance data of the hierarchical load balancing method on up to $16,384$ cores of Ranger cluster (at TACC) and $65,536$ cores of a Blue Gene/P at Argonne National Laboratory for a synthetic benchmark. We also demonstrate the successful deployment of the method in a scientific application, NAMD with results on the Blue Gene/P machine at ANL.

Keywords: load balancing, scalability, hierarchical algorithms, parallel applications, performance study

1 Introduction

Parallel machines with over hundred thousand processors are already in use. It is being speculated that by the end of this decade, Exaflop/s computing systems that may have tens of millions of cores will emerge. Such machines will provide unprecedented computing power to solve scientific and engineering problems. Modern parallel applications which use such large supercomputers often involve simulation of dynamic and complex systems (Phillips et al. 2002; Weirs et al. 2005). They use techniques such as multiple time stepping and adaptive refinements which often result in load imbalance and poor scaling. For such applications, load balancing techniques are crucial to achieve high performance on very large scale machines (Devine et al. 2005; Bhatele et al. 2008).

Several state-of-the-art scientific and engineering applications such as NAMD (Phillips et al. 2002) and ChaNGa (Jetley et al. 2008) adopt a centralized load balancing strategy, where load balancing decisions are made on one specific processor, based on the load data collected at runtime. Since global load information is readily available on a single processor, the load balancing algorithm can make excellent load balancing decisions. Centralized load balancing strategies have been proven to work very well on up to a few thousand processors (Phillips et al. 2002; Bhatele et al. 2008). However, they face scalability problems, especially on machines with relatively small amount of memory. Such problems can be overcome by using distributed algorithms. Fully distributed load balancing, where each processor exchanges workload information only with neighboring processors, decentralizes the load balancing process. Such strategies are inherently scalable, but tend to yield poor load balance on very large machines due to incomplete information (Ahmad and Ghafoor 1990).

It is evident that for petascale/exascale machines, the number of cores and nature of the load imbalance problem will necessitate the development of a qualitatively different class of load balancers. First, we need to develop algorithmically efficient techniques because increasing machine and problem sizes leads to more complex load balance issues. Efforts involved in balancing the load can itself become a performance bottleneck if the performance gain due to better load balance is offset by the high cost of the load balancing process itself. Further, at large scales, it might be impossible to store load information that is used for making load balancing decisions on a single processor. Hence, we need to develop effective techniques to use information that is distributed over many processors. This paper will present a load balancing strategy designed for very large scale machines. It overcomes the scalability challenges discussed above by exploiting a tree-based hierarchical approach.

The basic idea in our hierarchical approach is to divide the processors into independent autonomous groups and to organize the groups in a hierarchy, thereby decentralizing the load balancing task. At each

level, the processor at a given node balances load across all processors in its sub-tree. The root of the tree balances load across all the groups. This method reduces the time and memory required for load balancing since the groups are much smaller than the entire set of processors. We present the following ideas in this paper: techniques to construct the tree using machine topology information to minimize communication and improve locality; a method that explicitly controls and reduces the amount of load data aggregated to the higher levels of the tree; and a token-based load balancing scheme to minimize the cost of migration of tasks. We also demonstrate that this hierarchical approach does not significantly compromise the quality of load balance achieved, even though we do not have global load information available at each load balancing group.

We demonstrate the proposed hierarchical approach within a measurement-based load balancing framework in CHARM++ (Kalé and Krishnan 1993; Zheng 2005), which explicitly targets applications that exhibit persistent computational and communication patterns. Our experience shows that a large class of complex scientific and engineering applications with dynamic computational structure exhibit such behavior. To perform load balancing on these applications, load balancing metadata (i.e. application's computational load and communication information) can be obtained automatically by runtime instrumentation. It is also important to note that the idea of our proposed hierarchical approach may also apply to applications that do not exhibit such patterns, for example, those expressed in master-workers style, where the work load can be approximated by the number of tasks in the task pool.

The remainder of the paper is organized as follows: Section 2 describes CHARM++ and its load balancing framework, which is the infrastructure on which the proposed hierarchical load balancers are implemented. Design and implementation of the hierarchical load balancing method is presented in Section 3. Performance results using the hierarchical load balancers for a synthetic benchmark and for a production scientific application, NAMD are provided in Section 4. Section 5 discusses existing work for scalable load balancing strategies. Finally, Section 6 concludes the paper with some future plans.

2 Periodic Load Balancing

Load balancing is a technique of distributing computational and communication load evenly across processors of a parallel machine so that no single processor is overloaded. In order to achieve global load balance, some schemes allow only migration of newly created tasks, such as those in the field of *task scheduling* problem, while other schemes allow migration tasks in progress. In all these schemes, detailed computation and communication information need to be maintained continuously to be used as metadata for making load balancing decisions. In this paper, we mainly consider periodic load balancing schemes, in which the loads

are balanced only when needed by migrating existing tasks. With periodic load balancing, expensive load balancing decision making and task data migration occur only at the balancing times.

A large class of load balancing strategies in CHARM++ belong to the category of the periodic load balancing scheme (Zheng 2005). Periodic load balancing schemes are suitable for a class of iterative scientific applications such as NAMD (Phillips et al. 2002), FEM (Lawlor et al. 2006) and climate simulation, where the computation typically consists of a number of time steps, a number of iterations (as in iterative linear system solvers), or a combination of both. A computational task in these applications is executed for a long period of time, and tends to be persistent. During execution, partially executed tasks will be moved to different processors to achieve global load balance. These characteristics make it challenging to apply the load balancing method in the field of *task scheduling*, where it is often assumed that once a task is started, it must be able to execute to completion (Dinan, Larkins, Sadayappan, Krishnamoorthy, and Nieplocha 2009).

One difficulty of all the load balancing schemes is how to obtain the most current detailed computation and communication information for making load balancing decisions. CHARM++ uses a heuristic known as the *principle of persistence* for iterative applications. It posits that, empirically, for certain classes of scientific and engineering applications, when they are expressed in terms of natural objects (as CHARM++ objects or threads), the computational loads and communication patterns *tend to* persist over time, even in dynamically evolving computations. This has led to the development of measurement-based load balancing strategies that use the recent past as a guideline for the near future. These strategies have proved themselves to be useful for a large class of applications (such as NAMD (Bhatele et al. 2008), ChaNGa (Jetley et al. 2008), Fractography3D (Mangala et al. 2007)), up to thousands of processors. In measurement-based load balancing strategies, the runtime automatically collects the computational load for each object and its communication patterns and records them in a load "database" on each processor. The advantage of this method is that it provides an automatic application-independent method to obtain load information without users giving hints or manually predicting the load.

The run-time assesses the load database periodically and determines if load imbalance has occurred. Load imbalance can be computed as:

$$\sigma = \frac{L_{max}}{L_{avg}} - 1 \tag{1}$$

where L_{max} is the load of the most overloaded processor, and L_{avg} is the average load of all the processors. Since a parallel program can only complete when the most loaded processor completes its work, L_{max} represents the actual execution time of the program, while L_{avg} represents the performance in the best scenario when the load is balanced. Note that even when load imbalance occurs ($\sigma > 0$), it may not be profitable to start a new load balancing step due to the overhead of load balancing itself. When the run-

time determines that load balancing would be profitable, the load balancing decision module uses the load database to compute a new assignment of objects to physical processors and informs the run-time to execute the migration decisions.

2.1 Migratable Object-based Load Balancing Model in Charm++

In our design of the hierarchical load balancing scheme, we consider a petascale application as a massive collection of migratable objects communicating via messages, distributed on a very large number of processors. Migrating objects and their associated work from an overloaded processor to an underloaded processor helps in achieving load balance. Our implementation takes advantage of the existing CHARM++ load balancing framework (Zheng 2005) that has been implemented based on such an object model (Lawlor and Kalé 2003).

Most of the existing load balancing strategies used in production CHARM++ applications are based on centralized schemes. We have demonstrated the overheads of centralized load balancing in the past in a simulation environment (Zheng 2005). A benchmark that creates a specified number of tasks, n on a number of processors, p (where $n >> p$) was used. In the benchmark, tasks communicate in a two-dimensional mesh pattern. The load balancing module collects load information for each task on every processor. Information per task includes the task ID, computation time, and data for each communication edge including source and destination task ID, communication times and volume. Processor level load information is also collected for each processor, including number of tasks on each processor, processor's background load and idle time.

We measured the memory usage on the central processor for various experimental configurations. The results are shown in Table 1. The memory usage reported is the total memory needed for storing the task-communication graph on the central processor. The intermediate memory allocation due to the execution of the load balancing algorithm itself is not included.

As the results show, the memory overhead of storing the load information in a centralized load balancing strategy increases significantly as the number of tasks increases. In particular, for an application with 1 million tasks running on $65,536$ processors, the database alone requires around 450 MB of memory, which is non-trivial for machines with relatively low memory. This clearly becomes a bottleneck when executing a realistic load balancing algorithm on a million core system with even more task units.

This motivated the work in this paper to design a hierarchical load balancing scheme that allows the scaling of load balancing strategies to very large number of processors without sacrificing the quality of the load balance achieved. We hope and expect that the techniques we present are of use to other periodic load balancing systems to solve scalability challenges in load balancing.

3 Hierarchical Load Balancing

The basic idea in our hierarchical strategy is to divide the processors into independent autonomous groups and to organize the groups in a hierarchy, thereby decentralizing the load balancing task. For example, a binary-tree hierarchical organization of an eight-processor system is illustrated in Figure 1. In the figure, groups are organized in three hierarchies. At each level, a root node of the sub-tree and all its children form a load balancing group.

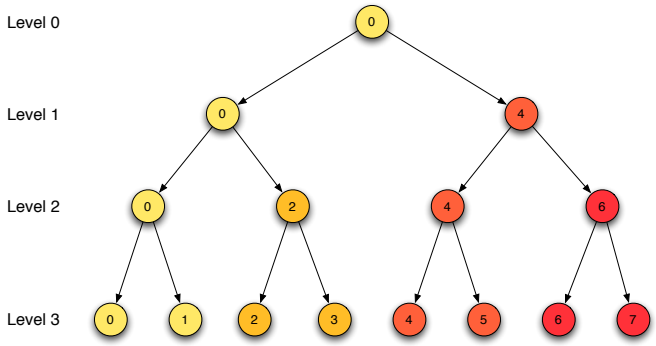

Figure 1: Hierarchical organization of an eight processor system

Generalizing this scheme, an intermediate node at level l_i and its immediate children at level l_{i-1} form a load balancing group or domain, with the root node as a group leader. Group leaders are in charge of balancing load inside their domains, playing a role similar to the central node in a centralized load balancing scheme. Root processors at level l_i also participate in the load balancing process controlled by their group leaders at level l_{i+1}. Processors in the subtree of the group leaders at level l_i do not participate in the load balancing process at level l_{i+1}.

During load balancing, processors at the lowest level of the tree send their object load information to their domain leaders (parent processors) respectively. At each level, load and communication data are converted such that domain leaders represent their entire sub-domains. In particular, load data are converted so that it appears as if all objects belong to the domain leader, and all “outgoing message" records from senders inside the domain are now represented as messages from the domain leader. With the aggregated load and communication database, a general centralized load balancing strategy can be applied within each individual sub-domain by its domain leader. From a software engineering point of view, this method is beneficial in that it takes advantage of the many existing centralized load balancing algorithms.

When the tree is uniform for a domain leader, the size of its load balancing sub-domains (i.e. the number of processors in its subtrees) is the same. The centralized load balancing strategy then distributes the load evenly to its sub-domains. However, when the tree is not balanced for a domain leader, every

sub-domain should receive work proportional to its size. This is done by assigning normalized CPU speeds to each sub-domain (the sub-domain leader acts as a representative processor of its domain) such that a smaller sized sub-domain is represented by a slower CPU speed. CHARM++ centralized load balancing strategies take these CPU speeds into account when making load balancing decisions, i.e. faster processors take proportionally bigger work load, and vice versa.

In this scheme, as load balancing moves up in the tree, the load balancing cost increases as the size of the aggregated load balancing metadata grows. Our design goal for the hierarchical load balancing scheme therefore focus on the optimizations that reduce communication, minimize memory usage, and limit data migration. We now discuss these optimizations.

Topology-aware Tree Construction: The tree can be built in several ways to take advantage of the topology characteristics of the architectures. In particular, a tree can be built according to the machine's network topology to minimize communication overhead. The CHARM++ runtime has capabilities to obtain information about the physical topology for some classes of supercomputers such as IBM Blue Gene and Cray XT machines (Bhatelé et al. 2010). This can be used to optimize the tree construction for reducing communication. For example, for a three-dimensional (3D) torus topology, the load balancing domain can be constructed at the lowest level by simply slicing the 3D torus along the largest dimension. The advantage of topology-aware tree construction is in minimizing network contention since processors are topologically close to each other in a domain and the algorithms tend to reassign the work within the domain as much as possible.

Load Data Reduction: As load information propagates to a node at a higher level, the amount of load balancing metadata including computation and communication load increases considerably since the domain size increases. Hence, much larger memory is required on the higher level nodes to store the integrated load database. Therefore, it is necessary to shrink load data while propagating it to higher levels. Based on the physical memory available on a processor and the application memory requirements, we can calculate a limit on the memory available for the load balancer ($\bar{M}$). During load balancing, the amount of memory actually needed for storing the load database at level i can be calculated as:

$$M_i = N_i * sizeof(ObjData) + C_i * sizeof(CommData) \tag{2}$$

where N_i is the total number of objects , and C_i is the number of communication records at the level i. *ObjData* is the data structure which records the load data per object and *CommData* is the data structure for the communication data.

The runtime uses a memory usage estimator to monitor the load data memory usage. When $M_i > \bar{M}$, load data needs to be shrunk at level i to fit in memory. Three strategies to reduce the memory usage have been explored:

- At each level, communication data can be shrunk by deleting some trivial communication records between objects. The heuristic applied is that it is more important for the load balancing algorithm to optimize communication of the most heavily communicating objects.
- Use a coarsening scheme to reduce the number of objects and their load balancing metadata. This is done by consolidating multiple objects to one single *container* object, whose load is the sum of the load of the objects it represents. The coarsening method can be done by calling METIS library to partition the object communication graph into fewer groups, where each group is now a container object. The advantage of using METIS is to take object communication into account so that most communicating objects can be group together into a container object. When a migration decision is made to a container object, all the objects it represents will migrate together.
- When the amount of load data is prohibitively large at a certain level, a dramatic shrinking scheme is required. In this case, only the total load information (sum of all object loads) is sent to the higher level and load balancing at this domain switches to a different mode — "semi-centralized load balancing".

In the semi-centralized scheme, a group leader of a domain does not make detailed migration decisions about which object migrates to which processor. It only makes decisions on the *amount* of load of a sub-domain to be transferred to another sub-domain. It is up to the group leaders of each sub-domain to independently select objects to migrate to other sub-domains according to the decisions made by the parent processor.

Token-based Load Balancing: In the hierarchical load balancing scheme, one of the challenges is to balance load across multiple domains, while at the same time, minimizing data migration. Some hierarchical load balancing schemes (such as (Willebeek-LeMair and Reeves 1993)) balance the domains from bottom up. This method incurs repeated load balancing effort when ascending the tree. Even when each sub-tree is balanced, a higher level domain will re-balance all the domains in its sub-tree. This behavior can lead to unnecessary multiple hops of data migration across domains before the final destination is reached, which is not efficient due to the cost of migrating objects.

Our load balancing scheme overcomes this challenge by using a top down token-based approach to reduce the excessive object migration. As shown in Figure 2, the actual load balancing decisions are made starting from the top level, after load statistics are collected and coarsened at the top level. A refinement-based load

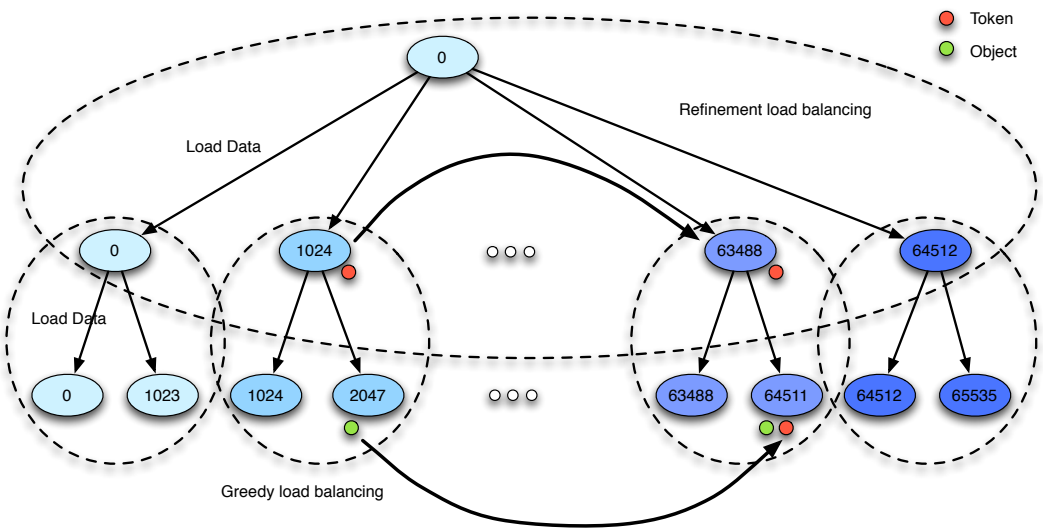

Figure 2: Hierarchical token-based load balancing scheme

balancing algorithm is invoked to make global load balancing decisions across the sub-domains. When load balancing decisions are made, lightweight tokens that carry only the objects' workload data are created and sent to the destination group leaders of the sub-domains. The tokens represent the movement of objects from an overloaded domain to an underloaded domain. When the tokens that represent the incoming objects arrive at the destination group leader, their load data are integrated into the existing load database on that processor. After this phase, the load database of all the group leaders at the lower level domains is updated, reflecting the load balancing decisions made – new load database entries are created for the incoming objects, and load database entries corresponding to the outgoing objects are removed from the database. This new database can then be used to make load balancing decisions at that level. At the intermediate levels of the tree, load balancing decisions are made in the form of which object migrates to which sub-domain. This process repeats until load balancing reaches the lowest level, where final load balancing decisions are made on migrating objects and their final destination processors.

At this point, tokens representing a migration of an object may have traveled across several load balancing domains, therefore its original processor needs to know which final destination processor the token has traveled to. In order to match original processors with their tokens, a global collective operation is performed on the tree. By sending tokens instead of actual object data in the intermediate load balancing phases of the hierarchical tree, this load balancing scheme ensures that objects are only migrated once after all the final migration decisions are made.

3.1 Deciding the number of levels

The general problem of determining an optimal tree depends on factors such as the use of varying branching factors and different load balancing algorithms at different levels of the tree. This general problem of determining the optimal tree is beyond the scope of this paper. However, this section offers an analysis of the complexity of a hierarchical load balancing algorithm in a simplified but typical scenario we commonly use.

Load Balancing Time: For illustration, we assume that the branching factor of the tree (G) is same across all levels, and a refinement load balancing algorithm (RefineLB) is applied at each level of the tree. RefineLB uses an algorithm that strives to reduce the cost of migration by moving only a few objects from overloaded processors to underloaded ones so that load of all processors gets close to the average. This is done by examining every object on an overloaded processor and looking for the best choice of an underloaded processor to migrate the object to. processors in each load balancing domain. The complexity of this algorithm is $\mathcal{O}(GlogG + N_i logG)$, where N_i is the number of migratable objects in each load balancing domain at level i and G is the number of processors in the domain. In this formula, $\mathcal{O}(GlogG)$ is the time it takes to build an initial max heap of processor loads for overloaded processors and $\mathcal{O}(N_i logG)$ is the time it takes to examine objects on overloaded processors and update the max heap with the decision of where the object moves.

When the memory usage reaches a threshold at certain level, the semi-centralized load balancing algorithm is used to reduce the memory footprint of the algorithm. The complexity of this algorithm is $\mathcal{O}(GlogG+N_i)$. Here, $\mathcal{O}(GlogG)$ is the time it takes to calculate average load among sub-domains, build an initial max heap of processor loads for overloaded sub-domains, and compute the amount of total load for each overloaded sub-domain that migrates to an underloaded sub-domains; $\mathcal{O}(N_i)$ is the time on each sub-domains to make decisions on which object to move.

At level i from the top, the number of objects in a load balancing domain at that level is $N_i = N/G^i$, assuming even distribution of the objects to processors. When $N_l > M$ at level l, where M is the threshold to start load data reduction, the load balancing algorithm switches from refinement strategy to the semi-centralized load balancing strategy to reduce memory usage. The total cost of the hierarchical load balancing algorithm is the summation of the cost on each level of the tree (except the lowest level where there is no load balancing), where the first l levels (from the top of the tree) invoke the semi-centralized load balancing

algorithm, and the rest of the levels invoke the regular refinement algorithm:

$$\mathcal{O}(\sum_{i=0}^{l}(GlogG + N_i) + \sum_{i=l+1}^{L-2}(GlogG + N_i logG))$$

where the number of levels, $L = log_G P + 1$ (from $G = \sqrt[L-1]{P}$). Also $l = log_G(\frac{N}{M})$, which is the switching point of the load balancing algorithms when load data reduction occurs.

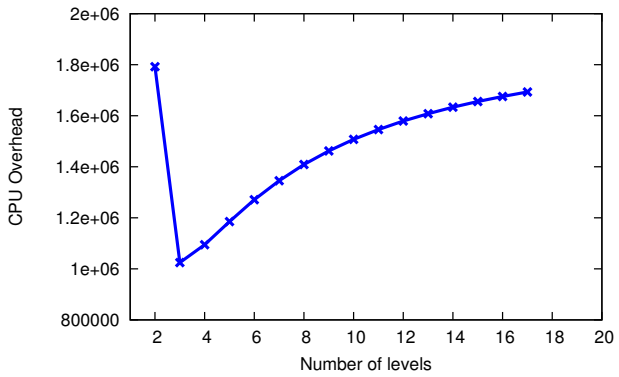

Figure 3: Plot showing the dependence of load balancing time on the number of levels in the tree when using RefineLB

As an example, Figure 3 shows a plot of computational overhead for load balancing an application that has $1000,000$ parallel objects on $65,536$ processors for varying number of levels of the tree. The threshold for load data reduction is when the number of objects in a load balancing sub-domain is greater than $500,000$. We can see that, with a three-level tree, the load balancing time is the shortest. In Section 4.1.3, we will use a synthetic benchmark to demonstrate this effect.

Intuitively, when the tree has fewer levels, the quality of load balancing tends to be better since, with increasing sub-domain size at lower levels, more global information becomes available. In particular, when the tree comes down to two levels (i.e. depth of one), hierarchical load balancing becomes equivalent to the centralized load balancing, however losing the benefits of multi-level load balancing. Therefore, to achieve good load balance that is close to the centralized load balancing algorithm, the heuristic for building the tree is to maximize the branching factor at the lower level to the point that the cost of the algorithm is just affordable. This allows us to exploit the advantages of the centralized load balancing algorithm to the extent possible at the lowest level. In our experiments, we found this three-level tree with high branching factor at the lowest level is generally good.

Communication Overhead: We analyze the communication overhead in terms of the total number of messages generated by the hierarchical load balancing scheme. The messages due to migrating object data

are not counted, because the migration pattern depends on both the pre-condition of the load balance and the types of load balancing algorithms used.

Given level i in a hierarchical tree (i starts from 0, which is the top level), there are P/G^{L-1-i} load balancing domains at a level i of a hierarchical tree, where L is the total number of levels. In the first phase of the hierarchical load balancing when collecting load statistics, messages are sent from the leaves to the root, therefore, each domain processor receives G load data messages sent from its children, yielding a total number of

$$\sum_{i=0}^{L-2} \frac{P}{G^{L-1-i}} G = \frac{PG - G}{G - 1}$$

messages. Note that $G^{L-1} = P$.

In the second phase when each load balancing domain leader makes load balancing decisions starting from top to bottom along the tree, the communication pattern is similar to the first phase but in the reverse order. This generates the same number of messages, i.e. $\frac{PG-G}{G-1}$.

In the third phase of the load balancing, a global collective operation is performed to propagate load balancing decisions to all processors with a communication pattern similar to phase 1. Another same number of messages is generated. Therefore the total number of messages in one hierarchical load balancing is,

$$3 \times \sum_{i=0}^{L-1} \frac{P}{G^i} G = 3 \times \frac{PG - G}{G - 1}$$

For example, given a binary tree (i.e. $G = 2$) , the total number of messages in load balancing is $6P - 6$. Given a machine of 64K processors, Figure 4 shows a plot of the number of messages varying the number of levels of the hierarchical trees (note that $G = \sqrt[L-1]{P}$). It can be seen that as the number of levels increases, the number of load balancing messages increases dramatically. This suggests that using a tree of small number of levels is beneficial in reducing communication overhead.

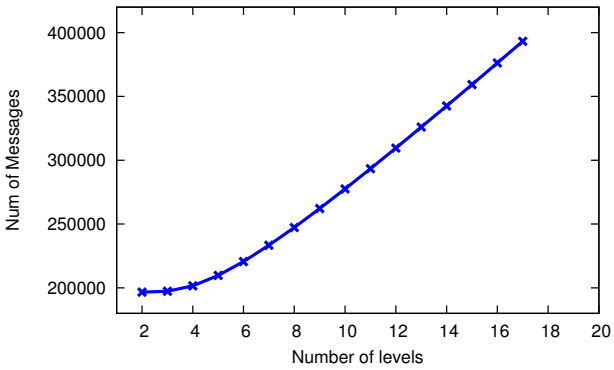

Figure 4: Plot showing the dependence of number of messages on the number of levels in the tree

4 Performance Results

One question that remains to be answered is whether these benefits of decrease in memory usage and increase in efficiency are attained by sacrificing the quality of the load balance achieved. To answer this question, we evaluate the performance of hierarchical load balancing schemes on two major HPC systems for up to $16,384$ and $65,536$ cores respectively using a synthetic benchmark and a production application, NAMD. In the next two sections, we show that the hierarchical load balancing strategy does not compromise application performance, despite the fact that there is lack of global load information. In addition, we demonstrate the dramatic improvement of using the hierarchical load balancing strategy in reducing the load balancing decision making time, and the resource required.

4.1 Synthetic Benchmark

This section offers a comparative evaluation between hierarchical and centralized load balancers using a synthetic benchmark. This benchmark provides a scenario where it is possible to control load imbalance. We will call the benchmark "lb_test". It creates a given number of objects, distributed across all the processors. The number of objects is much larger than the number of processors. The work done by each object in each iteration is randomized in a parameterized range. At each step, objects communicate in a ring pattern with neighboring objects to get boundary data and do some computation before entering the next step. All objects are sorted by their load or computation time and this ordering is used to place them on all processors assigning equal number of objects to each. This placement scheme creates a very challenging load imbalance scenario that has the most overloaded processors at one end, and least overloaded processors at the other end.

4.1.1 Evaluation on the Ranger Cluster

Our first test environment is a Sun Constellation Linux Cluster called Ranger installed at the Texas Advanced Computing Center. It is one of the largest computational resources in the world. Ranger is comprised of $3,936$ 16-way SMP compute nodes providing a total of $62,976$ compute cores. All Ranger nodes are interconnected using InfiniBand technology. The experiments were run on $16,384$ cores, where a three-level tree is optimal. Specifically, the tree for $16,384$ cores is built as following: groups of 512 processors form load balancing domains at the first (lowest) level and 32 such domains form the second level load balancing domain. The same branching factor of 512 at the lowest level is also used for building three-level trees for $4,096$ and $8,192$ core cases. We use greedy-based load balancing algorithms at the lowest level and a refinement-based load balancing algorithm (with data shrinking) at the high level.

As comparison, three different centralized algorithms are used. The first one is a simple scheme called *GreedyLB*, which always picks the heaviest unassigned object and assigns it to the currently least loaded processor. For N objects and $N >> P$, this is an $O(NlogN)$ algorithm. The second one is called *GreedyCommLB*. It is an algorithm similar to GreedyLB, however is much more expensive in that it takes communication into account. When making assignment for a heaviest object, it not only checks against the least loaded processor, but also checks the processors that the object communicates with in order to find the best possible choice. The third one is *RefineLB*. Unlike the previous two load balancing strategies which make load balancing decision from scratch, RefineLB improves the load balance by incrementally adjusting the existing object distribution. Refinement is used with an overload threshold. From example, with a threshold of 1.03, all processors with a load greater than 1.03 times of the average load (i.e. with 3% overload) are considered overloaded, and objects will migrate out of the overloaded processors. This algorithm is implemented with an additional heuristic: it starts with a higher overload threshold, and uses a binary search scheme to reduce the threshold to achieve a load balance close to average load.

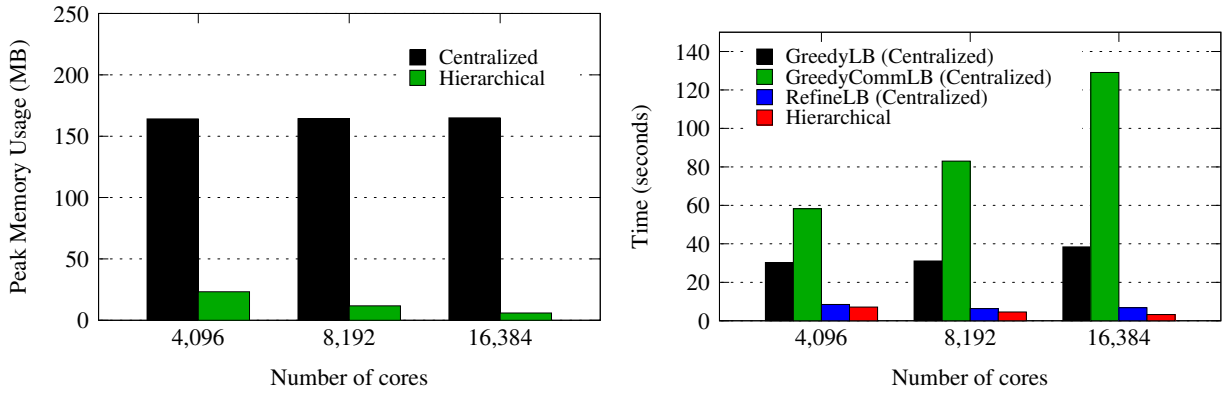

Figure 5: Comparison of peak memory usage and time for load balancing using CentralLB versus HybridLB for lb_test (on Ranger)

We first measured the maximum memory usage of all processors due to the collection of the load metadata in a hierarchical load balancing scheme (HybridLB), and compared to the cases of using centralized load balancing strategies. In these tests, the lb_test program creates a total of 1 M(1,048,576) objects running on varying number of processors on up to 16,384 cores of Ranger, the maximum allowed job size on the Ranger cluster. The results are shown in Figure 5 (left plot). Comparing with the memory usage in the centralized load balancing strategies, the memory usage of the hierarchical load balancer (HybridLB) is significantly reduced. Furthermore, the maximum memory usage of HybridLB decreases as the number of processors increases, while the memory usage in case of centralized load balancing remains almost the same. This is because with the fixed problem size in these tests, when the number of processor doubles, each

domain has half the number of objects, and the size of the object load database on each group leader reduces accordingly. In the case of centralized load balancing, however, all object load information is collected on the central processor regardless of the number of processors. Therefore, the size of the load database in that case is about the same.

Figure 5 (right plot) compares the load balancing time spent in HybridLB and the three centralized load balancers for the same problem size that has 1M objects running on up to 16,384 cores. The results show that the hierarchical load balancer is very efficient compared with the greedy-based centralized load balancers. This is largely due to the fact that given much smaller load balancing domains and a smaller load balancing problem for each sub-domain to solve, the load balancing algorithms run much faster. Further, HybridLB exploits more parallelism by allowing execution of load balancing concurrently on independent load balancing domains. Compared to the greedy-based load balancers, RefineLB is much more efficient, This is due to the fact that RefineLB only migrates a fraction of objects, thus reducing the data migration time. This will become clear when we look at the total number of objects each load balancing strategy decided to migrate. As shown in the Table 2, RefineLB migrates much smaller number of objects, while greedy-based load balancers migrate almost all of the 1M objects. Note that HybridLB uses a greedy load balancing algorithm at the lowest level of the tree, HybridLB migrates almost all of the 1M objects, similar to centralized greedy-based load balancers.

The breakdown of the time spent in load balancing in lb_test benchmark in Table 3 illustrates where the reduction in load balancing time comes from. The time is shown in three different phases — data collection, load balancing algorithm execution (strategy time), and data migration. Three centralized load balancing schemes, GreedyLB, GreedyCommLB and RefineLB are compared with the HybridLB. We see that GreedyCommLB that takes communication into account is much more expensive than the GreedyLB which only considers load. HybridLB reduces time in all three phases. The most significant reduction happens in the time spent on the strategy where the time of the centralized cases increases as the number of processors increases. The time in data migration phase is also considerably reduced by using hierarchical strategy. This is due to the fact that in the greedy-based centralized strategies, very large messages that contain load balancing decisions for around 1M objects are generated and broadcast from the central processor to every processor, leading to a communication bottleneck on the central processor. Hierarchical strategy, however, avoids this bottleneck by doing independent load balancing in each sub-domain. Although RefineLB is almost as efficient as HybridLB, it suffers from the same memory constraint as other centralized strategies (as illustrated in Figure 5).

To compare the quality of load balance achieved, we also compared the performance of lb_test after using the hierarchical and centralized load balancers and the results are shown in Figure 6. The first bar in

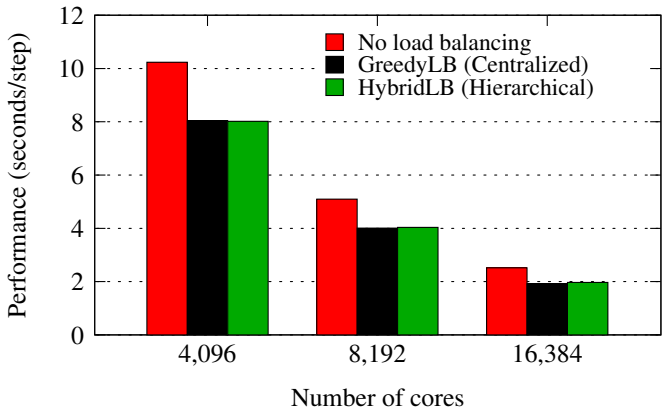

Figure 6: Performance (time per step) for lb_test to compare the quality of load balance with different load balancing strategies (on Ranger)

each cluster shows the time per step in lb_test without load balancing. The other two bars in each cluster represent the time per step after applying the centralized and hierarchical load balancers, respectively. We can see that both HybridLB and centralized GreedyLB improve lb_test performance, and HybridLB performs comparably to GreedyLB.

4.1.2 Evaluation on Blue Gene/P

To further study how the hierarchical load balancer performs at even larger scales, we ran the same experiments on Intrepid, a Blue Gene/P installation at Argonne National Laboratory. Intrepid has 40 racks, each of them containing 1024 compute nodes. A node consists of four PowerPC450 cores running at 850 MHz. Each node only has 2 GB of memory, making it a difficult environment for using centralized load balancing strategies due to the low memory available.

Indeed, lb_test with 1M (1,048,576) objects does run out of memory at load balancing in the phase of load metadata collection. Thus, only half a million objects (*i.e.* 524,288) are created in lb_test in the following experiments. Figure 7 illustrates the load balancing time using two centralized load balancers (GreedyLB and RefineLB), and the hierarchical load balancer (HybridLB). For each experiment, the breakdown of the time in the three load balancing phases are displayed in three different colors. We see that the hierarchical load balancer is very efficient even as number of cores increase. Specifically, HybridLB on $65,536$ cores only takes 0.55 seconds. The RefineLB centralized load balancer executes faster than the GreedyLB when number of cores is less than $65,536$, largely due to the much smaller number of object migrating as shown in Table 4. However, its load balancing decision making time (strategy time) keeps increasing as the number of cores increases, possibly due to the heuristic on the overload threshold that leads to expensive binary search for the better load balance decisions .

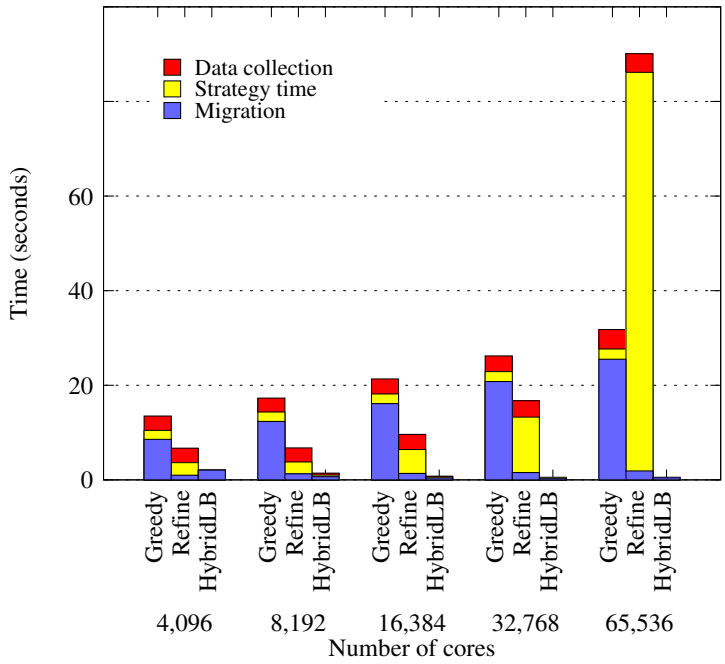

Figure 7: Comparison of time for load balancing using centralized load balancers (GreedyLB and RefineLB) versus HybridLB for lb_test (on Blue Gene/P)

4.1.3 Performance Study of the Number of Tree Levels

As analyzed in Section 3.1, the number of levels of the hierarchical tree and the level when load data reduction occurs and the load balancing algorithm switches can significantly affect the performance of the hierarchical load balancer.

In an experiment with lb_test benchmark on 4096 cores of Blue Gene/P, we evaluated the variation in load balancing time with various number of levels of the tree. Unlike the tree used in the previous experiments, which was a three-level tree with a large branching factor ($G = 512$) at the lowest level, the tree used in these experiments are uniform with the same branching factor at all levels. For example, for 4096 cores, a four level tree has a branching factor 16.

In most of these experiments, the threshold that triggers data reduction and switching load balancing algorithms is when the number of objects in a load balancing sub-domain at level i is greater than $65,536$ (i.e. $N_i > 65,536$). For example, for a 3 level tree with $G = 64$, the data reduction occurs at level 1, even though the number of objects at that level is only 8192. This is because otherwise at its parent level (level 0, the root of the tree), the number of objects collected from all its 64 children would reach $524,288$, which is beyond the threshold. The results of the HybridLB load balancing time are shown in Table 5. The "shrink level" in the third column is the level when load balancing reaches the threshold of doing data reduction, and "# objects" in the fourth column is the estimated number of objects in the load balancing sub-domain

at that time. We can see that the three-level tree performs the best in this particular test scenario. This reflects the analysis of the load balancing tree in Section 3.1.

4.2 Application Study – NAMD

NAMD (Phillips et al. 2002; Bhatele et al. 2008) is a scalable parallel application for molecular dynamics simulations written using the Charm++ programming model. The load balancing framework in Charm++ is deployed in NAMD for balancing computation across processors. Load balancing is measurement-based – a few time-steps of NAMD are instrumented to obtain load information about the objects and processors. This information is used in making the load balancing decisions. Two load balancers are used in NAMD:

1. A **comprehensive** load balancer is invoked at start-up which does the initial load balancing and moves most of the objects around.

2. A **refinement** load balancer is called several times during the execution to refine the load by moving load from overloaded processors and bringing the maximum load on any processor closer to the average.

A greedy strategy is used in both load balancers, where we repeatedly pick the heaviest object and find an underloaded processor to place it on. This is repeated until the load of the most overloaded processor is within a certain percentage of the average. More details on the load balancing techniques and their significance for NAMD performance can be found in (Bhatelé et al. 2009; Kalé et al. 1998).

Traditionally, the load balancers in NAMD have been centralized where the load balancing statistics are collected on a certain processor (typically, processor 0) and this processor is in charge of making the decisions. This is becoming a bottleneck for very large simulations using NAMD on large supercomputers. Load balancing can sometimes take as long as a thousand time steps of the actual simulation. Table 6 shows the time processor 0 takes to calculate load balancing solutions in the centralized case. As we scale from $1,024$ to $16,384$ processors, the time for refinement load balancing increases by 315 times!

The main driver, from the point of view of the NAMD user, to deploy the hierarchical load balancing scheme, has been to reduce the time for load balancing. We now describe the process of using the hierarchical load balancing schemes in this production code. For the hierarchical case, we build a tree with three levels and eight sub-domains. The final selection of the number of levels in the tree and the number of sub-domains may seem arbitrary, but we observed good results with this particular combination. To form each sub-domain, we simply group consecutive processors together, using the processor ID assigned by the Charm++ runtime.

Since our hierarchical load balancing scheme applies centralized strategies within each sub-domain of the tree, this allows NAMD to still use its optimized centralized load balancing algorithms, but within a

much smaller sub-domain. We still need to extend the existing comprehensive and refinement algorithms so that they work well with a subset of processors and relatively incomplete global information. Cross-domain load balancing is done according to the semi-centralized load balancing scheme described in Section 3. The root of the tree makes global load balancing decisions about the percentages of load to be moved from overloaded sub-domains to underloaded ones. The group leaders of overloaded sub-domains make detailed load balancing decisions about which objects to move.

To evaluate our new hierarchical load balancing strategy, we ran NAMD with a molecular system consisting of $1,066,628$ atoms. This system is called STMV (short for Satellite Tobacco Mosaic Virus) and it was run with the long-range electrostatic force component (PME) disabled. There are approximately $100,000$ to $400,000$ objects for this system (depending on the decomposition) under the control of the load balancer. All the runs were executed on Intrepid, a Blue Gene/P installation at Argonne National Laboratory. Intrepid has 40 racks, each of them containing 1024 compute nodes. A node consists of four PowerPC450 cores running at 850 MHz.

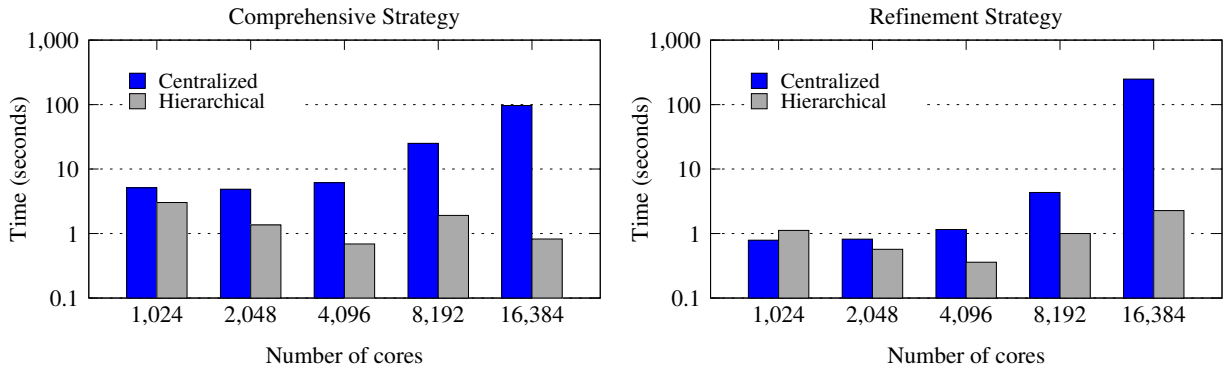

Figure 8: Comparison of time to solution for centralized and hierarchical load balancers for NAMD on Blue Gene/P (Molecular System: STMV)

Figure 8 (left plot) presents a comparison of the time spent in load balancing between the centralized and the hierarchical approach for the comprehensive load balancers. The load balancing time in the centralized case does not increase necessarily with the increase in number for cores because heuristic techniques are being used. We can see that the hierarchical strategy outperforms the centralized scheme by a large margin on all core counts (Note: the y-axis has a logarithmic scale). For instance, on $4,096$ cores, the centralized approach takes 6.16 seconds to balance the load. In contrast, the hierarchical load balancer takes only 0.69 seconds, which is a speedup of 8.9. Larger the core count, higher is the speedup. On $16,384$ cores, the hierarchical load balancers are faster by 117 times! The results for the refinement load balancing phase are similar. Figure 8 (right plot) compares the centralized and hierarchical balancers for the refinement load balancer. The highest reduction in load balancing time occurs at $16,384$ cores, where the time taken reduces

from 249.1 to 2.27 seconds, giving a speedup of 110.

A breakdown of the time spent in load balancing into three different phases – data collection, load balancing algorithm within the sub-domains (strategy time) and sending migration decisions would give us a better idea of where does the reduction happen. Table 7 shows a breakdown of the time spent in the comprehensive strategy for NAMD for the centralized and hierarchical algorithms. The most important reduction happens in the time spent on the strategy in the centralized case which increases as we scale to a larger number of processors. The time spent in sending migration decisions also benefits from the use of hierarchical strategies.

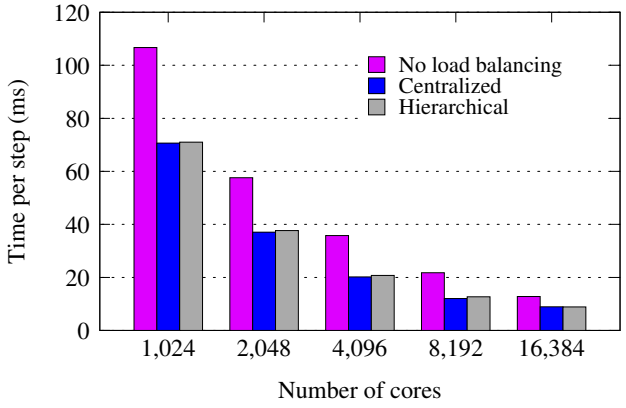

Figure 9: Comparison of NAMD's performance on Blue Gene/P when using centralized versus hierarchical load balancers (Molecular System: STMV)

We further demonstrate that the hierarchical load balancing strategy performs no worse than the centralized strategy in balancing the load in NAMD. Figure 9 shows the time per step performance of NAMD when the centralized and hierarchical load balancers are used (compared with no load balancing base line performance). On $4,096$ cores, the simulation time is 35.78 ms per step when no load balancing is performed, 20.21 ms per step for the centralized load balancing and 20.75 ms per step for the hierarchical load balancing case, showing negligible slowdown. Very similar results can be observed from $1,024$ to $16,384$ cores. This shows that the hierarchical load balancing strategy performs equally well in balancing the load compared to the centralized load balancers, while using less memory and taking significantly less time in computing the load balancing decisions.

5 Related Work

Load balancing is a challenging problem and has been studied extensively in the past. Load balancing strategies can be divided into two broad categories – those for applications where new tasks are created and

scheduled during execution (i.e. task scheduling) and those for iterative applications with persistent load patterns (i.e. periodic load balancing).

Much work has been done to study scalable load balancing strategies in the field of task scheduling, where applications can be expressed through the use of *task pools* (a *task* is a basic unit of work for load balancing). This task pool abstraction captures the execution style of many applications such as master-workers and state-space search computations. Such applications are typically non-iterative.

Neighborhood averaging schemes present one way of solving the fully distributed scalable load balancing problem (Corradi et al. 1999; Ha'c and Jin 1987; Kalé 1988; Shu and Kalé 1989; Sinha and Kalé 1993; Willebeek-LeMair and Reeves 1993). In these load balancing schemes, each processor exchanges state information with other processors in its neighborhood and neighborhood average loads are calculated. Each processor requests work from the processor with the greatest load in its neighborhood, to achieve load balance. Although these load balancing methods are designed to be scalable, they tend to yield poor load balance on extremely large machines or tend to take much longer time to yield good solutions due to a great degree of randomness involved in a rapidly changing environment (Ahmad and Ghafoor 1990).

Randomized work stealing is yet another distributed dynamic load balancing technique, which is used in some runtime systems such as Cilk (Frigo et al. 1998). Recent work in (Dinan et al. 2009) extends this work using the PGAS programming model and RDMA to scale work stealing to 8192 processors for three benchmarks. ATLAS (Baldeschwieler et al. 1996) and Satin (Nieuwpoort et al. 2000) use hierarchical work stealing for clusters and grids, both supporting JAVA programming on distributed systems.

Several other hierarchical or multi-level load balancing strategies (Ahmad and Ghafoor 1990; Furuichi et al. 1990) have been proposed and studied. Ahmad and Ghafoor (Ahmad and Ghafoor 1990) propose a two-level hierarchical scheduling scheme which involves partitioning a hypercube system into independent regions (spheres) centered at some nodes. At the first level, tasks can migrate between different spheres in the system; while at the second level, the central nodes schedule within their individual spheres. Although our work shares a common purpose with such work, we deal with scalability issues of load balancing encountered at very large scale in the context of production applications running on supercomputers. Most of the previous work presents results via simulation studies using synthetic benchmarks.

In the above load balancing strategies in the field of task scheduling, it is often assumed that the cost associated with migrating tasks is small, and once a task is started, it must be able to execute to completion (Dinan et al. 2009). This assumption is to avoid the need to migrate a partially executed task when load balancing is needed. This assumption holds true for divide and conquer and state-space search type of applications but not for iterative scientific applications. For scientific applications, a different class of load balancers is needed such as those in CHARM++ (Zheng 2005; Kalé et al. 1998) and Zoltan (Catalyurek et al.

2007). These load balancers support the migration of a task and associated data during the lifetime of the task. Unlike the task scheduling problem, migration of tasks and their data can be costly especially when user data is large and migration occurs frequently. Therefore, to reduce the excessive migration of tasks, these strategies typically invoke load balancing in a periodic fashion, that is, load balancing happens only when needed. These load balancing schemes are suitable for a class of iterative scientific applications such as NAMD (Phillips et al. 2002), FEM (Lawlor et al. 2006) and climate simulation, where the computation typically consists of a number of time steps, a number of iterations (as in iterative linear system solvers), or a combination of both. Phase-based load balancing strategies for iterative applications are the main focus of this paper.

Hierarchical phased-based load balancing strategies are also studied in the context of the iterative applications. Zoltan toolkit web site (Zoltan User's Guide) describes a hierarchical partitioning and dynamic load balancing scheme, where different balancing procedures are used in different parts of the parallel environment. However, it mainly considers machine hierarchy of clusters that consist of a network of multiprocessors, and does not consider the performance issues involved when load balancing on very large parallel machines.

6 Conclusion

Load balancing for parallel applications running on tens of thousands of processors is a difficult task. When running at scale, running time for the load balancing algorithm and memory requirements for the instrumented data become important considerations. It is impractical to collect information on a single processor and load balance in a centralized fashion. In this paper, we presented a hierarchical load balancing method which combines the advantages of centralized and fully distributed schemes. The proposed load balancing scheme adopts a phase-based load balancing approach that is designed for iterative applications that exhibit persistent computational and communication patterns. This hierarchical method is demonstrated within a measurement-based load balancing framework in Charm++. We discussed several techniques to deal with scalability challenges of load balancing which are found at very large scale in the context of production applications.

We presented results for a synthetic benchmark and a scientific application, NAMD on up to $65,536$ and $16,384$ cores respectively. Using hierarchical schemes, we were able to considerably reduce the memory requirements and the running time of the load balancing algorithm for the synthetic benchmark. Similar benefits were obtained for NAMD and the application performance was similar for the hierarchical and centralized load balancers. In the future, we would like to deploy the hierarchical load balancers in other applications. We also plan to extend our hierarchical load balancing algorithms to provide interconnect

topology aware strategies that map the communication graph on the processor topology to minimize network contention. This is especially useful for current supercomputers such as IBM Blue Gene and Cray XT machines.

Tables

No. of tasks	128K	256K	512K	1M
Memory (MB)	61	117	230	457

Table 1: Memory usage (in MB) on the central processor for centralized load balancing when running on $65,536$ cores (simulation data)

	GreedyLB	GreedyCommLB	RefineLB	HybridLB
4096	1048321	1048310	65994	1046721
8192	1048447	1048455	64777	1046662
16384	1048568	1048507	60097	1046715

Table 2: The number of objects load balancing strategies decide to migrate in lb_test benchmark with total of 1048576 objects (Ranger cluster)

	Data collection				Strategy time				Migration			
	Greedy	*Comm*	*Refine*	*Hier*	*Greedy*	*Comm*	*Refine*	*Hier*	*Greedy*	*Comm*	*Refine*	*Hier*
4096	4.79	2.43	2.10	1.22	0.71	27.47	3.07	0.42	24.67	28.32	3.30	6.66
8192	2.14	3.45	2.16	0.56	1.40	49.13	2.13	0.17	32.24	31.16	2.05	3.83
16384	2.44	2.70	2.35	0.44	1.42	94.18	2.10	0.09	34.54	32.22	2.38	2.70

Table 3: Breakdown of load balancing time (in seconds) into three phases for the three centralized (GreedyLB, GreedyCommLB, RefineLB) and hierarchical load balancing strategies in lb_test benchmark with 1 million objects (Ranger cluster)

	GreedyLB	RefineLB	HybridLB
4096	524166	30437	523326
8192	524232	29977	523338
16384	524255	25581	523306
32768	524274	20196	523360
65536	524283	17310	523364

Table 4: The number of objects load balancing strategies decide to migrate in lb_test benchmark with total of 524288 objects (Blue Gene/P)

Tree levels	Branching factor	shrink level	# objects	LB Time (s)
3	64	1	8192	0.52
4	16	1	32768	2.82
5	8	1	65536	21.47
7	4	2	32768	7.51
13	2	3	65536	59.63

Table 5: Comparison of using different levels of hierarchical trees in HybridLB in the lb_test benchmark on 4096 processors of Blue Gene/P

No. of cores	1024	2048	4096	8192	16384
Comprehensive	5.12	4.87	6.16	25.09	96.84
Refinement	0.79	0.82	1.16	4.33	249.10

Table 6: Time (in seconds) for centralized load balancing in NAMD

	Data collection		Strategy time		Migration	
	Cent	*Hier*	*Cent*	*Hier*	*Cent*	*Hier*
1024	0.36	0.78	3.77	1.84	0.99	0.41
2048	0.36	0.39	3.36	0.76	1.13	0.21
4096	0.41	0.20	4.12	0.38	1.63	0.11
8192	1.06	0.29	17.06	1.44	6.98	0.18
16384	1.25	0.15	84.44	0.46	11.15	0.22

Table 7: Breakdown of load balancing time (in seconds) into three phases for the centralized and hierarchical load balancing strategy (for comprehensive strategy in NAMD)

Acknowledgments

This work was supported in part by the NSF Grant OCI-0725070 for Blue Waters, the DOE Grant B341494 funded by Center for Simulation of Advanced Rockets, the DOE Grant DE-SC0001845 for HPC Colony II and by the Institute for Advanced Computing Applications and Technologies (IACAT). We used running time on the Blue Gene/P at Argonne National Laboratory, which is supported by DOE under contract DE-AC02-06CH11357. Runs on Ranger were done under the TeraGrid (Catlett et al. 2007) allocation grant ASC050040N supported by NSF.

Biographies

Gengbin Zheng: received the BS (1995) and MS (1998) degrees in Computer Science from the Peking University, Beijing, China. His Master's thesis was on High Performance Fortran compiler. He received his Ph.D. degree in the Computer Science Department at University of Illinois at Urbana-Champaign in 2005. He is currently a research scientist at the Parallel Programming Lab at the University of Illinois, working with Prof. Laxmikant V. Kalé. His research interests include parallel runtime support with dynamic load balancing for highly adaptive parallel applications, simulation-based performance prediction for large parallel machines and fault tolerance. A paper co-authored by him on scaling the molecular dynamics program NAMD was one of the winners of the Gordon Bell award in SC2002.

Abhinav Bhatelé: Abhinav received a B. Tech. degree in Computer Science and Engineering from I.I.T. Kanpur (India) in May 2005 and M. S. and Ph. D. degrees in Computer Science from the University of Illinois at Urbana-Champaign in 2007 and 2010 respectively. Currently, he is a post-doctoral research associate at the Parallel Programming Lab at the University of Illinois, working with Prof. Laxmikant V. Kalé. His research is centered around topology aware mapping and load balancing for parallel applications. Abhinav has received the David J. Kuck Outstanding MS Thesis Award in 2009, Third Prize in the ACM Student Research Competition at SC 2008, a Distinguished Paper Award at Euro-Par 2009 and the George Michael HPC Fellowship Award at SC 2009.

Esteban Meneses: received a B. Eng. degree in Computing Engineering (2001) and a M.Sc. degree in Computer Science (2007) from the Costa Rica Institute of Technology. Currently, he is a third year PhD student working with Prof. Laxmikant V. Kalé at the Parallel Programming Laboratory in the University of Illinois at Urbana-Champaign. From 2007 to 2009, Esteban held a Fulbright-LASPAU scholarship for the first two years of his PhD program. His research interests span the areas of scalable fault tolerance and load

balancing for supercomputing applications.

Laxmikant V Kalé: Professor Laxmikant Kalé has been working on various aspects of parallel computing, with a focus on enhancing performance and productivity via adaptive runtime systems, and with the belief that only interdisciplinary research involving multiple CSE and other applications can bring back well-honed abstractions into Computer Science that will have a long-term impact on the state-of-art. His collaborations include the widely used Gordon-Bell award winning (SC'2002) biomolecular simulation program NAMD, and other collaborations on computational cosmology, quantum chemistry, rocket simulation, space-time meshes, and other unstructured mesh applications. He takes pride in his group's success in distributing and supporting software embodying his research ideas, including Charm++, Adaptive MPI and the ParFUM framework.

L. V. Kalé received the B.Tech degree in Electronics Engineering from Benares Hindu University, Varanasi, India in 1977, and a M.E. degree in Computer Science from Indian Institute of Science in Bangalore, India, in 1979. He received a Ph.D. in computer science in from State University of New York, Stony Brook, in 1985. He worked as a scientist at the Tata Institute of Fundamental Research from 1979 to 1981. He joined the faculty of the University of Illinois at Urbana-Champaign as an Assistant Professor in 1985, where he is currently employed as a Professor.

References

Ahmad, I. and A. Ghafoor (1990). A semi distributed task allocation strategy for large hypercube supercomputers. In *Proceedings of the 1990 conference on Supercomputing*, New York, NY, pp. 898–907. IEEE Computer Society Press.

Baldeschwieler, J. E., R. D. Blumofe, and E. A. Brewer (1996). Atlas: an infrastructure for global computing. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, New York, NY, USA, pp. 165–172. ACM.

Bhatelé, A., E. Bohm, and L. V. Kalé (2010). Optimizing communication for Charm++ applications by reducing network contention. *Concurrency and Computation: Practice and Experience*.

Bhatelé, A., L. V. Kalé, and S. Kumar (2009). Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *23rd ACM International Conference on Supercomputing*.

Bhatele, A., S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale (2008, April). Overcoming scaling challenges in biomolecular simulations across multiple platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*.

Catalyurek, U., E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen (2007). Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE. Best Algorithms Paper Award.

Catlett, C. et al. (2007). TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications. In L. Grandinetti (Ed.), *HPC and Grids in Action*, Amsterdam. IOS Press.

Corradi, A., L. Leonardi, and F. Zambonelli (1999). Diffusive load balancing policies for dynamic applications. In *IEEE Concurrency*, pp. 7(1):22–31.

Devine, K. D., E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio (2005). New challenges in dynamic load balancing. *Appl. Numer. Math. 52*(2–3), 133–152.

Dinan, J., D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha (2009). Scalable work stealing. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, New York, NY, USA, pp. 1–11. ACM.

Frigo, M., C. E. Leiserson, and K. H. Randall (1998, June). The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, Volume 33 of *ACM Sigplan Notices*, Montreal, Quebec, Canada, pp. 212–223.

Furuichi, M., K. Taki, and N. Ichiyoshi (1990). A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Ha'c, A. and X. Jin (1987, April). Dynamic load balancing in distributed system using a decentralized algorithm. In *Proc. of 7-th Intl. Conf. on Distributed Computing Systems*.

Jetley, P., F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn (2008). Massively parallel cosmological simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*.

Kalé, L. and S. Krishnan (1993, September). CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke (Ed.), *Proceedings of OOPSLA'93*, pp. 91–108. ACM Press.

Kalé, L. V. (1988, August). Comparing the performance of two dynamic load distribution methods. In *Proceedings of the 1988 International Conference on Parallel Processing*, St. Charles, IL, pp. 8–11.

Kalé, L. V., M. Bhandarkar, and R. Brunner (1998). Load balancing in parallel molecular dynamics. In *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*, Volume 1457 of *Lecture Notes in Computer Science*, pp. 251–261.

Lawlor, O., S. Chakravorty, T. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. Kale (2006, September). Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers 22*(3-4), 215–235.

Lawlor, O. S. and L. V. Kalé (2003). Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience 15*, 371–393.

Mangala, S., T. Wilmarth, S. Chakravorty, N. Choudhury, L. V. Kale, and P. H. Geubelle (2007, December). Parallel adaptive simulations of dynamic fracture events. *Engineering with Computers 24*, 341–358.

Nieuwpoort, R. V. V., T. Kielmann, and H. E. Bal (2000). Satin: Efficient parallel divide-and-conquer in java. In *Lecture Notes in Computer Science*. Springer.

Phillips, J. C., G. Zheng, S. Kumar, and L. V. Kalé (2002, September). NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, MD, pp. 1–18.

Shu, W. W. and L. V. Kalé (1989, November). A dynamic load balancing strategy for the Chare Kernel system. In *Proceedings of Supercomputing '89*, pp. 389–398.

Sinha, A. and L. Kalé (1993, April). A load balancing strategy for prioritized execution of tasks. In *International Parallel Processing Symposium*, New Port Beach, CA., pp. 230–237.

Weirs, G., V. Dwarkadas, T. Plewa, C. Tomkins, and M. Marr-Lyon (2005). Validating the Flash code: vortex-dominated flows. In *Astrophysics and Space Science*, Volume 298, pp. 341–346. Springer Netherlands.

Willebeek-LeMair, M. H. and A. P. Reeves (1993, September). Strategies for dynamic load balancing on highly parallel computers. In *IEEE Transactions on Parallel and Distributed Systems*, Volume 4.

Zheng, G. (2005). *Achieving high performance on extremely large parallel machines: performance prediction and load balancing*. Ph. D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

Zoltan User's Guide. Zoltan Hierarchical Partitioning. http://www.cs.sandia.gov/Zoltan/ug_html/.