THE INTERNATIONAL JOURNAL of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
25(4) 371–385
© The Author(s) 2011
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342010394383
hpc.sagepub.com
SAGE

Periodic hierarchical load balancing for large supercomputers

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

Abstract

Large parallel machines with hundreds of thousands of processors are becoming more prevalent. 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 a relatively small amount of memory. Fully distributed load balancing algorithms, on the other hand, tend to take longer to arrive at good solutions. In this paper, we present an automatic dynamic hierarchical load balancing method that overcomes the scalability challenges of centralized schemes and longer running times of traditional distributed schemes. Our solution overcomes these issues 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 discuss techniques to deal with scalability challenges of load balancing at very large scale. We present performance data of the hierarchical load balancing method on up to 16,384 cores of Ranger (at the Texas Advanced Computing Center) and 65,536 cores of Intrepid (the 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 Intrepid.

Keywords

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

1 Introduction

Parallel machines with over a hundred thousand processors are already in use. It is speculated that by the end of this decade Exaflop/s computing systems that may have hundreds of millions of cores will emerge, providing unprecedented computing power to solve scientific and engineering problems. Modern parallel applications that 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 that often result in load imbalance and poor scaling. For such applications, load balancing techniques are crucial to achieving 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 a 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). Fully distributed load balancing also tends to take longer to arrive at good solutions.

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

Department of Computer Science, University of Illinois at Urbana-Champaign, USA.

Corresponding author:
Gengbin Zheng, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
Email: gzheng@illinois.edu

algorithmically efficient techniques because increasing machine and problem sizes lead to more complex load balance issues. Efforts involved in balancing the load can 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 presents 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 subtree. 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: 1) techniques to construct the tree using machine topology information to minimize communication and improve locality; 2) a method that explicitly controls and reduces the amount of load data aggregated to the higher levels of the tree; and 3) 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), that 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. the application's computational load and communication information) can be obtained automatically by runtime instrumentation. 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 for 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 the migration of newly created tasks only, such as those in the field of *task scheduling* problems, while other schemes also allow migration of tasks in progress. In all these schemes, detailed computation and communication information needs to be maintained continually to be used as metadata for making load balancing decisions. In this paper, we mainly consider periodic load balancing schemes, in which the load is 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 periodic load balancing schemes (Zheng, 2005). Periodic load balancing schemes are suitable for a class of iterative scientific applications such as NAMD (Phillips et al., 2002), Finite Element Method (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 are 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 et al., 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 principle 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)). In measurement-based load balancing strategies, the runtime automatically collects the computational load and communication patterns for each object 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:

$$\sigma = \frac{L_{\max}}{L_{\text{avg}}} - 1, \qquad (1)$$

where $L_{\max}$ (also referred to as the critical path) 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 $n \gg p$, 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 the number of tasks on each processor and each 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

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

Number of tasks	128K	256K	512K	1M
Memory usage (MB)	61	117	230	457

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 cores, the database alone requires around 450 MB of memory, which is non-trivial for machines with relatively little memory. This large amount of information clearly becomes a bottleneck when executing a realistic load balancing algorithm on a million-core system with even more task units.

The challenge of dealing with a large amount of load information motivated the work in this paper to design a hierarchical load balancing scheme that allows the scaling of load balancing strategies to a very large number of processors without sacrificing the quality of the load balance achieved. We 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.

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 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 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). 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

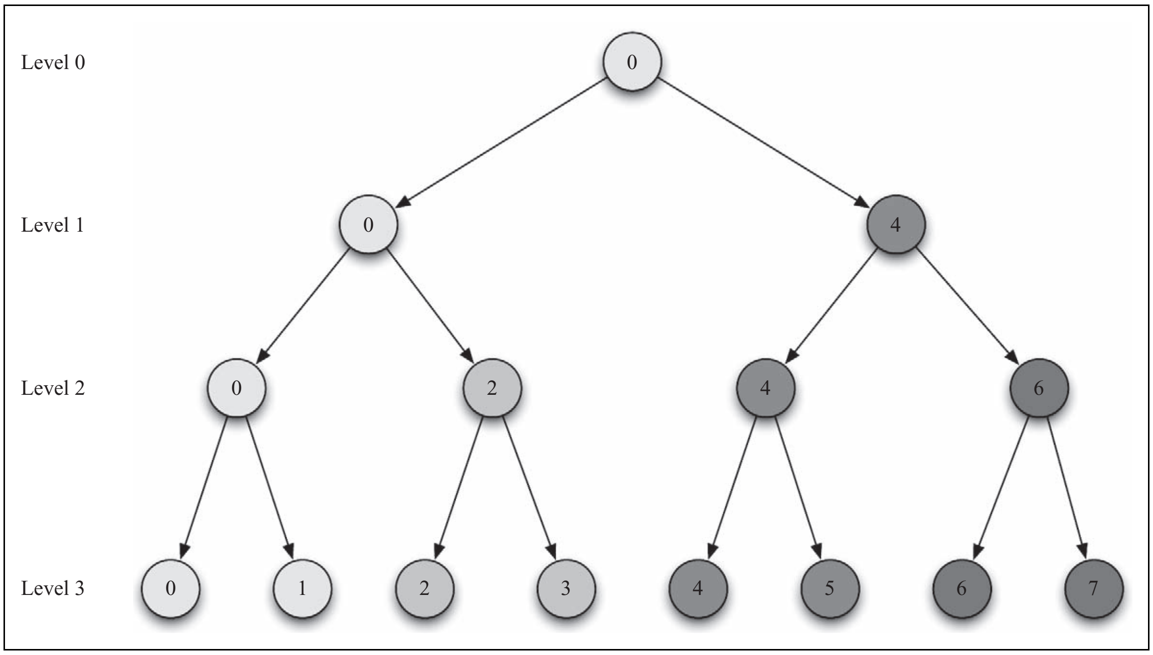

Figure 1. Hierarchical organization of an eight processor system

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. We achieve this balanced distribution 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 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 a proportionally bigger workload.

In the hierarchical scheme, as we move up the tree, the load balancing cost increases as the size of the aggregated load balancing metadata grows. Our design goals 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 advantages of exploiting the machine topology for tree algorithms is well known. For hierarchical load balancing, the advantages of topology-aware tree construction lie in minimizing network contention, since processors within a domain are topologically close to one another and the load balancing strategies tend to reassign the work within the domain as much as possible. The CHARM++ runtime can 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 information 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.

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

$$M_i = N_i * sizeof\ (ObjData) \ + \ C_i * sizeof\ (CommData), \quad (2)$$

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 that 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. We have explored three strategies to reduce memory usage:

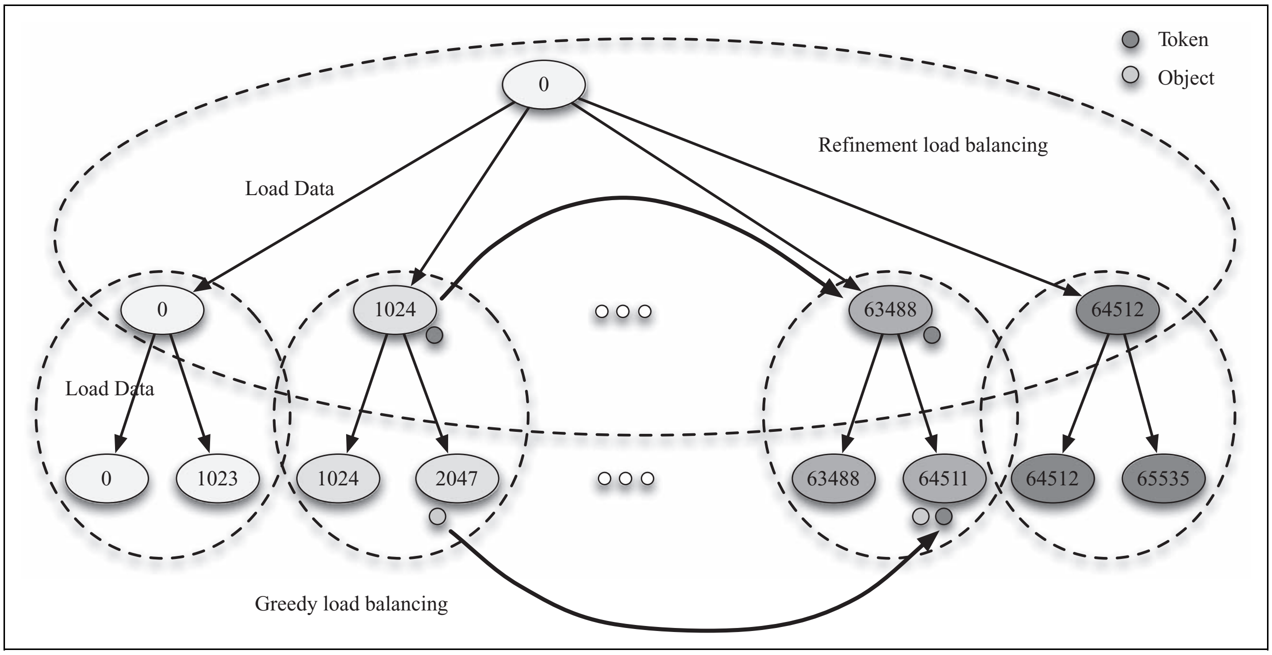

Figure 2. Hierarchical token-based load balancing scheme

- 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 the 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, for which the load is the sum of the load of the objects it represents. The coarsening method can be done by calling the METIS library (Karypis and Kumar, 1998) 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 the most heavily communicating objects can be grouped together into a container object. When a migration decision is made for 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 this scheme, the group leader of a domain does not make detailed migration decisions about individual objects. 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 (Willebeek-LeMair and Reeves, 1993) balance the domains from the 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 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 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 each of 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

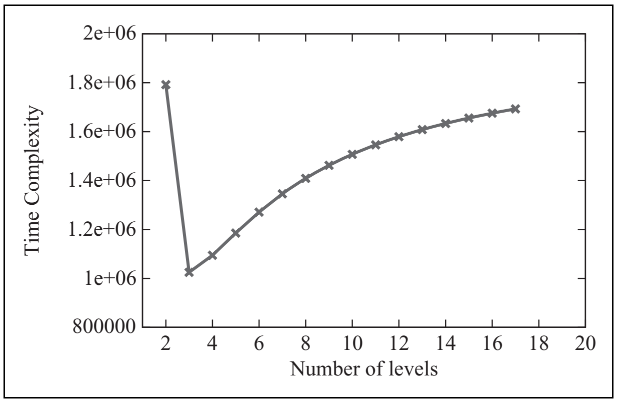

Figure 3. Plot showing the time complexity of load balancing as a function of the number of levels in the tree when using RefineLB

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 to which final destination processor the token has traveled. In order to match original processors with their tokens, a global collective operation is performed on the tree. This global collective operation is a fully distributed operation, and therefore it is reasonably efficient. 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 migrated only once after all the final migration decisions are made.

3.1 Complexity analysis of hierarchical load balancing

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 that we commonly use.

Load balancing time: For illustration, we assume that the branching factor of the tree (G) is the same across all levels, and a refinement load balancing algorithm (RefineLB) is applied at each level of the tree. RefineLB strives to reduce the cost of migration by moving only a few objects from overloaded processors to underloaded ones so that the load of each processor comes close to the average. RefineLB works by examining every object on an overloaded processor and finding the best choice of underloaded processor to which the object can be migrated. 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 to move the object.

When the memory usage reaches a threshold at a 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 the average load among sub-domains, build an initial max heap of processor loads for overloaded sub-domains, and compute the total amount of load for each overloaded sub-domain that migrates to an underloaded sub-domain; $\mathcal{O}(N_i)$ is the time on each sub-domain 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 the 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 at 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.

As an example, Figure 3 shows a plot of the time complexity for load balancing an application that has 1,000,000 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 centralized load balancing, 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

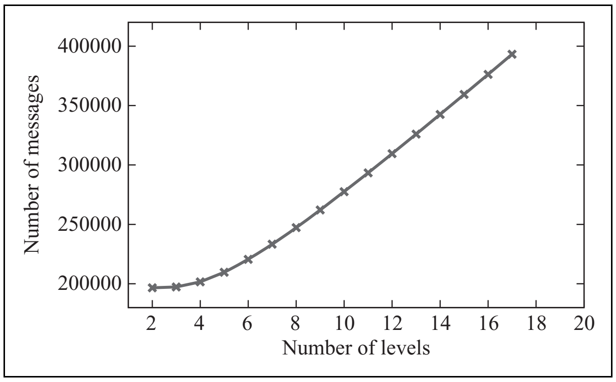

Figure 4. Plot showing the dependence of number of messages on the number of levels in the tree

is to maximize the branching factor at the lowest level until 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 generally performs well.

Communication overhead: We analyze the communication overhead in terms of the total number of messages generated by the hierarchical load balancing scheme. For simplicity, network contention due to several processors interacting with a single group leader is not considered. Network contention is only important as a contribution to overhead when the volume of data being communicated becomes a significant fraction of the available network bandwidth, which can be controlled by the branching factor of the tree. Further, the messages due to migrating object data are not counted, because the migration pattern depends both on the initial load balance and the type of load balancing algorithms used.

At a 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, 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 process 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. Again, the 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 with 65,536 processors, Figure 4 shows the number of messages for varying numbers of levels of the hierarchical tree (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 with a small number of levels is beneficial in reducing communication overhead.

4 Performance results

One question that remains to be answered is whether these benefits, 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 global information is not used. In addition, we demonstrate the dramatic improvements obtained in both time and memory requirements from using the hierarchical load balancing strategy.

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 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 numbers of objects to each processor. This placement scheme creates a very challenging load imbalance scenario that has the most overloaded processors at one end and the least overloaded processors at the other end.

4.1.1 Evaluation on Ranger. Our first test environment is a Sun Constellation Linux Cluster called Ranger, installed at the Texas Advanced Computing Center. Ranger is comprised of 3936 16-way SMP compute nodes providing a

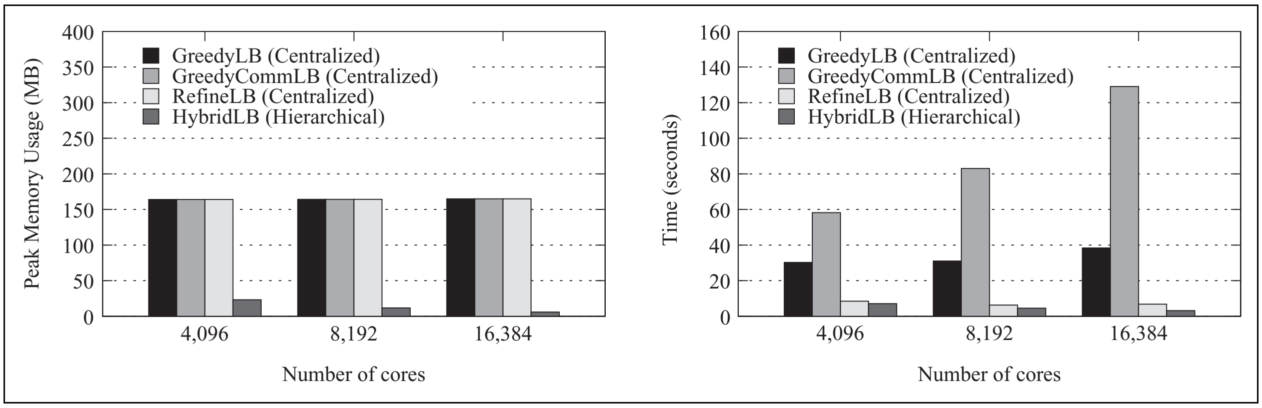

Figure 5. Comparison of the memory usage (for load metadata) and load balancing time for centralized and hierarchical strategies (for lb_test on Ranger)

total of 62,976 compute cores. Ranger nodes are connected using the 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 follows: groups of 512 cores 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 4096- and 8192-core tests. We use greedy load balancing strategies at the lowest level and a refinement-based load balancing strategy (with data shrinking) at the highest level.

Three different centralized strategies are used for comparison. The first one is a simple scheme called *GreedyLB*, that always picks the heaviest unassigned object and assigns it to the currently least loaded processor. For N objects and $N>>P$, this is an $\mathcal{O}(NlogN)$ algorithm. The second one is called *GreedyCommLB*. It is similar to GreedyLB in its choice of processors, however it is much more expensive in that it takes communication into account. When making an 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 strategy is called *RefineLB*. Unlike the previous two load balancing strategies, which make load balancing decisions from scratch, RefineLB improves the load balance by incrementally adjusting the existing object distribution. Refinement is used with an overload threshold. For example, with a threshold of 1.03, all processors with a load greater than 1.03 times the average load (i.e. with 3% overload) are considered overloaded, and objects are migrated from such 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.

We first measured the memory usage due to the collection of the load metadata in a hierarchical load balancing scheme (*HybridLB*), and compared it with the centralized load balancing strategies. The memory usage measured is only for the load metadata, not including the temporary data structures created by each centralized load balancing strategy, therefore the memory usage is the same for all centralized strategies. In these tests, the lb_test program creates a total of 1,048,576 objects running on varying numbers of cores up to 16,384, 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 cores increases, while the memory usage for centralized load balancing remains almost the same. This is due to the fact that, for the fixed problem size in these tests, when the number of processors 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 with 1,048,576 objects running on up to 16,384 cores. The results show that the hierarchical load balancer is very efficient compared with the greedy centralized strategies. 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. Furthermore, HybridLB exploits more parallelism by allowing execution of load balancing concurrently on independent load balancing domains. Compared to the greedy schemes, RefineLB is much more efficient, because RefineLB only migrates a fraction of the objects, thus reducing the data migration time. This advantage will become clear when we look at the total number of objects migrated by each load balancing strategy. As shown in the Table 2, RefineLB migrates significantly fewer objects, while greedy load balancing strategies migrate almost all objects. HybridLB uses a greedy strategy at the lowest level

Table 2. The number of objects migrated by different load balancing strategies for the lb_test benchmark with a total of 1,048,576 objects (on Ranger)

Number of cores	GreedyLB	GreedyCommLB	RefineLB	HybridLB
4096	1,048,321	1,048,310	65,994	1,046,721
8192	1,048,447	1,048,455	64,777	1,046,662
16,384	1,048,568	1,048,507	60,097	1,046,715

of the tree and hence migrates almost all objects, similar to the centralized greedy schemes.

The breakdown of the time spent in load balancing in the lb_test benchmark in Figure 6 illustrates the cause of the reduction in load balancing time. The left plot shows the timings obtained on Ranger. The time has been separated into three different phases – time for data collection, execution time for the load balancing strategy (strategy time), and time taken for object migration. HybridLB reduces time for all three phases. The most significant reduction happens in the time spent on the strategy, where the time for the centralized cases increases as the number of processors increase. The time in the data migration phase is also considerably reduced by using the hierarchical strategy. In the greedy centralized strategies, large messages that contain load balancing decisions for around one million objects are generated and broadcast from the central processor to every processor, leading to a communication bottleneck on the central processor. The 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 constraints as other centralized strategies (as illustrated in Figure 5).

In order 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 7. The first bar in each cluster shows the time per step performance of lb_test without load balancing. The next three bars in each cluster represent the time per step after applying the centralized load balancers and the last bar in each cluster represents the time per step after applying the hierarchical strategy. We can see that the hierarchical scheme performs comparably to the centralized schemes, and all of them improve the performance of lb_test in comparison to the no load balancing case.

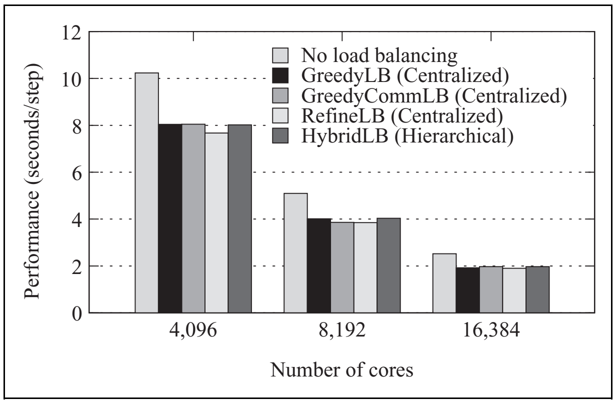

Figure 7. Performance (time per step) for lb_test, to compare the quality of load balance with different load balancing strategies (on Ranger)

4.1.2 Evaluation on Intrepid. To further study how the hierarchical load balancer performs at even larger scales, we ran similar experiments using lb_test 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 has 2 GB of memory, making it a difficult environment for using centralized load balancing strategies due to the low memory available per core.

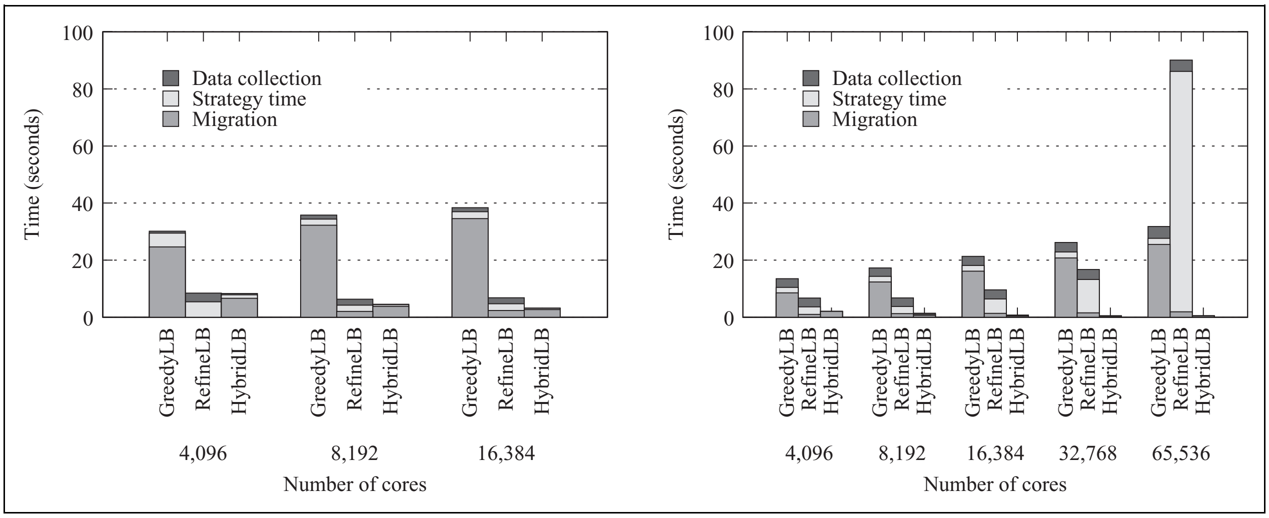

Figure 6. Comparison of time for load balancing using centralized load balancers (GreedyLB and RefineLB) versus the hierarchical load balancer (HybridLB) for lb_test (on Ranger and Blue Gene/P)

Table 3. The number of objects migrated by different load balancing strategies for the lb_test benchmark with a total of 524,288 objects (on Blue Gene/P)

Number of cores	GreedyLB	RefineLB	HybridLB
4096	524,166	30,437	523,326
8192	524,232	29,977	523,338
16,384	524,255	25,581	523,306
32,768	524,274	20,196	523,360
65,536	524,283	17,310	523,364

Indeed, lb_test with 1,048,576 objects runs out of memory during load balancing in the phase of load metadata collection. Thus, only 524,288 objects are created in lb_test in the following experiments. Figure 6 (right plot) illustrates the breakdown of the load balancing time using two centralized load balancers (GreedyLB and RefineLB), and the hierarchical load balancer (HybridLB). We see that the hierarchical load balancer is very efficient, especially when the number of cores increases. Specifically, HybridLB on 65,536 cores only takes 0.55 s. The RefineLB centralized load balancer executes faster than GreedyLB when the number of cores is less than 65, 536, largely due to a significantly smaller number of objects being migrated, as shown in Table 3. However, its load balancing decision making time (strategy time) keeps increasing as the number of cores increases, possibly due to the heuristic of overload threshold that leads to an expensive binary search for better load balance decisions.

4.1.3 Performance study of the number of tree levels. Section 3.1 explains that the number of levels of the hierarchical tree and the level at which load data reduction occurs and the load balancing algorithm switches, can significantly affect the performance of the hierarchical load balancer. In an experiment with the lb_test benchmark on 4096 cores of Blue Gene/P, we evaluated the variation in load balancing time with various numbers 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 trees used in these experiments are uniform, with the same branching factor at all levels. For example, for 4096 cores a four level tree is used that has a branching factor of 16.

In most of these experiments, the threshold that triggers data reduction and switching of load balancing strategies 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 three-level tree with $G = 64$, the data reduction occurs at level 1, even though the number of objects at that level is only 8192. Otherwise, at its parent level (level 0, the root of the tree), the number of objects collected from its 64 children would reach 524,288, which is beyond the threshold. The results of the HybridLB load balancing time are shown in Table 4. The "Shrink Level" in the third column is the level at which load balancing reaches the threshold of data reduction, and "No. of Objects" in the fourth column is the estimated number of objects in the load balancing sub-domain at that time. We can see that a three-level tree performs best for this particular test scenario. This result is in agreement with the analysis 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 which uses 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 that is invoked at start-up. It performs the initial load balancing and moves most of the objects around.
2. A **refinement** load balancer is called several times during 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 on which to place it. This process 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 elsewhere (Kalé et al., 1998; Bhatelé et al., 2009).

Table 4. Effect on the load balancing time of using different numbers of levels for the hierarchical trees in HybridLB (for the lb_test benchmark running on 4096 cores of Blue Gene/P)

No. of levels	Branching factor	Shrink Level	No. of Objects	LB Time (s)
3	64	1	8192	0.52
4	16	1	32,768	2.82
5	8	1	65,536	21.47
7	4	2	32,768	7.51
13	2	3	65,536	59.63

Table 5. Time (in seconds) for centralized load balancing in NAMD (on Blue Gene/P)

Number of cores	1024	2048	4096	8192	16,384
Comprehensive	5.12	4.87	6.16	25.09	96.84
Refinement	0.79	0.82	1.16	4.33	249.10

Traditionally, NAMD uses strategies that collect all load balancing statistics to a central location (typically processor 0) where all load balancing decisions are made. These strategies are 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 5 shows the time processor 0 takes to calculate load balancing solutions in the centralized case. As we scale from 1024 to 16,384 processors, the time for refinement load balancing increases by a factor of 315.

The main driver, from the point of view of the NAMD user, for deploying the hierarchical load balancing scheme has been to reduce the time taken for load balancing. However, with the use of NAMD for increasingly larger simulations, memory is also bound to become a bottleneck. 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 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.

Figure 8 (left plot) presents a comparison of the time spent in load balancing between the centralized and the hierarchical approaches for the comprehensive load balancers. The load balancing time in the centralized case does not increase necessarily with the increase in the number of 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 that the y-axis has a logarithmic scale). For instance, on 4096 cores, the centralized approach takes 6.16 s to balance the load. In contrast, the hierarchical load balancer takes only 0.69 s, which is a speedup of 8.9. Speedup increases with increasing core count. On 16,384 cores, the hierarchical load balancer is 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 strategy. The highest reduction in load balancing time occurs at 16,384 cores, where the time taken is reduced from 249.1 s to 2.27 s, giving a speedup of 110.

A breakdown of the time spent in load balancing into three different phases – time for data collection, execution time for the load balancing strategy within the sub-domains (strategy time) and time for sending migration decisions – indicates the source of the reduction. Table 6 shows a breakdown of the time spent in the comprehensive strategy for NAMD for the centralized and hierarchical algorithms.

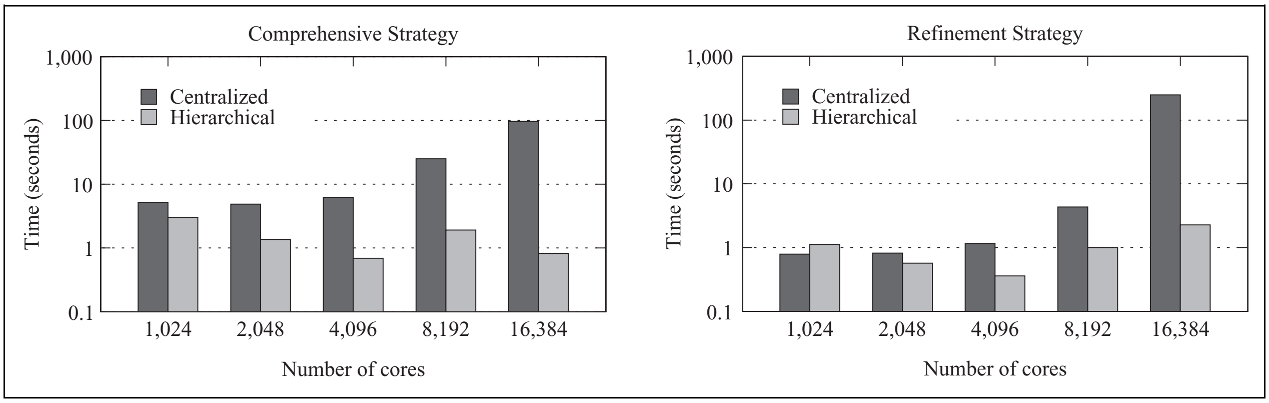

Figure 8. Comparison of time to solution for centralized and hierarchical load balancers for NAMD on Blue Gene/P (molecular system: STMV)

Table 6. Breakdown of load balancing time (in seconds) into three phases for the centralized and hierarchical load balancing strategy (for comprehensive strategy in NAMD running on Blue Gene/P)

	Data collection		Strategy time		Migration	
Number of cores	*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
16,384	1.25	0.15	84.44	0.46	11.15	0.22

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.

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 the no load balancing base line performance). On 4096 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 1024 to 16,384 cores. These results show 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.

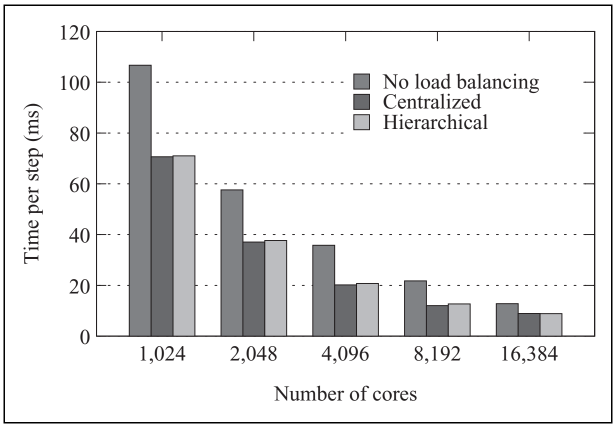

Figure 9. Comparison of NAMD 's performance on Blue Gene/P when using centralized versus hierarchical load balancers (molecular system: STMV)

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 (Ha'c and Jin, 1987; Kalé, 1988; Shu and Kalé, 1989; Sinha and Kalé, 1993; Willebeek-LeMair and Reeves, 1993; Corradi et al., 1999). 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 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 (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 that 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; 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 work 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 (Dinam et al., 2009).

This assumption avoids migration of 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++ (Kalé et al., 1998; Zheng, 2005) 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. Such load balancing schemes are suitable for a class of iterative scientific applications such as NAMD (Phillips et al., 2002), Finite Element Method (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. Periodic load balancing strategies for iterative applications are the main focus of this paper.

Hierarchical periodic load balancing strategies have also been studied in the context of iterative applications. The 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 the 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 that scale, the execution time of the load balancing strategy and the 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 that combines the advantages of centralized and fully distributed schemes. The proposed load balancing scheme adopts a periodic 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 discuss several techniques to deal with scalability challenges of load balancing that are found at very large scale in the context of production applications.

We presented results for a synthetic benchmark on up to 65,536 cores and a scientific application, NAMD, on up to 16,384 cores. Using hierarchical schemes, we were able to reduce considerably the memory requirements and the runtime 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 will deploy the hierarchical load balancers in other applications. We exploited the machine topology only for the tree construction. We also plan to extend our hierarchical load balancing strategies to be topology-aware, such that they map the communication graph on the processor topology to minimize network contention.

Funding

This work was supported in part by the US National Science Foundation (NSF Grant OCI-0725070 for Blue Waters), the US Department of Energy (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 the US Department Of Energy (contract DE-AC02-06CH11357). Runs on Ranger were supported by the US National Science Foundation (TeraGrid (Catlett et al., 2007) allocation grant ASC050040N).

Conflict of interest

None declared.

References

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

Baldeschwieler JE, Blumofe RD, and Brewer EA (1996) Atlas: an infrastructure for global computing. In: EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop. New York: ACM, 165–72.

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

Bhatelé A, Kalé LV, and Kumar S (2009) Dynamic topology aware load balancing algorithms for molecular dynamics applications. In: *23rd ACM International Conference on Supercomputing*, 110–6.

Bhatelé A, Kumar S, Mei C, Phillips JC, Zheng G, and Kalé LV (2008) Overcoming scaling challenges in biomolecular simulations across multiple platforms. In: *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 1–12.

Catalyurek U, Boman E, Devine K, Bozdag D, Heaphy R, and Riesen L (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, 1–11.

Catlett C et al. (2007) TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of

Applications. In: Grandinetti L (ed.) *HPC and Grids in Action*. Amsterdam: IOS Press. 16: 225–49.

Corradi A, Leonardi L, and Zambonelli F (1999) Diffusive load balancing policies for dynamic applications. *IEEE Concurrency* 7(1):22–31. URL http://polaris.ing.unimo.it/Zambonelli/PDF/Concurrency.pdf

Devine KD, Boman EG, Heaphy RT, Hendrickson BA, Teresco JD, Faik J, et al. (2005) New challenges in dynamic load balancing. *Appl. Numer. Math.* 52(2–3): 133–52.

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

Frigo M, Leiserson CE, and Randall KH (1998) 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, 212–23.

Furuichi M, Taki K, and Ichiyoshi N (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*, 50–9.

Ha'c A and Jin X (1987) Dynamic load balancing in distributed system using a decentralized algorithm. In: *Proc. of 7-th Intl. Conf. on Distributed Computing Systems*, 170–7.

Jetley P, Gioachin F, Mendes C, Kale LV, and Quinn TR (2008) Massively parallel cosmological simulations with ChaNGa. In: *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 1–12.

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

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

Kalé LV, Bhandarkar M, and Brunner R (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*, 251–61.

Karypis G and Kumar V (1998) Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing* 48: 96–129. URL http://www-users.cs.umn.edu/~karypis/publications/Papers/PDF/mlevel_kparallel.pdf

Lawlor O, Chakravorty S, Wilmarth T, Choudhury N, Dooley I, Zheng G, et al. (2006) Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers* 22(3–4): 215–35.

Lawlor OS and Kalé LV (2003) Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience* 15: 371–93.

Mangala S, Wilmarth T, Chakravorty S, Choudhury N, Kale LV, and Geubelle PH (2007) Parallel adaptive simulations of dynamic fracture events. *Engineering with Computers* 24: 341–58.

Nieuwpoort RVV, Kielmann T, and Bal HE (2000) Satin: Efficient parallel divide-and-conquer in java. In: *Lecture Notes in Computer Science*. Springer. 1900/2000: 690–9.

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

Shu WW and Kalé LV (1989) A dynamic load balancing strategy for the Chare Kernel system. In: *Proceedings of Supercomputing '89*, 389–98.

Sinha A and Kalé L (1993) A load balancing strategy for prioritized execution of tasks. In: *International Parallel Processing Symposium*, New Port Beach, CA. 230–7.

Weirs G, Dwarkadas V, Plewa T, Tomkins C, and Marr-Lyon M (2005) Validating the Flash code: vortex-dominated flows. In: *Astrophysics and Space Science*, 298: 341–6.

Willebeek-LeMair MH and Reeves AP (1993) Strategies for dynamic load balancing on highly parallel computers. In: *IEEE Transactions on Parallel and Distributed Systems*, 4: 979–93.

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

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

Author's Biographies

Gengbin Zheng received the BS (1995) and MS (1998) degrees in Computer Science from Peking University, Beijing, China. His Master's thesis was on the High Performance Fortran compiler. He received his PhD 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 Laboratory at the University of Illinois, working with Professor 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 at SC 2002.

Abhinav Bhatelé received a BTech degree in Computer Science and Engineering from IIT Kanpur (India) in May 2005 and MS and PhD 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 University of Illinois, working with Professors William D Gropp and 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 BEng degree in Computing Engineering (2001) and a MSc degree in Computer Science (2007) from the Costa Rica Institute of Technology. Currently, he is a third year PhD student working with Professor 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 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) bio-molecular 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.

LV Kalé received the BTech degree in Electronics Engineering from Benares Hindu University, Varanasi, India in 1977, and a ME degree in Computer Science from the Indian Institute of Science in Bangalore, India, in 1979. He received a PhD in computer science from the 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.