Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers

Gengbin Zheng, Esteban Meneses, Abhinav Bhatelé and Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: {gzheng, emenese2, bhatele, kale}@illinois.edu

Abstract— **Large parallel machines with hundreds of thousands of processors are being built. Recent studies have shown that 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 aggressive 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 (at TACC) 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.**

I. Introduction

Parallel machines with hundreds of thousands of 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 [1], [2]. 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 [3], [4].

Several state-of-the-art scientific and engineering applications such as NAMD [1] and ChaNGa [5] adopt a centralized load balancing strategy, where load balancing decisions are made on one specific processor, based on the load data instrumented 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 [4], [6]. 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.

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. Load balancing can actually 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 several 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 subtree and 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 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++ [7], [8], 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 pattern) 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 II discusses existing work for scalable load balancing strategies. Section III 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 IV. Performance results using the hierarchical load balancers for a synthetic benchmark and for a production scientific application, NAMD are provided in Section V. Section VI concludes the paper with some future plans.

II. RELATED WORK

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. It 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.

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 [9], [10], [11], [12], [13], [14]. 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 [15].

Randomized work stealing is yet another distributed dynamic load balancing technique, which is used in some runtime systems such as Cilk [16]. Recent work in [17] extends this work using the PGAS programming model and RDMA to scale work stealing to 8192 processors for three benchmarks. ATLAS [18] and Satin [19] use hierarchical work stealing for clusters and grids, both supporting JAVA programming on distributed systems.

Several other hierarchical or multi-level load balancing strategies [15], [20] have been proposed and studied. Ahmad and Ghafoor [15] 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 srategies 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. This assumption is to avoid the need to migrate a partially executed task when load balancing is needed [17]. 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++ [8], [21] and Zoltan [22]. 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 "phase-based" fashion, that is, load balancing happens periodically when needed. These load balancing schemes are suitable for a class of iterative scientific applications such as NAMD [1], FEM [23] 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 [24] 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.

In this paper, we focus on a hierarchical load balancing scheme that is designed to scale on existing petascale machines. We aim at scientific and engineering applications that exhibit persistent computational and communication patterns, even in dynamically evolving simulations, such as those in molecular dynamics applications or iterative solvers. The new hierarchical load balancing scheme adopts an object migration approach when migrating encapsulated data and tasks between processors. It takes advantage of the existing work in CHARM++ for supporting object migration.

III. OBJECT-BASED LOAD BALANCING 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 that has been implemented based on such an object model [8].

Many load balancing strategies in CHARM++ are based on a heuristic known as the *principle of persistence*. 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 [4], ChaNGa [5], Fractography3D [25]), up to thousands of processors. In measurement-based load balancing strategies, the runtime automatically instruments 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 occurs. 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. 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. In practice, the load imbalance threshold can be chosen based on a heuristic. Let us assume that the load of the most overloaded processor *after* load balancing is L'_{max}. The gain from load balancing is equal to $L_{max} - L'_{max}$. Load balancing should be done if the gain from load balancing ($L_{max} - L'_{max}$) is greater than the estimated cost of load balancing (C_{lb}). That is:

$$L_{max} - L'_{max} > C_{lb} \tag{2}$$

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.

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 [8]. 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 I. 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.

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

TABLE I
MEMORY USAGE (IN MB) ON THE CENTRAL PROCESSOR FOR CENTRALIZED LOAD BALANCING WHEN RUNNING ON $65,536$ CORES (SIMULATION DATA)

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 phase-based load balancing systems to solve scalability challenges in load balancing.

IV. 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 Fig. 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.

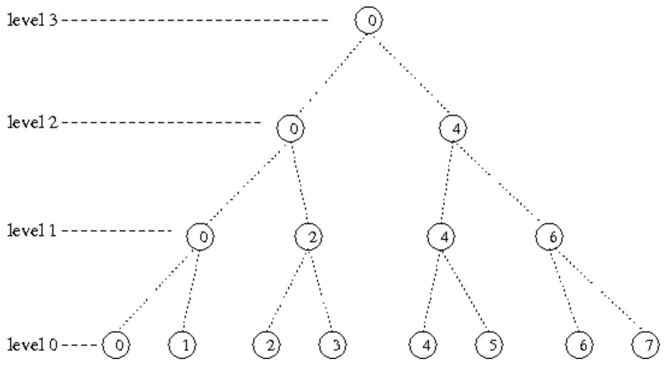

Fig. 1. Hierarchical organization of an eight processor system

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 (root 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 to each individual sub-domain by its domain leaders.

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.

Our design goal for the hierarchical load balancing scheme is to 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 [26]. 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 one 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 data 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 ($\bar{M}$). The amount of memory needed for storing the load database can be calculated as:

$$m = N * \mathit{sizeof}(\mathit{ObjData}) + C * \mathit{sizeof}(\mathit{CommData}) \quad (3)$$

where N is the total number of objects, C is the number of communication records, *ObjData* is the data structure which records the load data per object and *CommData* is the data structure which records the communication data.

The runtime uses a memory usage estimator to monitor the load data memory usage. When $m > \bar{M}$, load data needs to be shrunk to fit in memory. Two 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.
- 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 [14]) 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.

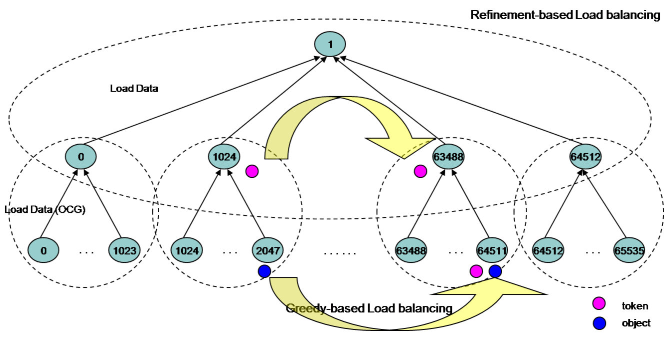

Fig. 2. Hierarchical token-based load balancing scheme

Our load balancing scheme overcomes this challenge by using a top down token-based approach. As shown in Fig. 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 among the 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 child group leaders at the lower level domains are updated, reflecting the load balancing decisions made on the higher level – new load database entries are created for the incoming objects from other domains, 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 the current level. This process repeats until load balancing reaches the lowest level.

At this point, all load balancing decisions have been made when load balancing from the highest to the lowest level. 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.

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. For illustration, we use a simplified example, where we assume that the branching factor of the tree 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. For simplicity, we assume that there are only a small number of overloaded processors in each load balancing domain. The complexity of this algorithm is $\mathcal{O}(GlogG + \frac{N_i}{G} logG)$, where N_i is the number of migratable objects 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 $\mathcal{O}(\frac{N_i}{G} logG)$ is the time it takes to examine objects on overloaded processors and update the max heap with the decision of where the object moves.

At a certain level i from the top, the number of objects at that level, $N_i = N/G^i$. Thus, 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):

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

where the number of levels, $L = log_G P + 1$.

Compared to the computational cost, the communication cost of the hierarchical load balancing algorithm is minimal, as it involves only two rounds of tree reductions and broadcasts. Therefore, this formula is useful to determine the optimal number of levels of the tree for this particular case. As an example, Fig. 3 shows a plot of computational overhead for load balancing an application that has 1 million parallel objects on $65,536$ processors for varying number of levels of the tree. We can see that, with a three-level tree, the execution time of

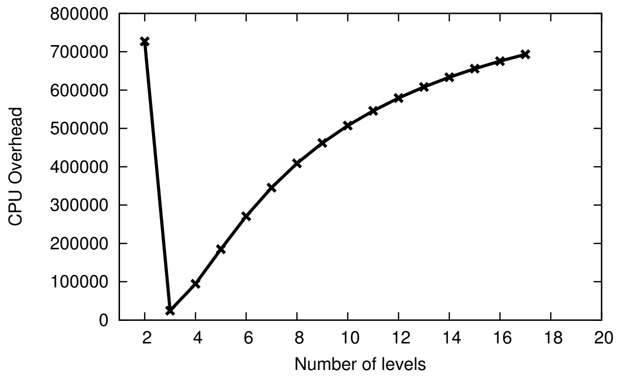

Fig. 3. Plot showing the dependence of load balancing time on the number of levels in the tree when using RefineLB

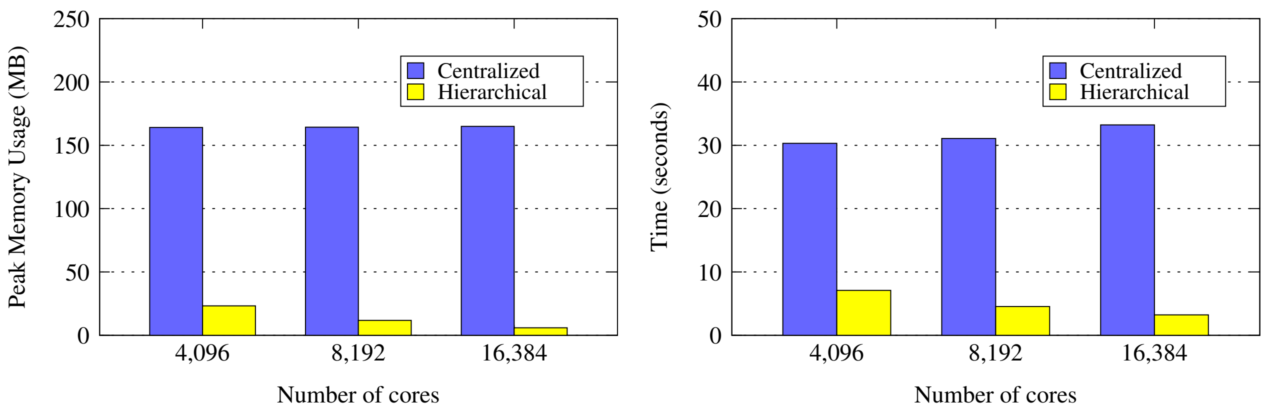

Fig. 4. Comparison of peak memory usage and time for load balancing using CentralLB versus HybridLB for lb_test (on Ranger)

the algorithm is the lowest.

In practice, however, there are many factors that affect the choice of number of levels as mentioned earlier. This general problem of determining the optimal tree is beyond the scope of this paper. However, 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 a three-level tree with high branching factor at the lowest level is generally good.

V. 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 compare the hierarchical and centralized load balancing schemes 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.

A. 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.

Our 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 a greedy-based load balancing algorithm at the first level and a refinement-based load balancing algorithm (with data shrinking) at the second level.

We first measured the peak memory usage for the load database across all the group leaders for various experimental configurations and compared with the centralized load balancing scheme. In these tests, the lb_test program creates a total of 1 million objects on varying number of processors. The results are shown in Fig. 4 (left plot). Comparing with the memory usage in a centralized load balancing strategy, 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

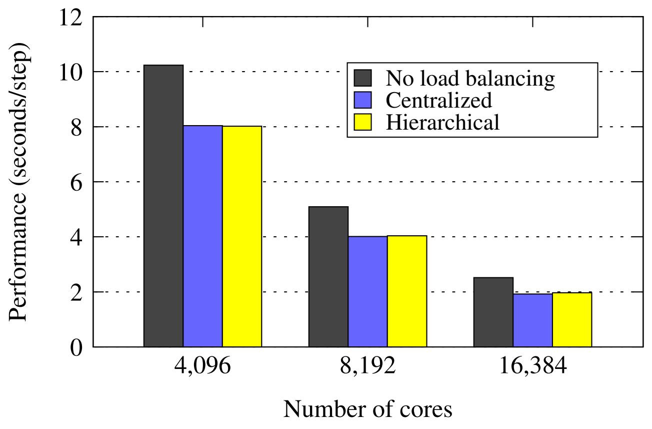

Fig. 5. Performance (time per step) for lb_test to compare the quality of load balance with different load balancing strategies (on Ranger)

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.

Fig. 4 (right plot) compares the load balancing time spent in HybridLB and a greedy heuristic based centralized load balancing strategy (CentralLB) for the same sized problem with 1 million objects running on up to $16,384$ cores. The results show that the hierarchical load balancer is very efficient compared with the centralized load balancer. 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.

We also compared the performance of lb_test when using the hierarchical and centralized load balancers and the results are shown in Fig. 5. The first bar in each cluster shows the time per step of 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 HybridLB performs comparably to CentralLB.

B. NAMD

NAMD [1], [4] 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 [6], [21].

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. Fig. 6 shows the time processor 0 takes to calculate load balancing solutions in the centralized case. As we scale from 256 to $2,048$ processors, the time for refinement load balancing increases by 275 times!

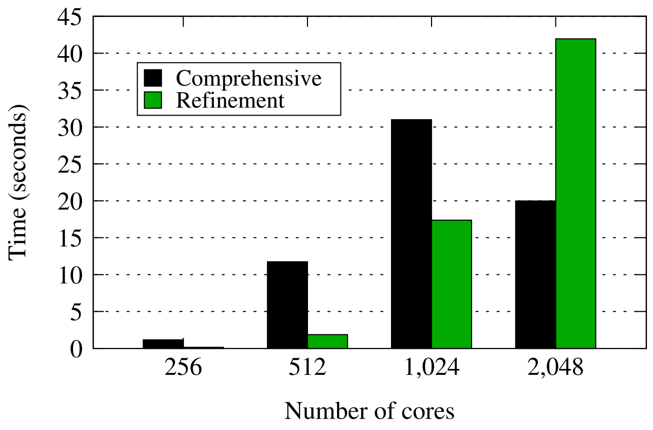

Fig. 6. Increase in time for load balancing of NAMD as we scale to large number of cores on Blue Gene/P (for comprehensive and refinement strategies)

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

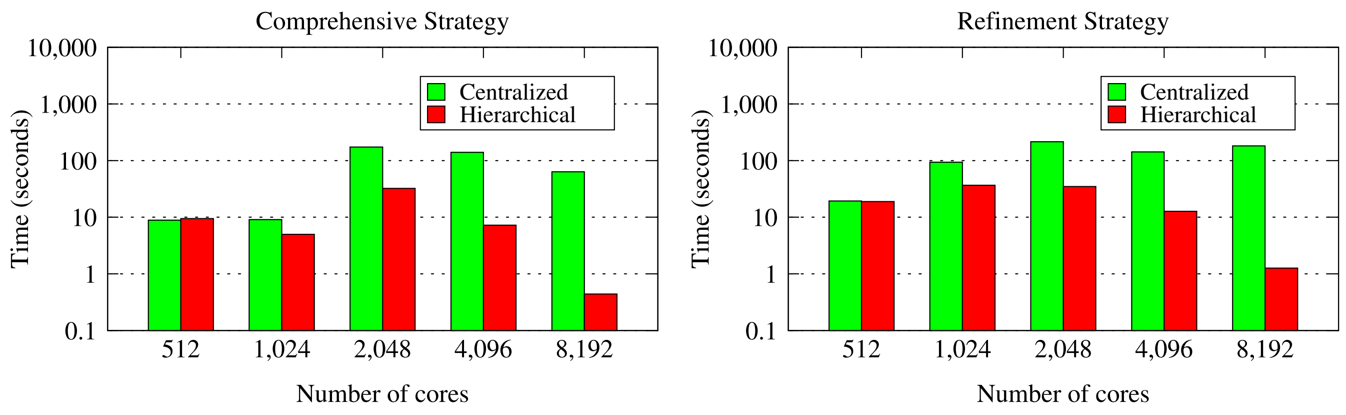

Fig. 7. Comparison of time to solution for centralized and hierarchical load balancers for NAMD on Blue Gene/P

load balancing is done according to the semi-centralized load balancing scheme described in Section IV. 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 the standard molecular system used by biophysicists for benchmarking, Apolipoprotein-A1 (ApoA1) with PME enabled and a 2AwayXYZ decomposition. This decomposition leads to approximately $113,000$ objects 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.

Fig. 7 (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 $2,048$ cores, the centralized approach takes 173.71 seconds to balance the load. In contrast, the hierarchical load balancer takes 32.23 seconds, which is a speedup of 5.4. Larger the core count, higher is the speedup. On $8,192$ cores, the hierarchical load balancers are faster by 145 times! The results for the refinement load balancing phase are similar. Fig. 7 (right plot) compares the centralized and hierarchical balancers for the refinement load balancer. The highest absolute reduction in load balancing time occurs at $2,048$ cores, where the time taken reduces from 215.19 to 34.74 seconds, giving a speedup of 6.2. On $8,192$ cores, we obtain the maximum speedup of 145.

We further demonstrate that the hierarchical load balancing strategy performs no worse than the centralized strategy in balancing the load in NAMD. Fig. 8 shows the time per step

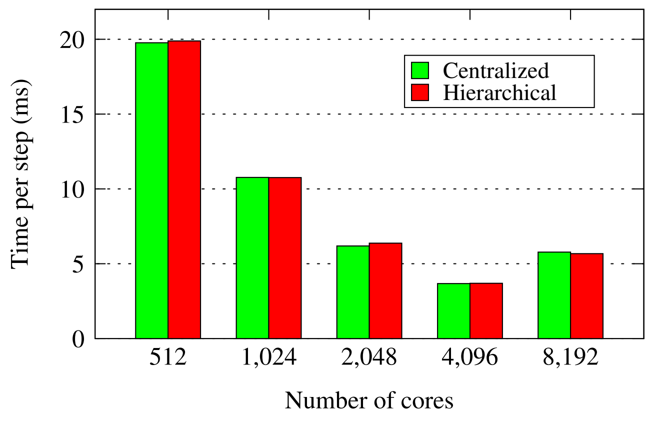

Fig. 8. Comparison of NAMD's performance when using centralized versus hierarchical load balancers (on Blue Gene/P)

performance of NAMD when the centralized and hierarchical load balancers are used. On 512 cores, the simulation time is 19.77 ms per step for the centralized load balancing and 19.89 ms per step for the hierarchical load balancing case, showing negligible slowdown. Very similar results can be observed from $1,024$ to $8,192$ 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. We note that the performance at $8,192$ cores is worse than that on $4,096$ cores for both strategies. However, this is orthogonal to the issue of load balancing and its discussion is out of the scope of this paper.

VI. 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 $16,384$ and $8,192$ 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.

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 [27] allocation grant ASC050040N supported by NSF.

REFERENCES

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

[2] G. Weirs, V. Dwarkadas, T. Plewa, C. Tomkins, and M. Marr-Lyon, "Validating the Flash code: vortex-dominated flows," in *Astrophysics and Space Science*. Springer Netherlands, 2005, vol. 298, pp. 341–346.

[3] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, "New challenges in dynamic load balancing," *Appl. Numer. Math.*, vol. 52, no. 2–3, pp. 133–152, 2005.

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

[5] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[6] A. Bhatelé, L. V. Kalé, and S. Kumar, "Dynamic topology aware load balancing algorithms for molecular dynamics applications," in *23rd ACM International Conference on Supercomputing*, 2009.

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

[8] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

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

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

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

[12] W. W. Shu and L. V. Kalé, "A dynamic load balancing strategy for the Chare Kernel system," in *Proceedings of Supercomputing '89*, November 1989, pp. 389–398.

[13] A. Sinha and L. Kalé, "A load balancing strategy for prioritized execution of tasks," in *International Parallel Processing Symposium*, New Port Beach, CA., April 1993, pp. 230–237.

[14] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, September 1993.

[15] I. Ahmad and A. Ghafoor, "A semi distributed task allocation strategy for large hypercube supercomputers," in *Proceedings of the 1990 conference on Supercomputing*. New York, NY: IEEE Computer Society Press, 1990, pp. 898–907.

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

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

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

[19] R. V. V. Nieuwpoort, T. Kielmann, and H. E. Bal, "Satin: Efficient parallel divide-and-conquer in java," in *Lecture Notes in Computer Science*. Springer, 2000.

[20] M. Furuichi, K. Taki, and N. Ichiyoshi, "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*, 1990.

[21] L. V. Kalé, M. Bhandarkar, and R. Brunner, "Load balancing in parallel molecular dynamics," in *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*, ser. Lecture Notes in Computer Science, vol. 1457, 1998, pp. 251–261.

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

[23] O. Lawlor, S. Chakravorty, T. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. Kale, "Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications," *Engineering with Computers*, vol. 22, no. 3-4, pp. 215–235, September 2006.

[24] Zoltan User's Guide, "Zoltan Hierarchical Partitioning," http://www.cs.sandia.gov/zoltan/ug_html/ug_alg_hier.html.

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

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

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