A Distributed Dynamic Load Balancer for Iterative Applications

Harshitha Menon, Laxmikant Kalé
Department of Computer Science, University of Illinois at Urbana-Champaign
{gplkrsh2,kale}@illinois.edu

ABSTRACT

For many applications, computation load varies over time. Such applications require dynamic load balancing to improve performance. Centralized load balancing schemes, which perform the load balancing decisions at a central location, are not scalable. In contrast, fully distributed strategies are scalable but typically do not produce a balanced work distribution as they tend to consider only local information.

This paper describes a fully distributed algorithm for load balancing that uses partial information about the global state of the system to perform load balancing. This algorithm, referred to as *GrapevineLB*, consists of two stages: global information propagation using a lightweight algorithm inspired by *epidemic* [21] algorithms, and work unit transfer using a randomized algorithm. We provide analysis of the algorithm along with detailed simulation and performance comparison with other load balancing strategies. We demonstrate the effectiveness of *GrapevineLB* for adaptive mesh refinement and molecular dynamics on up to 131,072 cores of BlueGene/Q.

General Terms

Algorithms, Performance

Keywords

load balancing, distributed load balancer, epidemic algorithm

1. INTRODUCTION

Load imbalance is an insidious factor that can reduce the performance of a parallel application significantly. For some applications, such as basic stencil codes for structured grids, the load is easy to predict and does not vary dynamically. However, for a significant class of applications, load represented by pieces of computations varies over time, and may be harder to predict. This is becoming increasingly prevalent with the emergence of sophisticated applications. For example, atoms moving in a molecular dynamics simulation will lead to (almost) no imbalance when they are distributed statically to processors. But, they create imbalance *when* spatial partitioning of atoms is performed for more sophisticated and efficient force evaluation algorithms. The presence of moisture and clouds in weather simulations, elements turning from elastic to plastic in structural dynamics simulations and dynamic adaptive mesh refinements are all examples of sophisticated applications which have a strong tendency for load imbalance.

All the examples above are of "iterative" applications: the program executes series of time-steps, or iterations, leading to convergence of some error metric. Consecutive iterations have relatively similar patterns of communication and computation. There is another class of applications, such as combinatorial search, that involves dynamic creation of work and therefore has a tendency for imbalance. This class of applications has distinct characteristics and load balancing needs, and has been addressed by much past work such as work-stealing [25, 3, 32]. This paper does *not* focus on such applications but instead on the iterative applications, which are predominant in science and engineering. We also do not focus on approaches that partition the fine-grained application data. For example, in unstructured mesh based applications, the entire mesh (consisting of billions of elements) may be partitioned by a library such as METIS [13]. This approach is expensive and not widely applicable; instead we focus on scenarios where the application work has already been partitioned into coarser work units.

For iterative applications, the basic scheme we follow is: The application is assumed to consist of a large number of migratable units (for example, these could be chunks of meshes in adaptive mesh refinement application). The application pauses after every so many iterations, and the load balancer decides whether to migrate some of these units to restore balance. Load balancing is expensive in these scenarios and is performed infrequently or whenever significant imbalance is detected. Note that a reactive strategy such as work-stealing, which is triggered when a processor is idle, is almost infeasible (e.g. Communication to existing tasks must be redirected on the fly). Schemes for arriving at a distributed consensus on *when* and how often to balance load [28], and how to avoid the pause (carrying out load balancing asynchronously with the application) have been addressed in the past. In this paper we focus on a synchronous load balancer. Since scientific applications have synchronizations at various points, this can be used without extra overhead of synchronization.

SC '13, November 17-21, 2013, Denver, Colorado, USA

http://dx.doi.org/10.1145/2503210.2503284

Various strategies have been proposed to address the load balancing problem. Many applications employ centralized load balancing strategies, where load information is collected on to a single processor, and their decision algorithm is run sequentially. Such strategies have been shown to be effective for a few hundred to thousand processors, because the total number of work units is relatively small (on the order of ten to hundred per processor). However, they present a clear performance bottleneck beyond a few thousand processors, and may become infeasible due to the memory capacity bottleneck on a single processor.

An alternative to centralized strategies are distributed strategies that use local information, e.g. diffusion based [9]. In a distributed strategy, each processor makes autonomous decisions based on its local view of the system. The local view typically consists of the load of its neighboring processors. Such strategies are scalable, but tend to yield poor load balance due to the limited local information [1].

Hierarchical strategies [35, 23, 1] overcome some of the aforementioned disadvantages. They create subgroups of processors, and collect information at the root of each subgroup. Higher levels in the hierarchy only receive aggregate information and deliver decisions in aggregate terms. Although effective in reducing memory costs, and ensuring good balance, these strategies may suffer from excessive data collection at the lowest level of the hierarchy and work being done at multiple levels.

We propose a fully distributed strategy, *GrapevineLB*, that has been designed to overcome the drawback of other distributed strategies by obtaining a partial representation of the global state of the system and basing the load balancing decisions on this. We describe a light weight information propagation algorithm based on *epidemic* algorithm [21](also known as the gossip protocol [10]) to propagate the load information about the underloaded processors in the system to the overloaded processors. This spreads the information in the same fashion as gossip spreads through the grapevine in a society. Based on this information, *GrapevineLB* makes probabilistic transfer of work units to obtain good load distribution. The proposed algorithm is scalable and can be tuned to optimize for either cost or performance.

The primary contributions of this paper are:

- *GrapevineLB*, a fully distributed load balancing algorithm that attains a load balancing quality comparable to the centralized strategies while incurring significantly less overhead.
- Analysis of propagation algorithm used by *GrapevineLB* which leads us to an interesting observation that good load balance can be achieved with significantly less information about underloaded processors in the system.
- Detailed evaluations that experimentally demonstrate the scalability and quality of *GrapevineLB* using simulation.
- Demonstration of its effectiveness in comparison to several other load balancing strategies for adaptive mesh refinement and molecular dynamics on up to 131,072 cores of BlueGene/Q.

2. BACKGROUND

Load characteristics in dynamic applications can change over time. Therefore, such applications require periodic load balancing to maintain good system utilization. To enable load balancing, a popular approach is overdecomposition. The application writer exposes parallelism by overdecomposing the computation into tasks or objects. The problem is decomposed into communicating objects and the run-time system can assign these objects to processors and perform rebalancing.

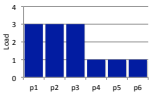
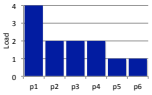
Processor Load	L_{avg}	L_{max}	σ	$\mathcal{I}$
	2	3	$\sqrt{6}$	0.5
	2	4	$\sqrt{6}$	1

Table 1: Choice of load imbalance metric

The load balancing problem in our context can be summarized as: given a distributed collection of work units, each with a load estimate, decide which work units should be moved to which processors, to reduce the load imbalance. The load balancer needs information about the loads presented by each work-unit. This can be based on a model (simple examples being associating a fixed amount of work with each grid point, or particle). But for many applications, another metric turns out to be more accurate. For these applications, a heuristic called *principle of persistence* [20] holds which allows us to use recent instrumented history as a guide to predicting load in near-future iterations. The load balancing strategy we describe can be used with either model-based or persistence-based load predictions. In persistence-based load balancer, the statistics about the load of each task on a processor is collected at that processor. The database containing the task information is used by the load balancers to produce a new mapping. The run-time system then migrates the tasks based on this mapping.

It is important to choose the right metric to quantify load imbalance in the system. Using standard deviation to measure load imbalance may seem like an appropriate metric, but consider the two scenarios shown in Table 1. In both the cases, the average load of the system is 2. If we consider standard deviation, σ, to be a measure of imbalance, then we find that in case 1 and case 2 we obtain the same σ of $\sqrt{6}$ whereas the utilization and the total application times differ. A better indicator of load imbalance in the system is the ratio of maximum load to average load. More formally, load imbalance ($\mathcal{I}$) can be measured using

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

In case 1, $\mathcal{I}$ is 0.5 and in case 2, $\mathcal{I}$ is 1. We use this metric of load imbalance as one of the evaluation criteria to measure the performance of the load balancing strategy. Notice that this criteria is dominated by the load of the single processor — viz. the most overloaded processor — because of the *max* operator. This is correct, since the execution time is determined by the worst-loaded processor and others must wait for it to complete its step.

Apart from how well the load balancer can balance load,

it is important to incur less overhead due to load balancing. Otherwise, the benefit of load balancing is lost in the overhead. Therefore, we evaluate quality of load balance, cost of the load balancing strategy and the total application time.

3. RELATED WORK

Load balancing has been studied extensively in the literature. For applications with regular load, *static* load balancing can be performed where load balance is achieved by carefully mapping the data onto processors. Numerous algorithms have been developed for statically partitioning a computational mesh [12, 5, 6, 16]. These model the computation as a graph and use graph partitioning algorithms to divide the graph among processors. Graph and hypergraph partitioning techniques have been used to map tasks on to processors to balance load while considering the locality. They are generally used as a pre-processing step and tend to be expensive. Our algorithm is employed where the application work has already been partitioned and used to balance the computation load imbalance that arises as the application progresses. Our algorithm also takes into consideration the existing mapping and moves tasks only if a processor is overloaded.

For irregular applications, work stealing is employed in task scheduling and is part of runtime systems such as Cilk [3]. Work stealing is traditionally used for task parallelism of the kind seen in combinatorial search or divide-and-conquer applications, where tasks are being generated continuously. A recent work by Dinan et al. [11] scales work stealing to 8192 processors using the PGAS programming model and RDMA. In work that followed, a hierarchical technique described as retentive work stealing was employed to scale work-stealing to over 150K cores by exploiting the principle of persistence to iteratively refine the load balance of task-based applications [23]. CHAOS [31] provides an inspector-executor approach to load balancing for irregular applications. Here the data and the associated computation balance is evaluated at runtime before the start of the first iteration to rebalance. The proposed strategy is more focused towards iterative computational science applications, where computational tasks tend to be persistent.

Dynamic load balancing algorithms for iterative applications can be broadly classified as centralized, distributed and hierarchical. Centralized strategies [7, 29] tend to yield good load balance but exhibit poor scalability. Alternatively, several distributed algorithms have been proposed in which processors autonomously make load balancing decisions based on localized workload information. Popular nearest neighbor algorithms are dimension-exchange [34] and the diffusion methods. Dimension-exchange method is performed in an iterative fashion and is described in terms of a hypercube architecture. A processor performs load balancing with its neighbor in each dimension of the hypercube. Diffusion based load balancing algorithms were first proposed by Cybenko [9] and independently by Boillat [4]. This algorithm suffers from slow convergence to the balanced state. Hu and Blake [17] proposed a non-local method to determine the flow which is minimal in the l_2-norm but requires global communication. The token distribution problem was studied by Peleg and Upfal [30] where the load is considered to be a token. Several diffusive load balancing policies, like direct neighborhood, average neighborhood, have been proposed in [8, 14, 19]. In [33], a sender-initiated model is compared with receiver-initiated in an asychronous setting. It also compares Gradient Method [24], Hierarchical Method and DEM (Dimension exchange). The diffusion based load balancers are incremental and scale well with number of processors. But, they can be invoked only to improve load balance rather than obtaining global balance. If global balance is required, multiple iterations might be required to converge [15]. To overcome the disadvantages of centralized and distributed, hierarchical [35, 23, 1] strategies have been proposed. It is another type of scheme which provides good performance and scaling.

In our proposed algorithm, global information is spread using a variant of gossip protocol [10]. Probabilistic gossip-based protocols have been used as robust and scalable methods for information dissemination. Demers et al. use a gossip-based protocol to resolve inconsistencies among the Clearinghouse database servers [10]. Birman et al. [2] employ gossip-based scheme for bi-modal multicast which they show to be reliable and scalable. Apart from these, gossip-based protocols have been adapted to implement failure detection, garbage collection, aggregate computation etc.

4. GRAPEVINE LOAD BALANCER

Our distributed load balancing strategy, referred to as *GrapevineLB*, can be conceptually thought of as having two stages. *1) Propagation*: Construction of the local representation of the global state at each processor. *2) Transfer*: Load distribution based on the local representation.

At the beginning of the load balancing step, the average load is calculated in parallel using an efficient tree based all-reduce. This is followed by the propagation stage, where the information about the underloaded processors in the system is spread to the overloaded processors. Only the processor ID and load of the underloaded processors is propagated. An underloaded processor starts the propagation by selecting other processors randomly to send information. The receiving processors further spread the information in a similar manner.

Once the overloaded processors have received the information about the underloaded processors, they autonomously make decisions about the transfer of the work units. Since various processors do not coordinate at this stage, the transfer has to happen such that the probability that an underloaded processor becomes overloaded is low. We propose a randomized algorithm that meets this goal. We elaborate further upon the above two stages in the following sections.

4.1 Information propagation

To propagate the information about the underloaded processors in the system, *GrapevineLB* follows a protocol which is inspired by the epidemic algorithm [21] (also known as the gossip protocol [10]). In our case, the goal is to spread the information about the underloaded processors such that every overloaded processor receives this information with high probability. An underloaded processor starts the 'infection' by sending its information to a randomly chosen subset of processors. The size of the subset is called *fanout*, f. An infected processor further spreads the infection by forwarding all the information it has to another set of randomly selected f processors. Here, each processor makes an independent random selection of peers to send the information.

We show that the number of rounds required for all processors to receive the information with high probability is

Algorithm 1 Informed selection at each processor $P_i \in P$

Input:
f - Fanout
L_{avg} - Average load of the system.
k - Target number of rounds
L_i - Load of this processor

$S \longleftarrow \varnothing$ ▷ Set of underloaded processors
$L \longleftarrow \varnothing$ ▷ Load of underloaded processors
if $(L_i < L_{avg})$ **then**
  $S \longleftarrow P_i$; $L \longleftarrow L_i$
  Randomly sample $\{P^1, \ldots, P^f\} \in P$
  Send (S, L) to $\{P^1, \ldots, P^f\}$
end if
for $(round = 2 \rightarrow k)$ **do**
  if (received msg in previous round) **then**
    $R \longleftarrow P \setminus S$ ▷ Informed selection
    Randomly sample $\{P^1, \ldots, P^f\} \in R$
    Send (S, L) to $\{P^1, \ldots, P^f\}$
  end if
end for

when (S_{new}, L_{new}) is received ▷ New message
$S \longleftarrow S \cup S_{new}$; $L \longleftarrow L \cup L_{new}$ ▷ Merge information

$O(\log_f n)$, where n is the number of processors. We propose two randomized strategies of peer selection as described below. Note that although we discuss various strategies in terms of rounds for the sake of clarity, there is no explicit synchronization for rounds in our implementation.

Naive Selection: In this selection strategy, each underloaded processor independently initiates the propagation by sending its information to a randomly selected set of f peers. A receiving processor updates its knowledge with the new information. It then randomly selects f processors, out of the total of n processors, and forwards its current knowledge. This selection may include other underloaded processors.

Informed Selection: This strategy is similar to the *Naive* strategy except that the selection of peers to send the information is done incorporating the current knowledge. Since the current knowledge includes a partial list of underloaded processors, the selection process is biased to not include these processors. This helps propagate information to the overloaded processors in fewer number of rounds. This strategy is depicted in Algorithm 1.

4.2 Probabilistic transfer of load

In our distributed scheme the decision making for transfer of load is decentralized. Every processor needs to make these decisions in isolation given the information from the propagation stage. We propose two randomized schemes to transfer load.

Naive Transfer: The simplest strategy to transfer load is to select processors uniformly at random from the list of underloaded processors. An overloaded processor transfers load until its load is below a specified threshold. The value of threshold indicates how much of an imbalance is acceptable. As one would expect, this random selection results in overloading processors whose load is closer to the average. This is illustrated in Figure 1 and described in detail in Section 7.1.

Informed Transfer: A more informed transfer can be made by randomly selecting underloaded processors based on their initial load. We achieve this by assigning to each processor a probability that is inversely proportional to its load in the following manner:

Algorithm 2 Informed transfer at each processor $P_i \in P$

Input:
O - Set of objects in this processor
S - Set of underloaded processors
T - Threshold to transfer
L_i - Load of this processor
L_{avg} - Average load of the system

Compute $p_j \quad \forall\, P_j \in S$ ▷ Using eq. 2
Compute $F_j = \sum_{k<j} p_k$ ▷ Using eq. 3
while $(L_i > (T \times L_{avg}))$ **do**
  Select object $O_i \in O$
  Randomly sample $X \in S$ using F ▷ Using eq. 4
  if $(L_X + load(O_i) < L_{avg})$ **then**
    $L_X = L_X + load(O_i)$
    $L_i = L_i - load(O_i)$
    $O \longleftarrow O \setminus O_i$
  end if
end while

$$p_i = \frac{1}{Z} \times \left(1 - \frac{L_i}{L_{avg}}\right) \tag{2a}$$

$$Z = \sum_{1}^{N} \left(1 - \frac{L_i}{L_{avg}}\right) \tag{2b}$$

Here p_i is the probability assigned to the ith processor, L_i its load, L_{avg} is the average load of the system and Z is a normalization constant. To select processors according to this distribution we use the inversion method for generating samples from a probability distribution. More formally if $p(x)$ is a probability density function, then the cumulative distribution function $F(y)$ is defined as:

$$F(y) = p(x < y) = \int_{-\infty}^{y} p(x)dx \tag{3}$$

Given a uniformly distributed random sample $r_s \in [0, 1]$, a sample from the target distribution can be computed by:

$$y_s = F^{-1}(r_s) \tag{4}$$

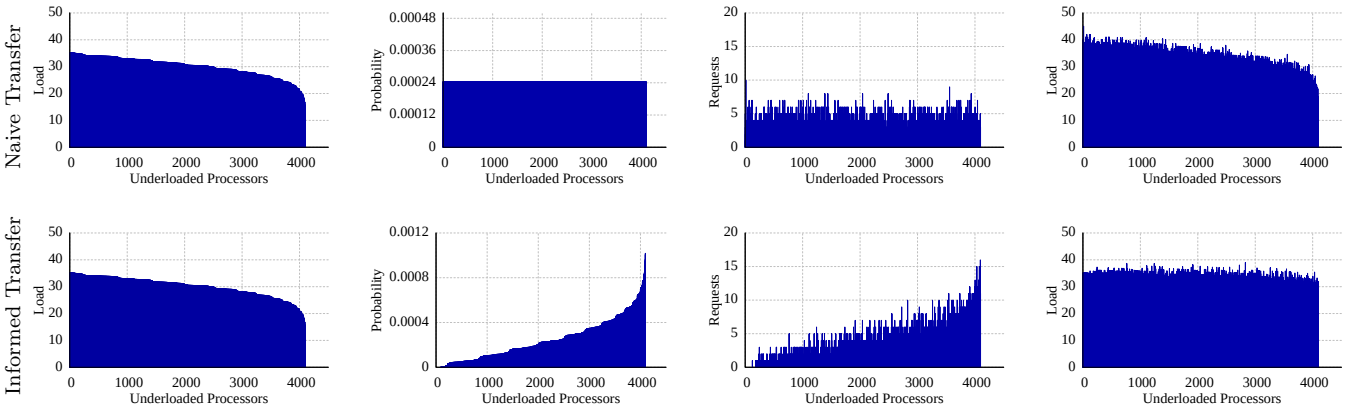
Using the above, we randomly select the processors according to p_i for transferring load. This is summarized in Algorithm 2. Figure 1 illustrates the results.

4.3 Partial Propagation

An interesting question to ask is what happens if the overloaded processors have incomplete information. This may happen with high probability if the propagation stage is terminated earlier than $\log n$ rounds. We hypothesize that to obtain good load balance, information about all the underloaded processors is not necessary. An overloaded processor can have a partial set of underloaded processors and still achieve good balance. We empirically confirm our hypothesis by a set of experiments in Section 7.1.

4.4 Grapevine+

Even though the scheme where every processor makes autonomous decision for randomized transfer of work is less



(a) Load of Underloaded Processors (b) Probability Distribution (c) Transfers Received (d) Load After LB

Figure 1: (a) Initial load of the underloaded processors, (b) Probabilities assigned to each of the processors, (c) Work units transferred to each underloaded processor, (d) Final load of the underloaded processors after transfer.

likely to cause underloaded processors to become overloaded, this may still happen. To guarantee that none of the underloaded processors get overloaded after the transfer, we propose an improvement over the original *GrapevineLB* strategy. In the improved scheme, referred to as *Grapevine+LB*, we employ a negative-acknowledgement based mechanism to allow an underloaded processor to reject a transfer of work unit. For every potential work unit transfer, the sender initially sends a message to the receiver which contains details about the load of the work unit. The receiver, depending on the current load, chooses to either accept or reject. If accepting the work unit makes the receiver overloaded, then it rejects with a Nack (negative-acknowledgement). A sender on receiving a Nack will try to find another processor from the list of underloaded processors. This trial is carried out for a limited number of times after which the processor gives up. This scheme will ensure that no underloaded processor gets overloaded. Although this requires exchanging additional messages, the cost is not significant as the communication is overlapped with the decision making process.

5. ANALYSIS OF THE ALGORITHM

This section presents an analysis of the information propagation algorithm. We consider a system of n processors and, for simplicity, assume that the processors communicate in synchronous rounds with a fanout f. Note that in practice the communication is asynchronous (Section 6). We show that the expected number of rounds required to propagate information to all the processors in the system with high probability is $O(\log_f n)$. Although we analyze the case of single sender, the results are same for multiple senders since they communicate concurrently and independently.

In round $r = 1$, one processor initiates the information propagation by sending out f messages. In all successive rounds, each processor that received a message in the previous round sends out f messages. We are interested in the probability, p_s, that any processor P_i received the message by the end of round s. We can compute it by $p_s = 1 - q_s$, where q_s is the probability that the processor P_i did *not* receive any message by the end of round s.

Probability that a processor P_i did not receive a message sent by some other processor is $(1-\frac{1}{n-1}) \approx (1-\frac{1}{n}), \because n \gg 1$. Further, the number of messages sent out in round r is f^r, since the fan-out is f.

Clearly,

$$q_1 = \left(1 - \frac{1}{n}\right)^f \tag{5}$$

Therefore, the probability that P_i did not receive any message in any of the $r \in \{1, \ldots, s\}$ rounds is

$$\begin{aligned} q_s &= \prod_{r=1}^{s} \left(1 - \frac{1}{n}\right)^{f^r} = \left(1 - \frac{1}{n}\right)^{(f+f^2+f^3+\cdots+f^s)} \\ &= \left(1 - \frac{1}{n}\right)^{f\frac{f^s-1}{f-1}} \\ &\approx \left(1 - \frac{1}{n}\right)^{\gamma f^s}, \quad \text{Where} \quad \gamma = \frac{f}{f-1} \end{aligned}$$

Here $f^s - 1 \approx f^s, \because f^s \gg 1$. Taking log of both sides

$$\log q_s \approx \gamma f^s \log\left(1 - \frac{1}{n}\right) \approx \frac{-\gamma f^s}{n}$$

$$\therefore \quad q_s \approx \exp\left(\frac{-\gamma f^s}{n}\right)$$

Approximating by the first two terms of the Taylor expansion of e^x

$$q_s \approx 1 - \frac{\gamma f^s}{n}$$

Since we want to ensure that the probability that a processor P_i did not receive any message in s rounds is very low i.e. $q_s \approx 0$, substituting this in the above yields

$$\begin{aligned} \gamma f^s &\approx n \quad \text{As } q_s \to 0 \\ \therefore \quad s \log f &\approx \log n - \log \gamma \\ s &\approx \log_f n - \log_f\left(\frac{f}{f-1}\right) \\ &= O(\log_f n) \end{aligned}$$

□

Our simulation results shown in figure 3 concur with the above analysis. It is evident that increasing the fan-out results in significant reduction of the number of rounds required to propagate the information.

6. IMPLEMENTATION

We provide an implementation of the proposed algorithm as a load balancing strategy in CHARM++. CHARM++ is a parallel programming model which has message driven parallel objects, *chares*, which can be migrated from one processor to another. *Chares* are basic units of parallel computation in CHARM++, which are mapped onto processors initially using a default mapping or any custom mapping. *--withing* CHARM++ load balancing framework supports instrumenting load information of work units from the recent past and using it as a guideline for the near future. The key advantage of this approach is that it is application independent, and has been shown to be effective for a large class of applications, such as NAMD [27] and ChaNGa [18].

Charm++ has a user-friendly interface for obtaining dynamic measurements about *chares*. The load balancers, which are pluggable modules in CHARM++, can use this instrumented load information to make the load balancing decisions. Based on these decisions Charm++ RTS migrates the *chares*. Since the CHARM++ RTS stores information about *chares* and processors in a distributed database, it is compatible with *GrapevineLB*'s implementation requirements.

Although we have described the *GrapevineLB* algorithm in terms of rounds, an implementation using barriers to enforce the rounds will incur considerable overhead. Therefore, we take an asynchronous approach for our implementation. But such an approach poses the challenge of limiting the number of messages in the system. We overcome this by using a TTL (Time To Live) based mechanism which limits the circulation of information forever. It is implemented as a counter embedded in the messages being propagated. The first message initiated by an underloaded processor is initialized with the TTL of desired number of rounds before being sent. A receiving processor incorporates the information and sends out a new message with updated information and decremented TTL. A message with $TTL = 0$ is not forwarded and is considered expired. The key challenge that remains is to detect quiescence, i.e. when all the messages have expired. To this end, we use a distributed termination detection algorithm [26].

7. EVALUATION

We evaluate various stages of *GrapevineLB* with simulations using real data and compare it with alternative strategies using real world applications.

7.1 Evaluation using Simulation

We first present results of simulation of *GrapevineLB* strategy using real data on a single processor. This simulation allows us to demonstrate the effect of various choices made in different stages of the algorithm. For the simulations, the system model is a set of 8192 processors, initialized with load from a real run of an adaptive mesh refinement application with same number of cores on IBM BG/Q. This application was decomposed into 253,405 work units. Figure 2 shows the load distribution for this application when the load balancer was invoked. The average load of the system is 35, the maximum load is 66, therefore $\mathcal{I}$, metric for imbalance from Equation 1, is 0.88. Note that the value of $I \approx 0$ indicates perfect balance in the system. Among the 8192 processors, 4095 are overloaded and 4097 are either underloaded or have their

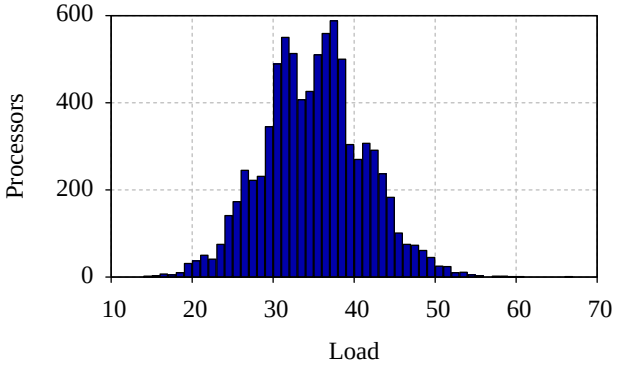


Figure 2: Load distribution for a run of AMR used in simulation. Counts of processors for various loads are depicted.

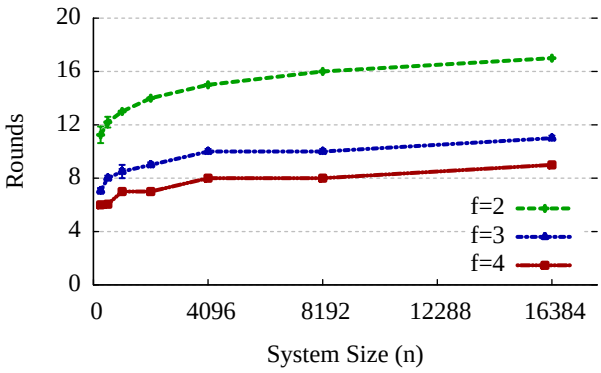


Figure 3: Expected number of rounds taken to spread information from one source to 99% of the overloaded processors for different system sizes and fanouts.

load close to average. We perform a step-by-step analysis of all the stages of the proposed algorithm based on this system model. It is to be noted that we have simulated synchronous rounds. The experiments were run 50 times and we report the results as mean along with its standard deviation.

Number of Rounds and Fanout: Figure 3 illustrates the dependence of expected number of rounds required to spread information on the system size. Here we consider only one source initiating the propagation and report when 99% of processors have received the information. As the system size (n) increases, the expected number of rounds increase logarithmically, $O(\log n)$, for a fixed fanout. This is in accordance with our analysis in Section 5. Note that the number of rounds decreases with increase in the fanout used for the information propagation. A system size of $16K$, fanout of 2, requires 17 rounds to propagate information to 99% processors whereas, fanout of 4, takes 8 rounds.

Naive vs Informed Propagation: Figure 4 compares the expected number of rounds taken to propagate information using *Naive* and *Informed* propagation schemes. Although, the expected number of rounds for both the schemes is on the order of $O(\log n)$, the *Informed* scheme takes one less round to propagate the information. This directly results in the reduction of the number of messages as most of the messages are sent in the later rounds. We can also choose to vary the fanout adaptively to reduce

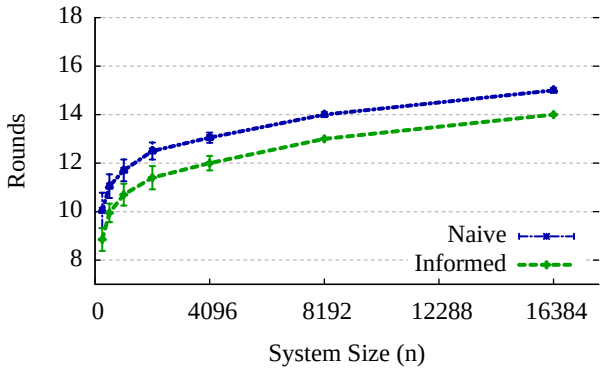


Figure 4: Expected number of rounds taken to spread information from one source to 99% of the overloaded processors using Naive and Informed schemes for different system sizes. Here $f = 2$ and 50% of the system size is underloaded

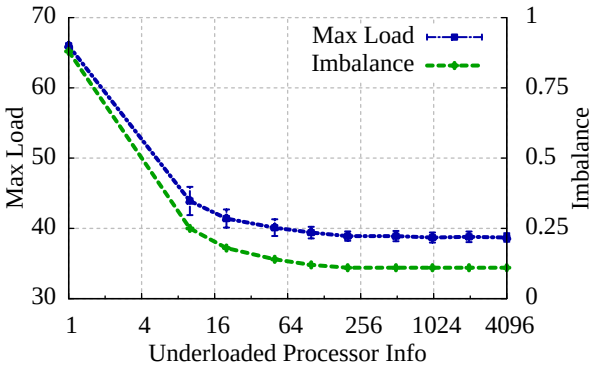


Figure 5: Evaluation of load balancer with partial information. Max load(left) and Imbalance(right) decrease as more information about underloaded processors is available. It is evident that complete information is not necessary to obtain good performance.

the number of rounds required, while not increasing the number of messages significantly. Instead of having a fixed fanout, we increase the fanout in the later stages. This is based on the observation that messages in the initial stages do not carry a lot of information. We evaluated this for a system of 4096 processors where 50% were overloaded. Information propagation without the adaptive variation requires 13 rounds with a total of 79600 messages. While an adaptive fanout strategy, where we use a fanout of 2 initially and increase the fanout to 3 beyond 5 rounds and further increase to 4 beyond 7 rounds, helps reduce the number of rounds to 10 with a total of 86400 messages.

Naive vs Informed Transfer: We compare the performance of the two randomized strategies for transfer given in Section 4. Figure 1 shows the *Naive* scheme for the transfer of load where an underloaded processor is selected uniformly at random. Here we also show the probability distribution of the underloaded processors for the *Informed* transfer strategy using the equation 2 and the transfer of load which follows this distribution which are shown in Figure 1. It shows the initial load distribution of the underloaded processors, probability assigned to each processor (uniform distribution), number of transfers based on the probability distribution and the final load of the underloaded processors. It can be seen that the maximum load of the initially underloaded processors is 44 while the average is 35. Comparison with Figure 1 clearly shows that the final distribution of load is much more reasonable. Further, the maximum load of the underloaded processors is 38 while the system average is 35.

Evaluation of a Pathological Case: We evaluate the behavior of the proposed algorithm under the pathological case where just one out of 8192 processors is significantly overloaded ($\mathcal{I}$ is 6.18). Analysis in Section 5 shows that q_s decreases rapidly with rounds for a particular source. Since all underloaded processors will initiate information propagation, this scenario shouldn't be any worse in expectation. We experimentally verify this and find that for a fanout value of 2 and using the *Naive* strategy for information propagation, it takes a maximum of 14 rounds to propagate the information which is similar to the case where many processors are overloaded. Once the information is available at the overloaded processor, it randomly transfers the work units, reducing the $\mathcal{I}$ from 6.18 to 0.001.

Evaluation of Quality of Load Balancing: To answer the question posed in the earlier section as to what happens if the overloaded processors have incomplete information, we simulate this scenario by providing information about only a partial subset of underloaded processors to the overloaded processors. The subset of underloaded processors for each processor is selected uniformly at random from the set of underloaded processors and the probabilistic transfer of load is then carried out based on this partial information. The quality is evaluated based on the metric $\mathcal{I}$ given by equation 1. Figure 5 shows the expected maximum load of the system along with standard deviation, σ and the value of $\mathcal{I}$ metric. It can be seen that on one hand having less information, $10 - 50$ underloaded processors, yields considerable improvement of load balance although not the optimal possible. On the other hand, having complete information is also not necessary to obtain good load balance. Therefore, this gives us an opportunity to trade-off between the overhead incurred and load balance achieved.

Evaluation of Information Propagation: Based on the earlier experiment, it is evident that complete information about the underloaded processors is not required for good load balance. Therefore, we evaluate the expected number of rounds taken to propagate partial information about the underloaded processors to all the overloaded processors. Figure 6 shows the percentage of overloaded processors that received the information as the rounds progress for a fanout of 2. The x-axis is the number of rounds and the y-axis is the percentage of overloaded processors who received the information. We plot the number of rounds required to propagate information about $200, 400, 2048, 4097$ underloaded processors to all the overloaded processors. In the case of propagating information about at least 200 underloaded processors in the system, 100% of the overloaded processors receive information about at least 200 underloaded processors in 12 rounds and 99.8% received in 9 rounds. It took 18 rounds to propagate information about all the underloaded processors in the system to all the overloaded processors. This clearly indicates that if we require only partial information, the total number of rounds can be reduced which will result in reduction of the load balancing cost.

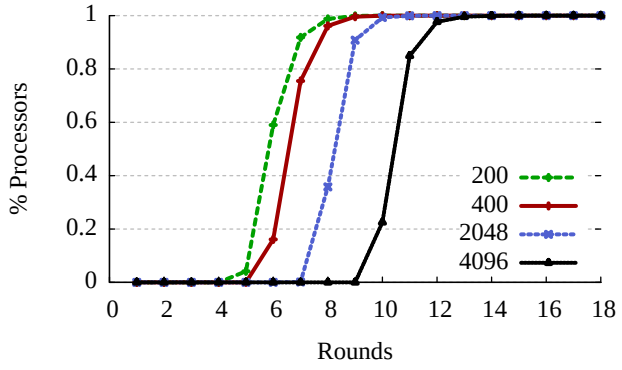


Figure 6: Percentage of processors having various amounts of partial information as rounds progress. There are a total of 4096 underloaded processors. 99% receive information about 400 processors by 8th round while it takes 12 rounds for all the 4096 underloaded processors.

From the above experiments, it is evident that good load balance could be attained with partial information. This is particularly useful as propagating partial information takes fewer number of rounds and incurs lesser overhead. We utilize this observation to choose a value of TTL much lower than $\log n$ for comparison with other strategies on real applications.

7.2 Evaluation using Applications

We evaluate our *GrapevineLB* load balancing strategy on two applications, LeanMD and adaptive mesh refinement (AMR), by comparing against various load balancing strategies. We use *GrapevineLB* with a fixed set of configurations, $\{f = 2,\ TTL = 0.4 \times \log_2 n$, Informed Propagation, Informed Transfer $\}$, and focus on comparing with other load balancing strategies. Results presented here are obtained from experiments run on IBM BG/Q Mira. Mira is a $49,152$ node Blue Gene/Q installation at the ALCF. Each node consists of 16 64-bit PowerPC A2 cores run at 1.6GHz. The interconnect in this system is a $5D$ torus. In the following sections, we first provide details about the applications and the load balancers and then present our evaluation results.

7.2.1 Applications

Adaptive Mesh Refinement: AMR is an efficient technique used to perform simulations on very large meshes which would otherwise be difficult to simulate even on modern-day supercomputers. This application simulates a popular yet simple partial differential equation called Advection. It uses a first-order upwind method in 2D space for solving the advection equation. The simulation begins on a coarse-grained structured grid of uniform size. As the simulation progresses, individual grids are either refined or coarsened. This leads to slowly-growing load imbalance which requires frequent load balancing to maintain high efficiency of the system. This application has been implemented using the object-based decomposition approach in Charm++ [22].

LeanMD: It is a molecular dynamics simulation program written in Charm++, that simulates the behavior of atoms based on the Lennard-Jones potential. The computations performed in this code are similar to the short-range non-bonded force calculation in NAMD [27], an application that has won the Gordon Bell award. The three-dimensional simulation space consisting of atoms is divided into cells. In each iteration, force calculations are done for all pairs of atoms that are within a specified cutoff distance. For a pair of cells, the force calculation is assigned to a set of objects called the *computes*. After the force calculation is performed by the computes, the cells update the acceleration, velocity and position of the atoms within their space. The load imbalance in LeanMD is primarily due to the variable number of atoms in a cell. The load on computes is proportional to the the number of atoms in the cells which changes over time as the atoms move based on the force calculation. We present simulation of LeanMD for a 2.8 million atom system. The load imbalance is gradual therefore load balancing is performed infrequently.

7.2.2 Load Balancers

We compare the performance of *GrapevineLB* against several other strategies including centralized, distributed and hierarchical strategies. The load balancing strategies are

GreedyLB: A centralized strategy that uses greedy heuristic to assign heaviest tasks onto least loaded processors iteratively. This strategy does not take into consideration the current assignment of tasks to processors.

AmrLB: A centralized strategy that does refinement based load balancing taking into account the current distribution of work units. This is tuned for the AMR application [22].

HierchLB: A hierarchical strategy [35] in which processors are divided into independent groups and groups are organized in a hierarchical manner. At each level of the hierarchy, the root node performs the load balancing for the processors in its sub-tree. This strategy can use different load balancing algorithms at different levels. It is an optimized implementation that is used in strong scaling NAMD to more than $200K$ cores.

DiffusLB: A *neighborhood averaging* diffusion strategy [8, 33] where each processor sends information to its neighbors in a domain and load is exchanged based on this information. A domain constitutes of a node and all its neighbors where the neighborhood is determined by physical topology. On receiving the load information from all its neighbors, a node will compute the average of the domain and determines the amount of work units to be transfered to each of its neighbors. This is a two phase algorithm: in the first phase tokens are sent and in the second phase actual movement of work units is performed. There are multiple iterations of token exchange and termination is detected via quiescence [26].

We use the following metrics to evaluate the performance of various load balancing strategies: 1) *Execution time per step* for the application, which indicates the quality of the load balancing strategy. 2) *Load balancing overhead*, which is the time taken by a load balancing strategy. 3) *Total application time*, which includes the time for each iteration as well as the time for load balancing strategy.

7.2.3 Evaluation with AMR

We present an evaluation of different load balancing strategies on the AMR application on BG/Q ranging from 4096 to 131072 cores. AMR requires frequent load balancing to run efficiently because coarsening and refinement of the mesh introduces dynamic load imbalance.

Time per Iteration: First we compare the execution

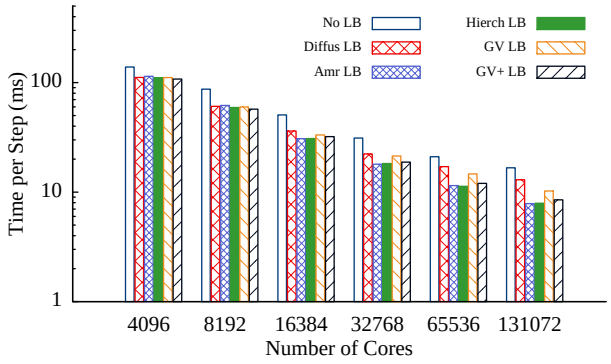


Figure 7: Comparison of time per step (excluding load balancing time) for various load balancing strategies for AMR on Mira (IBM BG/Q). GV+ achieves quality similar to other best performing strategies. Note that axes are log scale.

LB	Number of Cores					
	4K	8K	16K	32K	65K	131K
Hierc	9.347	5.505	2.120	0.888	0.560	0.291
Amr	2.018	3.321	4.475	7.836	11.721	21.147
Diff	0.018	0.017	0.016	0.016	0.016	0.015
Gv	0.012	0.012	0.013	0.014	0.016	0.018
Gv+	0.012	0.013	0.013	0.014	0.016	0.018

Table 2: Average cost (in seconds) per load balancing step of various strategies for AMR

time per iteration of the application to evaluate the quality of the load balancers. This directly relates to $\mathcal{I}$ metric given in equation 1 because as $\mathcal{I} \rightarrow 0$, the maximum load of the system approaches the average load, resulting in least time per iteration. Figure 7 shows, on logarithmic scale, the time taken per iteration with various load balancing strategies. The base run was made without any load balancing and is referred to as *NoLB*. It is evident that with *NoLB* the efficiency of the application reduces as it is scaled to higher number of cores. The *Grapevine+LB* load balancer (shown as GV+ LB) reduces the iteration time by 22% on $4K$ cores and 50% on $131K$ cores. *AmrLB* and *HierchLB* also show comparable performance for this metric. We see an increase in gain because on larger number of cores, the load imbalance becomes significant. This is because the number of work units per processor decreases and the chance that a processor becomes overloaded increases. *DiffusLB* also shows some improvement but much less than the aforementioned ones on larger scale. For $131K$, it reduces the time per step by 22% while others (*AmrLB*, *HierchLB* and *Grapevine+LB*) reduce it by 50%. An interesting thing to note here is that, *Grapevine+LB* load balancer performs better than *GrapevineLB* (shown as GV LB) for core counts more than $32K$. This is due to the fact that *Grapevine+LB* ensures that no underloaded processor gets overloaded using a Nack mechanism. From this it is evident that the quality of load balance performed by *Grapevine+LB* is at-par with the quality of the centralized and hierarchical strategies.

Overhead: Table 2 shows the overhead incurred by various load balancers in one load balancing step for different system sizes. The overhead(load balancing cost) includes the time for finding the new assignment of objects to processors and the time for migrating the objects. The overhead incurred by *AmrLB* is 2.01 s for $4K$ cores and increases with the increase in the system size to a maximum of 21.14 s for $131K$ cores. *HierchLB* incurs an overhead of 5.5 s for $8K$ cores and thereafter the cost reduces to a minimum of 0.29 s for $131K$ cores. This is due to the fact that as the number of processors increases, the number of sub groups also increase resulting in a reduction of work units per group. Hence, the time taken for the root to carry out the load balancing strategy reduces. The distributed load balancing strategies, *GrapevineLB* and *DiffusLB*, incur considerably less overhead in comparison to other strategies.

LB	Number of Cores					
	4K	8K	16K	32K	65K	131K
No	27.61	17.30	10.06	6.11	3.98	2.94
Hierc	87.58	41.23	21.06	9.84	6.03	3.25
Amr	36.98	35.40	37.55	58.42	84.19	149.22
Diff	22.26	12.16	7.23	4.41	3.24	2.21
Gv	22.21	12.00	6.56	4.21	2.76	1.69
Gv+	21.50	11.48	6.44	3.73	2.34	1.48

Table 3: Total application time (in seconds) for AMR on BG/Q. Proposed strategies Gv and Gv+ perform the best across all scales.

Total Application Time: The total application time using various strategies is given in Table 3. In this application frequent load balancing is required. The overhead of the centralized strategies diminishes the benefit of load balancing. *AmrLB* does not improve the total application time because of the overhead of load balancing. This is true for the hierarchical strategy as well. The *DiffusLB* results in a reduction of the execution time by 28% for $16K$ cores and 24.8% for $131K$ cores where as *GrapevineLB* gives a reduction of 35% and 49.6% respectively. *GrapevineLB* provides a large performance gain by achieving a better load balance and incurring less overhead. It enables more frequent load balancing to improve the efficiency. A future direction would be to use MetaBalancer [28] to choose the ideal load balancing period.

7.2.4 Evaluation with LeanMD

We evaluate LeanMD by executing a 1000 iterations and invoking the load balancer first time at the $10th$ iteration and periodically every 300 iterations there after.

Execution time per iteration: We compare the execution time per iteration of the application to evaluate the quality of the load balancers. For $4K$ to $16K$ cores, the centralized, hierarchical and *GrapevineLB* strategies improve the balance up to 42%. The diffusion-based strategy improves the balance only by 35% at $8K$ cores and there after it shows diminishing gains. *GrapevineLB* on the other hand performs at-par to the centralized load balancer up to $32K$. At $131K$ cores, it only gives an improvement of 25% in comparison to 36% given by centralized scheme. This reduction is because the number of tasks per processor decreases to 4 at $131K$, causing refinement-based load balancers to perform suboptimally. *GrapevineLB* is consistently better than the *DiffusLB* because it has a representation of the global state of the system which helps it make better load balancing decisions.

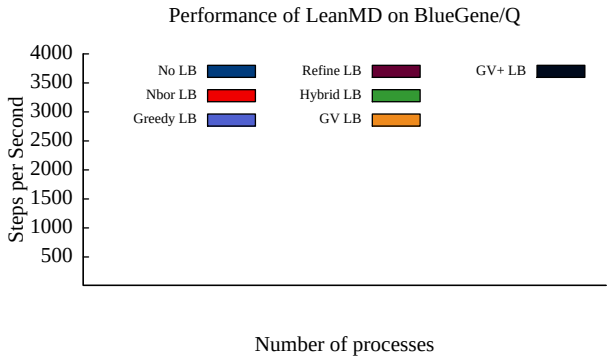


Figure 8: Comparison of time per step (excluding load balancing time) for various load balancing strategies for LeanMD on Mira (IBM BG/Q). Note that axes are log scale.

LB	Number of Cores					
	4K	8K	16K	32K	65K	131K
Hierc	3.721	1.804	0.912	0.494	0.242	0.262
Grdy	7.272	7.567	8.392	12.406	18.792	21.913
Diff	0.080	0.057	0.051	0.035	0.027	0.018
Gv	0.017	0.013	0.014	0.016	0.015	0.018
Gv+	0.017	0.013	0.013	0.015	0.015	0.018

Table 4: Average cost per load balancing step (in seconds) of various strategies for LeanMD

Overhead: Table 4 presents a comparison of overhead incurred by various strategies for a single load balancing step. The load balancing cost of the centralized strategy is very high and is on the order of tens of seconds. The high overhead of *GreedyLB* is due to the overhead of statistics collection, making the decision at the central location and the migration cost. The hierarchical strategy, *HierchLB*, incurs less overhead. It takes 3.7 s for $4K$ cores and decreases to 0.26 s as the system size increases to $131K$. The overhead of *DiffusLB* is 0.080 s for $4K$ cores and decreases thereafter. This is because the number of work units per core decreases as the number of cores increase. Finally, we observe that *GrapevineLB* has an overhead of 0.017 s for $4K$ cores and decreases with increase in system size to 0.013 s for $16K$ cores and thereafter increases to 0.018 s for $131K$. The load balancing cost for *GrapevineLB* includes the time for information propagation and transfer of work units. At $4K$ cores the load balancing time is dominated by the transfer of work units. As the system size increases, the work units per processor decreases. This results in cost being dominated by information propagation.

Total Application Time: Table 5 shows the total application time for LeanMD. The centralized strategy improves the total application time but only for core counts up to $16K$. Beyond $16K$ cores, the overhead due to load balancing exceeds the gains and results in increasing the total application time. *DiffusLB* incurs less overhead in comparison to the centralized and hierarchical strategies but it does not show substantial gains because the quality of load balance is not good. At $32K$ cores, it gives a reduction of 12% in total execution time while *GrapevineLB* gives 34% and *HierchLB* gives 33%. *HierchLB* incurs less overhead in comparison to the centralized strategies. It reduces the total execution time by 37% for $8K$ cores while *GrapevineLB* reduces it by 42%. *GrapevineLB* consistently gives better performance than other load balancing strategies. *Grapevine+LB* gives the maximum performance benefit by reducing the total application time by 20% for $131K$, 40% for $16K$ cores, around 42% for $4K$ and $8K$ cores. Thus, *GrapevineLB* and *Grapevine+LB* provide an improvement in performance by achieving a high quality load balance with significantly less overhead.

LB	Number of Cores					
	4K	8K	16K	32K	65K	131K
No	519.19	263.30	131.56	67.19	41.49	27.20
Hierc	325.00	163.65	84.62	44.56	33.49	22.43
Grdy	336.34	184.09	112.23	90.19	99.51	105.35
Diff	342.15	170.41	99.67	58.47	34.91	24.29
Gv	311.12	157.34	80.45	45.58	31.91	22.79
Gv+	305.20	152.21	79.94	43.88	31.30	21.53

Table 5: Total application time (in seconds) for LeanMD on BG/Q

8. CONCLUSION

We have presented *GrapevineLB*, a novel algorithm for distributed load balancing. It includes a light weight information propagation stage based on gossip protocol to obtain partial information about the global state of the system. Exploiting this information, *GrapevineLB* probabilistically transfers work units to obtain high quality load distribution.

We have demonstrated performance gains of *GrapevineLB* by comparing against various centralized, distributed and hierarchical load balancing strategies for molecular dynamics simulation and adaptive mesh refinement. *GrapevineLB* is shown to match the quality of centralized strategies, in terms of the time per iteration, while avoiding associated bottlenecks. Our experiments demonstrate that it significantly reduces the total application time in comparison to other load balancing strategies as it achieves good load distribution while incurring less overhead.

Acknowledgment

The authors would like to thank Phil Miller, Jonathan Lifflander and Nikhil Jain for their valuable help in proofreading. This research was supported in part by the US Department of Energy under grant DOE DE-SC0001845 and by NSF ITR-HECURA-0833188. This research also used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. Experiments for this work were performed on Mira and esta, IBM Blue Gene/Q installations at Argonne National Laboratory. The authors would like to acknowledge PEACEndStation and PARTS projects for the machine allocations provided by them.

9. REFERENCES

[1] I. Ahmad and A. Ghafoor. A semi distributed task allocation strategy for large hypercube supercomputers. In *Conference on Supercomputing*, 1990.

[2] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 1999.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP*, 1995.

[4] J. E. Boillat. Load balancing and poisson equation in a graph. *Concurrency: Practice and Experience*, 2(4):289–313, 1990.

[5] 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)*, pages 1–11. IEEE, 2007. Best Algorithms Paper Award.

[6] C. Chevalier, F. Pellegrini, I. Futurs, and U. B. I. Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. In *In Proceedings of Euro-Par 2006, LNCS*, pages 243–252, 2006.

[7] Y.-C. Chow and W. H. Kohler. Models for dynamic load balancing in homogeneous multiple processor systems. In *IEEE Transactions on Computers*, 1982.

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

[9] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of parallel and distributed computing*, 7(2):279–301, 1989.

[10] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *ACM Symposium on Principles of distributed computing*, 1987.

[11] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[12] George Karypis and Vipin Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Proc. of the 8th SIAM conference on Parallel Processing for Scientific Computing*, 1997.

[13] George Karypis and Vipin Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48:96–129 , 1998.

[14] A. Ha'c and X. Jin. Dynamic load balancing in distributed system using a decentralized algorithm. In *Intl. Conf. on Distributed Computing Systems*, 1987.

[15] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184(2):485–500, 2000.

[16] B. Hendrickson and R. Leland. The Chaco user's guide. Technical Report SAND 93-2339, Sandia National Laboratories, Albuquerque, NM, Oct. 1993.

[17] Y. Hu and R. Blake. An optimal dynamic load balancing algorithm. Technical report, Daresbury Laboratory, 1995.

[18] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn. Massively parallel cosmological simulations with ChaNGa. In *IPDPS*, 2008.

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

[20] L. V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[21] W. Kermack and A. McKendrick. Contributions to the mathematical theory of epidemics. ii. the problem of endemicity. *Proceedings of the Royal society of London. Series A*, 138(834):55–83, 1932.

[22] A. Langer, J. Lifflander, P. Miller, K.-C. Pan, L. V. Kale, and P. Ricker. Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement. In *SBAC-PAD 2012*, New York, USA, October 2012.

[23] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *HPDC*, 2012.

[24] F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *Software Engineering, IEEE Transactions on*, (1):32–38, 1987.

[25] Y.-J. Lin and V. Kumar. And-parallel execution of logic programs on a shared-memory multiprocessor. *J. Log. Program.*, 10(1/2/3&4):155–178, 1991.

[26] F. Mattern. Algorithms for distributed termination detection. *Distributed computing*, 2(3):161–175, 1987.

[27] C. Mei and L. V. K. et al. Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*.

[28] H. Menon, N. Jain, G. Zheng, and L. V. Kalé. Automated load balancing invocation based on application characteristics. In *IEEE Cluster*, 2012.

[29] L. M. Ni and K. Hwang. Optimal load balancing in a multiple processor system with many job classes. In *IEEE Trans. on Software Eng.*, volume SE-11, 1985.

[30] D. Peleg and E. Upfal. The token distribution problem. *SIAM Journal on Computing*, 18(2):229–243, 1989.

[31] S. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing 1994*, Nov. 1994.

[32] Y. Sun, G. Zheng, P. Jetley, and L. V. Kale. An Adaptive Framework for Large-scale State Space Search. In *IPDPS*, 2011.

[33] 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*, September 1993.

[34] C. Xu, F. C. M. Lau, and R. Diekmann. Decentralized remapping of data parallel applications in distributed memory multiprocessors. *Concurrency - Practice and Experience*, 9(12):1351–1376, 1997.

[35] G. Zheng, A. Bhatele, E. Meneses, and L. V. Kale. Periodic Hierarchical Load Balancing for Large Supercomputers. *IJHPCA*, March 2011.