Automated Load Balancing Invocation based on Application Characteristics

Harshitha Menon, Nikhil Jain, Gengbin Zheng and Laxmikant Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, Illinois
Email: {gplkrsh2,nikhil,gzheng,kale}@illinois.edu

***Abstract*—Performance of applications executed on large parallel systems suffer due to load imbalance. Load balancing is required to scale such applications to large systems. However, performing load balancing incurs a cost which may not be known a priori. In addition, application characteristics may change due to its dynamic nature and the parallel system used for execution. As a result, deciding when to balance the load to obtain the best performance is challenging. Existing approaches put this burden on the users, who rely on educated guess and extrapolation techniques to decide on a reasonable load balancing period, which may not be feasible and efficient.**

In this paper, we propose the Meta-Balancer framework which relieves the application programmers of deciding when to balance load. By continuously monitoring the application characteristics and using a set of guiding principles, Meta-Balancer invokes load balancing on its own without any prior application knowledge. We demonstrate that Meta-Balancer improves or matches the best performance that can be obtained by fine tuning periodic load balancing. We also show that in some cases Meta-Balancer improves performance by 18% whereas periodic load balancing gives only a 1.5% benefit.

***Keywords*-load balancing, automated, parallel, simulation**

I. Introduction

Modern parallel applications running on large clusters often involve simulations of dynamic and complex systems [1, 2]. A significant amount of effort is spent on writing these parallel applications in order to fully exploit the processing power of large systems and show scalability. For such applications, load balancing techniques are crucial to achieve high performance on large scale systems [3], because load imbalance among processors leads to significant drop in system utilization and hampers application's scalability. With ever-growing parallelism available in supercomputers of today, tackling the imbalance in an efficient manner is a difficult problem.

In a large class of scientific applications such as NAMD [1], FEM [4] and climate simulation, the problem is broken into smaller data and work units that execute on multiple processors. The computation being performed consists of a number of time steps and/or iterations with frequent interaction among data/work units via messages. Independent of the programming paradigm being used (such as MPI and Charm++ [5]), handling load imbalance in such applications is a multi-faceted problem and involves the following common tasks:

1) Identify movable work units and estimate their load
2) Make load balancing decisions, including how often to balance load
3) Move the work units

One method to estimate the work load is using performance modeling technique with a cost function that models the work load based on the programmer's a priori knowledge of the application domain. Another method, which is adopted in Charm++, is based on instrumenting the load information from the recent past as a guideline for the near future, using a heuristic known as the *principle of persistence* [6]. It posits that, empirically, the computational loads and communication patterns of the work units *tend to* persist over time, even in dynamically evolving computations. Therefore, load balancer can use the instrumented load information to make load balancing decisions. The key advantage of this approach is that it is application independent, and it has been shown to be effective for a large class of applications, such as NAMD [7], ChaNGa [8] and Fractography3D [9].

Performing load balancing entails overheads which includes the time spent in finding the new placement of work units and the time spent in moving the work units. Due to the cost of load balancing, it is important to determine if invoking the load balancer is profitable, i.e., whether the overhead due to load balancing is less than the gain obtained after load balancing for a period of time. Typically, application behavior depends on the size of system being simulated and the parallel system being used for simulation. As a result, finding the time steps (or iterations) at which load balancing should be invoked to obtain best performance is a difficult task. Most runtime systems (RTS) depend on the application programmers to decide when to balance the load. A common practice is to choose a fixed period to invoke the load balancer; for example every 100 time steps. This, however, prevents the load balancing from adapting to the changing application behavior.

In this paper, we introduce the Meta-Balancer framework which is a step towards automating load balancing related decision making. Based on the application characteristics observed at runtime and a set of guiding principles, Meta-Balancer relieves the application programmer from the critical task of deciding when the load balancer should be invoked. Unlike many existing models, which rely only on the most recent data and do not make predictions based on dynamic nature of applications [10, 11], Meta-Balancer

continuously monitors the application and predicts load behavior. Using a linear prediction model on the collected information, Meta-Balancer predicts the time steps (or iterations) at which load balancing should be performed for optimal performance. In addition, Meta-Balancer monitors for sudden changes in the application behavior and invokes the load balancer if needed.

We have implemented Meta-Balancer on top of Charm++ load balancing framework in order to take advantage of its support for load balancing. We demonstrate that Meta-Balancer improves application performance by choosing the correct time steps to invoke load balancer for iterative applications. We show that, using Meta-Balancer, performance of LeanMD, a molecular dynamics simulation program, can be improved by upto 18% in cases where a fine-tuned fixed load balancing period provides marginal gains of 1.5%. For Fractography3D, we demonstrate that Meta-Balancer identifies the dynamic characteristics of the application without any input from the user, and at least matches the performance of periodic load balancing with a carefully chosen fixed period. Note that working of Meta-Balancer is transparent to a user and only a trivial change in application is required to use Meta-Balancer. Moreover, the same concepts can be used for any other parallel programming paradigm such as MPI.

The key contributions of this paper are as follows:

- We introduce a generic concept that can be used to automatically decide when to invoke the load balancer based on application characteristics.
- We present an implementation of our concept as Meta-Balancer in Charm++ using asynchronous algorithms, which executes in the background and is interleaved with application's execution.
- We demonstrate that Meta-Balancer takes correct decisions regarding invocation of the load balancing without any input from the user for two real world applications, and improves performance in most cases.

In §II, we provide a background on the load balancing framework in Charm++ which is followed by a description of Meta-Balancer in §III. Thereafter, results on using Meta-Balancer with two real world applications are presented in §IV. Finally, previous work in presented in §V followed by conclusion and future work in §VI.

II. Background

In our design, we consider a large scale application as a collection of migratable objects distributed on a large number of processors, communicating via messages. Load balancing framework can migrate these objects and the associated work from an overloaded processor to an underloaded processor. Our implementation takes advantage of the existing Charm++ load balancing framework that is based on such an execution model [10].

A. Charm++ and its Load Balancing Framework

Charm++ is a parallel programming model which implements message-driven parallel objects (*chares*), which are migratable among processors. An application written in Charm++ is comprised of a collection of chares, distributed among the processors and communicating via messages. When there is imbalance of work among the processors, migrating the objects from an overloaded processor to an underloaded processor helps in achieving balance and thereby improves the performance of the application.

The load balancing framework in Charm++ is a measurement based framework, and is responsible for two key tasks. Firstly, it instruments the application code at a very fine-grain level and provides the vital statistics for load balancing. Secondly, it executes the load balancing strategy to determine a mapping of objects onto processors and performs the migration.

Charm++'s object model simplifies the task of application instrumentation. The runtime system (RTS) instruments the start and the end time of each method invocation on the chares. The advantage of this method is that it provides an automatic application-independent method to obtain load information without user input or manual prediction of the load. Further, the Charm++ RTS can record chare-to-chare and collective communication patterns as every communication initiated by chares is eventually handled by the RTS. The RTS also records the idle time and the background load on a processor. However, the task of initiating load balancing and selection of load balancing strategy is the responsibility of the programmer. The load balancing strategies are plugins in Charm++.

Algorithm 1 Application Code on every Chare

```
1: when ResumeWork invoked
2: perform work
3: if (curr_iter % fixed_period == 0) then
4:     call AtSync
5: else
6:     call ResumeWork
7: end if
8: curr_iter++
```

In Algorithm 1, we present the iterative component of a typical Charm++ program. In Charm++, execution proceeds when functions are invoked for chares by RTS on receiving messages for them. An application run begins with RTS invoking appropriate functions (ResumeWork in our example) for some chares. Most function calls (such as *call ResumeWork*) results in a message send to RTS which subsequently results in a function invocation. After the message is sent, execution resumes assuming that the function call returned. Each object calls *AtSync*, a blocking collective, when it is ready for load balancing.

Once all chares on a processor call *AtSync*, the load balancing framework takes control. Thereafter, the load statistics associated with all the processors and chares are sent either to a central processor (if using a centralized strategy) or to a set of processors (if using a hybrid strategy) [12]. At these hub(s), the load balancing framework computes a new mapping of chares to processors using the collected

statistics, and the strategy specified by the programmer either as a run time argument or during code compilation. Once the new mapping is computed, the load balancing decision is broadcast to every processor involved and the migrations are performed. Eventually, the chares resume their execution on being invoked by the RTS.

III. Meta-Balancer

Meta-Balancer is designed as a framework which, given an application and a load balancer, automatically makes decision at runtime on when to invoke the load balancer, taking into account the application characteristics. It is implemented on top of Charm++ load balancing framework (§II-A). Meta-Balancer relies on a heuristic known as *the principle of persistence* described in §I. The idea is to let the runtime continuously monitor the application's load behavior, and based on the collected statistics, predict the trend of the change of the load, and make decisions on when to invoke load balancing. Meta-Balancer consists of three major components, namely, asynchronous statistics collection, decision making module for the ideal LB period and consensus of LB period, which are described below.

A. Meta-Balancer Statistics Collection

Meta-Balancer collects load information about the running application aggressively to determine if load balancing is needed. Using the same *AtSync* interface as described in §II-A, every chare informs its work load to the Meta-Balancer and moves on to the next iteration. Once all the chares residing on a processor have deposited their load for an iteration, Meta-Balancer gathers these statistics via an asynchronous reduction as shown in Figure 1. However, instead of calling *AtSync* only when load balancing is actually needed, *AtSync* is now called more frequently, for example in every iteration, so that Meta-Balancer can examine the overall load information continuously.

Since Meta-Balancer requires frequent aggregation of statistics at a central location, this may incur significant communication overhead on large systems. In order to reduce the overhead, we select a minimal set of statistics to be collected periodically by the Meta-Balancer. These statistics include the *maximum load*, *average load* and the *minimum utilization* on all processors in the system. We have found that these vital statistics are sufficient for deciding the LB period for optimal performance. Further, the overheads are mitigated by the use of Charm++'s asynchronous reduction of the minimal statistics that runs in the background and overlaps with the normal execution of the application, thanks to Charm++'s asynchronous message driven execution model.

B. Ideal Load Balancing Period Computation

Using the aggregated result of the load statistics, Meta-Balancer determines whether there is load imbalance, which can be calculated by

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

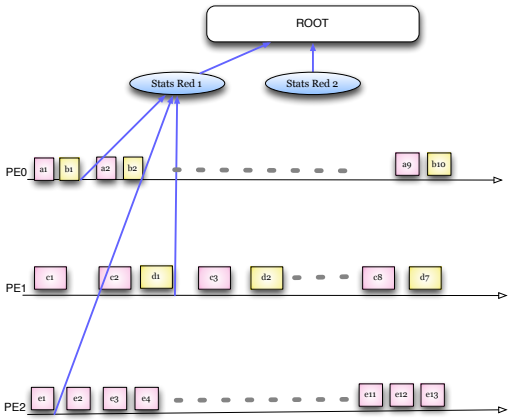


Figure 1. Periodic Statistics Collection

where L_{max} is the load on the most loaded processor and L_{avg} is the average load of all the processors. If there is load imbalance in the system ($\zeta > 0$), it will lead to performance loss. However, presence of load imbalance does not necessarily entail for load balancing as it may not be profitable to do the load balancing due to the overhead in load balancing step.

With load balancing, the total execution time of an application is sum of the load balancing overhead and the time spent in running the application which can be affected by the load balancing. The goal is to minimize the total execution time. This can be a challenging problem since we need to model the effectiveness of load balancer on the application and how the application load evolves over time after load balancing. We present a simple mathematical analysis based on an assumption that the maximum and average load can be modeled linearly with time (i.e. iterations). A linear model has been chosen because more complex models in the proximity can be approximated to piecewise linear. The mathematical analysis helps derive the ideal load balancing period which can be used by Meta-Balancer to decide the next iteration at which load balancing should be performed. Let,

τ be the ideal LB period,
γ be the total iterations an application executes,
Γ be the total application execution time, and
Δ be the cost associated with load balancing

Let the average load be represented by the line equation:

$$L_{avg} = at + l_a \tag{2}$$

where a is the slope and l_a is the average load for the first iteration.

Let the maximum time per iteration, approximately equal to the maximum load on the most loaded processor, be represented by the line equation:

$$L_{max} = mt + l_m \tag{3}$$

where m is the slope w.r.t. to the average load line, l_m is the difference of maximum load and average load for the first iteration and t is the time steps (or iterations).

Application execution time, Γ, can be computed by an integral of maximum time per iteration over the total iterations and load balancing cost as shown below:

$$\begin{aligned}\Gamma &= \frac{\gamma}{\tau} \times (\int_0^{\tau} (mt + l_m)dt + \Delta) + \int_0^{\gamma} (at + l_a)dt \\ \Gamma &= \frac{\gamma}{\tau} \times (\frac{m\tau^2}{2} + l_m\tau + \Delta) + \gamma \times (\frac{a\gamma}{2} + l_a) \\ \Gamma &= \gamma \times (\frac{m\tau}{2} + l_m + \frac{\Delta}{\tau} + \frac{a\gamma}{2} + l_a)\end{aligned}$$

Note that $\frac{\gamma}{\tau}$ represents the number of times the load balancing is invoked during the execution of an application. Also, for the purpose of this analysis, we have assumed that the load balancing leads to a perfect balance. In order to minimize Γ, we differentiate it with respect to τ, and obtain the following value of τ used by Meta-Balancer as the ideal load balancing period.

$$\begin{aligned}\frac{d}{d\tau}(\Gamma) &= \gamma \times (\frac{m}{2} - \frac{\Delta}{\tau^2}) = 0 \\ \tau &= \sqrt{\frac{2\Delta}{m}} \end{aligned} \quad (4)$$

Eq 4 effectively states that once the load balancer is invoked, the next invocation should be performed only when the cost for load balancing invocation has been covered. The load balancing cost is estimated using the cost incurred during the previous invocation. The cost for a load balancing is covered by the gains which are obtained as load balancing reduces the iteration time represented by the area of the triangle in Figure 2. The ideal LB period is calculated and continuously refined by Meta-Balancer, using Eq 4, as the application executes. The simplifying assumption that the load balancing leads to a perfect balance is handled by shifting the average curve upwards, if a perfect balance is not achieved, during the gain calculation.

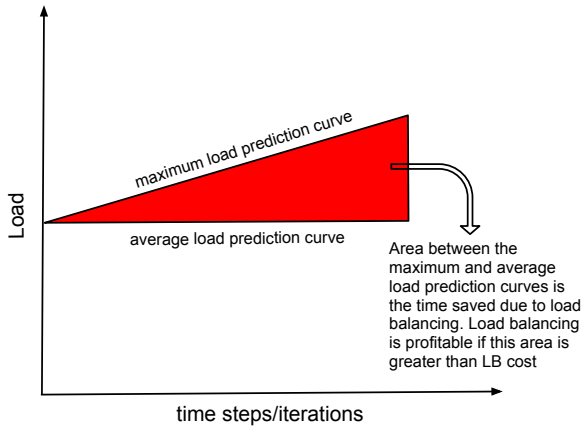


Figure 2. Ideal Load Balancing Period

C. Distributed Consensus

In the original case without using Meta-Balancer, to perform load balancing, all chares enter the load balancing phase in the same iteration, controlled by the fixed load balancing period. After the load balancing, the execution resumes on each chare. However, when using Meta-Balancer, there may be a race condition scenario that causes the program to hang. As an example, let Meta-Balancer's decision of the next load balancing time be iteration number i. Consider a chare a, which receives the notification of load balancing at iteration i before it reaches iteration i. When this chare arrives at iteration i, it waits for the load balancing to be done. Consider another chare b, which is already at the iteration $i+1$ when the notification of load balancing at iteration i is delivered to it. As a result, it will not join other chares waiting for the load balancing to be done. This scenario is possible for applications that have no explicit global synchronization at each iteration, because chares perform the computation work at their own speed, and the load balancing decisions are taken asynchronously and communicated to the processors asynchronously. Since a centralized load balancing strategy enforces a global barrier which requires the participation of all chares, load balancing will never happen in this scenario as chare b missed the iteration i, causing the application to hang.

To avoid such a scenario, all the chares need to reach a consensus on the iteration number that chares can reach to enter the load balancing stage. Since the chares can be in different iterations, we use the following scheme shown in Figure 3 to obtain the consensus.

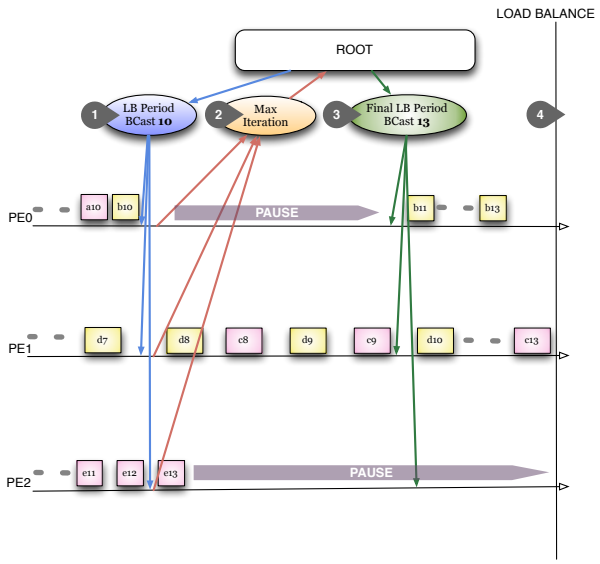


Figure 3. Three Step Consensus Mechanism in Meta-Balancer

In the first step, the central processor (root) broadcasts the calculated ideal LB period as a tentative decision for the next load balancing time. Whenever a processor receives the

tentative LB period, it sets its own local tentative LB period to be the maximum of the received tentative LB period and the maximum iteration of any chare that resides on it and prevents any chare from going beyond that. It contributes its local iteration number via a reduction to find the maximum iteration number any chare is executing. Recall that the reductions in Charm++ are asynchronous. In the final step, when the root receives the maximum iteration number, it sets the final load balancing period to be the maximum of the tentative load balancing period and the maximum iteration number that was received. This final load balancing period is then broadcast to every other processor. Note that it is guaranteed that no chare would have moved beyond this final load balancing period as RTS on each processor blocks a chare which has reached its local tentative LB period.

D. Implementation

In this section, we describe the implementation of Meta-Balancer and its interaction with the application and Charm++ RTS. As mentioned earlier, a typical application can be depicted using Algorithm 1. Periodically, the application invokes *AtSync* which passes the control to Charm++ RTS for potentially invoking the load balancer. The functionality of Charm++ RTS for a chare is shown in Algorithm 2. When *AtSync* is invoked, the Charm++ RTS registers the chare load for the previous iteration with Meta-Balancer. It also performs load balancing if the chare has reached the LB period. In case the consensus mechanism is active and the chare reaches the tentative LB period, it blocks any further execution of the chare. If none of these conditions are met, execution of next iteration is initiated for the chare.

Meta-Balancer code on each processor is presented in Algorithm 3. When the Charm++ RTS registers the load for a chare, Meta-Balancer contributes to the minimal statistics reduction if the load for all chares on that processor has been registered. Thereafter, when the central processor receives the result of this reduction, as shown in Algorithm 4, it finds the ideal LB period and follows the consensus mechanism described in §III-C to find the final load balancing period. The root broadcasts this final load balancing period to all processors. On receiving the final load balancing period, the RTS on each processor either initiates load balancing or invokes next iteration on the chares.

Algorithm 2 Charm RTS on each Chare

```
when AtSync invoked
update chare load in Meta-Balancer
if (reached LB period) then
    perform load balancing
else if (reached tentative LB period) then
    wait for final LB period
else
    call ResumeWork
end if
```

```
when received final LB period
if (curr_iter == finalLBperiod) then
    perform load balancing
else
    call ResumeWork
end if
```

Algorithm 3 Meta-Balancer on every Processor

```
when received chare load
if (all chares have registered their load for an iteration) then
    contribute to reduction for statistics collection
end if
```

```
when received tentative LB period
find maximum iteration number of chares
contribute to reduction for maximum iteration number
```

IV. Experimental Results

In this section, we present a comparison of the performance of Meta-Balancer with respect to periodic load balancing using two real world applications *LeanMD* and *Fractography3D*. We show that Meta-Balancer is able to identify the ideal load balancing period which changes as the application evolves and extracts the best performance for the applications automatically at runtime. For the experiments, we use two machines - Ranger and Jaguar. Ranger is a SUN constellation cluster located at the Texas Advanced Computing Center consisting of $3,936$ nodes connected via a full-CLOS Infiniband interconnect providing 1 GB/s of peer-to-peer bandwidth. Jaguar is a Cray system at Oak Ridge Leadership Computing Facility consisting equipped with Cray's new high performance Gemini network.

A. LeanMD

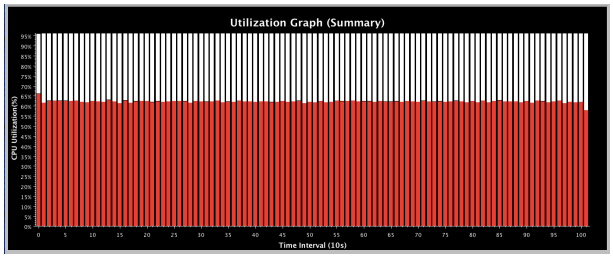
LeanMD is a molecular dynamics simulation program written in Charm++. It simulates the behavior of atoms based on the Lennard-Jones potential, which is an effective potential that describes the interaction between two uncharged molecules or atoms. The computation performed in this code mimics the short-range non-bonded force calculation in NAMD [7], an application widely used by biophysicists, that won the Gordon Bell award.

The force calculation in Lennard-Jones dynamics is done within a cutoff-radius, r_c for every atom. The three-dimensional (3D) simulation space consisting of atoms is divided into cells of dimensions that are equal to the sum of the cutoff distance, r_c and a margin. In each iteration, force calculations are done for all pairs of atoms that are within the cutoff distance. The force calculation for a pair of cells is assigned to a different set of objects called *computes*. Based on the forces sent by the computes, the cells perform the force integration and update various properties of their atoms – acceleration, velocity and positions. The load imbalance in LeandMD is due to the variation in the number of atoms that reside in a cell. The load on computes is directly proportional to the product of the number of atoms in the cells for which the force is being computed. LeanMD is a computation intensive benchmark, in which load imbalance is high when the application begins.

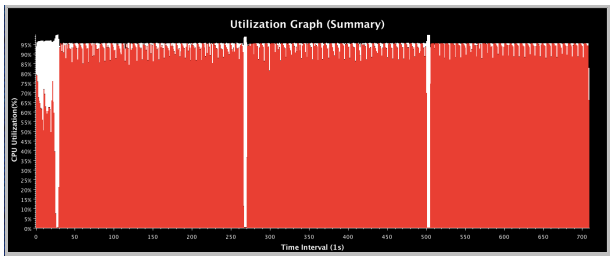
Algorithm 4 Meta-Balancer on Central Processor

1: when received result of **statistics reduction**
2: find tentative LB period
3: inform tentative LB period to all processors

1: when received result of **maximum iteration reduction**
2: set final LB period as max{tentative LB period, maximum iteration}
3: inform final LB period to all processors



(a) No Load Balancing



(b) Meta-Balancer

Figure 4. Processor Utilization of LeanMD on 256 cores

We use LeanMD to study the behavior of 1 million and 300,000 atom system for 2000 time steps on Jaguar and Ranger respectively. First we describe the results of the runs on Jaguar followed by the runs on Ranger. On Jaguar, the base runs for LeanMD were made for a range of core counts $(128, 256, \cdots, 4096)$ without any load balancing. The processor utilization graph for running LeandMD on 256 cores without load balancing is shown in Figure 4(a). On the y-axis, we have the average percentage utilization for all the processors in the system, and the x-axis represents time progression as the simulation proceeds. The key thing to note is that, there is no significant variation in processor loads as the simulation progress. However, the utilization is as low as 60% for the entire run.

In the next step, we ran LeanMD with periodic load balancing over a range of periods $(10, 20, \cdots, 700)$, result for which is shown in Figure 5. There are two important points to note in these results: 1) the LB period which gives the best period varies with the system size, and 2) in some cases, such as 4096 processors, periodic load balancing provides only marginal improvement in performance. In such scenarios, it is very difficult and time consuming for the user to find and use a LB period which gives the best performance.

In Figure 4(b), we present processor utilization for the case where LeandMD is run with Meta-Balancer on 256

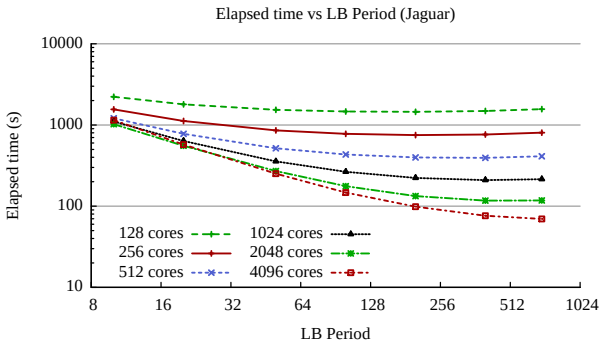


Figure 5. Variation in LB Period for LeanMD on Jaguar

Core	No LB (s)	Periodic LB (Period) (s)	Meta-Balancer (s)
128	1945.16	1451.30 (200)	1388.29
256	1005.22	750.11 (200)	695.55
512	516.47	393.30 (400)	355.85
1024	264.15	209.64 (400)	190.52
2048	135.92	116.69 (400)	94.33
4096	70.68	69.6 (700)	57.83

Table I. LeanMD Application Time on Jaguar

cores. Note that to run Meta-Balancer, the only change in the application was to change the frequency at which *AtSync* was invoked. For our experiments, we invoke *AtSync* every 5 iterations. The vertical notches in the plot indicate the time when load balancing was performed. It can be seen that Meta-Balancer invokes load balancing at the very beginning due to the load imbalance. Thereafter, since the processor utilization is very high (95%) with insignificant variation, load balancing is invoked very infrequently. This translates into performance improvement of 31% as shown in Table I. For large core count of 4096, we observe that while periodic load balancing provides marginal gains of 1.5%, Meta-Balancer improves the performance by 18%. For smaller core counts, Meta-Balancer outperforms any fixed LB period used.

We also ran LeanMD on Ranger cluster to simulate a $300,000$ atom system for 2000 time steps. Figure 6 presents the performance of periodic load balancing when the period is varied from 10 to 700 iterations. For runs on 128, 256 and 512 cores, we observe that the best performance is obtained at different LB periods when compared with the runs on Jaguar. This suggests that the LB period at which the best performance is obtained also changes with the system being simulated and the system being used to execute the application. In Table II, a comparison of performance of Meta-Balancer with other runs is shown. It can be seen that Meta-Balancer consistently outperforms periodic load balancing, and improves the performance over base runs by upto 28%. These experiments on Ranger and Jaguar highlight the utility of Meta-Balancer in identifying the characteristics of an application and invoking load balancing appropriately to obtain good performance without any input

from the user.

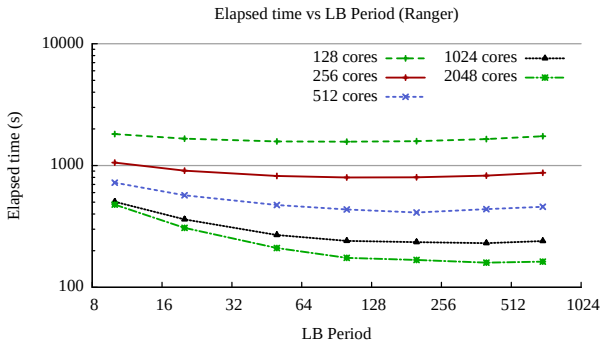


Figure 6. Variation in LB Period for LeanMD on Ranger

Core	No LB (s)	Periodic LB (Period) (s)	Meta-Balancer (s)
128	2169.85	1570.45 (100)	1545.9
256	1087.39	798.28 (100)	787.01
512	552.96	411.71 (200)	401.78
1024	285.8	230.39 (400)	228.55
2048	203.29	159.42 (400)	159.28

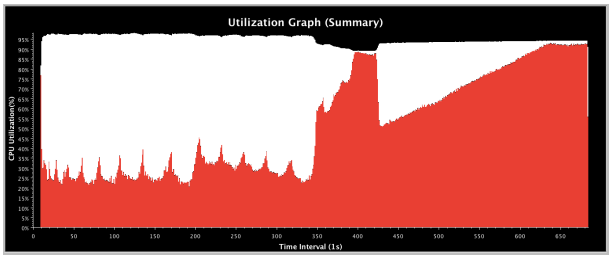
Table II. LeanMD Application Time on Ranger

B. *Fractography3D*

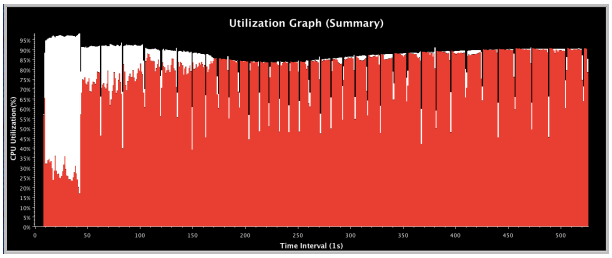
Fractography is used to study fracture surfaces of materials. Fractographic methods are used to determine the cause of failure in engineering structures and evaluate theoretical models of crack growth behavior. Our simulation program, called *Fractography3D*, is written using Charm++ FEM framework [4].

In Fractography3D, the framework discretizes a 3-D volume into tetrahedras. Typically, the number of elements is very large, and they are grouped into a number of chunks distributed across processors. During the simulation, each tetrahedral element is considered to have one of two material properties: elastic or plastic. When an external force is applied to the material under study, the initially elastic response of the material may change to plastic as stress increases, resulting in much more expensive computation in that region. This in turn causes some of the mesh partitions to spend more time on computation per timestep than other partitions, resulting in load imbalance.

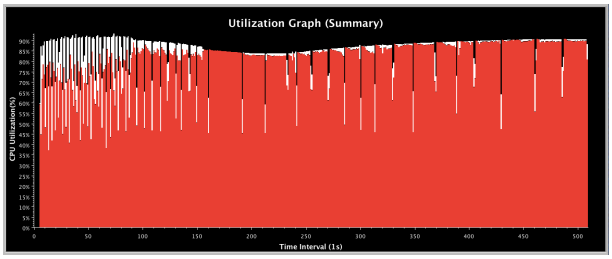
Using Fractography3D, we study the effect of application of external force on a bar. The bar is represented using 88641 points in 3D space which are used to generate tetrahedras. The simulation is performed for 3.6 ms of real world time with a time step of 32 micro seconds. Therefore, there are approximately 11,200 iterations executed during the simulation. For the base runs, we ran Fractography3D on Jaguar without any load balancing being performed for core counts ranging $64, 128, \cdots, 1024$. Figure 7(a) shows the processor utilization graph generated when Fractorgraphy3D is run on 64 cores of Jaguar. On the y-axis, we have the average percentage utilization for all the cores in the system, and the x-axis represents time progression as the simulation proceeds. It can be seen that Fractography3D has a large variation in processor utilization during a simulation run. Also, for a large portion of the execution, substantial amount of processor resources are wasted. Similar trend was found in processor utilization on other core counts as well.



(a) No Load Balancing



(b) Periodic Load Balancing (300 iterations)



(c) Meta-Balancer

Figure 7. Processor Utilization of Fractography3D on 64 cores

Following the base runs, we ran Fractography3D with load balancing being performed periodically. We experimented with a large range of LB periods $(5, 10, 20, \cdots, 7000)$ to find the period which gives the best performance. Figure 8 shows the application run time for Fractography3D using these LB periods on various core counts. We find a significant variation in application execution time as the LB period is varied. If the load balancing is done very frequently, the overheads of load balancing overshoots the gains of load balancing, and results in bad performance. On the other hand, if load balancing is done very infrequently, the load imbalance in the system reduces the gains due to load balancing. However, for intermediate periods, such as 300 iterations, best performance is obtained. In Figure 7(b), we present the processor utilization graph for Fractography3D on 64 cores when the load balancing is performed every 300 iterations. The key thing to note in Figure 7(b) is the

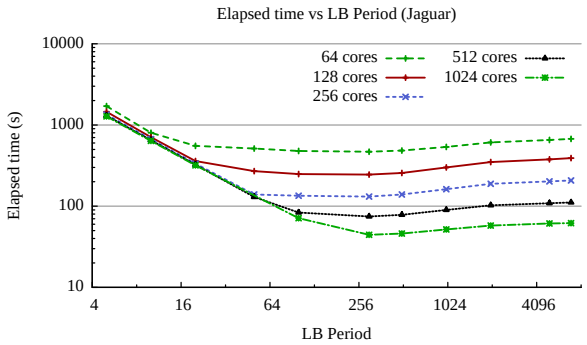


Figure 8. Variation in LB Period for Fractography3D on Jaguar

substantial increase in the processor utilization due to periodic load balancing that results in reduction in application execution time by 28%.

Finally, we ran Fractography3D using Meta-Balancer on the same range of core counts as used for the earlier cases. As mentioned earlier, the only change in the user code required for using Meta-Balancer is the invocation of *AtSync* frequently (every 5 iterations). Figure 7(c) shows the processor utilization graph generated when Fractography3D is run on 64 cores with Meta-Balancer. It can been seen that Meta-Balancer increases processor utilization which results in a performance gain of 31% in comparison to the case in which no load balancing is performed. An interesting thing to note is the frequent invocation of load balancing by Meta-Balancer in the first quarter of the execution as seen by the vertical notches in the plot. This is because of the fact that the load variation among processors changes frequently in the first quarter. Thereafter, when the load variation decreases in the second half of execution, the frequency of load balancing also goes down.

In Table III, a comparison of total execution time of Fractography3D for the following three cases is presented - without load balancing, periodic load balancing every 300 iterations and Meta-Balancer. We observe that in most cases Meta-Balancer either matches or improves the best performance obtained by periodic load balancing. The only exception is at 1024 cores which we believe is because of very small run time of the application. Both Meta-Balancer and periodic load balancing outperform the base case with no load balancing by 28% to 31%. Thus, we have shown that, for Fractography3D, Meta-Balancer is able to invoke load balancing whenever required without any input from the user, and at least match the performance of periodic load balancing. This shows the utility of Meta-Balancer in automating the load balancing decisions, and helping the user avoid numerous runs to find the best LB period for an application. Note that, unlike benchmark runs, the users do not have the luxury of repeating the runs to find the best load balancing period as their objective is to get their runs as fast as possible.

In order to measure the overhead of using Meta-Balancer,

Core	No LB (s)	Periodic LB - 300 (s)	Meta-Balancer (s)
64	654.5	468.35	448.76
128	375.51	244.9	231.36
256	200.78	131.4	127.25
512	109.45	74.6	74.03
1024	59.49	44.4	48.8

Table III. Fractography3D Application Time

LeanMD and Fractography3D were run using Meta-Balancer with a constraint that irrespective of the load balancing period determined by Meta-Balancer, load balancing was not invoked. In comparison to the base case, negligible performance drop was observed. The absence of significant overhead can be attributed to the asynchronous manner in which Meta-Balancer is run by the RTS and its overlap with the application run.

V. Previous Work

Dynamic load balancing strategies have been studied extensively in the past [13, 14]. One important category of load balancing scheme is the periodic load balancing for iterative applications with persistent load patterns. Exemplar runtime systems implementing this approach are Zoltan [3], Chombo [15], and Charm++ [10]. Similar schemes have also been proposed and used in MPI applications [16, 17]. This paper proposes concepts which build upon these existing frameworks in order to make decisions related to load balancing to get good performance.

As described in [14], deciding when to invoke load balancing is a critical step in a load balancing process. This decision depends on determining if performing load balancing at an instance will improve overall application performance. A simple model based on load imbalance factor $\phi(t)$ is proposed in [14], which is based on the estimate of the potential gain through load balancing at time t. However, this model does not consider the dynamic behavior of the application. In contrast, the proposed work uses a linearized extrapolation model for predicting load based on recent past which is used to predict the time steps at which load balancing should be performed.

A more complex scheme to decide a good load balancing period is proposed by Siegell et.al. [18]. Several factors such as interaction overhead, load balancing overhead and application time quantum are measured at run time, and are used to decide the time at which next load balancing should be invoked. The main drawback of this approach is its reliance on users for several inputs to decide the acceptable granularity of each of these factors. The proposed work in this paper requires no input from the user, and hence results in complete automation. Moreover, the proposed work is based on the concept of load balancing cost recovery which has not been explored in any of the previous work.

Techniques for automation of load balancing related decisions are also presented in a recent work by Pearce et.al. [11] in press. The primary focus of this work is on selection

of load balancing strategy based on simulation of multiple strategies. For load prediction, the dynamic nature of application is not considered, and a synchronous global barrier based scheme is used for making decisions. In contrast, our work focuses on deciding when to invoke load balancing based on prediction of application load characteristics. We also avoid barriers by using asynchronous communication which may be beneficial on large systems.

VI. Conclusion

Load imbalance is a key factor that affects performance and scalability of an application. Leaving it to the application programmer to manually handle the load imbalance in a dynamic application, and to find an optimum load distribution throughout the run of the application, is unreasonable and inefficient. In this paper, we presented techniques for deciding when to invoke load balancing based on application characteristics and their embodiment in the Meta-Balancer. Meta-Balancer represents an application independent concept which is helpful in extracting the best performance of an application without requiring the user to make multiple benchmark runs or use domain specific knowledge to estimate the load balancing period. We also presented details related to a practical implementation of Meta-Balancer on top of Charm++.

We demonstrated the adaptive nature of Meta-Balancer in the context of two real world applications. We showed that Meta-Balancer is able to identify the ideal load balancing period which changes as the application evolves and extracts the best performance automatically. In the process, we presented scenarios in which Meta-Balancer is able to extract substantial gains whereas periodic load balancing provides only marginal gains.

In future, we plan to extend Meta-Balancer to automate the selection of load balancing strategy. This includes strategy selection based on application characteristics, such as computational load and communication volume, system characteristics, such as centralized and distributed algorithms etc. We plan to include higher order algorithms for predicting load. We also plan to work on newer methods to better estimate the load imbalance and use them to take load balancing decisions.

Acknowledgment

The authors would like to thank Esteban Meneses for helpful discussion. This research was supported in part by the Blue Waters: Leadership Petascale System project (which is supported by the NSF grant OCI 07-25070) and by the US Department of Energy under grant DOE DE-SC0001845. This work used machine resources from Teragrid under award ASC050039N.

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, 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.*, 2005.

[4] 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.

[5] 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.

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

[7] 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*.

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

[9] 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.

[10] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Department of Computer Science, UIUC, 2005.

[11] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato, "Quantifying the effectiveness of load balance algorithms," in *26th ACM international conference on Supercomputing*, ser. ICS '12, 2012, pp. 185–194.

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

[13] 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.

[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*, September 1993.

[15] "Chombo Software Package for AMR Applications," http://seesar.lbl.gov/anag/chombo.

[16] J. Corbalán, A. Duran, and J. Labarta, "Dynamic load balancing of mpi+openmp applications," in *ICPP*, 2004, pp. 195–202.

[17] I. Banicescu and S. F. Hummel, "Balancing processor loads and exploiting data locality in n-body simulations," in *In Proceedings of Supercomputing95 (CD-ROM*, 1995.

[18] B. S. Siegell and P. A. Steenkiste, "Automatic selection of load balancing parameters using compile-time and run-time information," 1996.