META-BALANCER: AUTOMATED LOAD BALANCING BASED ON APPLICATION BEHAVIOR

BY

HARSHITHA MENON GOPALAKRISHNAN MENON

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Professor Laxmikant Kale

Abstract

With the dawn of petascale, and with exascale in the near future, it has become significantly difficult to write parallel applications that fully exploit the processing power, and scale to large systems. Load imbalance, both computationally and communication induced, presents itself as one of the important challenges in achieving scalability and high performance. Problem sizes and system sizes have become so large that manually handling the imbalance in dynamic applications, and finding an optimum distribution of load has become a herculean task. Charm++ [6] provides the user with a run time system that performs dynamic load balancing. To enable Charm++ to perform load balancing in an efficient manner, the user takes certain decisions such as when to load balance and which strategy to use, and informs the Charm++ run-time system of these decisions. Many a times, taking these important decisions involve hand tuning each application by observing various runs of the application.

In this thesis, a Meta-Balancer which relieves the user from the effort of making the load balancing related decisions, is presented. The Meta-Balancer is a part of the Charm++ load balancing framework. It identifies the characteristics of the application, and based on the *principle of persistence* and the accrued information, makes load balancing related decisions. We study the performance of the Meta-Balancer in the context of *leanmd* mini application. We also evaluate the Meta-Balancer in the context of micro benchmarks such as *kNeighbor* and *jacobi2D*.

We also present several new load balancing strategies, that have been incorporated into Charm++, and study their impact on the performance of applications. These new strategies are: 1) RefineSwapLB, which is a refinement based load balancing strategy, 2) CommAwar-

eRefineLB, which is a communication aware refinement strategy, 3) ScotchRefineLB, which is a refinement based graph partitioning strategy using Scotch [2] – a graph partitioner, and 4) ZoltanLB, which is a multicast aware load balancing strategy using Zoltan [3] – a hypergraph partitioner.

To my parents, Gopalakrishnan Parat and Surya Gopalakrishnan, for their love and support.

Acknowledgments

I would like to thank my advisor, Prof. Kale, for his guidance, support and his faith in my capability without which this thesis would not have materialized.

I would like to thank PPL members, Nikhil Jain, Gengbin Zheng and Esteban Meneses, who provided valuable suggestions and insights into the problem.

Finally and most importantly, I would like to thank my family for their support, encouragement and being the pillar of my life.

Table of Contents

List of Figures

Chapter 1

Introduction

With the dawn of petascale, and with exascale in the near future, it has become significantly difficult to write parallel applications that fully exploit the processing power and show scalability. Many parallel programming paradigms have been proposed, and are being used to answer the following key issues: 1) How to obtain scalable performance? 2) How to make parallel programming easy without loss of performance? One of the proposed paradigms, which has gained momentum and shown good results, relies on division of labor between the application writers and the run-time system (RTS). In this paradigm, the application writer is responsible for decomposing the problem at hand as it seems best from the application's perspective. The RTS takes the decomposed application, and runs it on the given system. Charm++ [6], ParalleX [5], FG-MPI [7] and Adaptive MPI [9] are some of the practical implementations of the frameworks that belong to this family of programming paradigm.

In Charm++, computation is over-decomposed into fine-grained tasks or objects, where the number of such tasks is much larger than the number of processors (typically an order of magnitude larger). *Overdecomposition* offers many advantages over its simple *processor based decomposition* counterpart, in which only one task is run per processor. It provides an overlap of computation with communication without any effort from the programmer. It also enables adaptive resource management and dynamic load balancing. In addition, fault tolerance can be achieved with minimal efforts from the programmer. High Performance Computing applications, such as Molecular Dynamics [11], Fractography, Weather Simulation [12], Cosmology [4] to name a few, have fine-grain parallelism and, have been shown to achieve scalability and high performance using Charm++.

Obtaining good performance with fine-grained parallelism essentially requires a smart and adaptive resource allocation to maintain load balance, in terms of computation and communication. Expecting application programmers to handle load imbalance, and perform resource allocation in dynamic applications is unrealistic. To this end, Charm++ provides a comprehensive measurement-based dynamic load balancing framework. Charm++ run-time system performs automatic measurement of task load, processor load and communication pattern, which is subsequently used by the load balancing framework [1] to migrate over-decomposed objects to balance computational load and reduce the communication overhead at runtime.

To enable Charm++ to perform load balancing in an optimal manner, the programmer takes certain decisions related to load balancing, such as when to perform load balancing and which strategy to use. Many a times, coming up with these decisions involve hand tuning each application by observing application characteristics over multiple runs and experimenting with the available options. For example, if the application has significant communication overhead, it is beneficial to use a communication aware load balancing strategy rather than a strategy that balances the computational load. But, if the application is computationally heavy, then using a computational balancing strategy will help improve the performance. Similarly, the decision on frequency of load balancing, i.e., how often to call the load balancer, depends on the overheads and gains associated with load balancing for an application. These overheads include the time spent in computing the new mapping of tasks to processors as well as the time spent in migration. The benefit of performing load balancing is the improved performance in subsequent iterations. However, if the load balancing is done frequently, the overheads incurred may nullify performance gains and may results in worse execution time.

We have observed that, in many dynamic applications, the above mentioned decisions should not be taken statically. The ideal load balancing period may vary as the application behavior changes. Similarly, the best strategy for doing load balancing may change over

the period of time. Moreover, the gains obtained by performing load balancing may or may not be significant, and one may wish to use a strategy which incurs proportionate overheads. However, it is not possible for the application writer to be aware of these dynamic variations and take the correct decisions statically. But the Charm++ RTS has the necessary dynamic information about the characteristics of the application, and about the potential gains, and overheads of performing load balancing. Therefore, it will benefit the application performance if the RTS can automatically decide and control the load balancing decisions. In this thesis, we present a Meta-Balancer which relieves the application writer from these load balancing related decisions. The Meta-Balancer, which is a part of the Charm++ RTS, identifies the characteristics of the application and, based on the principle of persistence and the accrued information, makes these load balancing related decisions. We study the performance of Meta-Balancer in the context of several mini applications such as *leanmd* and NPB. We also present the impact of using the Meta-Balancer on micro benchmarks such as *kNeighbor* and *jacobi2D*.

Chapter 2

Load Balancing in Charm++

Overdecomposition is key to writing scalable applications in Charm++. The *overdecomposition* of computation into fine-grain tasks by the programmer results in the creation of *chares* or virtual objects. From the programmer's perspective, operations are performed by the chares. In general, the programmer need not be concerned with the underlying details, such as the processor on which a chare executes, how the scheduling of multiple chares on a single processor is done etc. Any application run begins with an initial static placement of chares which can be specified by the programmer or left at RTS's discretion. This placement, however, can be changed as the execution progresses by migrating chares from one processor to other processors. Such a migration is needed if the initial placement leads to a load imbalance, either computational or communication induced. An intelligent and need based migration is facilitated by the Charm++ load balancing framework [1]. The framework relies on the *principle of persistence*, which states that in most applications the load either changes slowly, or changes abruptly, but infrequently. In such situations, data from the recent past is a good predictor of the near future, and is being used by Charm++ RTS. For other unexpected situations, the application can provide performance estimates for the chares to supplant the measurements.

2.1 Charm++ Load Balancing Framework

The load balancing framework in Charm++, which is a part of Charm++ RTS, 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 chares onto processors and performs the migration.

The task of chare migration, performed by load balancing framework, is facilitated by Charm++'s object model. A typical Charm++ application consists of parallel C++ objects or chares, which promotes data encapsulation. For each chare, there is a well-defined region of memory on which it operates. This simplifies the packing of data for migration at level of chares, in comparison to a process level migration or thread level migration. Delivery of messages to a processor on which the (migrated) chare resides is handled by Charm++ RTS. As stated earlier, from a programmer's point of view, chares are location independent; messages are usually delivered to chares instead of processors. Thus, there is no processor-specific state that an application writer needs to worry about for chare migration; Charm++ RTS takes care of the run-time state associated with the migrating chares.

Charm++ object model also simplifies the task of application instrumentation. The RTS treats chares uniformly by instrumenting the start and the end time of each method invocation on the chares, rather than deriving execution time from some application-specific knowledge. Further, the Charm++ RTS can record chare-to-chare and collective communication patterns as every communication initiated by chare is eventually handled by the RTS. The RTS also separates the idle time from communication overhead.

To summarize, during an application run, several chare specific and processor specific statistics are collected. For a chare, these statistics include the amount of time it is active on a processor, and the communication (number of messages exchanged and volume of data exchanged) it does with every other chare. For a processor, the framework records the idle time and the background load. Current mapping of chares to processors is also provided by the load balancing framework.

However, the task of initiating load balancing and selection of load balancing strategy is the responsibility of the programmer. In a typical program, the load balancing is initiated

by making *AtSync* call in chares. Once triggered, the load balancing framework takes control on every processor. Thereafter, the 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 processor (if using a hybrid strategy). At these hub(s), the load balancing framework computes the 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, migrations are initiated. Eventually, the chares resumes their execution as *ResumeFromSync* function is invoked on them by the RTS.

2.2 Existing Load Balancers

There are several in-built load balancing strategies in Charm++ that can be used by application developers, some of which are described here for completeness (and for the benefit of the reader to understand the presented results better). A broad classification of these load balancers can be made on two key criteria: 1) Is the current mapping being given any importance? 2) Which performance metric is being given importance? If a load balancer completely ignores the current mapping, and computes a new mapping afresh, we say that it is a *comprehensive* load balancer. In contrast, if a load balancer gives importance to current mapping, and makes a few changes to it to derive a new mapping, we call it a *refinement* load balancer. In terms of application performance metric, a load balancer may favor reducing either computation imbalance or communication overhead. The following strategies are commonly used by application writers in Charm++:

GreedyLB: A comprehensive load balancer which only considers computational load. It is based on the greedy heuristic that maps the heaviest chares on to the least loaded processors until the load on all the processors is close to the average load.

RefineLB: A refinement load balancer is generally used after GreedyLB for computational centric load balancing. It migrates chares from processors with greater than average

load (starting with the most overloaded processor) to those with less than average load. This strategy also aims to reduce the number of chares migrated.

MetisLB: A comprehensive strategy that uses Metis [8] to partition the chares into partitions equal to the number of processors. Charm++ RTS provides the computation and the communication graph to METIS, which uses the recursive graph partitioning algorithm. The aim of this strategy is to minimize the cut, among the partitions, induced in the communication graph.

ScotchLB: A comprehensive strategy that uses Scotch [2] to perform the partitioning. Scotch, like Metis, aims to obtain partitions with minimum cut.

RefineCommLB: A refinement strategy similar to RefineLB, that also considers the communication overhead along with the computation load, and then performs the refinement strategy, in which chares from heavily loaded processors are moved to light weight processors.

2.3 New Load Balancers

Several new load balancing strategies have been added to Charm++ which are described in detail in this section. Among these new strategies, RefineSwapLB, ScotchRefineLB and CommAwareRefineLB are refinement load balancers. ZoltanLB, which is based on Zoltan [3] hypergraph partitioner, is a comprehensive load balancer useful for applications with multicasts (one-to-many communication).

2.3.1 RefineSwapLB

This is a refinement based load balancing strategy, which is an improvement over RefineLB. RefineLB is an algorithm which improves the load balance by incrementally adjusting the existing chare distribution. Refinement is used with an overload threshold, which is typically

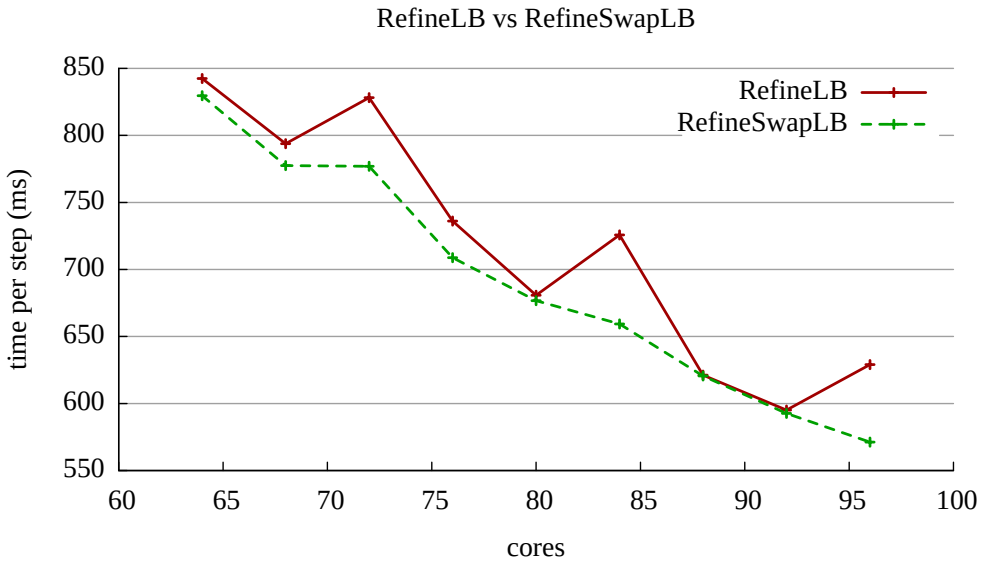

Figure 2.1: Comparison of RefineLB with RefineSwapLB for leanmd

set as 1.003 times the average load across all processors. A processor is considered to be overloaded, if its load is above this threshold. The cost of the algorithm is low because only the overloaded processors are examined, and it results in only a few chares being migrated. But RefineLB may get stuck in a local minima if it cannot move any chare from the most overloaded processor to any other processor without making it overloaded. To handle this scenario, RefineSwapLB has been implemented.

Similar to RefineLB, RefineSwapLB tries to move chares from overloaded processor to less loaded processor without causing them to become overloaded. If it cannot find such a processor, then it swaps chares between processors in order to reduce the load on the overloaded processor without overloading the other processor.

Performance analysis

To compare the performance of RefineSwapLB with RefineLB, we use *leanmd*, in which we observed that RefineLB gets stuck in a local minima. *leanmd* is a molecular dynamics simulation program written in Charm++. We ran the application for non-power of two number of processors, where RefineLB was not able to balance the load effectively. It

can be seen in Figure 2.1, that RefineSwapLB improves the balance, and reduces the per step iteration time incomparison to RefineLB. We attribute this difference to the ability of RefineSwapLB to swap chares when it is unable to move chares from the overloaded processors.

2.3.2 ZoltanLB

ZoltanLB uses Zoltan, a hypergraph partitioner, to partition chares among procesors. Hypergraph is a generalization of a graph, where a hyperedge can connect any number of vertices. Hyperedge is a useful way of representing a group of vertices, which are connected in some respect (an edge is special case of hyperedge which connects two vertices). In a multicast communication, which is common in many applications, a message is sent from a chare to a group of chares. These chares form the vertices of the hypergraph and the multicast forms a hyperedge in the hypergraph. This hypergraph is provided to Zoltan which partitions the chares and maps them onto processors. This load balancer should be used for multicast aware load balancing.

Performance analysis

To compare the performance of ZoltanLB with MetisLB and ScotchLB, we use *leanmd* application. *leanmd* is a molecular dynamics simulation program written in Charm++. In each iteration of *leanmd*, the atoms contained in a cell are sent to every compute that needs them. This transfer of atoms from cells to computes is performed in an efficient manner using multicast.

	1-away (27 chares in multicast)	2-away (45 chares in multicast)
ZoltanLB	5.5	7
MetisLB	8.5	10.15
ScotchLB	8.2	10.9

Table 2.1: Comparison of ZoltanLB with MetisLB and ScotchLB for leanmd

Table 2.1 shows the impact of using ZoltanLB over MetisLB and ScotchLB for multicast aware load balancing. It can be seen that ZoltanLB reduces the number of multicast messages

by mapping chares participating in a multicast onto same processors.

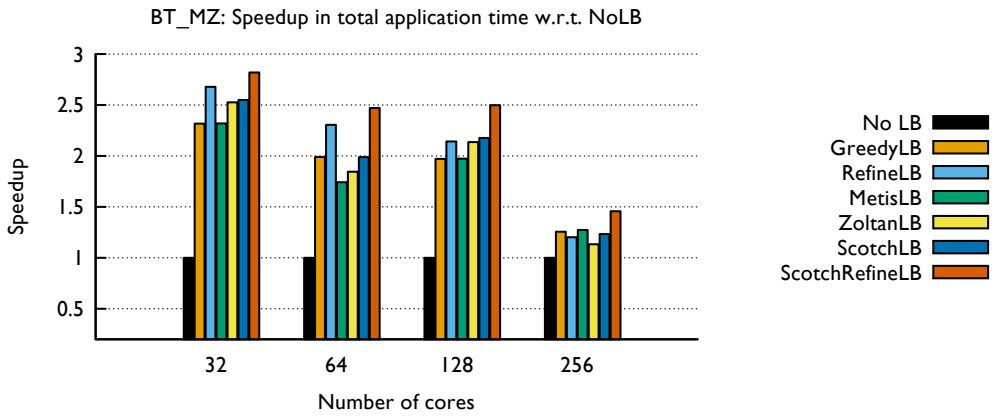

Figure 2.2: Speedup in the total application time for BT MZ on Steele

2.3.3 ScotchRefineLB

A new feature of Scotch 6.0 is the ability to compute remappings of a graph, based on an existing mapping. Every vertex of the graph to be remapped is associated with a fictitious edge that connects it to a fictitious vertex that represents the old partition. Doing so allows us to integrate the migration cost within the existing edge cut minimization process. All of the fictitious edges are weighted, with a weight that represents the cost of migrating the vertex to another partition. A new refinement load balancer, ScotchRefineLB, has been implemented to advantage of this functionality

Performance analysis

We present the results for the *NAS BT* multi-zone benchmark runs on Steele, which is an infiniband cluster located at Purdue University. For these experiments, the number of chares per processor varies with the class of the benchmark used and the system size. For example, a run of class D on 256 processors will have, on an average, four chares per processor. The baseline run, in which no load balancing is performed, is called NoLB. We ran class C on 32 and 64 cores, and class D on 128 and 256 cores. The result for a complete run of *BT-MZ*, in which load balancing is performed once in every 1000 iterations,

is presented in Figure 2.2. The application time primarily consists of the time per step and the migration time. Refinement-based load balancers, RefineLB and ScotchRefineLB, which take the migration cost into account, migrate very few chares. ScotchRefineLB performs best among all the load balancers and shows a speedup of 1.5 to 2.8 in comparison to NoLB. In comparison to other load balancers, ScotchRefineLB performs better by 11%.

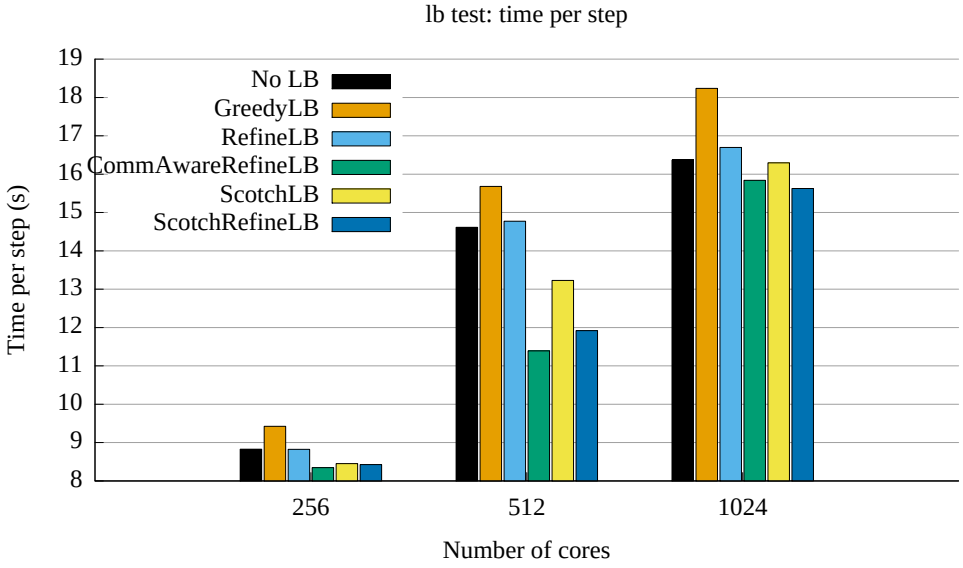

Figure 2.3: Time per step for lbtest

2.3.4 CommAwareRefineLB

This is a refinement based load balancing strategy, similar to RefineLB, which also takes communication into account while deciding to migrate chares. It minimizes the load imbalance by moving chares from overloaded processors to less loaded processors with which it is already communicating. This strategy keeps track of all the processor a chare communicates with and migrates the chare from the overload processor to less loaded processor among the communcating processors. If none of the communicating processors are underloaded, the chare is moved to any available underloaded processor. This reduces the communication overhead as well as the number of chares migrated.

Performance analysis

We present results for *lbtest* on Blue Gene/P in Figure 2.3. *lbtest* is a micro-benchmark distributed with Charm++. It creates a communication graph of chares (which can be random or regular), and assigns variable loads, in the user specified range, to the chares. In each iteration, every chare communicates with its neighbors in the graph, and performs the alloted amount of computation. It can be seen that CommAwareRefineLB gives a performance better than ScotchLB, and comparable to ScotchRefineLB, which are graph based partitioners. In the case of 512 cores, CommAwareRefineLB gives a performance gain of 22% over NoLB (when load balancing is not performed), and 4% over ScotchRefineLB. CommAwareRefineLB, apart from reducing the number of chares migrated, also takes lesser time for strategy in comparison with graph based partitioners.

Chapter 3

Meta-Balancer

The current load balancing framework in Charm++ is application driven. It is invoked when the application makes a call to *AtSync*, and uses a strategy specified by the user as a command line argument. In this chapter, we introduce Meta-Balancer, which is a new component added to the load balancing framework of Charm++ RTS. It is responsible for complete automation of the load balancing process, and makes it RTS driven.

3.1 Motivation

Understanding the characteristics of an application, and taking appropriate load balancing decisions is important to improving its performance. Some of these load balancing related decisions, which programmer has to take are, the frequency of load balancing and the strategy to use. In the case of a dynamic application, it becomes challenging to identify the characteristics of the application and take these decisions a priori. Many a times, application characteristics change over time, and affect the load balancing related decision making. For such applications, it is difficult and suboptimal to decide upfront on how frequently the load balancing should be done and which type of load balancing strategy should be used. To this end, we propose Meta-Balancer which relieves application writers of such decision making related to load balancing and improve the overall performance.

3.2 Iterative Applications

Meta-Balancer has been written keeping in mind applications which adhere to the *principle of persistence* (§2). From Meta-Balancer's perspective, an application run consists of multiple iterations. The load balancing framework is triggered frequently using *AtSync* at suitable places by the application writer. Meta-Balancer takes control at each *AtSync* call and performs various tasks such as statistics collection, decision making, load balancing, migration etc., each of which is described in detail in the following sections.

3.3 Decisions in Meta-Balancer

The key decisions related to load balancing which Meta-Balancer takes, based on the characteristics of the application, are:

- Frequency of load balancing
- Adaptive triggering of load balancing
- Strategy Selection
  - Communication vs Computation strategy
  - Comprehensive vs Refinement strategy

3.4 Statistics Collection

The statistics collected by Charm++ RTS are chare specific and processor specific (§2.1). Some of these statistics such as the chare load, the processor load and the processor idle time are continuously monitored by the Meta-Balancer. These statistics presents a holistic picture of the application. Meta-Balancer collects these minimal statistics at a central location, and based on the characteristics of the application, controls the load balancing decisions as described in this section.

3.4.1 Existing Synchronous Statistics Collection

In Charm++, a special set of chares called *LB Manager*, are initialized by the Charm++ RTS on each processor. During the execution of an application, the *LB Manager* on each processor, monitors the chares executing on that processor. It collects the necessary statistics and stores them using *LB Database*, which is another set of chare array initialized by the RTS. When a particular chare is being executed, RTS notifies the *LB Manager* to update the corresponding timers. Similarly, when RTS receives request for communication by a chare, it intimates the *LB Manager* which make corresponding changes in the *LB Database*.

In the exising framework, when *AtSync* is called, a synchronous gather of processor and chare statistics is performed. The collection of statistics at a central processor requires two barriers - local and global. On each processor, a local barrier is invoked on the chares that reside on it. Once all the chares have reached the local barrier on a processor, it takes part in a global barrier, during which the statistics are sent to the central processor. Having obtained the statistics, the central processor makes decisions regarding the placement of chares onto processors and informs rest of the processors. Currently, the statistics sent by the processor to the central processor includes all the statistics that have been collected since last load balancing. Note that, in the current framework, the invocation of load balancing using *AtSync* is determined by the programmer and is an infrequent event.

3.4.2 Asynchronous Collection of Statistics via Reduction

For the Meta-Balancer to control the load balancing decision, frequent aggregation of statistics at a central location is necessary. However, frequent collection of entire statistics is time consuming and results in substantial overheads. In order to prevent these overheads, only minimal statistics (§3.4) needs to be collected periodically. Moreover, instead of performing a gather of these minimal statistics at a central location, a reduction whose results are delivered to the central location is sufficient.

It is to be noted that, with the minimal statistics being reduced periodically and frequently, it is not advisable to have a local and a global barrier. Presence of a frequent local and global barrier may result in substantial overheads and a slow-down of the application. The overheads and slow-down has been mitigated by the use of Charm++'s asynchronous reduction of the minimal statistics. Reduction, which is done via a spanning tree in Charm++, ensures that there is not much communication introduced by the Meta-Balancer.

3.4.3 Extracting per Iteration Information

At the end of every iteration, when *AtSync* is called by a chare, it updates its load information by intimating *AdaptiveLB Manager*, which is a chare array initiated by RTS for Meta-balancer. Once the chare has updated its load information, it moves onto the next iteration. Therefore, on a processor, different chares may be executing different iterations. This asynchronous scheme, without local barrier, accrues precise information about per chare per iteration. But it complicates the process of obtaining processor specific information. To obtain the utilization statistics of the processor for an iteration, it is required to approximate the idle time for that iteration. The idle time for a processor for an iteration i is obtained using

$$idle\_time = (total\ idle\ time\ until\ all\ chares\ finish\ the\ iteration) \times \left(\frac{number\ of\ chares}{sum\ of\ current\ iteration\ of\ chares}\right) \tag{3.1}$$

where the chares being considered are the ones residing on the processor

This is approximately equivalent to idle time till now divided by the total iterations. Utilization is calculated using

$$utilization = \frac{idle\ time}{processor\ load + idle\ time}. \tag{3.2}$$

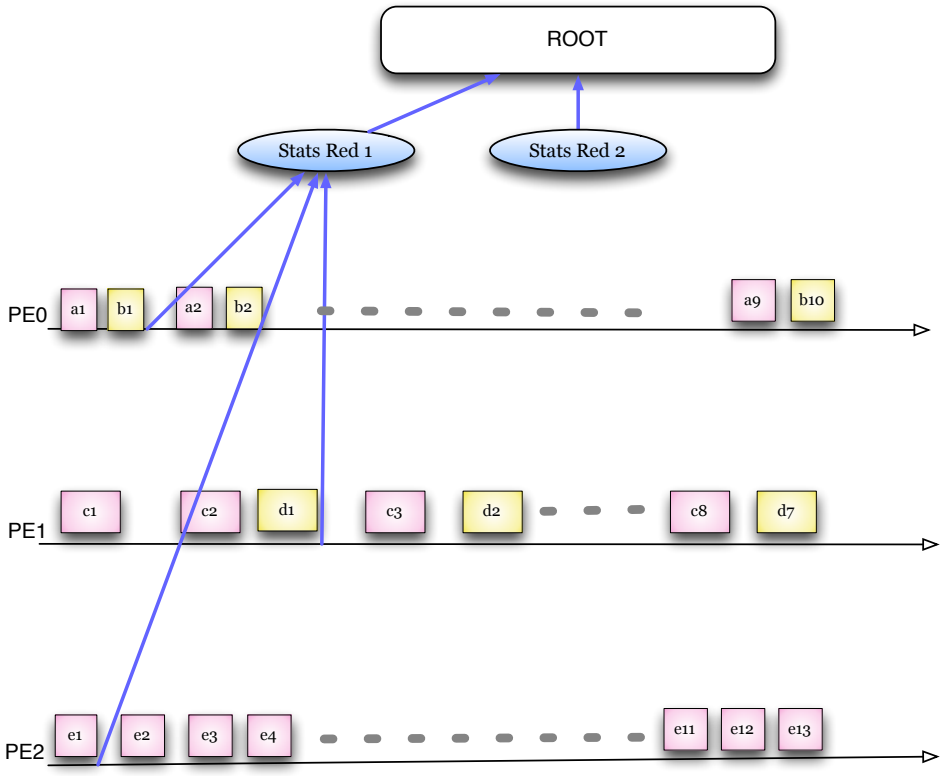

Figure 3.1: Periodic Statistics Collection

Once all the chares residing on a processor have updated their load for an iteration, the *AdaptiveLB Manager* contributes minimal statistics to the reduction, which delivers the reduced statistics to the central processor as shown in Figure 3.1. The reduction is performed in such a manner that the central processor receives the average load per processor, maximum processor load and the minimum utilization ratio.

3.5 Load Balancing Period

Load balancing is performed to remove the load imbalance, which results in the improvement of application performance. However, there is an overhead or cost associated with load balancing. The cost consist of the time spent in the load balancing strategy to find new mapping of chares onto processors, and the cost of migrating them if needed. Therefore, if

load balancing is performed very frequently, it may degrade the performance or reduce the gains from performing load balancing. In contrast, if load balancing is performed very infrequently, load imbalance may result in suboptimal performance. The optimal performance is obtained only if load balancing is performed at an ideal frequency, where the gains obtained from load balancing is maximized despite the incurred overheads.

3.5.1 Ideal LB Period

In this section, we describe the theory of identifying the ideal load balancing period which is used by Meta-Balancer. Let,

τ be the ideal load balancing period,

γ be the total iterations an application executes,

Γ be the total application execution time, and

θ be the cost associated with load balancing

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

$$y = mx + c_m \tag{3.3}$$

where m is the slope and c_m is the offset with respect to the maximum load value after load balancing.

Similarly, let the average time per iteration, approximately equal to the average load on the processors, be represented by the line equation

$$y = ax + c_a \tag{3.4}$$

where a is the slope and c_a is the offset.

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:

$$\begin{aligned}
\Gamma &= \frac{\gamma}{\tau} \times (\int_0^{\tau} (mx + c_m)dx + \theta) + \int_0^{\gamma} (ax + c_a)dx \\
\Gamma &= \frac{\gamma}{\tau} \times (\frac{m\tau^2}{2} + c_m\tau + \theta) + \gamma \times (\frac{a\gamma}{2} + c_a) \\
\Gamma &= \gamma \times (\frac{m\tau}{2} + c_m + \frac{\theta}{\tau} + \frac{a\gamma}{2} + c_a)
\end{aligned}$$

In order to minimize Γ, we differentiate it with respect to τ , and obtain the following value of τ, which is used by the Meta-Balancer as the ideal load balancing period.

$$\begin{aligned}
\frac{d}{d\tau}(\Gamma) &= \gamma \times (\frac{m}{2} - \frac{\theta}{\tau^2}) = 0 \\
\tau &= \sqrt{\frac{2\theta}{m}}
\end{aligned} \tag{3.5}$$

Note that, the load balancing period is calculated and continuously refined, using Eq 3.5, as the application executes. At any instant in the application run, the current value of load balancing period is used by the Meta-Balancer to determine when next to perform load balancing.

3.5.2 Verification

The relationship between the total application time and the load balancing period for *jacobi2D* is presented in Figure 3.2. *jacobi2D* is a benchmark that performs a 5-point stencil computation over a given two dimensional set of points. It has a communication pattern, which is representative of communication in Weather Research and Forecast Model [10]. For this experiment, the total load and the load distribution is continuously changed as the benchmark executes. The experiment was run on Intrepid, which is an IBM's Blue Gene/P

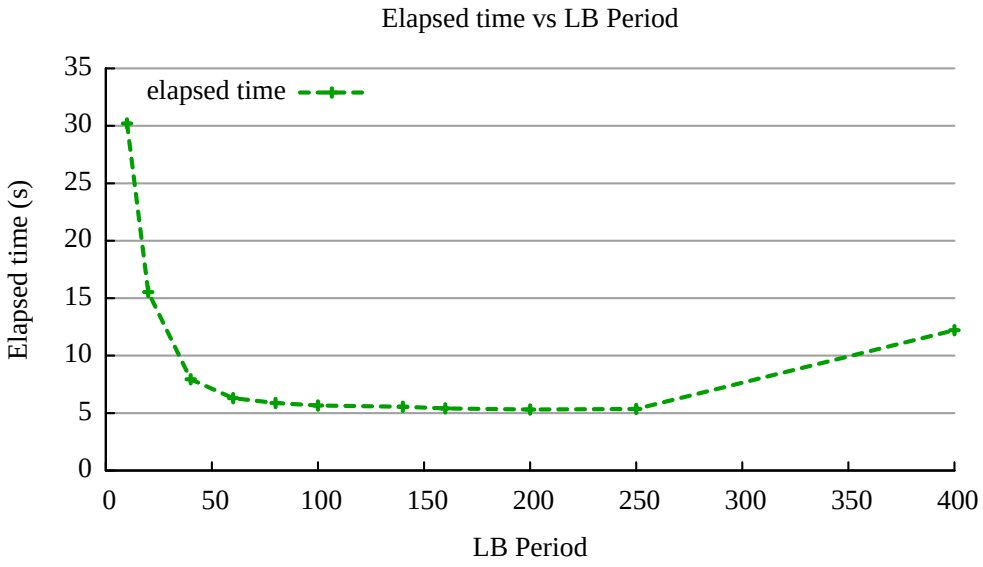

Figure 3.2: Elapsed time vs LB Period for *jacobi2D*

machine installed at ANL. It can be observed from the Figure 3.2 that, if the load balancing is performed frequently, it leads to low performance. On the other hand, if load balancing is done infrequently, it results in bad performance due to load imbalance. However, there exists an ideal LB period in the range of 150 to 250 iterations, where the application achieves maximum gains due to load balancing.

For the same benchmark, Meta-Balancer calculates the ideal load balancing period based on the theory described earlier (§3.5.1). Figure 3.3 shows the LB period identified by the Meta-Balancer. As seen in the figure, the Meta-Balancer identifies that the load balancing has to be performed initially in the 14^{th} iteration, because of the high computation load imbalance. Following the initial load balancing, based on the cost of LB, Meta-Balancer identifies an LB period of 210 and 180 iterations. This is in the range of the ideal period as shown in Figure 3.2, and gives the best performance possible. This experiment demonstrates that the Meta-Balancer, by observing the application behavior, is able to extract the maximum possible gains due to load balancing, which otherwise is obtained by trying multiple values in different runs.

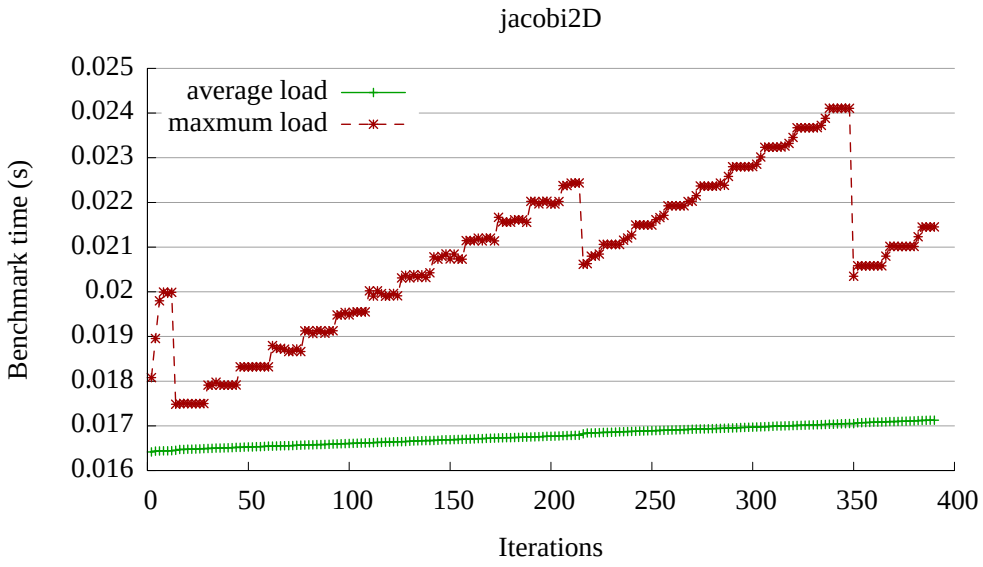

Figure 3.3: Identifying LB Period for *jacobi2D*

3.5.3 Implementation: Computation Intensive Application

In this section, we present the details of how Meta-Balancer identifies the load balancing period for a computationally intensive application. As stated earlier, Meta-Balancer periodically collects minimal statistics (§3.4.3). Once these minimal statistics have been collected for a few iterations, a linear-extrapolation of these values is done, relying on *the principle of persistence*. Hence, we obtain curves that represent predicted maximum load and the predicted average load for the application run. Using them, we obtain the maximum load curve equation with respect to the average curve (Eq 3.3 in §3.5.1). Finally, the cost of load balancing along with the other calculated values, are substituted in Eq 3.5 to obtain the ideal LB period. This LB period is broadcast to all the processors. As more statistics are collected, the LB period is refined and the refined value is sent to all the processors.

When the load balancing is performed, the expectation is that the load balancing strategy will make the maximum load equal to or close to the average load. Hence, we use the maximum load curve with respect to the average curve as the curve required by Eq 3.3. But, it can so happen that the load balancing strategy is not able to balance beyond a

certain threshold. This inefficiency in LB strategies needs be incorporated into the LB period calculation. We deal with this situation by shifting the average curve up by the imbalance ratio which a load balancing strategy could not even out. An expected value of the imbalance ratio is obtained by performing some post processing on the new mapping and the statistics collected so far. The expected maximum load after load balancing is calculated to be the predicted average load times the imbalance ratio. Thereafter, maximum load curve is calculated with respect to this expected load, and the LB period is calculated as described in the previous section (§3.5.1). This adjustment to the expected maximum load after load balancing prevents frequent load balancing in cases, where the load balancer cannot improve the balance of load.

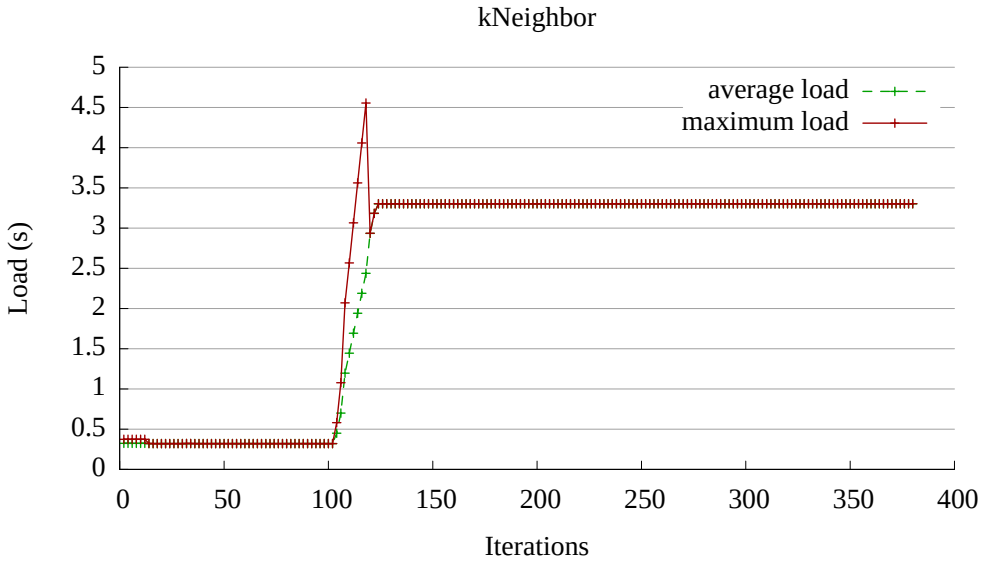

Figure 3.4: Dynamic triggering of LB for *kNeighbor*

3.5.4 Implementation: Communication Intensive Application

For communication intensive application, deciding LB period based on only computation load will result in suboptimal performance. However, we observed that a small variation of the above scheme can be used for such application. In addition to considering computation

load of processors for extrapolation, and calculation of LB period, Meta-Balancer also adds the idle time observed on each processor to the predicted maximum load value, and to the predicted average load value. This ensures that the time spent while waiting for data from other processors is accounted for in the ideal load balancing period calculation.

3.5.5 Dynamic Triggers

In dynamic applications, the load in the system can change unexpectedly. If the load balancing period has been decided, the refinement based on the new statistics collected can take time. It can result in a delay in performing load balancing, which reduces the performance gains due to load balancing. Since, the Meta-Balancer periodically monitors the load in the system, it is in a position to trigger load balancing immediately in such scenarios. In the current set up, if the Meta-Balancer notices that the imbalance ratio, given by $\frac{maximum\ load}{average\ load}$, is beyond a threshold of 1.1, it triggers the load balancing as soon as possible.

The verification of this aspect of Meta-Balancer was done using *kNeighbor* benchmark, in which a sudden load imbalance was introduced in middle of the benchmark run. It can be seen in Figure 3.4 that, when the load in the system changes suddenly, Meta-Balancer identifies this change in the load distribution of the application and triggers the load balancing process.

3.6 LB Period Intimation

Once the ideal load balancing period has been identified (or a dynamic trigger is identified), all the processors needs to be intimated about it, so that they take part in the load balancing process. However, this load balancing period may not be suitable for performing load balancing, as it may lead to application getting hung, as described in detail in the next section.

3.6.1 Consensus on LB Period

During the load balancing process, all the processors reach a local and a global barrier, and send their detailed statistics to a central processor. Centralized load balancing process involves executing the strategy to identify the new mapping of chares on to the processors, and migrating them. When the load balancing is carried out, we expect the chares to remain idle. To enforce this, whenever a chare finishes an iteration, at *AtSync*, it checks if it needs to go into load balancing stage. On reaching the communicated LB period, the chares enter the load balancing stage. If it is not time for load balancing, the chare resumes its work. However, chares on a processor can be in different iterations depending on their load, the processor load and the communication dependencies. To ensure that the application does not hang, all the chares have to be at the same iteration to perform centralized load balancing. An example, where an application can hang if this constraint is not met is as follows. Let the ideal load balancing period intimated be iteration i. Consider a chare a, which finished iteration i, and hence enters the load balancing stage. Consider another chare b, which is executing iteration $i + 1$, and is waiting for a message from chare a, which chare a was to send in iteration $i + 1$. Since the chare a has become idle, b remains waiting for the message from a and cannot complete its iteration. However, load balancing cannot proceed until b enters the load balancing phase, and sends its data. This causes the application to hang.

To avoid such a scenario, which causes the application to hang, all the chares need to reach a consensus on the iteration number to enter the load balancing stage. Since the chares can be in different iterations, we use the following scheme shown in Figure 3.5 to obtain consensus.

When the central processor identifies the ideal load balancing period, which is a tentative LB period, it broadcasts this information to all the processors. On receiving the tentative LB period, the processors informs the root with information about the maximum iteration, which any chare residing on the processor is executing. Upon receiving the maximum iteration of

all the chares in the system, the root decides the final load balancing period. If the tentative LB period is beyond the maximum iteration, i.e. no chare has reached the tentative LB period, the LB period is set to the tentative value. But, if any chare has gone past the ideal load balancing period, the final load balancing period is set to be that maximum iteration. This final LB period is broadcasted to all the processors.

Meanwhile, on each processor, whenever a chare finishes its iteration, before entering the next iteration, it checks if it has reached the maximum of the tentative LB period or the value the processor had sent. If it has not reached the limit, it resumes its work. If it has reached the limit, it waits for the final verdict on the LB period. It is said to be in a *pause* state. On receiving the final LB period, a chare in *pause* state, decides to either resume its work, or enter the load balancing stage. Any chare which reaches the final lb period, enters into LB stage. This scheme ensures that all the chares arrive at a consensus regarding the load balancing period and enters the LB stage at the same iteration.

3.6.2 Dynamic Refinement of LB Period

As the statistics collection proceeds, the predicted load might change and become refined. This in turn leads to the refinement of ideal LB period. When the ideal LB period changes, this is intimated to all the processors and chares using the scheme described in the previous section (§3.6.1). Unless any chare has entered the load balancing phase, it is possible to extend the LB period. Similarly, the LB period can be reduced from the previously announced period if no chare has gone beyond that period.

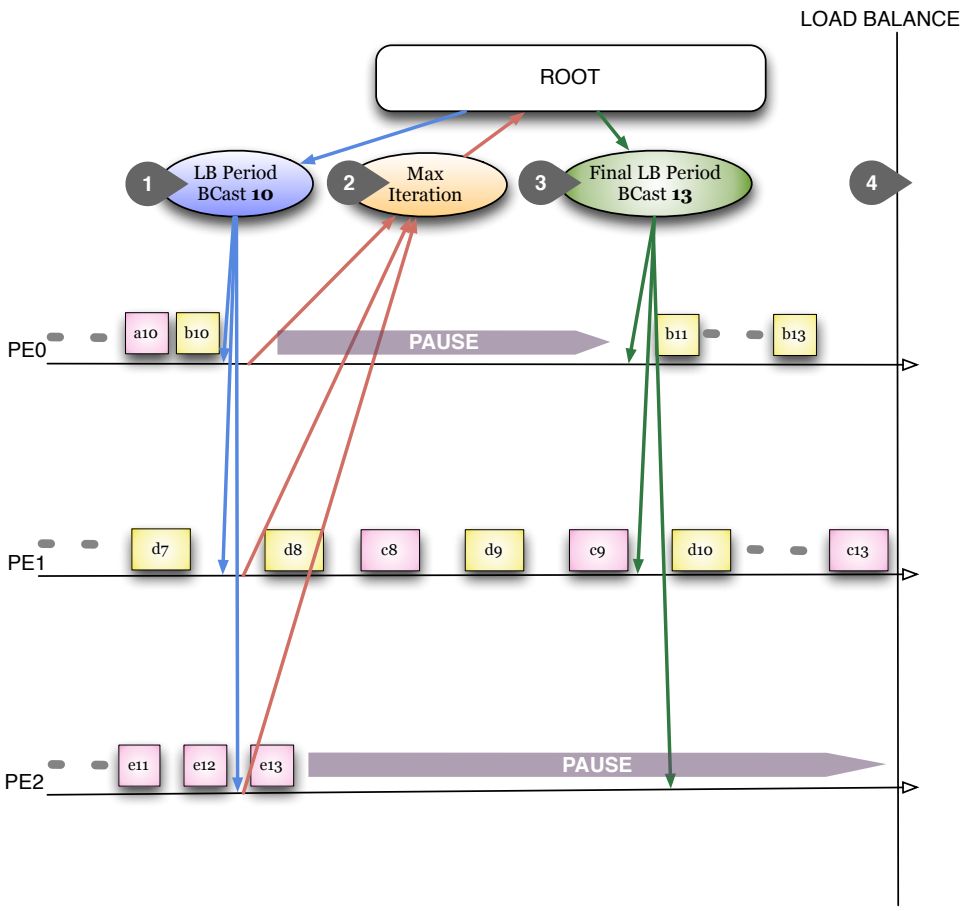

Figure 3.5: Intimating the calculated LB period

3.7 Strategy Selection

3.7.1 Communication vs Computation Strategy

Data and their associated computations are distributed across several interconnected processing elements (processors) which work in parallel. Accessing data on remote processors requires inter-processor communication. Consequently, the efficient use of such distributed memory parallel machines requires spreading the computation load evenly across different processors and minimizing the communication overhead. Depending on the type of application, preference should be given to load balancing strategies that either minimizes load imbalance or that minimizes communication overhead.

Applications have various characteristics that helps determine whether it is communication intensive or computation intensive. Presence of idle time is an indicator of the need to perform load balancing. This idle time can be either due to computational load imbalance, or due to high communication volume. In presence of idle time, Meta-Balancer uses $alpha-beta$ cost of an application to determine if it is a communication intensive application. We define $alpha - beta$ cost of an application as

$$alpha - beta = \alpha * m_n + \beta * m_v \tag{3.6}$$

where m_n is number of messages sent

m_v is the total number of bytes transferred

α is per message start up cost, and

β is per byte send cost

If the $alpha-beta$ cost of an application is at least 10% of the total load, it indicates that this is a communication intensive application. To minimize the communication overhead, graph partitioner based load balancing strategies, such as MetisLB or ScotchLB, are used by the Meta-Balancer. In case a refinement based strategy is required, CommAwareRefineLB or ScotchRefineLB, is used.

If an application is not communication intensive, then imbalance in load will result in heavy degradation of performance. For such applications, strategies that balances out the load, such as GreedyLB, RefineLB or RefineSwapLB, are used by the Meta-Balancer. Imbalance in load can be identified by the ratio of $\frac{maximum\ load\ on\ a\ processor}{average\ load\ per\ processor}$. If this ratio is beyond a threshold of 1.1, then there is considerable imbalance.

These features of Meta-Balancer were verified by using *leanmd* and $kNeighbor$. *leanmd* is a computation intensive benchmark, in which load imbalance is high when the application begins. However, there is very little variation in the load of a chare through out the application run. As shown in Figure 3.6, Meta-Balancer identifies *leanmd* to be a compu-

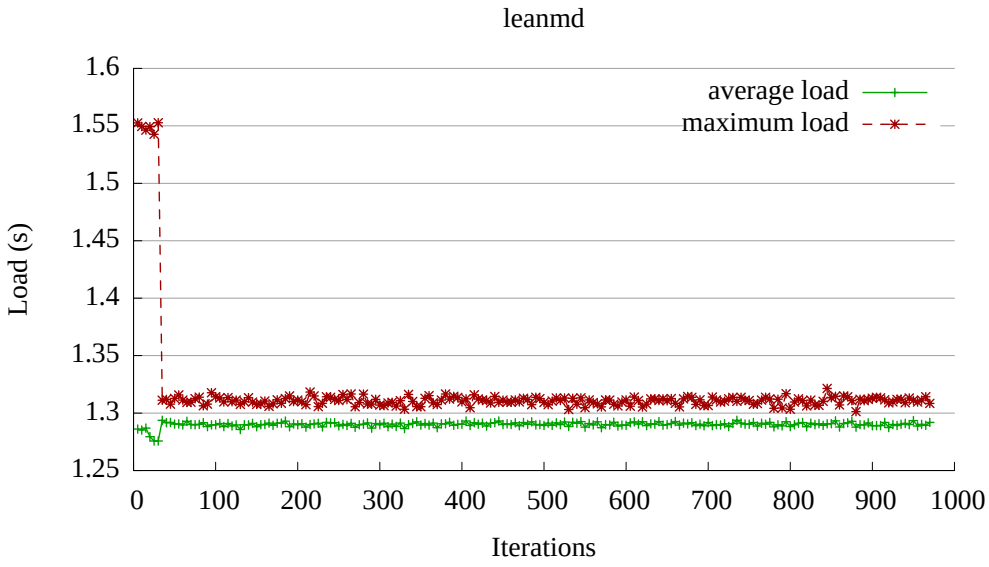

Figure 3.6: *leanmd* mini-application

tation intensive benchmark, and invokes GreedyLB, which results in perfect load balance and low per iteration time. In contrast, for a *kNeighbor* run with large communication, Meta-Balancer invoked MetisLB, and reduced the iteration time as shown in Figure 3.7.

3.7.2 Refinement vs Comprehensive Strategy

The load balancing strategies map chares onto the processors based on the computation load or on the communication pattern. There are two categories of algorithms used for load balancing. One is comprehensive algorithms, also called as from-scratch, that does not take the existing chare mapping into account. The other is the refinement-based algorithms that takes the existing mapping into account in order to limit the number of chare movements. Reducing the chare movements leads to lesser migration cost. The computation based comprehensive strategies include GreedyLB and computation based refine strategies include RefineLB and RefineSwapLB. In the case of communication aware load balancing strategies, comprehensive strategies include MetisLB, ScotchLB and ZoltanLB, and refinement based strategies include ScotchRefineLB and CommAwareRefineLB. Meta-Balancer

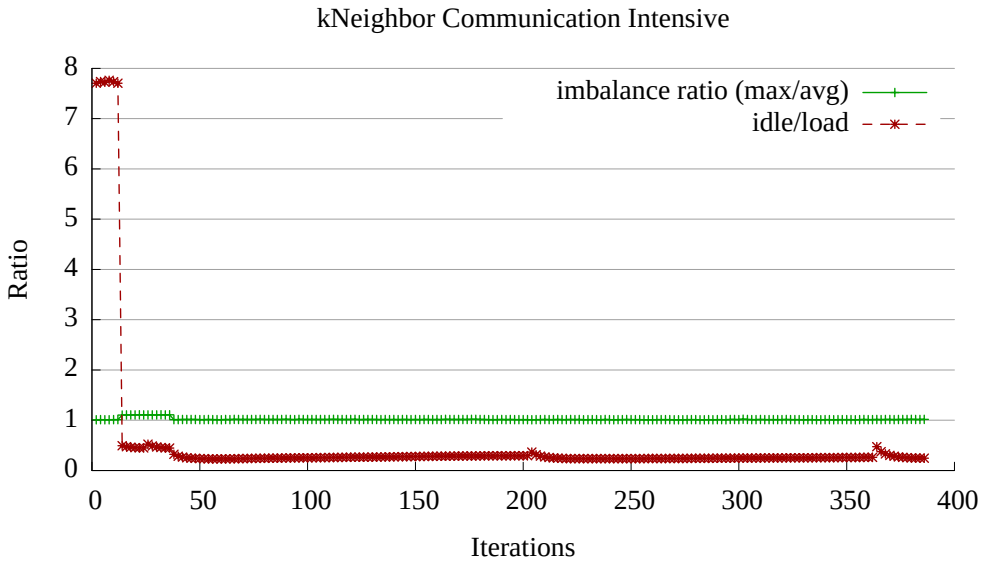

Figure 3.7: *kNeighbor* with high communication

begins every application run with a call to comprehensive strategy. Thereafter, refinement strategies are invoked unless it observes that refinement strategies performed poorly in comparison to comprehensive strategies. The Meta-Balancer takes history into account, and invokes the strategies based on the quality of results they provided.

3.7.3 Overall Strategy Selection

Figure 3.8 shows the flowchart describing the strategy selection for communication vs computation, and comprehensive vs refinement. Whenever Meta-Balancer receives statistics, it checks to see if there is load imbalance. Load imbalance is calculated by $\frac{maximum\ load}{average\ load}$. If the load imbalance is within certain threshold, currently set to 1.10, it indicates that there is not much load imbalance in the application. But, this does not necessarily mean that there is no communication overhead. At this point, Meta-Balancer checks to see if the utilization is low (or idle time is high). If there is no idle time, Meta-Balancer concludes that the application is perfectly balanced. Otherwise, it calculates the $alpha - beta$ cost. If the $alpha - beta$ cost is more than 10% of the total load, it invoke the communication minimization LB strategy.

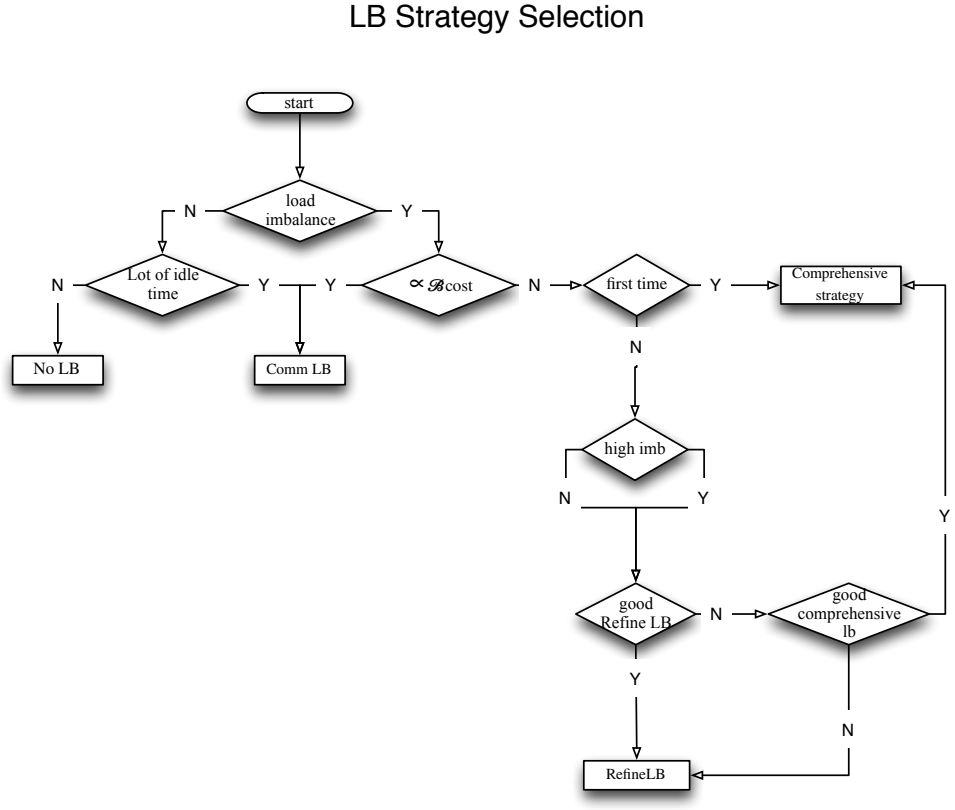

Figure 3.8: Flowchart describing strategy selection

If there is not significant amount of communication in the application, and there is load imbalance, it invoke strategies that balances the load. It is preferable to use refinement based strategies instead of comprehensive strategies since they reduce the number of chare migrations. Hence, only if it is the first time that the load balancing is called, then comprehensive strategy is invoked. In the consecutive runs, refinement-based strategies are preferred over comprehensive, if their performance is comparable.

Chapter 4

Conclusions and Future Work

Load imbalance, both computationally and communication induced, is a key factor that affects performance and scalability of an application. Leaving it to the application programmer to manually handle this imbalance in a dynamic application, and to find an optimum load distribution throughout the run of the application, is unreasonable and inefficient. Charm++ provides the user with a RTS that performs dynamic load balancing. In this thesis, we proposed a Meta-Balancer, which is a part of Charm++ RTS, that controls the load balancing decision based on the application characteristics. We studied the performance of Meta-Balancer in the context of multiple mini-applications and micro-benchmarks. It was shown that Meta-Balancer is able to identify an ideal load balancing period which gives the maximum performance gains. In the case of *leanmd* mini-application, it could be seen that after the initial load balancing, the Meta-Balancer identified that there was no significant imbalance to trigger load balancing and the ideal load balancing period was long. On the other hand, in *kNeighbor*, the sudden change in the load imbalance caused Meta-Balancer to trigger load balancing. We also observed that Meta-Balancer was able to identify the key characteristics of the application, either computation or communication bound, and select the strategy accordingly. This shows that Meta-Balancer is successful in controlling and selecting load balancing decisions based on the characteristics of the application without much overhead.

Future work involves expanding the strategy selection to Hierarchical vs Centralized strategy. Another area of exploration is the use of topology-aware mapping strategies vs non topology-aware.

References

[1] Robert K. Brunner, *Versatile automatic load balancing with migratable objects*, TR 00-01, January 2000.

[2] Cdric Chevalier, Franois Pellegrini, Inria Futurs, and Universit Bordeaux I, *Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework*, In Proceedings of Euro-Par 2006, LNCS 4128:243252, 2006, pp. 243–252.

[3] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, and Luis G. Gervasio, *New challenges in dynamic load balancing*, Appl. Numer. Math. **52** (2005), no. 2–3, 133–152.

[4] Filippo Gioachin, Amit Sharma, Sayantan Chakravorty, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn, *Scalable cosmology simulations on parallel machines*, VECPAR 2006, LNCS 4395, pp. 476-489, 2007.

[5] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling, *Parallex an advanced parallel execution model for scaling-impaired applications*, ICPPW '09: Proceedings of the 2009 International Conference on Parallel Processing Workshops (Washington, DC, USA), IEEE Computer Society, 2009, pp. 394–401.

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

[7] Humaira Kamal and Alan Wagner, *FG-MPI: Fine-Grain MPI for multicore and clusters*, The 11th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDESC), IEEE, April 2010.

[8] George Karypis and Vipin Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, TR 95-035, Computer Science Department, University of Minnesota, Minneapolis, MN 55414, May 1995.

[9] Orion Lawlor, Milind Bhandarkar, and Laxmikant V. Kalé, *Adaptive mpi*, Tech. Report 02-05, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2002.

[10] Michalakes, J., J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang, *The Weather Research and Forecast Model: Software Architecture and Performance*, Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology, October 2004.

[11] James Phillips, Gengbin Zheng, and Laxmikant V. Kalé, *Namd: Biomolecular simulation on thousands of processors*, Workshop: Scaling to New Heights (Pittsburgh, PA), May 2002.

[12] Eduardo R. Rodrigues, Philippe O. A. Navaux, Jairo Panetta, Celso L. Mendes, and Laxmikant V. Kale, *Optimizing an MPI Weather Forecasting Model via Processor Virtualization*, Proceedings of International Conference on High Performance Computing (HiPC), 2010.