© 2012 Harshitha Menon

METABALANCER AUTOMATED LOAD BALANCING BASED ON APPLICATION CHARACTERISTICS

BY

HARSHITHA 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 programs that exploit the processing power and scale the applications. Load imbalance presents itself as one of the significant challenges to achieving scalability and high performance. Manually handling the imbalance in dynamic applications and finding an optimum distribution of load becomes a herculean task. Charm++ provides the user with a run time system that can carry out dynamic load balancing. To enable Charm++ to carry out load balancing, the user takes certain decisions related to the load balancing period and strategy and informs these decision to the Charm++ run-time system. Many a times, this involves hand tuning each application by observing various runs of the application. In this thesis, we present a Meta-Loadbalancer which is relieves the user from load balancing decisions. The Meta-Loadbalancer, which is a part of the Charm++ run-time system, identifies the characteristics of the application and, based on the principle of persistence, makes load balancing decisions. We study the performance of Meta-Balancer in the context of kNeighbor benchmark and mini applications like leanmd, barnes-hut, NPB, jacobi.

We also present new load balancing strategies implemented in Charm++ and study their impact on the performance of applications. The new strategies are RefineSwapLB, which is a refinement based load balancing strategy, CommAwareRefineLB, which is a communication aware refinement strategy, ScotchRefineLB, which is a refinement based graph partitioning strategy using Scotch and ZoltanLB, which is a multicast aware load balancing strategy using Zoltan which is a hypergraph partitioner.

To my parents 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 and Gengbin Zheng, who provided valuable suggestions and insights into the problem. I would like to thank my fiance Saurabh for his love and support. Finally and most importantly, I would like to thank my parents for their support, encouragement and being the pillar of my life.

Table of Contents

List of Tables . . . . **vi**

List of Figures . . . . **vii**

Chapter 1 Introduction **1**

Chapter 2 Load Balancing in Charm++ **3**
- 2.1 Charm++ Load Balancing Framework 3
- 2.2 Existing Load Balancers 4
- 2.3 New Load Balancers 5
  - 2.3.1 RefineSwapLB 6
  - 2.3.2 ZoltanLB 7

Chapter 3 Meta-Balancer **8**
- 3.1 Motivation 8
- 3.2 Statistics Collection 9
  - 3.2.1 Existing synchronous statistics collection 9
  - 3.2.2 Asynchronous Collection of Statistics via Reduction 10
  - 3.2.3 Extracting per iteration information 10
- 3.3 Load Balancing Period 11
  - 3.3.1 Ideal LB Period Calculation 12
  - 3.3.2 Computation Intensive Application 12
  - 3.3.3 Dynamic triggers 14
- 3.4 LB Period Intimation 14
  - 3.4.1 Dynamic Refinement of LB Period 16
- 3.5 Strategy Selection 17
  - 3.5.1 Communication vs Computation strategy 17
  - 3.5.2 Refinement vs From Scratch strategy 18

Chapter 4 Conclusions and Future Work **20**

References **21**

List of Tables

List of Figures

2.1 Comparison of RefineLB with RefineSwapLB for leanmd 6

3.1 periodic stats collection 11
3.2 Elapsed time vs LB Period for Jacobi 13
3.3 Identifying LB Period for Jacobi 14
3.4 Dynamic triggering of LB for kNeighbor benchmark 15
3.5 Intimating the calculated LB period 17
3.6 flowchart describing strategy selection 19

List of Abbreviations

LB	Load Balancer

Chapter 1

Introduction

With the dawn of petascale and with exascale in the near future, it has become significantly difficult to write parallel programs that exploit the processing power and scale the applications. In the case of programming models, such as CHARM++ [2], ParalleX [1], FG-MPI [3], Adaptive MPI [4] and others, computation is over-decomposed into ne-grained tasks or objects, where the number of such tasks, n is much greater than p. High Performance Computing applications, such as Molecular Dynamics and Fractography, have fine-grain parallelism to achieve scalability and high performance. Having such fine-grained parallelism would require a smart and adaptive resource allocation to maintain load balance. Expecting application programmers to handle load imbalance and carry out resource allocation in dynamic applications is unrealistic. Charm++, a migratable objects based programming model, provides a measurement-based dynamic load balancing framework. Charm++ runtime system performs automatic measurement of task load, processor load and communication pattern which is then used by the load balancing framework to migrate over-decomposed objects to balance computational load and reduce the communication overhead at runtime. To enable Charm++ to carry out load balancing, the user takes certain decisions related to the load balancing period and strategy and informs these decision to the Charm++ run-time system. Many a times, coming up with these decisions involve hand tuning for each application by observing application characteristics and experimenting with the options. For eg: If the application has communication overhead, then it is beneficial to use graph partitioner based load balancing strategy rather than the strategy that balances the load. But if the application is computationally heavy, then using a computational balancing strategy, like

greedy, would help improve the performance. The other decision that is left to the user is the frequency of load balancing, ie how often to call the load balancer. Load balancing incurs cost of carrying out the strategy to find the new mapping as well as the cost of migration. After doing load balancing, the performance of the application improves. But, if the load balancing is done frequently, then the cost of doing the load balancing supersedes the benefit. Therefore, there is an ideal load balancing period at which we obtain the best performance. Since the run-time system has the required information to identify the characteristics of the application, it will benefit the application programmer if the run-time system can automatically decide and control the load balancing decisions. In this thesis, we present a Meta-Balancer which is relieves the user from load balancing decisions such as when to do load balancing and which strategy to use. The Meta-Balancer, which is a part of the Charm++ run-time system, identifies the characteristics of the application and, based on the principle of persistence, makes load balancing decisions. We study the performance of Meta-Balancer in the context of kNeighbor benchmark and mini applications like leanmd, barnes-hut, NPB, jacobi.

Chapter 2

Load Balancing in Charm++

Applications written in CHARM++ over-decompose their computation into virtual processors or objects called chares which are then mapped on to physical processors by the runtime system. This initial static mapping can be changed as the execution progresses by migrating objects to other processors if the simulation leads to a load imbalance. This is facilitated by a load balancing framework that instruments the application to obtain the computational loads and the communication graph of the objects and uses them to make informed decisions for migrating objects. Measurement-based load balancing is effective when the load and communication pattern of the application either change slowly, or change abruptly but infrequently. In these situations, data from the recent past is a good predictor of the near future. For other situations, the application can provide performance estimates for the objects to supplant the measurements.

2.1 Charm++ Load Balancing Framework

CHARM++s object model is particularly well-suited to object-based load balancing. CHARM++ application is written in parallel C++ objects. C++ class promotes data encapsulation, which usually has well-dened regions of memory on which the class operates. This potentially simplies the packing of data for migration of objects compared to process migration or thread migration, where the threads entire stack must migrate to a new processor. Message forwarding after migration is automatically handled by the CHARM++ runtime system. From a users point of view, CHARM++ objects are location independent; messages are

usually delivered to objects instead of processors. Thus, there is no processor-specic state that an application writer needs to worry about for object migration; CHARM++ run-time system takes care of the run-time state associated with the migrating objects.

Using the CHARM++ object model, the run-time system treats application objects uniformly by instrumenting the start and end time of each method invocation on the objects, rather than deriving execution time from some application-specic knowledge. This makes the automatic measurement-based load balancing feasible. Further, the CHARM++ run-time system can automatically record object-to-object and collective communication patterns, so that load balancers can access the communication pattern information for making optimal load balancing decisions. CHARM++ can even separate the idle time from communication overhead. The CHARM++ run-time system can cleanly separate communication overhead from idle time. With the rich application load and communication statistics, better load balancing decisions can be made.

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 benet of the reader to understand the results better):

GreedyLB: A comprehensive load balancer based on the greedy heuristic that maps the heaviest objects on to the least loaded processors until the load of all processors is close to the average load.

ReneLB: A renement load balancer that migrates objects from processors ith greater than average load (starting with the most overloaded processor) to those with less than average load. The aim of this strategy is to reduce the number of objects migrated.

ReneCommLB: A renement strategy similar to ReneLB that also considers the communi-

cation between different objects when trying to choose the best underloaded processor to place an object on.

MetisLB: A strategy that passes the load information and the communication graph to METIS, a graph partitioning library, and uses the recursive graph partitioning algorithm in it for load balancing.

ScotchLB: A strategy that uses the load information and communication graph from Charm run-time system and uses Scotch, a graph partitioning library, to find the new mapping of the processes on to processor.

The CHARM++ runtime also encourages application developers to write application-specic load balancing strategies or use external libraries for the task. For this, it provides an easy interface to write new load balancers.

The runtime instruments a few time steps of the application before load balancing and this information is available in the LBDatabase. Using this information provided by the CHARM++ runtime, a load balancing strategy can be implemented that returns a new assignment for the processes. This information is then used by the runtime to migrate objects for the subsequent time steps. This setup facilitates the use of external load balancing algorithms/libraries for measurement-based dynamic load balancing of parallel applications, since they do not have to deal with the mechanics of instrumentation and object migration.

2.3 New Load Balancers

Three new load balancing strategies have been added to Charm++ which are described in detail in the following sections.

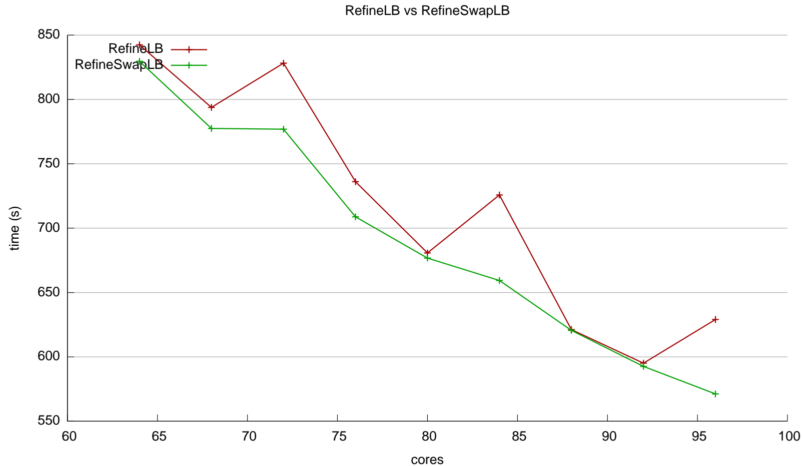

Figure 2.1: Comparison of RefineLB with RefineSwapLB for leanmd

2.3.1 RefineSwapLB

This is a refinement based load balancing strategy similar to that of RefineLB. ReneLB is an algorithm which improves the load balance by incrementally adjusting the existing object distribution. Renement is used with an overloaded threshold. Typically, this threshold is set at 1.003 times the average load across all processors. Any processor is considered overloaded if this load is above this threshold. The computational cost of the algorithm is low because only overloaded processors are examined, and it results in only a few objects being migrated. But RefineLB could get stuck in local minima if it cant move any object from the overloaded processor to another processor without making it overloaded. To handle this scenario, RefineSwapLB is implemented.

RefineSwapLB tries to move objects from overloaded processor to less loaded processor without causing them to become overloaded. If it cant find such a processor, then it swaps objects reducing the load on the overloaded processor.

Performance analysis:

2.3.2 ZoltanLB

ZoltanLB uses Zoltan, a hypergraph partitioner, to balance entities indicated by the application. This load balancer can be used for multicast aware load balancing. The result comparing ZoltanLB with MetisLB and ScotchLB is shown in figure. We use leanmd with multicast enabled.

Chapter 3

Meta-Balancer

3.1 Motivation

Understanding the characteristics of an application and taking appropriate load balancing decisions is key to improve its performance. Some of these decisions involve how frequently to call the load balancer or the type of strategy to use. In the case of a dynamic application, it becomes challenging to identify the characteristics of the application and take load balancing decisions apriori. 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 a metabalancer which will relieve application programmers of such key decision making related to load balancing and improve the overall performance. The key decisions related to load balancing are Frequency of load balancing

- Strategy Selection
- Communication vs Computation strategy
- Refinement vs From Scratch strategy
- Adaptive triggering of load balancing

Each of these decisions are based on the characteristics of the application, the details of which are given in the following sections. The Charm++ run-time system collects some statistics related to the object load, processor load, processor idle time and the communication pattern which is stored in the Load Balancing Database. These statistics presents a

holistic picture of the application. Meta-Balancer, which is a part of the Charm++ run-time system collects minimal stats at a central location and then based on the characteristics of the application, controls the load balancing decisions. Details of the stats collection is given in Section [].

3.2 Statistics Collection

3.2.1 Existing synchronous statistics collection

The CHARM++ run-time system (RTS) provides an accurate measurement of application load. During execution, the LB Manager, residing on each processor, monitors the load behavior of that processor. It collects object load, background load and idle time statistics into the LB Database. When a particular object is being executed, it noties the LB manager so that the manager may start the timing for its execution. The array manager also reports about communication initiated by the object.

In the centralized strategies, a dedicated processor gathers global information about the state of the entire machine and makes decisions for the migrations of tasks for every processor. Current mechanism for statistics collection for load balancing purpose is synchronous. All the processors come to a barrier and then send their statistics, which includes the object load, processor load and communication pattern, to a central processor. Whereas in fully distributed strategies, each processor executes load balancing algorithms by exchanging state information with other processors. The migration only happens between neighboring processors. For the Meta-Balancer to control the load balancing decision, some information related to the application needs to be collected periodically, like the average load per processor, maximum load on any processor in the system, lowest utilization on any processor in the system. To collect these stats periodically, it is not advisable to have a global barrier since that would unnecessarily cause an overhead and slow-down of the application. Therefore, there is a need for asynchronous collection of system stats periodically which is described in

detail in the following section.

3.2.2 Asynchronous Collection of Statistics via Reduction

The asynchronous collection of statistics periodically is motivated by the fact that any sort of barrier, global or local within a processor, would cause unnecessary slowdown of the application. Each processor has a collection of chares residing on it. The RTS has an accurate information of the load of each chare at any point of time. Typically, the stats would be collected every iteration of the application.

3.2.3 Extracting per iteration information

Periodically, chares send their load information since the last send iteration to the AdaptiveLB Manager, residing on each processor. Once the chare has sent its load information, it continues its work. To obtain the utilization of the processor during that iteration, it is required to obtain the idle time for that iteration. Since, there is no local barrier (barrier at the processor level), the idle time per iteration needs to be approximated. The idle time is considered to be total idle time until all chares finish the iteration * (number of chares per iteration / total number of contributions from all the chares till now). This is approximately equivalent to idle time till now / total iterations. Utilization is calculated as the idle time / load + idle time. Once all the chares residing on a processor has sent its load for that iteration, the AdaptiveLB Manager contributes this information to the central processor, which includes the load for the iteration and utilization of the processor, via a reduction tree. The central processor receives the average load per processor, maximum load in the system and the minimum utilization ratio. The figures shows the stats collection mechanism that is described above.

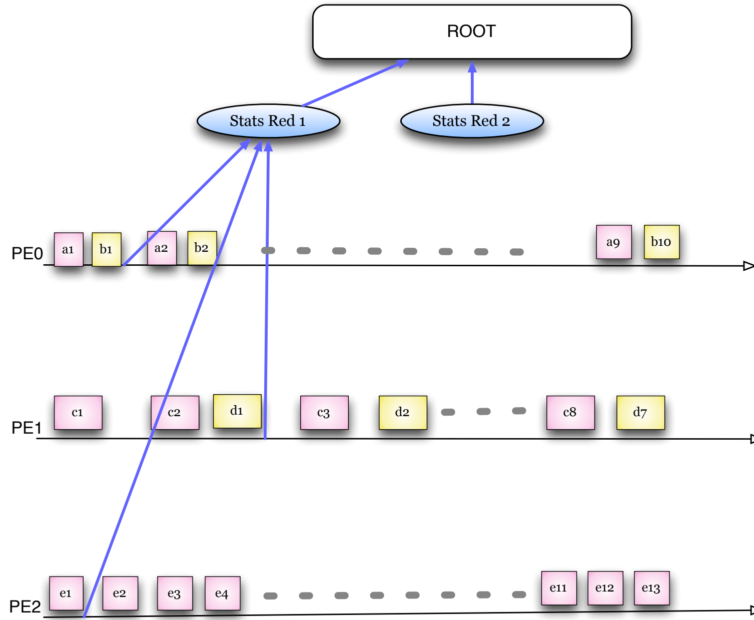

Figure 3.1: periodic stats collection

3.3 Load Balancing Period

Even though load balancing balances out the load imbalance which results in the improvement of performance, it incurs a cost. The cost involves carrying out the load balancing strategy to find new mapping of objects onto processors as well as the cost of migrating them. If load balancing is carried out very frequently, we would either loose performance or reduce the gains from doing load balancing. Therefore, there is an optimal load balancing period, where we obtain maximum gains.

τ be the ideal load balancing period

γ be the total iterations of the application

Γ be the total application time

θ be the cost associated with load balancing

Let the maximum time per iteration be denoted by the curve equation

$$y = mx + c_m \tag{3.1}$$

where m the slope is with respect to the average load linear curve.

$$\begin{aligned}
\Gamma &= \frac{\gamma}{\tau} \times (\int_0^{\tau} (mx + c_m) + \theta) + \int_0^{\gamma} (ax + c_a) \\
\Gamma &= \frac{\gamma}{\tau} \times (\frac{m\tau^2}{2} + c_m\tau + \theta) + \gamma \times (\frac{a\gamma}{2} + c_a) \\
\Gamma &= \gamma \times (\frac{m\tau}{2} + c_m + \frac{\theta}{\tau} + \frac{a\gamma}{2} + c_a) \\
\frac{d}{d\tau}(\Gamma) &= \gamma \times (m/2 - \theta/r^2) = 0 \\
\tau &= \sqrt{\frac{2\theta}{m}}
\end{aligned}$$

3.3.1 Ideal LB Period Calculation

The figure below shows the relationship between the total application time vs load balancing period for Jacobi application.

The figure below shows the lb period identified by the Meta-Balancer framework. As can be seen, the first load balancing step is at 14, followed by an lb period of 210 and 180.

Now we present how this theory is used by the Meta-Balancer to identify the load balancing period. For this, we consider a computation intensive application, like leanmd.

3.3.2 Computation Intensive Application

The Meta-Balancer periodically collects stats, which include average processor load, maximum load and utilization. When load balancing is done, the expectation is that the load balancing strategy would make the maximum load equal to or close to the average load per processor. Once a minimum set of stats are collected, we do a linear-extrapolation to predict

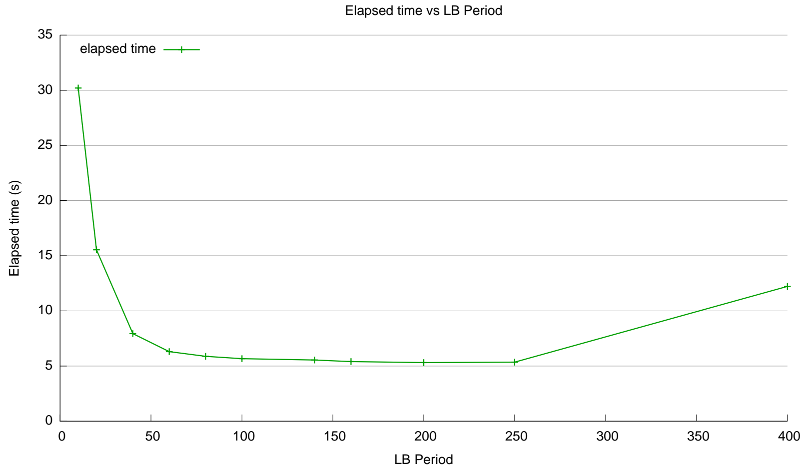

Figure 3.2: Elapsed time vs LB Period for Jacobi

the future load based on the principle of persistence. Thus, we obtain the curve that represents maximum load and average load for the application run. Once, we have the max curve equation with respect to the average curve, and the cost of load balancing, we substitute that in the above equation to obtain the ideal lb period. This lb period is informed to all the processors and chares. As more stats are collected, the lb period is refined and this refined value is informed to all the processors and chares.

We obtain the max curve equation with respect to average curve, as we expect that after load balancing, the max will become equal to the average. But it can so happen that the load balancing strategy can still cause some imbalance to remain. This inefficiency in LB strategies need be incorporated in the lb period calculation. The expected load after load balancing would depend on how well the load balancing strategy could even out the imbalance. The imbalance ratio after load balancing is calculated and the expected load is updated to be average load * imbalance ratio. The max curve is calculated with respect to this expected load. Once the max curve equation is identified, the lb period is calculated as described in the earlier section. This adjustment to the expected load after load balancing prevents frequent load balancing where the load balancer cannot improve the balance of load.

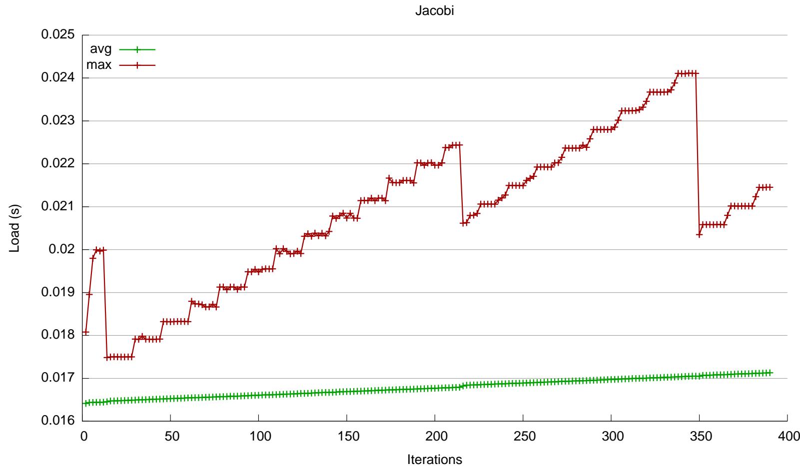

Figure 3.3: Identifying LB Period for Jacobi

3.3.3 Dynamic triggers

In dynamic applications, the load in the system can change suddenly. If the load balancing period has been fixed, then it could result in delayed call to the load balancer 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 if required. The Meta-Balancer, if sees that the imbalance ratio, given by max load/avg load, is beyond a threshold of 10earliest.

This can be seen in the following figure for kNeighbor benchmark, where the load in the system suddenly changes.

3.4 LB Period Intimation

Once the ideal load balancing period has been identified, it needs to be intimated to all the processors so that they take part in the load balancing. All the processors come to a barrier and then send their statistics, which includes the object load, processor load and communication pattern, to a central processor. While the load balancing is carried out, the chares remain idle. Once the load balancing period is informed, the chares enter the load

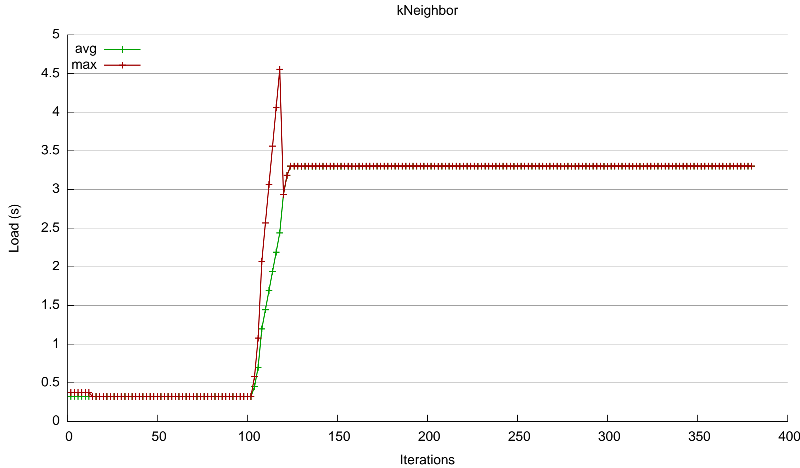

Figure 3.4: Dynamic triggering of LB for kNeighbor benchmark

balancing stage on reaching the informed period. Whenever a chare finishes its iteration and it checks to see if it needs to go into load balancing stage. If it isnt time for load balancing, it resumes its work. Chares can be in different iterations depending on its load, the processor load and communication dependencies. To ensure that the application doesnt hang, all the chares have to be at the same iteration to carry out centralized load balancing. Here is a scenario which could lead to a hung application. Consider a chare a, which is at iteration i, goes into load balancing and becomes idle. Another chare b is in the middle of iteration i+1, is waiting for a message from chare a. Since the chare a is idle, b cannot complete its iteration and the 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 load balancing stage. Since the chares can be in different iterations when the ideal load balancing period is informed, we use the following scheme to obtain consensus. Once the central processor, root, identifies the ideal load balancing period, which is tentative lb period, it broadcasts this information to all the processors. On receiving the tentative lb period, the processors informs the root about the maximum iteration of the chares residing on the processor. On receiving the maximum

iteration of the chares in the system, the root informs the final load balancing period. If the ideal period is beyond the maximum iteration, ie no chare has reached the ideal load balancing period, the lb period is fixed to the previously calculated value. But if any of the chare has gone past the ideal load balancing period, the new load balancing period is set to be the maximum iteration. This final lb period is informed to all the processors. On each processor, whenever a chare finishes its iteration, before entering the next iteration, checks to see if it has reached the lb period. If it hasnt reached the lb period, it resumes its work. If it has reached the tentative lb period, it waits for the final verdict of the lb period. It is said to be in a pause state. A chare in pause state, on receiving the final lb period, decides to either resume its work, if it hasnt reached the final lb period, or enters the load balancing stage, if it is at final lb period. Any chare which reaches the final lb period, enters into load balancing stage.

This scheme ensures that all the chares arrive at a consensus regarding the load balancing period and enters the load balancing stage at the same iteration. Figure shows the scheme.

3.4.1 Dynamic Refinement of LB Period

As the stats collection proceeds, the predicted load might change and become refined. This inturn 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 same scheme described in the previous section. Unless a 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.

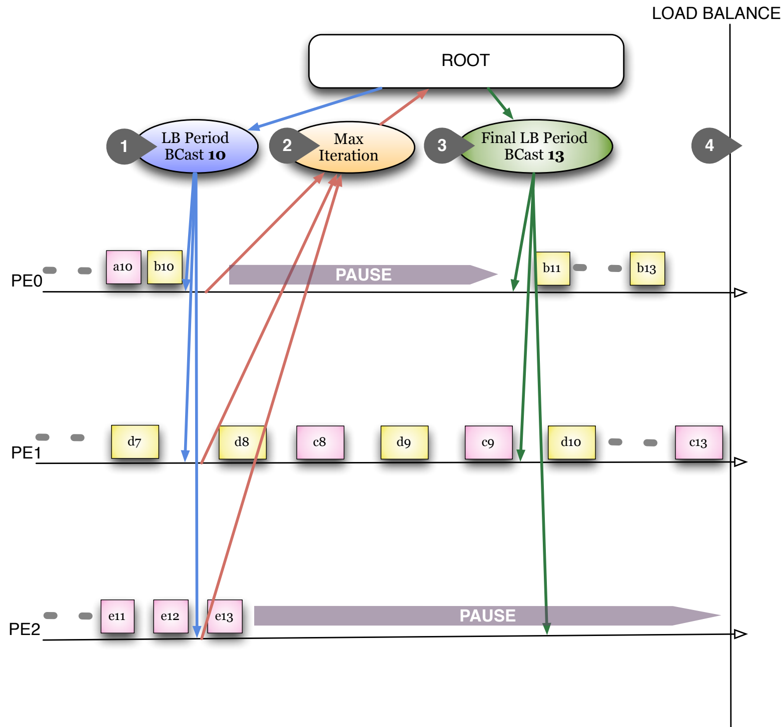

Figure 3.5: Intimating the calculated LB period

3.5 Strategy Selection

3.5.1 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 efcient 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 a computation intensive. If it is a computation intensive application, then imbalance in load will result in heavy degradation of performance. Load imbalance leads to

an increasing waste of resources as an application is scaled to more and more processors. Hence, for computation intensive applications, strategies that balances out the load, such as GreedyLB, RefineLB, RefineSwapLB, should be used. Imbalance in load can be identified by the ratio of maximum load/ average load per processor. If this ratio is beyond a threshold of 10then there is considerable imbalance. In Charm++ programs, communication is overlapped with computation. But if application has high communication volume, it would result in idle time due to message latency. If the alpha-beta cost of the communication in the application is atleast 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, ScotchLB, should be used.

3.5.2 Refinement vs From Scratch strategy

The load balancing strategies map objects to processors based on the objects computation load or based on the communication pattern. There are two categories of algorithms. One is greedy-based or from scratch algorithms that do not take the existing object mapping into account, and the other is renement-based algorithms that take the existing mapping into account in order to limit the number of object movements. Reducing the object movement indicates lesser migration cost. The computation based from-scratch strategies include GreedyLB and computation based refine strategies include RefineLB, RefineSwapLB. In the case of communication aware load balancing strategies, from-scratch strategies include MetisLB, ScotchLB, ZoltanLB and refinement based strategies include ScotchRefineLB, CommAwareRefineLB.

Figure shows the flowchart describing the strategy selection for communication vs computation and scratch vs refinement.

LB Strategy Selection

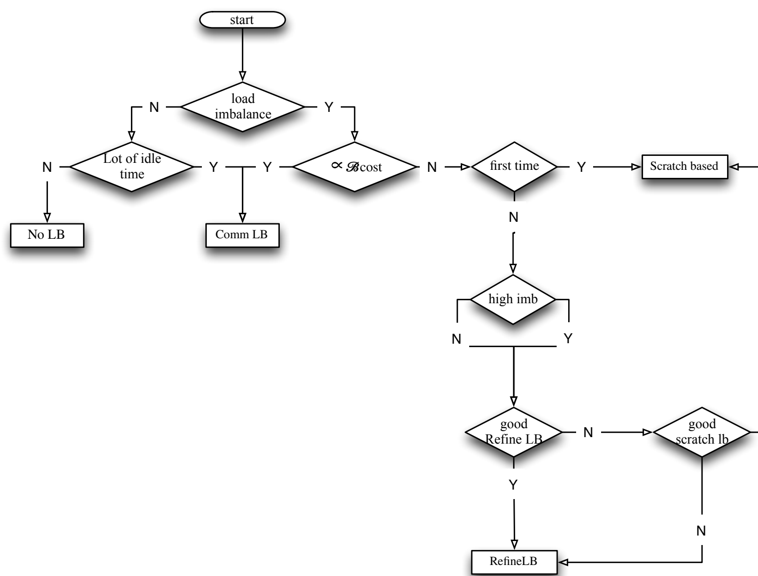

Figure 3.6: flowchart describing strategy selection

Chapter 4

Conclusions and Future Work

References

[1] 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.

[2] 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.