

© 2016 Harshitha Menon Gopalakrishnan Menon

ADAPTIVE LOAD BALANCING FOR HPC APPLICATIONS

BY

HARSHITHA MENON GOPALAKRISHNAN MENON

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kale, Chair
Professor William D. Gropp
Professor Josep Torrellas
Dr. Martin Schulz, Lawrence Livermore National Laboratory

Abstract

One of the critical factors that affect the performance of many applications is load imbalance. Applications are increasingly becoming sophisticated and are using irregular structures and adaptive refinement techniques, resulting in load imbalance. Moreover, systems are becoming more complex. The number of cores per node is increasing substantially and nodes are becoming heterogeneous. High variability in the performance of the hardware components introduces further imbalance. Load imbalance leads to drop in system utilization and degrades the performance. To address the load imbalance problem, many HPC applications employ dynamic load balancing algorithms to redistribute the work and balance the load.

Different application characteristics warrant different load balancing strategies. We need a variety of high-quality, scalable load balancing algorithms to cater to different applications. However, using an appropriate load balancer is insufficient to achieve good performance because performing load balancing incurs a cost. Moreover, due to the dynamic nature of the application, it is hard to decide when to perform load balancing. Therefore, deciding when to load balance and which strategy to use for load balancing may not be possible a priori.

With the ever increasing core counts on a node, there will be a vast amount of on-node parallelism. Due to the massive on-node parallelism, load imbalance occurring at the node level can be mitigated within the node instead of performing a global load balancing. However, having the application developer manage resources and handle dynamic imbalances is inefficient as well as is a burden on the programmer.

The focus of this dissertation is on developing scalable and adaptive techniques

for handling load imbalance. The dissertation presents different load balancing algorithms for handling inter and intra-node load imbalance. It also presents an introspective run-time system, which will monitor the application and system characteristics and make load balancing decisions automatically.

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 am forever indebted to him for accepting me as his student, for his kindness and unconditional support, especially during difficult times. I would like to thank the members of my dissertation committee for their time and valuable suggestions. I would like to thank our collaborator Prof. Thomas Quinn for introducing me to the fun of working on cosmology simulation application, ChaNGa. My parents have made me who I am today. I learned from my mother to work hard and believe that whatever happens is for best. I learned from my father to be dedicated and aim for perfection in whatever task I undertake. Their constant care and support have pulled me through tough times, and I cannot thank them enough for it. I would like to thank a very special person, my husband, Saurabh for his unwavering support and partnership in my life. Just thinking about my daughter, Samhita, makes my heart fill with joy. She has shown me child's unbounded curiosity and insatiable urge for learning. At the age of 18 months, she understands that she has to go to bed early so that her mother can finish the thesis work. Their presence in my life has made the Ph.D. journey more joyous. Finally, I would like to thank all the PPL members for *If I have seen further it is by standing on the shoulder of giants - Sir Isaac Newton.*

Grants

This research was supported in part by the Blue Waters sustained-petascale computing project, which is supported by NSF award number OCI 07-25070 and the state of Illinois. Blue Waters is a joint effort of the University of Illinois and NCSA. This work was supported in part supported by NSF award AST-1312913. This research was supported in part by the US Department of Energy under grant DOE DE-SC0001845 and by NSF ITR-HECURA-0833188. This research also used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. Experiments for this work were performed on Mira and Vesta, IBM Blue Gene/Q installations at Argonne National Laboratory. The authors would like to acknowledge PEACEndStation, PEACEndStation_2 and PARTS projects for the machine allocations provided by them. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357 (project allocations: PEACEndStation, PARTS). This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. I am thankful to Prof. Tarek Abdelzaher for letting me use the testbed for experimentation under grant NSF CNS 09-58314.

Table of Contents

List of Tables	ix
List of Figures	x
List of Algorithms	xiii
List of Abbreviations	xiv
1 Introduction	1
1.1 Motivation	1
1.2 Summary	3
1.3 Thesis Organization	4
2 Background	5
2.1 Over-decomposed Migratable Model to Support Load Balancing	5
2.2 CHARM++	5
3 Load Balancing Algorithm : GrapevineLB	8
3.1 Background	10
3.2 Related Work	12
3.3 Grapevine Load Balancer	14
3.3.1 Information Propagation	15
3.3.2 Probabilistic Transfer of Load	17
3.3.3 Partial Propagation	18
3.3.4 Grapevine+	19
3.4 Analysis of the Algorithm	19
3.4.1 Information Propagation	20
3.4.2 Randomized Transfer	21
3.4.3 Performance Analysis	23
3.4.4 Probabilistic Assignment	28
3.5 Implementation	29
3.6 Evaluation	30
3.6.1 Evaluation using Simulation	30
3.6.2 Evaluation using Applications	36
3.7 Conclusion	45

4	Graph Partitioner Based Load Balancers	46
4.1	Overview	46
4.2	Evaluation	49
4.3	Conclusion	50
5	Handling Imbalance in Distributed and Shared Memory	52
5.1	Related Work	54
5.2	Charm++ programming model for shared memory	56
5.3	Overview	58
5.4	Persistence Based Load balancing	59
5.5	Handling Residual and Transient Load Imbalance with Charm++ Task Model	61
5.5.1	Task API	61
5.5.2	Task Generation and Scheduling	61
5.5.3	Task Queue	64
5.6	Application Study	65
5.6.1	ChaNGa	66
5.6.2	NAMD	67
5.6.3	Kripke	69
5.7	Conclusion	72
6	Meta Balancer for LB Period	74
6.1	Background	77
6.1.1	Charm++ and its Load Balancing Framework	77
6.2	Meta-Balancer	79
6.2.1	Meta-Balancer Statistics Collection	79
6.2.2	Ideal Load Balancing Period Computation	81
6.2.3	Distributed Consensus	83
6.2.4	Implementation	85
6.3	Experimental Results	87
6.3.1	LeanMD	87
6.3.2	Fractography3D	92
6.4	Previous Work	96
6.5	Conclusion	97
7	Meta Balancer for LB Strategy	99
7.1	Meta-Balancer	100
7.1.1	Meta-Balancer Statistics Collection	100
7.1.2	Meta-Balancer Feature Construction	100
7.1.3	Load Balancing Strategy Selection	101
7.1.4	Load Balancing Strategy Selection Using Decision Tree	103
7.1.5	Load Balancing Strategy Selection Using Machine Learning	104
7.2	Experimental Results	105
7.3	Conclusion	106

8	Meta Balancer for Thermal Variation	108
8.1	Background	110
8.1.1	Charm++ and its Load Balancing Framework	111
8.1.2	Temperature Control using DVFS	111
8.2	Related Work	112
8.3	Limitations of Periodic Approach	113
8.4	MetaTempController	115
8.4.1	Temperature Control	116
8.4.2	Load Balancing	117
8.5	Results	117
8.5.1	Experimental Setup	118
8.5.2	Applications	118
8.5.3	Experimental Results	120
8.6	Conclusion	130
	References	131

List of Tables

3.1	Choice of load imbalance metric	11
3.2	Bisection bandwidth for different node counts	27
3.3	Time for gossip using α and β model	27
3.4	Gossip time taking bisection bandwidth into account	27
3.5	Migration time	29
3.6	Average cost (in seconds) per load balancing step of various strategies for AMR	41
3.7	Total application time (in seconds) for AMR on BG/Q. Proposed strategies Gv and Gv+ perform the best across all scales.	41
3.8	Average cost per load balancing step (in seconds) of various strategies for LeanMD	43
3.9	Total application time (in seconds) for LeanMD on BG/Q	43
4.1	Comparison between different load balancers using average number of multicast messages sent for <i>leanmd</i>	50
6.1	LeanMD Application Time on Jaguar	89
6.2	LeanMD Application Time on Ranger	91
6.3	Fractography3D Application Time	95
7.1	Meta-Balancer strategy selection statistics	101
7.2	Meta-Balancer strategy selection features	107

List of Figures

2.1	CHARM++ system view with over-decomposition	6
3.1	(a) Initial load of the underloaded processors, (b) Probabilities assigned to each of the processors, (c) Work units transferred to each underloaded processor, (d) Final load of the underloaded processors after transfer.	17
3.2	Probability distribution of maximum load.	23
3.3	Load distribution for a run of AMR used in simulation. Counts of processors for various loads are depicted.	31
3.4	Expected number of rounds taken to spread information from one source to 99% of the overloaded processors for different system sizes and fanouts(f).	32
3.5	Expected number of rounds taken to spread information from one source to 99% of the overloaded processors using Naive and Informed schemes for different system sizes. Here $f = 2$ and 50% of the system size is underloaded	33
3.6	Evaluation of load balancer with partial information. Max load(left) and Imbalance(right) decrease as more information about underloaded processors is available. It is evident that complete information is not necessary to obtain good performance.	35
3.7	Percentage of processors having various amounts of partial information as rounds progress. There are a total of 4096 underloaded processors. 99% receive information about 400 processors by 8th round while it takes 12 rounds for all the 4096 underloaded processors. . . .	36
3.8	Comparison of time per step (excluding load balancing time) for various load balancing strategies for AMR on Mira (IBM BG/Q). GV+ achieves quality similar to other best performing strategies. Note that axes are log scale.	39
3.9	Comparison of time per step (excluding load balancing time) for various load balancing strategies for LeanMD on Mira (IBM BG/Q). Note that axes are log scale.	42
4.1	Impact of increasing communication on the quality of load balance (kNeighbor running on 256 cores of Intrepid)	49
5.1	The potential benefits of intra-node work sharing on reducing load imbalance	53

5.2	Charm++ Parallel Programming System	57
5.3	Time line profile of ChaNGa for all the PEs (rows) on a SMP process for the 128K cores run. White shows idle time and colored bars indicate busy time. Fine-grained task parallelism achieves better distribution of work among PEs. The total time per step reduces from 5.0 seconds to 4.2 seconds.	67
5.4	ChaNGa strong-scaling performance on Blue Waters Cray XE6 system, using Charm++ alone, with integrated OpenMP, and with the <i>ParallelFor</i> extension. Both intra-node balancing mechanisms give more than 2X speedup at 128K cores.	68
5.5	Strong scaling results comparing the performance of original Charm++ with the new integrated task model for NAMD’s colvar benchmark on IBM Bluegene/Q.	69
5.6	Weak scaling Kripke with 4096 spatial zones per core on Blue Waters, the time per iteration is shown for MPI and AMPI with and without OpenMP. Numbers in parentheses indicate how many ranks were used per node.	72
6.1	Periodic Statistics Collection	80
6.2	Ideal Load Balancing Period	83
6.3	Three Step Consensus Mechanism in Meta-Balancer	85
6.4	Processor Utilization of LeanMD on 256 cores	89
6.5	Variation in LB Period for LeanMD on Jaguar	90
6.6	Variation in LB Period for LeanMD on Ranger	91
6.7	Processor Utilization of Fractography3D on 64 cores	93
6.8	Variation in LB Period for Fractography3D on Jaguar	95
7.1	Decision tree for choosing the load balancing strategy.	104
7.2	Accuracies on the training and test set along with confusion matrix. .	106
8.1	Comparison for maximum temperature and timing penalty for various user specified period	115
8.2	Maximum Temperature of the Processors for <i>wave2D</i>	119
8.3	Maximum Temperature of the Processors Over Time for <i>wave2D</i> . . .	119
8.4	Maximum Temperature of the Processors for kNeighbor	120
8.5	Maximum Temperature of the Processors Over Time for kNeighbor .	120
8.6	Maximum core temperature for the entire run for leanMD. This indicates region of hot-spots.	121
8.7	Maximum temperature on any core over time for leanMD. Without any control, temperature reaches 73° C and MetaTempController keeps it within threshold.	122
8.8	Average utilization over time with and without MetaTempControler .	123
8.9	Execution time and temperature for different strategies	123
8.10	Minimum frequency of the processors over time	126
8.11	Execution time breakdown into temp check and lb overhead	128

8.12 Machine energy normalized according to naive DVFS 129

List of Algorithms

1	Informed selection at each processor $P_i \in P$	15
2	Informed transfer at each processor $P_i \in P$	16
3	Recursive Splitting	63
4	Application Code on every Chare	78
5	Charm RTS on each Chare	86
6	Meta-Balancer on every Processor	87
7	Meta-Balancer on Central Processor	87
8	Periodic temperature-aware dynamic load balancing	114

List of Abbreviations

1D	one-dimensional
2D	two-dimensional
3D	three-dimensional
PE	Processing Element
BG/P	Blue Gene/P
BG/Q	Blue Gene/Q
NAMD	Nanoscale Molecular Dynamics
PGAS	Partitioned Global Address Space
RDMA	Remote direct memory access
DEM	Dimension exchange
AMR	Adaptive Mesh Refinement
API	Application Programming Interface
OS	Operating system
CPU	Central processing unit
DVFS	Dynamic voltage and frequency scaling
CRAC	Computer room air conditioning
HPC	High performance computing
CMP	Chip multiprocessor
IBM	International Business Machines Corporation
MILC	MIMD Lattice Computation
MPI	Message Passing Interface
ORB	Orthogonal Recursive Bisection

1 Introduction

Many of the breakthrough scientific research requires computational modeling using supercomputers. Consequently, high performance computing has a critical role to play in the advancement of science and engineering. However, many challenges need to be addressed to utilize the increasing power of parallel machines. One of the critical factors that affect the performance of many applications is load imbalance. Increasingly, science and engineering applications are becoming more complex and dynamic. Many applications use adaptive multiscale algorithms, such as adaptive mesh refinement, multi-grid methods, which introduce dynamically changing computation resulting in load imbalance. Moreover, systems are becoming more complex. The number of cores per node is increasing substantially and are becoming heterogeneous. The high variability in the performance of the hardware components introduces further imbalance due to heterogeneity [1]. As we move towards systems with billions of cores, even well-balanced applications will experience load imbalance due to variability and heterogeneity in the hardware.

1.1 Motivation

In scientific simulations the work is assigned to processing elements. Uneven distribution of work to processing elements can result in imbalance of load. Load imbalance is an insidious factor that can reduce the performance of a parallel application significantly. For some applications, such as basic stencil codes for structured grids, the load is easy to predict and does not vary dynamically. However, for a significant

class of applications, load represented by pieces of computations varies over time, and may be harder to predict. This is becoming increasingly prevalent with the emergence of sophisticated applications. For example, atoms moving in a molecular dynamics simulation will lead to (almost) no imbalance when they are distributed statically to processors. But, they create imbalance *when* spatial partitioning of atoms is performed for more sophisticated and efficient force evaluation algorithms. The presence of moisture and clouds in weather simulations, elements turning from elastic to plastic in structural dynamics simulations and dynamic adaptive mesh refinements are all examples of sophisticated applications which have a strong tendency for load imbalance.

All the examples above are of “iterative” applications: the program executes series of time-steps, or iterations, leading to convergence of some error metric. Consecutive iterations have relatively similar patterns of communication and computation. There is another class of applications, such as combinatorial search, that involves dynamic creation of work and therefore also has a tendency for imbalance. This class of applications has distinct characteristics and load balancing needs, and has been addressed by much past work such as work-stealing [2, 3, 4]. This thesis does *not* focus on such applications, but instead on the iterative applications, which are predominant in science and engineering.

Different application characteristics warrant different load balancing strategies. For example, an application that does substantial communication will benefit most from a load balancing strategy that takes communication into account and tries to minimize it. Alternatively, applications suffering from high compute load imbalance with little communication overhead will require load balancing strategies that are focused on balancing the computation load. This thesis presents various load balancing algorithms that are suitable for different application characteristics.

With ever increasing core counts on a node, there will be a vast amount of on-

node parallelism. There is heterogeneity in performance and the functionality of cores [1]. Due to the massive on-node parallelism, there is more opportunity to mitigate the load imbalance occurring at the node level within the node instead of performing a global load balancing. However, having the application developer manage resources and handle dynamic imbalances is inefficient as well as is a burden on the programmer. It is now widely accepted that the run-time system will be required to play a more active role in managing resources and handling imbalance.

However, using an appropriate load balancer is not sufficient to achieve good performance because performing load balancing incurs a cost. Moreover, due to the dynamic nature of the application, it is hard to decide when to perform load balancing. Therefore, deciding when to load balance and which strategy to use for load balancing may not be possible a priori.

This dissertation will demonstrate that an introspective run-time system that inspects the system resources and monitors the application characteristics can automatically make load balancing decisions. Since the run-time system is orchestrating the scheduling on processors and the communication between the work/data units, it can collect real time statistics about the state of the application and system to make informed load balancing decisions.

1.2 Summary

This thesis presents new load balancing strategies developed for different classes of applications. The focus is on *GrapevineLB*, which is a distributed load balancing algorithm implementing a high-quality balancer with very little overhead.

An important work of this thesis is an introspective run-time system component, called MetaBalancer, which monitors application and system performance, uses models to predict their behavior and applies machine learning algorithms to

make load balancing decisions automatically without any input from the user. The intra-node component of the introspective run-time system identifies load imbalances within a node and chooses a mechanism to share the work-load between compute units on a node. By performing intra-node load balancing to handle small and unpredictable variability in the application load and resources, we reduce the need for expensive application-level load balancing. The systemwide inter-node component identifies load imbalances across the nodes, calculates an ideal load balancing period weighing the cost benefit and also chooses the load balancing strategy to use based on the application and system characteristics.

1.3 Thesis Organization

This work is built on CHARM++ and Chapter 2 gives a brief introduction of the CHARM++ programming model and load balancing framework. Chapter 3 presents a novel, highly scalable, distributed load balancing strategy, which uses partial information about the global state of the system to give very good balance of load with low overhead. Chapter 4 discusses other load balancing strategies implemented. Chapter 5 presents how imbalance can be handled in distributed and shared memory by utilizing the large number of cores within a node. Chapters 6,7,8 presents the work on the introspective run-time system to automate the load balancing decisions.

2 Background

The load balancing strategies and the MetaBalancer framework are implemented in the CHARM++ run-time system. In this section, we elaborate on the CHARM++ programming model and the load balancing framework.

2.1 Over-decomposed Migratable Model to Support Load Balancing

For the run-time system to support load balancing, the problem needs to be over-decomposed into many more chunks than the number of processors. Over-decomposition refers to the division of a problem into a large number of work and data units, which can then be remapped to processors when load imbalance is detected. Migratability refers to the ability to move the work and data units from one processing element to another. Over-decomposition with migratability enables the run-time system to perform dynamic load balancing. This model has been implemented in the CHARM++ parallel programming model. In CHARM++ the programmer exposes parallelism by decomposing their computation into tasks or objects which are mapped and remapped on to the processors by the run-time system.

2.2 Charm++

The CHARM++ parallel programming model is based on asynchronous message driven execution. It has parallel objects, called chares, which are migratable and

communicate via messages. Chares are the basic units of parallel computation in CHARM++ and they are mapped onto processing elements (PEs) initially by the runtime system. CHARM++ applications are over-decomposed into many more work units than the number of processors. Overdecomposition along with migratability of chares empowers the CHARM++ run-time system to perform dynamic load balancing.

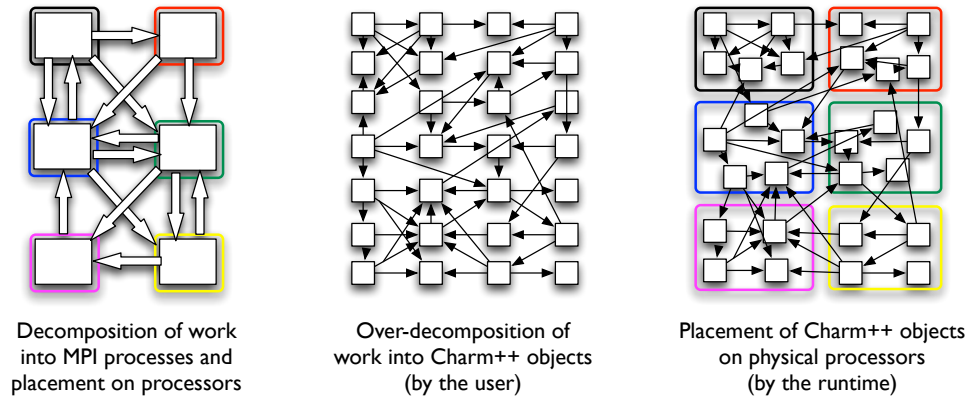


Figure 2.1: CHARM++ system view with over-decomposition

Applications written in CHARM++ over-decompose their computation into virtual processors or objects called “chares” which are then mapped onto physical processors by the runtime system (shown in Figure 2.1). The initial static mapping can be changed as the execution progresses if the application suffers from load imbalance by migrating chares to other processors. This is facilitated by the load balancing framework in CHARM++ [5]. Load balancing in CHARM++ is based on instrumenting the load from the recent past as a guideline for the near future, a heuristic known as the *principle of persistence* [6]. It posits that empirically, the computational loads and communication patterns of the tasks or objects *tend* to persist over time, even in dynamically evolving computations. Therefore, the load balancer can use 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].

CHARM++ provides a mature load balancing framework with a suite of load balancing strategies comprising of various centralized, distributed and hierarchical schemes for balancing computation load or communication [10]. Depending on the needs of applications, the user can invoke appropriate load balancers. The load balancing framework in CHARM++ instruments each chare as well as records the PE's load. In the *AtSync* mode of load balancing, all the chares pause their execution and call *AtSync*. The load statistics are collected and the user specified load balancing strategy is used to compute the new mapping. Once the load balancing decision is made, the framework handles the migration of the chares to the newly mapped PEs and resumes them.

3 Load Balancing Algorithm : GrapevineLB

Load imbalance is a critical factor that can significantly impact the performance of a parallel application. Many HPC applications employ dynamic load balancing algorithms to redistribute the work and address the load imbalance problem. Various strategies have been proposed to address the load balancing problem. Many applications employ a centralized load balancing strategy, where load information is gathered on to a single processor, and the decision algorithm is run sequentially. Such strategies have been shown to be effective for a few hundred to thousand processors, because the total number of work units is relatively small (on the order of ten to hundred per processor). However, they present a clear performance bottleneck beyond a few thousand processors, and may become infeasible due to the memory capacity bottleneck on a single processor.

An alternative to centralized strategies are distributed strategies that use local information, e.g., diffusion based [11]. In a distributed strategy, each processor makes autonomous decisions based on its local view of the system. The local view typically consists of the load of its neighboring processors. Such a strategy is scalable, but tend to yield poor load balance due to the limited information available locally [12].

Hierarchical strategies [13, 14, 12] overcome some of the aforementioned disadvantages. They create subgroups of processors and collect information at the root of each subgroup. Higher levels in the hierarchy only receive aggregate information and deliver decisions in aggregate terms. Although effective in reducing memory costs and ensuring good balance, these strategies may suffer from excessive data collection

at the lowest level of the hierarchy and work being done at multiple levels.

In this work we propose a fully distributed strategy, *GrapevineLB*, that has been designed to overcome the drawback of other distributed strategies by obtaining a partial representation of the global state of the system and basing the load balancing decisions on this. We describe a light weight information propagation algorithm based on the *epidemic* algorithm [15] (also known as the gossip protocol [16]) to propagate the load information about the underloaded processors in the system to the overloaded processors. This spreads the information in the same fashion as gossip spreads through the grapevine in a society. Based on this information, *GrapevineLB* makes probabilistic transfer of work units to obtain good load distribution. The proposed algorithm is scalable and can be tuned to optimize for either cost or performance.

The primary contributions of this chapter are:

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

3.1 Background

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

We focus on the iterative applications, which are predominant in science and engineering. For iterative applications, the basic scheme we follow is: The application is assumed to consist of a large number of migratable units (for example, these could be chunks of meshes in adaptive mesh refinement application). The application pauses after every so many iterations, and the load balancer decides whether to migrate some of these units to restore balance. Load balancing is expensive in these scenarios and is performed infrequently or whenever significant imbalance is detected. The load balancer needs information about the loads presented by each object. This can be based on a model (simple examples being associating a fixed amount of load with each grid point, or particle). However, for many iterative applications, another metric turns out to be more accurate. For these applications, a heuristic called *principle of persistence* [6] holds which posits that computational loads and communication patterns of objects *tend* to persist over time. This allows us to use recent instrumented history as a guide to predicting load in near-future iterations. The load balancing strategy we describe can be used with either model-based or persistence-based load predictions. In persistence-based load balancer, the statistics about the load of each task on a processor is collected at that processor. The database containing the task information is used by the load balancers to produce a new mapping. The run-time system then migrates the tasks based on this

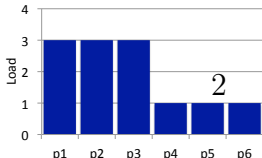
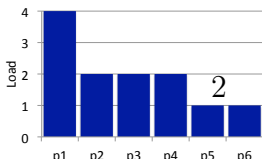
Processor Load	L_{avg}	L_{max}	σ	\mathcal{I}
	2	3	$\sqrt{6}$	0.5
	2	4	$\sqrt{6}$	1

Table 3.1: Choice of load imbalance metric

mapping.

It is important to choose the right metric to quantify load imbalance in the system. Using standard deviation to measure load imbalance may seem like an appropriate metric, but consider the two scenarios shown in Table 3.1. In both the cases, the average load of the system is 2. If we consider standard deviation, σ , to be a measure of imbalance, then we find that in case 1 and case 2 we obtain the same σ of $\sqrt{6}$ whereas the utilization and the total application times differ. A better indicator of load imbalance in the system is the ratio of maximum load to average load. More formally, load imbalance (\mathcal{I}) can be measured using

$$\mathcal{I} = \frac{L_{max}}{L_{avg}} - 1 \quad (3.1)$$

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

3.2 Related Work

Load balancing has been studied extensively in the literature. For applications with regular load, *static* load balancing can be performed where load balance is achieved by carefully mapping the data onto processors. Numerous algorithms have been developed for statically partitioning a computational mesh [17, 18, 19, 20]. These model the computation as a graph and use graph partitioning algorithms to divide the graph among processors. Graph and hypergraph partitioning techniques have been used to map tasks on to processors to balance load while considering locality. They are generally used as a pre-processing step and tend to be expensive. Our algorithm is employed after the application work has already been partitioned and used to balance the computation load imbalance that arises as the application progresses. Our algorithm also takes into consideration the existing mapping and moves tasks only if a processor is overloaded.

For irregular applications, work stealing is employed in task scheduling and is part of runtime systems such as Cilk [3]. Work stealing is traditionally used for task parallelism of the kind seen in combinatorial search or divide-and-conquer applications, where tasks are being generated continuously. A recent work by Dinan et al. [21] scales work stealing to 8192 processors using a PGAS programming model combined with RDMA. In work that followed, a hierarchical technique described as retentive work stealing was employed to scale work-stealing to over 150K cores by exploiting the principle of persistence to iteratively refine the load balance of task-based applications [14]. CHAOS [22] provides an inspector-executor approach to load balancing for irregular applications. Here the data and the associated computation balance is evaluated at runtime before the start of the first iteration to rebalance. The proposed strategy is more focused towards iterative computational science applications, where computational tasks tend to be persistent.

Dynamic load balancing algorithms for iterative applications can be broadly classified as centralized, distributed or hierarchical. Centralized strategies [23, 24] tend to yield good load balance, but exhibit poor scalability. Alternatively, several distributed algorithms have been proposed in which processors autonomously make load balancing decisions based on localized workload information. Popular nearest neighbor algorithms are dimension-exchange [25] and the diffusion methods. Dimension-exchange method is performed in an iterative fashion and is described in terms of a hypercube architecture. A processor performs load balancing with its neighbor in each dimension of the hypercube. Diffusion based load balancing algorithms were first proposed by Cybenko [11] and independently by Boillat [26]. This algorithm suffers from slow convergence to the balanced state. Hu and Blake [27] proposed a non-local method to determine the flow which is minimal in the l_2 -norm, but this approach requires global communication. The token distribution problem was studied by Peleg and Upfal [28] where the load is considered to be a token. Several diffusive load balancing policies, like direct neighborhood, average neighborhood, have been proposed in [29, 30, 31]. In [32], a sender-initiated model is compared with receiver-initiated in an asynchronous setting. It also compares Gradient Method [33], Hierarchical Method and DEM (Dimension exchange). The diffusion based load balancers are incremental and scale well with number of processors. However, they can be invoked only to improve load balance rather than obtaining global balance. If global balance is required, multiple iterations might be required to converge [34]. To overcome the disadvantages of centralized and distributed, hierarchical [13, 14, 12] strategies have been proposed. It is another type of scheme which provides good performance and scaling.

In our proposed algorithm, global information is spread using a variant of gossip protocol [16]. Probabilistic gossip-based protocols have been used as robust and scalable methods for information dissemination. Demers et al. use a gossip-based

protocol to resolve inconsistencies among the Clearinghouse database servers [16]. Birman et al. [35] employ gossip-based scheme for bi-modal multicast which they show to be reliable and scalable. Apart from these, gossip-based protocols have been adapted to implement failure detection, garbage collection, aggregate computation etc.

3.3 Grapevine Load Balancer

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

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

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

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

Input:

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

```
1:  $S \leftarrow \emptyset$  ▷ Set of underloaded processors
2:  $L \leftarrow \emptyset$  ▷ Load of underloaded processors
3: if ( $L_i < L_{avg}$ ) then
4:    $S \leftarrow P_i; L \leftarrow L_i$ 
5:   Randomly sample  $\{P^1, \dots, P^f\} \in P$ 
6:   Send ( $S, L$ ) to  $\{P^1, \dots, P^f\}$ 
7: end if
8: for ( $round = 2 \rightarrow k$ ) do
9:   if (received msg in previous round) then
10:     $R \leftarrow P \setminus S$  ▷ Informed selection
11:    Randomly sample  $\{P^1, \dots, P^f\} \in R$ 
12:    Send ( $S, L$ ) to  $\{P^1, \dots, P^f\}$ 
13:   end if
14: end for
```

```
1: when ( $S_{new}, L_{new}$ ) is received ▷ New message
2:  $S \leftarrow S \cup S_{new}; L \leftarrow L \cup L_{new}$  ▷ Merge information
```

3.3.1 Information Propagation

To propagate the information about the underloaded processors in the system, *GrapevineLB* follows a protocol that is inspired by the epidemic algorithm [15] (also known as a gossip protocol [16]). In our case, the goal is to spread the information about the underloaded processors such that every overloaded processor receives this information with high probability. An underloaded processor starts the ‘infection’ by sending its information to a randomly chosen subset of processors. The size of the subset is called *fanout*, f . An infected processor further spreads the infection by forwarding all the information it has to another set of randomly selected f processors. Each processor makes an independent random selection of peers to send the information.

We show that the number of rounds required for all processors to receive the information with high probability is $O(\log_f n)$, where n is the number of processors.

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

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

```
1: Compute  $p_j \quad \forall P_j \in S$  ▷ Using eq. 3.2
2: Compute  $F_j = \sum_{k < j} p_k$  ▷ Using eq. 3.21
3: while ( $L_i > (T \times L_{avg})$ ) do
4:   Select object  $O_i \in O$ 
5:   Randomly sample  $X \in S$  using  $F$  ▷ Using eq. 3.22
6:   if ( $L_X + load(O_i) < L_{avg}$ ) then
7:      $L_X = L_X + load(O_i)$ 
8:      $L_i = L_i - load(O_i)$ 
9:      $O \leftarrow O \setminus O_i$ 
10:  end if
11: end while
```

We propose two randomized strategies of peer selection as described below. Note that although we discuss various strategies in terms of rounds for the sake of clarity, there is no explicit synchronization between rounds in our implementation.

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

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

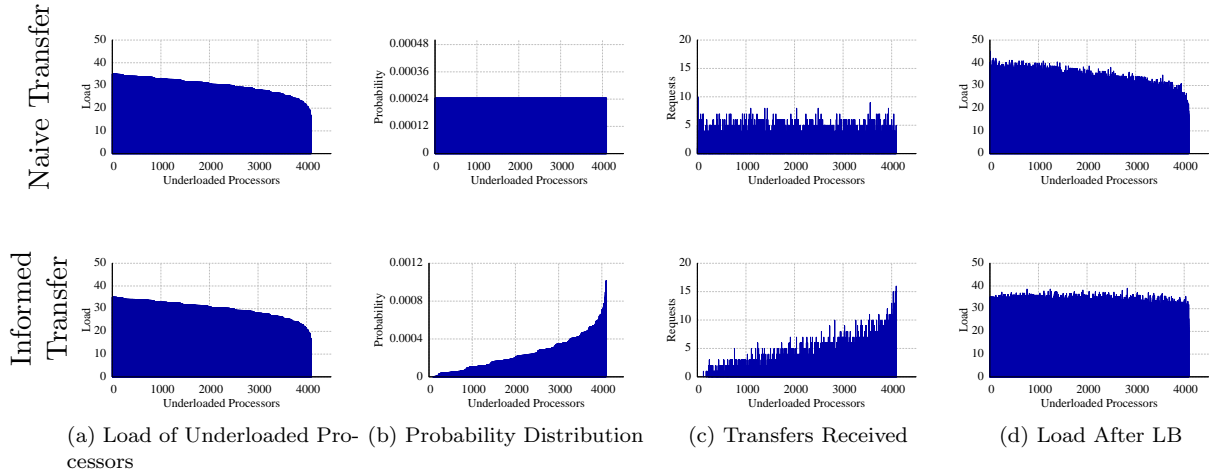


Figure 3.1: (a) Initial load of the underloaded processors, (b) Probabilities assigned to each of the processors, (c) Work units transferred to each underloaded processor, (d) Final load of the underloaded processors after transfer.

3.3.2 Probabilistic Transfer of Load

In our distributed scheme the decision making for transfer of load is decentralized. Every processor needs to make these decisions in isolation given the information from the propagation stage. We propose two randomized schemes to transfer load. **Naive Transfer:** The simplest strategy to transfer load is to select processors uniformly at random from the list of underloaded processors. An overloaded processor transfers load until its load is below a specified threshold. The value of threshold indicates how much of an imbalance is acceptable. As one would expect, this random selection results in overloading processors whose load is closer to the average. This is illustrated in the top row of Figure 3.1 and described in detail in Section 3.6.1.

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

$$p_i = \frac{1}{Z} \times \left(1 - \frac{L_i}{L_{avg}}\right) \quad (3.2a)$$

$$Z = \sum_1^N \left(1 - \frac{L_i}{L_{avg}}\right) \quad (3.2b)$$

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

$$F(y) = p(x < y) = \int_{-\infty}^y p(x)dx \quad (3.3)$$

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

$$y_s = F^{-1}(r_s) \quad (3.4)$$

Using the above, we randomly select the processors according to p_i for transferring load. This is summarized in Algorithm 2. Figure 3.1 illustrates the results.

3.3.3 Partial Propagation

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

achieve good balance. We empirically confirm our hypothesis by a set of experiments in Section 3.6.1.

3.3.4 Grapevine+

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

3.4 Analysis of the Algorithm

This section presents an analysis of the information propagation as well as the randomized transfer algorithm.

3.4.1 Information Propagation

We consider a system of n processors and, for simplicity, assume that the processors communicate in synchronous rounds with a fanout f . Note that in practice the communication is asynchronous (Section 3.5). We show that the expected number of rounds required to propagate information to all the processors in the system with high probability is $O(\log_f n)$. Although we analyze the case of a single sender, the results are the same for multiple senders since they communicate concurrently and independently.

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

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

Consequently,

$$q_1 = \left(1 - \frac{1}{n}\right)^f \quad (3.5)$$

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

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

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

$$\begin{aligned} \log q_s &\approx \gamma f^s \log \left(1 - \frac{1}{n} \right) \approx \frac{-\gamma f^s}{n} \\ \therefore q_s &\approx \exp \left(\frac{-\gamma f^s}{n} \right) \end{aligned}$$

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

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

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

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

Our simulation results shown in Figure 3.4 concur with the above analysis. It is evident that increasing the fan-out results in significant reduction of the number of rounds required to propagate the information.

3.4.2 Randomized Transfer

Due to the probabilistic transfer of load, some of the underloaded processors can become overloaded. After the information propagation phase, the overloaded processor will have the information about the underloaded processors in the system. Let us consider the case where number of rounds is $\log(n)$. This will ensure that the

overloaded processors will have all the information about the underloaded processors in the system with high probability.

The transfer can be considered as n independent Bernoulli trials, where n is the total number of transfers. A processor i with load L_i has a probability p_i , given in Eq 3.2, of being selected for transfer. This results in a binomial distribution. The expected load of an underloaded processor i due to the transfer is $E[X_i] = np_i$. This gives us only the mean, but for load balancing we are interested in the max load.

Using the results from [36], for $k \geq np_i$, following inequality holds:

$$P(X_i \leq k) > 1 - \frac{e^{-nD(p_i, k/n)}}{\max\left\{2, \sqrt{4\pi nD(p_i, k/n)}\right\}} \quad (3.10)$$

where $D(p_i, k/n)$ represents the Kullback-Leibler(KL) divergence between two Bernoulli variables with respective probabilities of success p_i and k/n and is given by

$$D(p_i, k/n) = (k/n)\log\frac{k/n}{p_i} + (1 - k/n)\log\frac{1 - k/n}{1 - p_i} \quad (3.11)$$

This can be written as the probability that processor i has at least k transfers is:

$$P(X_i > k) \leq \frac{e^{-nD(p_i, k/n)}}{\max\left\{2, \sqrt{4\pi nD(p_i, k/n)}\right\}} \quad (3.12)$$

If k is the transfer to processor i and L_i is the initial load, then the total load at processor i is $L_i + k$. The above equation can be written in terms of final load $M_i = L_i + k$ as:

$$P(Y_i > M_i) \leq \frac{e^{-nD(p_i, k/n)}}{\max\left\{2, \sqrt{4\pi nD(p_i, k/n)}\right\}} \quad (3.13)$$

Using union bound we determine that any of the processor has at least load M is

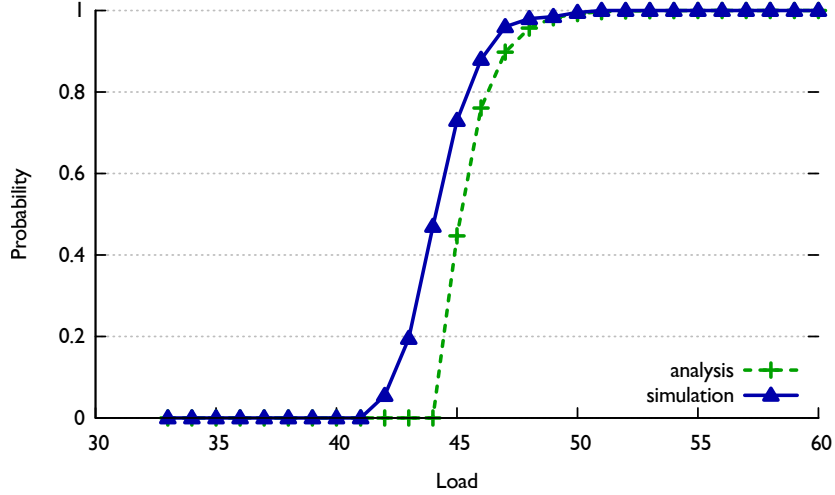


Figure 3.2: Probability distribution of maximum load.

given by:

$$P(Y > M) \leq \sum_i \frac{e^{-nD(p_i, k_i/n)}}{\max\left\{2, \sqrt{4\pi nD(p_i, k_i/n)}\right\}} \quad (3.14)$$

where $k_i = M - L_i$.

This can be written as the probability that the load of any processor less than M is given by:

$$P(X \leq M) > 1 - \sum_i \frac{e^{-nD(p_i, k_i/n)}}{\max\left\{2, \sqrt{4\pi nD(p_i, k_i/n)}\right\}} \quad (3.15)$$

Figure 3.2 shows the probability distribution based on a simulation of the real load data as well as the theoretical bound. This gives us a tight bound and we can use it to get an estimate of the maximum load of the system.

3.4.3 Performance Analysis

To estimate the cost of load balancing, we will divide it into load balancing strategy cost and migration cost. Let,

C_{lb} : the cost of load balancing which includes strategy and migration

C_{strat} : the cost of the load balancing strategy

C_{mig} : the cost of migration

T_{before} : the application time before load balancing

T_{after} : the application time after load balancing

T_i : the application run time at processor i

$$C_{lb} = C_{strat} + C_{mig} \quad (3.16)$$

To get a performance model for C_{strat} , we split the strategy phase into two stages:

1) information propagation via gossip; 2) probabilistic transfer of the work load.

Information Propagation

The information is propagated using a gossip protocol. At the very beginning, the underloaded processors in the system start the gossip by sending its information to two randomly chosen processors. On receiving this information, a processor updates its knowledge base, combines it with its knowledge and forwards it to two randomly chosen processors. In this information propagation protocol we don't necessarily need the entire information. To reduce the overhead of the algorithm, only partial information is forwarded. We can set this threshold based on the cost benefit analysis. For now let us assume that information about x processors is forwarded which amounts to n bytes.

We consider a system of p processors and, for simplicity, assume that the processors communicate in synchronous rounds. In round $r = 1$, underloaded processors initiate the information propagation by sending out two messages. In all successive rounds, each processor that received a message in the previous round sends out two messages. Let us assume that the number of underloaded processors is $O(p)$

(typically some fraction of the total processors in the system).

For each initiated gossip, in the first step two messages are sent. The probability that a processor P_i received any of these messages is $\frac{2}{p-1} \approx \frac{2}{p}$. The number of messages sent out in round r is 2^r . We have proven in Section 3.4 that with high probability all the processors will receive this message in $\log_2 n$ rounds. Consequently, each gossip is propagated $\log_2 p$ rounds. There are $O(p)$ such gossips initiated. Taking that into account in round one the number of messages received at P_i is $p \times \frac{2}{p} = 2$ and the number of messages received by P_i in round r is 2^r . Then, the number of messages received at P_i in $\log_2 p$ rounds is $2, 2^2, 2^3, \dots, 2^{\log_2 p}$, which is $2 \times (p - 1)$. The bytes sent in each message is n . The time taken by the gossip algorithm for TTL of $\log_2 p$ is

$$T_{log} = \log_2 p \times \alpha + 2 \times (p - 1)n\beta \quad (3.17)$$

In this algorithm, we limit the propagation of the messages by using TTL (Time To Live), which is set to a value such as $\log_2 n$ and whenever the message is received, TTL is decremented and that message is forwarded again only when the $TTL > 0$. With TTL of $\frac{1}{2}\log_2 n$, the messages received a processor P_i now becomes $2, 2^2, 2^3, \dots, 2^{\frac{1}{2}\log_2 p}$, which is $2 \times (\sqrt{p} - 1)$. Time for gossip algorithm for TTL of $\frac{1}{2}\log_2 p$ is

$$T_{0.5log} = \frac{1}{2}\log_2 p \times \alpha + 2 \times (\sqrt{p} - 1)n\beta \quad (3.18)$$

We ran the experiments on BlueGene/Q and used pingpong benchmark to obtain the α and β cost. It was run in three configurations. 1) Pingpong between two nodes with one rank per node, 2) Pingpong between 2 nodes with 32 ranks per node and 3) Pingpong between 512 nodes with 32 ranks per node. For 2 ranks, the values are $\alpha = 1.14\text{E}-5$ seconds and $\beta = 5.6\text{E}-10$ seconds/byte. For 64 and 16384 ranks, the

values are $\alpha = 1.37\text{E}-05$ seconds and $\beta = 8.74\text{E}-09$ seconds/byte.

Using the α and β cost model given in Eq 3.17,3.18, the timing for gossip is given in Table 3.3. A threshold of 100 was set for the amount of information forwarded, i.e., information of only 100 underloaded processors is forwarded. From Table 3.3, we can see that the actual time is about 3-4 times the time based on the model.

On further investigation, it was found that the time depends on the topology of the nodes for the job and the bisection bandwidth. To calculate the bisection bandwidth the individual link speed as well as the topology has to be considered. We used a tool to obtain the topology of the nodes for a particular job. The links of BlueGene/Q is bidirectional with maximum throughput of 2 GiB/s per direction. For 16,384 cores 512 nodes were used with 32 threads per node. The physical topology for our 512 nodes is $4 \times 4 \times 4 \times 4 \times 2$ with torus links, therefore the bisection bandwidth is $2 \times 4 \times 4 \times 4 \times 2 \times 4\text{GiB/s}$. The bisection rate β_{bi} for 512 nodes is $1/(2 \times 128 \times 4 \times 2^{30}) = 9.09\text{E}-13$ seconds/byte. For 8,192, which is 256 nodes, we use less than a midplane, therefore we use a mesh with a bisection rate of $3.63\text{E}-12$ seconds/byte. Table 3.2 gives the nodes, topology and the bisection bandwidth for different core counts. It is interesting to note that the topology obtained for 2048 nodes is $4 \times 4 \times 4 \times 4 \times 16 \times 2$ instead of $4 \times 4 \times 4 \times 8 \times 8 \times 2$. Since this is a randomized information propagation, in expectation half of the traffic will be using the bisection links. Using Eq 3.19,3.20 the time for information propagation due to limiting bisection bandwidth is shown in Table 3.3. The time predicted by the model is close to the actual time recorded from the experiments.

This analysis gives us more insights into the algorithm and shows that the gossip is communication intensive and affected by the injection bandwidth. We can consider improvements that reduce long hop large messages. For example: One modification for the algorithm could be to change the way peers are chosen to forward the information. When gossip starts, the message size is small and they can

Cores	Nodes	Topology	Bisection bandwidth (bytes/sec)	β_{bi} (secs/byte)
8192	256	$2 \times 4 \times 4 \times 4 \times 2$	2.75E+11	3.63E-12
16384	512	$4 \times 4 \times 4 \times 4 \times 2$	1.10E+12	9.09E-13
32768	1024	$4 \times 4 \times 4 \times 8 \times 2$	1.10E+12	9.09E-13
65536	2048	$4 \times 4 \times 4 \times 16 \times 2$	1.10E+12	9.09E-13

Table 3.2: Bisection bandwidth for different node counts

cores	$\log_2 p$ (ms)	model $\log_2 p$ (ms)	$\frac{1}{2} \log_2 p$	model $\frac{1}{2} \log_2 p$ (ms)
8192	4.40E+02	2.30E+02	3.40E+00	2.59E+00
16384	9.00E+02	4.60E+02	6.00E+00	3.64E+00
32768	4.54E+03	9.20E+02	1.40E+01	5.13E+00
65536	8.90E+03	1.84E+03	2.80E+01	7.23E+00

Table 3.3: Time for gossip using α and β model

be sent to far off nodes, but when the gossip messages start to increase in size considerably, then these can be forwarded to near by nodes.

$$T_{log} = \log_2 p \times \alpha + p \times (p - 1)n\beta_{bi} \quad (3.19)$$

$$T_{0.5log} = \frac{1}{2} \log_2 p \times \alpha + p \times (\sqrt{p} - 1)n\beta_{bi} \quad (3.20)$$

cores	$\log_2 p$ (ms)	model $\log_2 p$ using β_{bi} (ms)	$\frac{1}{2} \log_2 p$ (ms)	model $\frac{1}{2} \log_2 p$ using β_{bi} (ms)
8192	4.40E+02	3.90E+02	3.40E+00	4.35E+00
16384	9.00E+02	3.90E+02	6.00E+00	3.12E+00
32768	4.54E+03	1.56E+03	1.40E+01	8.68E+00
65536	8.90E+03	6.25E+03	2.80E+01	2.44E+01

Table 3.4: Gossip time taking bisection bandwidth into account

3.4.4 Probabilistic Assignment

In the assignment phase, each overloaded processor transfers the excess load to a randomly chosen underloaded processors from its list. To select processors we use the inversion method for generating samples from a probability distribution. More formally, if $p(x)$ is a probability density function, then the cumulative distribution function $F(y)$ is defined as:

$$F(y) = p(x < y) = \int_{-\infty}^y p(x)dx \quad (3.21)$$

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

$$y_s = F^{-1}(r_s) \quad (3.22)$$

Using the above, we randomly select the processors according to p_i for transferring load. To select the object to be transferred, a min heap is constructed out of all the object residing on that processor. Typically there are 20-50 objects per processing element. For each transfer, the computation required is $O(n)$ where n is the number of underloaded processors at the sender. This can be optimized to use binary search instead of linear search.

The clock speed of BlueGene/Q is 1.6 GHz, i.e., one cycle takes 6.25E-10 seconds. The overall time for this phase is a few tens of microseconds and is small in comparison to the other phases.

Migration

Migrating an object involves packing the object data into a message at the sender side, sending the message, and unpacking it at the receiver end and registering it

cores	migrations	migration time(ms)	model time(ms)
8192	330	3	1.7
16384	416	3	1.7
32768	1322	4	1.7
65536	1386	5	1.7

Table 3.5: Migration time

with the runtime system. In our experiments, the size of each object being migrated is $1.96\text{E}+02$ KB. The time for migration will depend on the number of messages received per processing unit. Table 3.5 shows the number of objects migrated and the performance analysis for it. In all these experiments, there is only one object received at the underloaded processors. Using the α and β cost model, the time taken to transfer the message is computed. This model does not use the time taken for packing and unpacking of the object thereby the model time is a lower bound.

3.5 Implementation

We provide an implementation of the proposed algorithm as a load balancing strategy in CHARM++. Details of the CHARM++ programming model are given in Chapter 2.

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

Although we have described the *GrapevineLB* algorithm in terms of rounds, an

implementation using barriers to enforce the rounds will incur considerable overhead. Therefore, we take an asynchronous approach for our implementation. But such an approach poses the challenge of limiting the number of messages in the system. We overcome this by using a *TTL* (Time To Live) based mechanism which limits the circulation of information. It is implemented as a counter embedded in the messages being propagated. The first message initiated by an underloaded processor is initialized with the *TTL* of desired number of rounds before being sent. A receiving processor incorporates the information and sends out a new message with updated information and decremented *TTL*. A message with $TTL = 0$ is not forwarded and is considered expired. The key challenge that remains is to detect quiescence, i.e. when all the messages have expired. To this end, we use a distributed termination detection algorithm [37].

3.6 Evaluation

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

3.6.1 Evaluation using Simulation

We first present simulation results of the *GrapevineLB* strategy on a single processor. This simulation allows us to demonstrate the effect of various choices made in different stages of the algorithm. For the simulations, the system model is a set of 8192 processors, initialized with load from a real run of an adaptive mesh refinement application with same number of cores on an IBM BG/Q. This application was decomposed into 253,405 work units. Figure 3.3 shows the load distribution for this application when the load balancer was invoked. The average load of the system is 35, the maximum load is 66, therefore \mathcal{I} , the metric for imbalance from

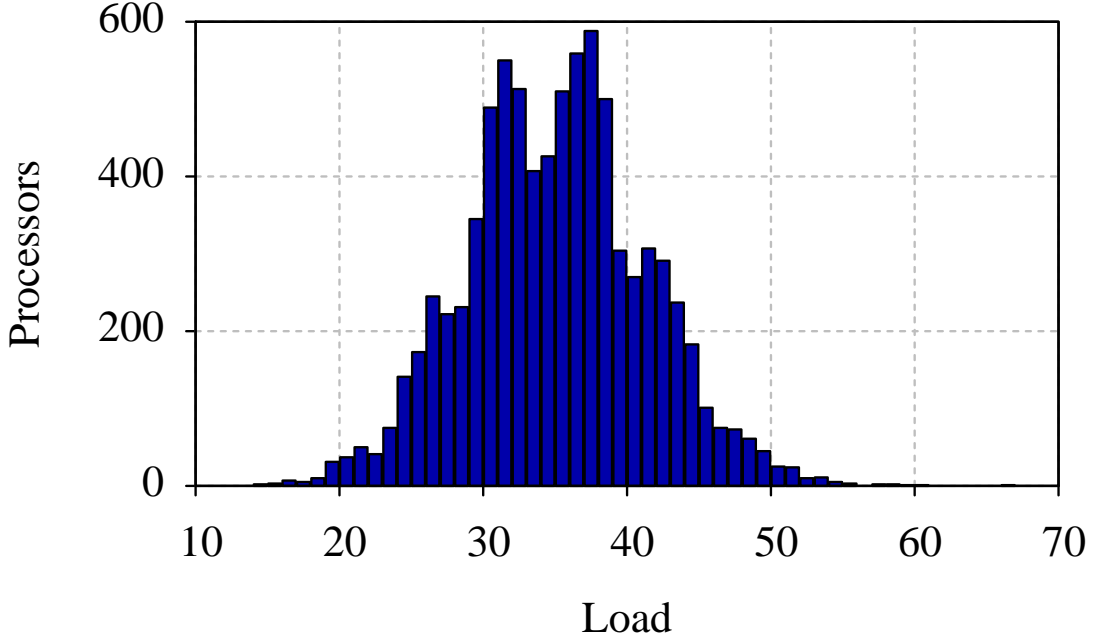


Figure 3.3: Load distribution for a run of AMR used in simulation. Counts of processors for various loads are depicted.

Equation 3.1, is 0.88. Note that the value of $I \approx 0$ indicates perfect balance in the system. Among the 8192 processors, 4095 are overloaded and 4097 are either underloaded or have their load close to average. We perform a step-by-step analysis of all the stages of the proposed algorithm based on this system model. It is to be noted that we have simulated synchronous rounds. The experiments were run 50 times and we report the results as mean along with their standard deviation.

Number of Rounds and Fanout: Figure 3.4 illustrates the expected number of rounds required to spread information on the system size. Here we consider only one source initiating the propagation and report when 99% of processors have received the information. As the system size (n) increases, the expected number of rounds increase logarithmically, $O(\log n)$, for a fixed fanout. This matches our analysis in Section 3.4. Note that the number of rounds decreases with increase in the fanout used for the information propagation. A system size of $16K$, fanout of 2, requires 17 rounds to propagate information to 99% processors whereas, fanout

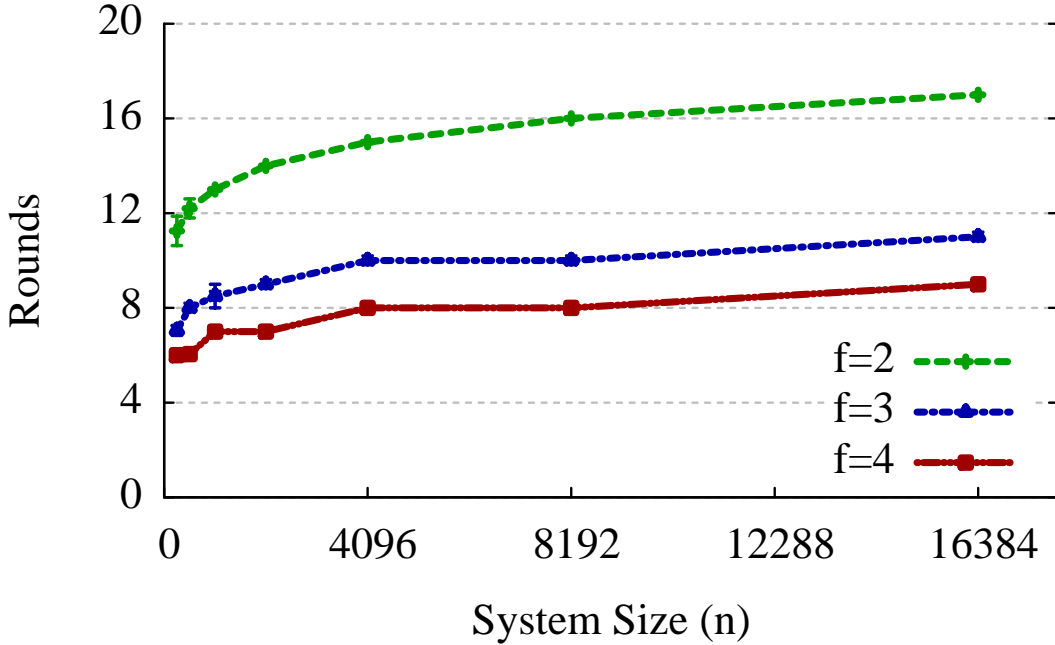


Figure 3.4: Expected number of rounds taken to spread information from one source to 99% of the overloaded processors for different system sizes and fanouts(f).

of 4, takes 8 rounds.

Naive vs. Informed Propagation: Figure 3.5 compares the expected number of rounds taken to propagate information using *Naive* and *Informed* propagation schemes. Although the expected number of rounds for both the schemes is on the order of $O(\log n)$, the *Informed* scheme takes one less round to propagate the information. This directly results in the reduction of the number of messages as most of the messages are sent in the later rounds. We can also choose to vary the fanout adaptively to reduce the number of rounds required, while not increasing the number of messages significantly. Instead of having a fixed fanout, we increase the fanout in the later stages. This is based on the observation that messages in the initial stages do not carry a lot of information. We evaluated this for a system of 4096 processors where 50% were overloaded. Information propagation without the adaptive variation requires 13 rounds with a total of 79600 messages. While an adaptive fanout strategy, where we use a fanout of 2 initially and increase the fanout to 3 beyond

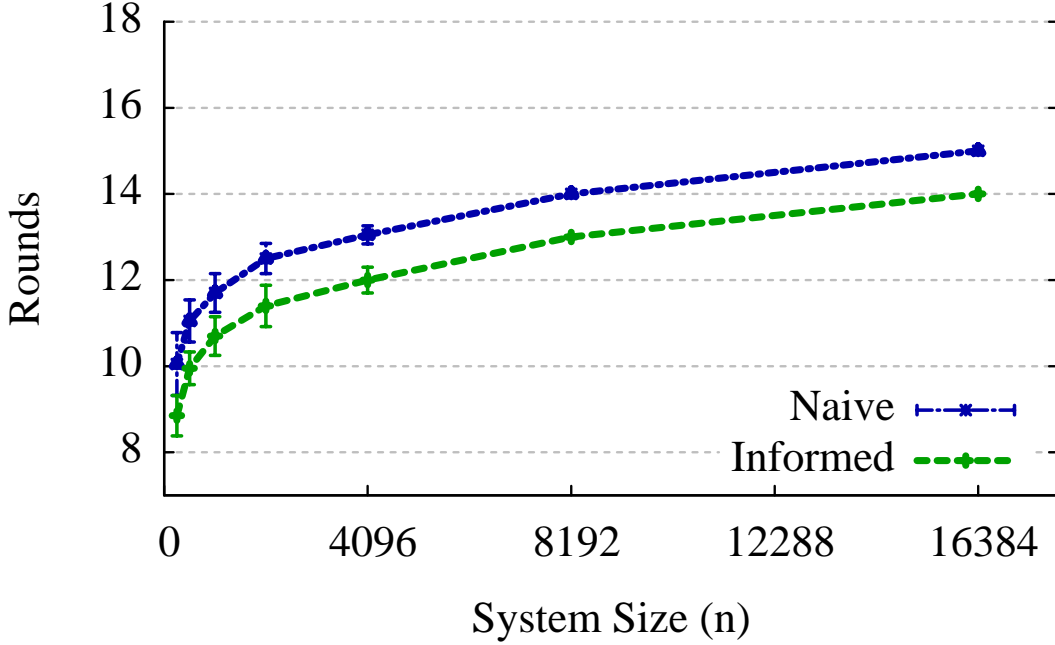


Figure 3.5: Expected number of rounds taken to spread information from one source to 99% of the overloaded processors using Naive and Informed schemes for different system sizes. Here $f = 2$ and 50% of the system size is underloaded

5 rounds and further increase to 4 beyond 7 rounds, helps reduce the number of rounds to 10 with a total of 86400 messages.

Naive vs Informed Transfer: We compare the performance of the two randomized strategies for transfer given in Section 3.3. Figure 3.1 shows the *Naive* scheme for the transfer of load where an underloaded processor is selected uniformly at random. Here we also show the probability distribution of the underloaded processors for the *Informed* transfer strategy using the Equation 3.2 and the transfer of load which follows this distribution, which are shown in Figure 3.1. It shows the initial load distribution of the underloaded processors, probability assigned to each processor (uniform distribution), number of transfers based on the probability distribution and the final load of the underloaded processors. It can be seen that the maximum load of the initially underloaded processors is 44 while the average is 35. Comparison with Figure 3.1 clearly shows that the final distribution of load is

much more reasonable. Further, the maximum load of the underloaded processors is 38 while the system average is 35.

Evaluation of a Pathological Case: We evaluate the behavior of the proposed algorithm under the pathological case where just one out of 8192 processors is significantly overloaded (\mathcal{I} is 6.18). Analysis in Section 3.4 shows that q_s decreases rapidly with rounds for a particular source. Since all underloaded processors will initiate information propagation, this scenario shouldn't be any worse in expectation. We experimentally verify this and find that for a fanout value of 2 and using the *Naive* strategy for information propagation, it takes a maximum of 14 rounds to propagate the information which is similar to the case where many processors are overloaded. Once the information is available at the overloaded processor, it randomly transfers the work units, reducing the \mathcal{I} from 6.18 to 0.001.

Evaluation of Quality of Load Balancing: To answer the question posed in the earlier section as to what happens if the overloaded processors have incomplete information, we simulate this scenario by providing information about only a partial subset of underloaded processors to the overloaded processors. The subset of underloaded processors for each processor is selected uniformly at random from the set of underloaded processors and the probabilistic transfer of load is then carried out based on this partial information. The quality is evaluated based on the metric \mathcal{I} given by Equation 3.1. Figure 3.6 shows the expected maximum load of the system along with standard deviation, σ and the value of \mathcal{I} metric. It can be seen that on one hand having less information, 10 – 50 underloaded processors, yields considerable improvement of load balance although not the optimal possible. On the other hand, having complete information is also not necessary to obtain good load balance. Therefore, this gives us an opportunity to trade-off between the overhead incurred and load balance achieved.

Evaluation of Information Propagation: Based on the earlier experiment,

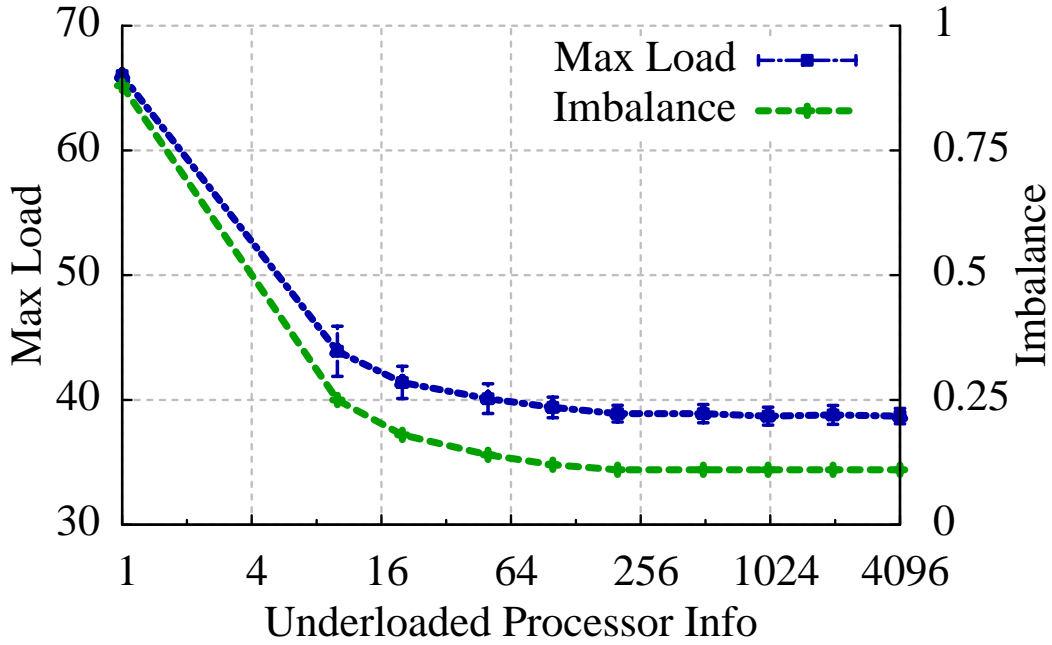


Figure 3.6: Evaluation of load balancer with partial information. Max load(left) and Imbalance(right) decrease as more information about underloaded processors is available. It is evident that complete information is not necessary to obtain good performance.

it is evident that complete information about the underloaded processors is not required for good load balance. Therefore, we evaluate the expected number of rounds taken to propagate partial information about the underloaded processors to all the overloaded processors. Figure 3.7 shows the percentage of overloaded processors that received the information as the rounds progress for a fanout of 2. The x-axis is the number of rounds and the y-axis is the percentage of overloaded processors who received the information. We plot the number of rounds required to propagate information from 200, 400, 2048 and 4097 underloaded processors to all overloaded processors. In the case of propagating information about at least 200 underloaded processors in the system, 100% of the overloaded processors receive information about at least 200 underloaded processors in 12 rounds and 99.8% received the information in 9 rounds. It took 18 rounds to propagate information about all the underloaded processors in the system to all the overloaded processors.

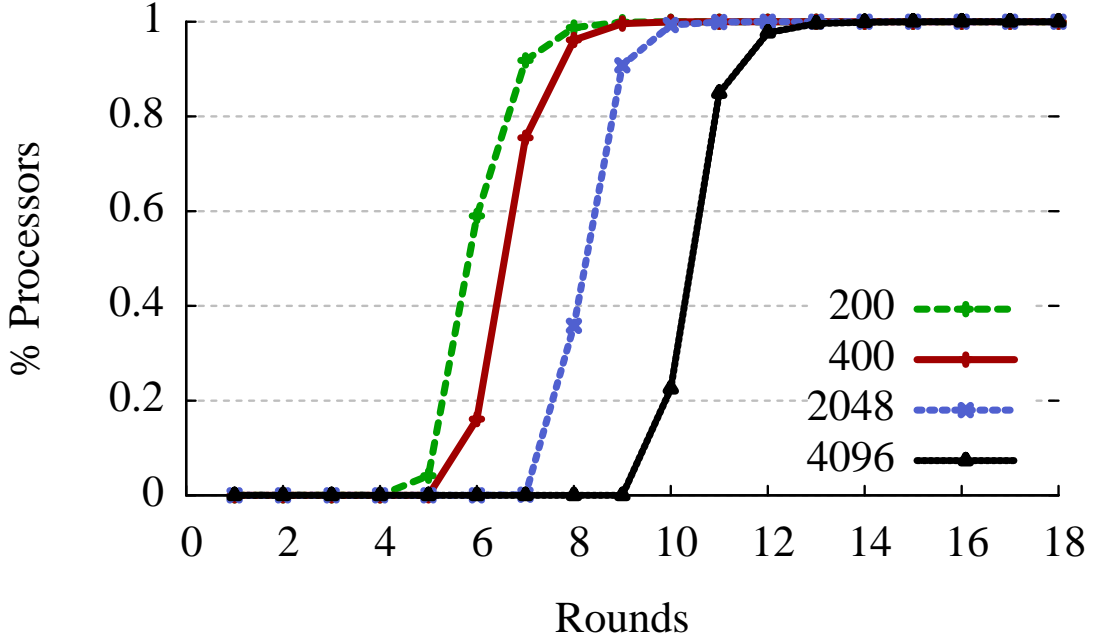


Figure 3.7: Percentage of processors having various amounts of partial information as rounds progress. There are a total of 4096 underloaded processors. 99% receive information about 400 processors by 8th round while it takes 12 rounds for all the 4096 underloaded processors.

This clearly indicates that if we require only partial information, and that the total number of rounds can be reduced, which will result in reduction of the load balancing cost.

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

3.6.2 Evaluation using Applications

We evaluate our *GrapevineLB* load balancing strategy on two applications, LeanMD and adaptive mesh refinement (AMR), by comparing against various other load balancing strategies. We use *GrapevineLB* with a fixed set of configurations, $\{f =$

2, $TTL = 0.4 \times \log_2 n$, Informed Propagation, Informed Transfer}, and focus on comparing with other load balancing strategies. Results presented here are obtained from experiments run on the IBM BG/Q Mira. Mira is a 49,152 node Blue Gene/Q installation at the ALCF. Each node consists of 16 64-bit PowerPC A2 cores run at 1.6GHz. The interconnect in this system is a 5D torus. In the following sections, we first provide details about the applications and the load balancers and then present our evaluation results.

Applications

Adaptive Mesh Refinement: AMR is an efficient technique used to perform simulations on very large meshes that would otherwise be difficult to simulate, even on modern-day supercomputers. Our AMR based application simulates a popular yet simple partial differential equation called Advection. It uses a first-order upwind method in 2D space for solving the advection equation. The simulation begins on a coarse-grained structured grid of uniform size. As the simulation progresses, individual grids are either refined or coarsened. This leads to slowly-growing load imbalance that requires frequent load balancing to maintain high efficiency of the system. This application has been implemented using the object-based decomposition approach in CHARM++ [38].

LeanMD: It is a molecular dynamics simulation program written in CHARM++ that simulates the behavior of atoms based on the Lennard-Jones potential. The computations performed in this code are similar to the short-range non-bonded force calculation in NAMD [39], an application that has won the Gordon Bell award. The three-dimensional simulation space consisting of atoms is divided into cells. In each iteration, force calculations are done for all pairs of atoms that are within a specified cutoff distance. For a pair of cells, the force calculation is assigned to a set of objects called the *computes*. After the force calculation is performed by the computes, the

cells update the acceleration, velocity and position of the atoms within their space. The load imbalance in LeanMD is primarily due to the variable number of atoms in a cell. The load on computes is proportional to the the number of atoms in the cells, which changes over time as the atoms move based on the force calculation. We present simulation results for a 2.8 million atom system. The load imbalance is gradual and therefore load balancing is performed infrequently.

Load Balancers

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

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

AmrLB: A centralized strategy that does refinement based load balancing taking into account the current distribution of work units. This is tuned for the AMR application [38].

HierchLB: A hierarchical strategy [13] in which processors are divided into independent groups and groups are organized in a hierarchical manner. At each level of the hierarchy, the root node performs the load balancing for the processors in its sub-tree. This strategy can use different load balancing algorithms at different levels. It is an optimized implementation that is used in strong scaling NAMD to more than 200K cores.

DiffusLB: A *neighborhood averaging* diffusion strategy [29, 32] where each processor sends information to its neighbors in a domain and load is exchanged based on this information. A domain constitutes of a node and all its neighbors where the neighborhood is determined by physical topology. On receiving the load infor-

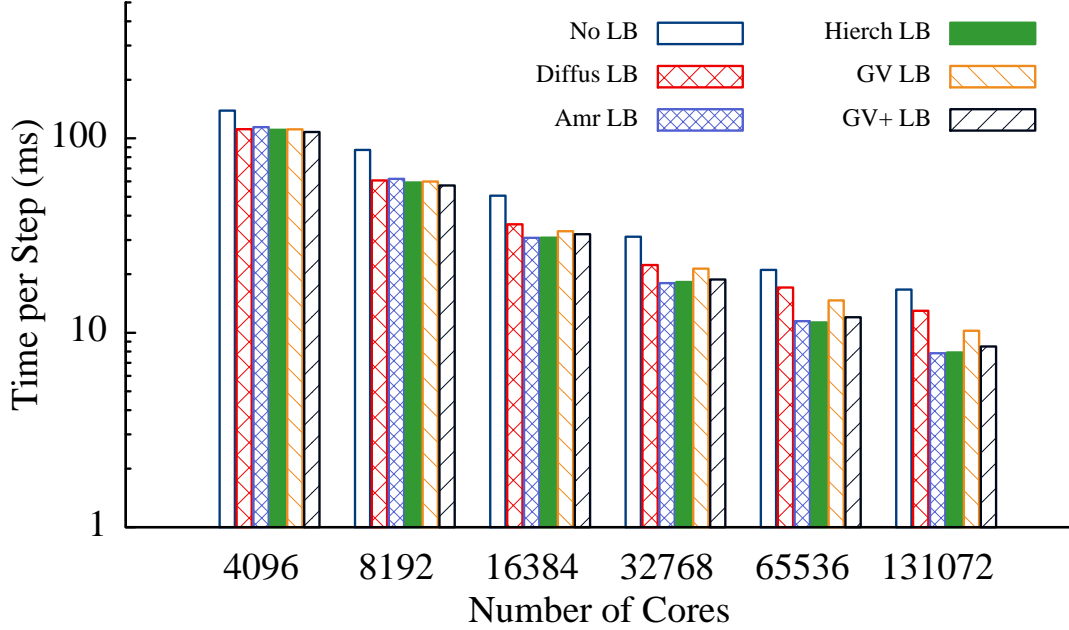


Figure 3.8: Comparison of time per step (excluding load balancing time) for various load balancing strategies for AMR on Mira (IBM BG/Q). GV+ achieves quality similar to other best performing strategies. Note that axes are log scale.

mation from all its neighbors, a node will compute the average of the domain and determines the amount of work units to be transferred to each of its neighbors. This is a two phase algorithm: in the first phase tokens are sent and in the second phase actual movement of work units is performed. There are multiple iterations of token exchange and termination is detected via quiescence [37].

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

Evaluation with AMR

We present an evaluation of different load balancing strategies on the AMR application on BG/Q ranging from 4096 to 131072 cores. AMR requires frequent load

balancing to run efficiently because coarsening and refinement of the mesh introduces dynamic load imbalance.

Time per Iteration: First we compare the execution time per iteration of the application to evaluate the quality of the load balancers. This directly relates to \mathcal{I} metric given in Equation 3.1 because as $\mathcal{I} \rightarrow 0$, the maximum load of the system approaches the average load, resulting in the least time per iteration. Figure 3.8 shows, on a logarithmic scale, the time taken per iteration with various load balancing strategies. The base run was made without any load balancing and is referred to as *NoLB*. It is evident that with *NoLB* the efficiency of the application reduces as it is scaled to higher number of cores. The *Grapevine+LB* load balancer (shown as GV+ LB) reduces the iteration time by 22% on 4K cores and 50% on 131K cores. *AmrLB* and *HierchLB* also show comparable performance for this metric. We see an increase in gain because, on larger number of cores, the load imbalance becomes significant. This is because the number of work units per processor decreases and the chance that a processor becomes overloaded increases. *DiffusLB* also shows some improvement, but much less than the aforementioned ones on larger scale. For 131K, it reduces the time per step by 22%, while others (*AmrLB*, *HierchLB* and *Grapevine+LB*) reduce it by 50%. An interesting thing to note here is that the *Grapevine+LB* load balancer performs better than *GrapevineLB* (shown as GV LB) for core counts of more than 32K. This is due to the fact that *Grapevine+LB* ensures that no underloaded processor gets overloaded using a Nack mechanism. From this it is evident that the quality of load balance performed by *Grapevine+LB* is at-par with the quality of the centralized and hierarchical strategies.

Overhead: Table 3.6 shows the overhead incurred by various load balancers in one load balancing step for different system sizes. The overhead(load balancing cost) includes the time for finding the new assignment of objects to processors and the time for migrating the objects. The overhead incurred by *AmrLB* is 2.01 s

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

Table 3.6: Average cost (in seconds) per load balancing step of various strategies for AMR

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

Table 3.7: Total application time (in seconds) for AMR on BG/Q. Proposed strategies Gv and Gv+ perform the best across all scales.

for 4K cores and increases with the increase in the system size to a maximum of 21.14 s for 131K cores. *HierchLB* incurs an overhead of 5.5 s for 8K cores and thereafter the cost reduces to a minimum of 0.29 s for 131K cores. This is due to the fact that as the number of processors increases, the number of sub groups also increase resulting in a reduction of work units per group. Hence, the time taken for the root to carry out the load balancing strategy reduces. The distributed load balancing strategies, *GrapevineLB* and *DiffusLB*, incur considerably less overhead in comparison to other strategies.

Total Application Time: The total application time using various strategies is given in Table 3.7. In this application frequent load balancing is required. The overhead of the centralized strategies diminishes the benefit of load balancing. *AmrLB* does not improve the total application time because of its overhead. This is true for the hierarchical strategy as well. The *DiffusLB* results in a reduction

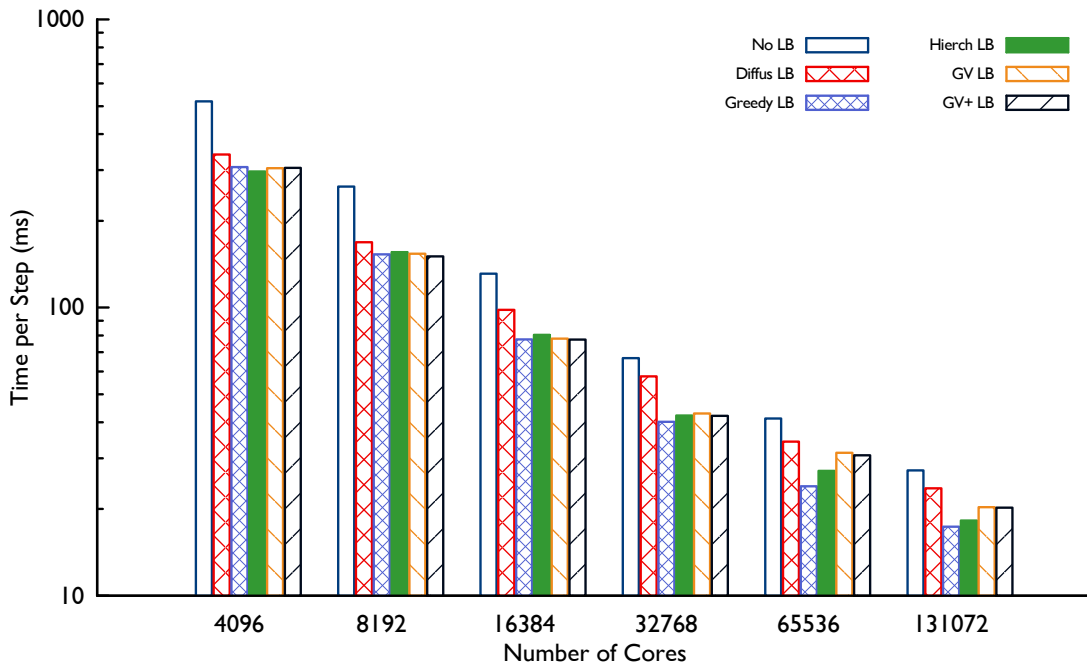


Figure 3.9: Comparison of time per step (excluding load balancing time) for various load balancing strategies for LeanMD on Mira (IBM BG/Q). Note that axes are log scale.

of the execution time by 28% for 16K cores and 24.8% for 131K cores where as *GrapevineLB* gives a reduction of 35% and 49.6% respectively. *GrapevineLB* provides a large performance gain by achieving a better load balance and incurring less overhead. It enables more frequent load balancing to improve the efficiency. A future direction would be to use MetaBalancer [40] to choose the ideal load balancing period.

Evaluation with LeanMD

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

Execution time per iteration: We compare the execution time per iteration of the application to evaluate the quality of the load balancers. For 4K to 16K cores, the centralized, hierarchical and *GrapevineLB* strategies improve the balance up to 42%. The diffusion-based strategy improves the balance only by 35% at 8K

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

Table 3.8: Average cost per load balancing step (in seconds) of various strategies for LeanMD

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

Table 3.9: Total application time (in seconds) for LeanMD on BG/Q

cores and there after it shows diminishing gains. *GrapevineLB* on the other hand performs at-par to the centralized load balancer up to 32K. At 131K cores, it only gives an improvement of 25% in comparison to 36% given by centralized scheme. This reduction is because the number of tasks per processor decreases to 4 at 131K, causing refinement-based load balancers to perform suboptimally. *GrapevineLB* is consistently better than the *DiffusLB* because it has a representation of the global state of the system which helps it make better load balancing decisions.

Overhead: Table 3.8 presents a comparison of overhead incurred by various strategies for a single load balancing step. The load balancing cost of the centralized strategy is very high and is on the order of tens of seconds. The high overhead of *GreedyLB* is due to the overhead of statistics collection, making the decision at the central location and the migration cost. The hierarchical strategy, *HierchLB*, incurs less overhead. It takes 3.7 s for 4K cores and decreases to 0.26 s as the system size increases to 131K. The overhead of *DiffusLB* is 0.080 s for 4K cores and decreases

thereafter. This is because the number of work units per core decreases as the number of cores increase. Finally, we observe that *GrapevineLB* has an overhead of 0.017 s for 4K cores and decreases with increase in system size to 0.013 s for 16K cores and thereafter increases to 0.018 s for 131K. The load balancing cost for *GrapevineLB* includes the time for information propagation and transfer of work units. At 4K cores the load balancing time is dominated by the transfer of work units. As the system size increases, the work units per processor decreases. This results in cost being dominated by information propagation.

Total Application Time: Table 3.9 shows the total application time for LeanMD. The centralized strategy improves the total application time but only for core counts up to 16K. Beyond 16K cores, the overhead due to load balancing exceeds the gains and results in increasing the total application time. *DiffusLB* incurs less overhead in comparison to the centralized and hierarchical strategies but it does not show substantial gains because the quality of load balance is not good. At 32K cores, it gives a reduction of 12% in total execution time while *GrapevineLB* gives 34% and *HierchLB* gives 33%. *HierchLB* incurs less overhead in comparison to the centralized strategies. It reduces the total execution time by 37% for 8K cores while *GrapevineLB* reduces it by 42%. *GrapevineLB* consistently gives better performance than other load balancing strategies. *Grapevine+LB* gives the maximum performance benefit by reducing the total application time by 20% for 131K, 40% for 16K cores, around 42% for 4K and 8K cores. Thus, *GrapevineLB* and *Grapevine+LB* provide an improvement in performance by achieving a high quality load balance with significantly less overhead.

3.7 Conclusion

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

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

4 Graph Partitioner Based Load Balancers

The efficient use of large parallel machines requires spreading the computational load evenly across all processors and minimizing the communication overhead. When the processes/tasks that perform the computation co-exist for the entire duration of the parallel program, the load balance problem can be modeled as a constrained graph partitioning problem on an undirected graph. The vertices of this process graph represent the computation to be performed and its edges represent inter-process communication. The problem of mapping these processes/tasks to processors can be viewed as the partitioning and mapping of a graph of n tasks to that of p processors. The aim is to assign the same load to all processors and to minimize the edge cut of the graph, which translates to reducing communication between processors.

This chapter evaluates the use of graph partitioning algorithms, traditionally used for partitioning physical domains/meshes, for measurement-based dynamic load balancing of parallel applications. We implement various load balancing strategies in Charm++ using graph partitioners, such as Metis, Scotch and Zoltan.

4.1 Overview

A process graph that models the computation is constructed to compute a mapping. The vertices of the process graph represent the computational load and the edges represent the communication between the processes. The set of processors onto which the processes are mapped is also modeled as a graph, called the target graph. The objective of the mapping algorithm is to obtain the load balance within

the specified imbalance threshold while minimizing communication cost. We implemented load balancing strategies in Charm++ that use graph partitioning methods available in Scotch, Metis and Zoltan graph partitioners. These are described below:

- **MetisLB:** A strategy that passes the load information and the communication graph to METIS, a graph partitioning library, and uses the recursive graph bipartitioning algorithm in it for load balancing.
- **ZoltanLB:** A hypergraph partitioning based load balancer which uses ZOLTAN.
- **ScotchLB:** A strategy that uses SCOTCH graph partitioning library to make load balancing decisions.

The two main classes of algorithms used to compute static mappings are direct k -way methods and recursive bipartitioning methods. Both k -way and bipartitioning methods are based on the multilevel graph partitioning paradigm which helps reduce the problem complexity and execution time. The multilevel paradigm consists of three phases: graph coarsening, initial mapping, and uncoarsening. In the graph coarsening phase, the graph is repeatedly coarsened into a series of smaller graphs. The graph at each step is derived from the previous graph by collapsing adjacent pairs of vertices. In the initial mapping phase, mapping is performed on the coarsest graph. Finally in the uncoarsening phase, the mapping of the coarsest graph is projected back to the original input graph [41, 42]. After each uncoarsening step, the mapping is refined to improve the quality using algorithms such as the Kernighan-Lin (KL) [43] and Fiduccia-Mattheyses (FM) [44]. The KL algorithm minimizes the edge cut by performing swaps between pairs of vertices, and hence the time complexity is quadratic in the number of vertices. This algorithm moves vertices between partitions, but it cannot perform major changes to the projected partitions. SCOTCH uses FM-based algorithms for bipartitioning and k -way mapping. Fiduccia-

Mattheyses is a modification of the KL algorithm that improves the time complexity without significantly decreasing the quality.

Multicast-Aware Load Balancer using Zoltan: In many HPC applications, the use of collective operations is crucial for performance and scalability. In a multicast communication, information is addressed to a group of destination processes or objects. Many applications use multicast to send information to a subset of objects. For example, in the case of LeanMD, which is a molecular dynamic simulation benchmark, the *Cell* objects send information to the *Compute* objects. The *Compute* objects calculate the forces for all pairs of atoms and sends it back to the corresponding *Cell* objects. This uses a multicast collective operation. The multicast operation is done via a multicast tree on the processors holding the destination objects. If the objects are spread across the system, this can result in a more expensive multicast operation.

We explore the use of the hypergraph partitioning algorithm in Zoltan for multicast-aware load balancing. 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 a special case of a hyperedge that connects two vertices). In a multicast communication a message is sent from a source to a group of destination objects. These objects form the vertices of the hypergraph and the multicast forms a hyperedge in the hypergraph. This hypergraph is provided to Zoltan, which partitions the objects and provides a mapping of them onto processors.

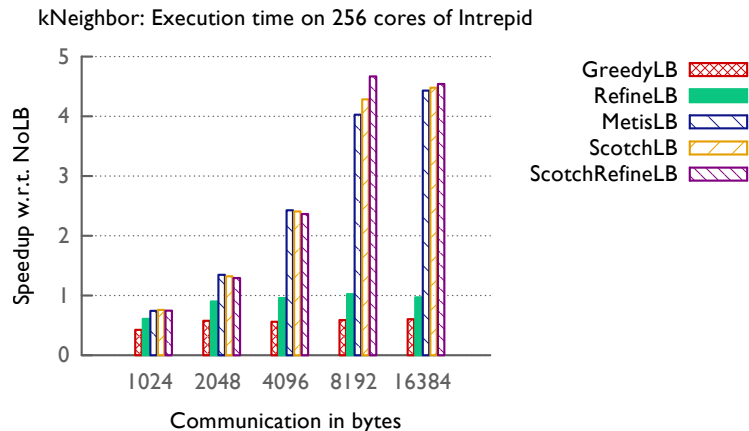


Figure 4.1: Impact of increasing communication on the quality of load balance (kNeighbor running on 256 cores of Intrepid)

4.2 Evaluation

We compare the performance of different load balancing strategies using a micro-benchmark. We first provide details about the kNeighbor benchmark and then present our evaluation results.

kNeighbor is a micro-benchmark with a near-neighbor communication pattern. In this benchmark, each object exchanges fixed sized messages with a fixed set of objects in every iteration. Each object is assigned a random computational load.

All the experiments were run on Intrepid, a 40960-node Blue Gene/P installation at the Argonne National Laboratory. Each node on Intrepid consists of four 850 MHz PowerPC cores. The primary interconnect for point-to-point communication in this system is a 3D torus with a bi-directional link bandwidth of 850 MB/s. The experiments were run in VN mode, i.e., using all four cores per node.

Figure 4.1 shows the speedup obtained in execution time per step with respect to NoLB for different message sizes. As we increase the message size from 2 KB to 16 KB, the improvement in the time per step using the graph partitioning based load balancers over RefineLB increases from 30% to 79%. When compared to the

baseline performance, the graph partitioning based load balancers give an overall speedup of 1.2 to 4.6 when varying the message size from 2 KB to 16 KB. Hence, graph partitioning based load balancers should definitely be used with applications that are communication-intensive.

Multicast-Aware LB We use the *leanmd* mini-application to compare the performance of multicast-aware load balancers, namely MultiCastLB, with MetisLB and ScotchLB. *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)
MultiCastLB	5.5	7
MetisLB	8.5	10.15
ScotchLB	8.2	10.9

Table 4.1: Comparison between different load balancers using average number of multicast messages sent for *leanmd*

Table 4.1 shows the impact of using MultiCastLB over MetisLB and ScotchLB for multicast-aware load balancing. We show that ZoltanLB reduces the number of average number of multicast messages by mapping chares participating in a multicast onto the same processors. Reducing the multicast messages will reduce the congestion and contention in the network and thereby improve the performance of the application.

4.3 Conclusion

Graph partitioners, such as Scotch, Metis and Zoltan, were used with the measurement based load balancing framework in Charm++. Various graph partitioner based load balancers were implemented and a performance comparison study was

done. Graph partitioner based load balancers were successful in improving the performance of communication intensive applications. ZoltanLB, which is a hyper-graph partitioner, was used for multicast-aware load balancing to reduce the number of multicast messages by placing the objects participating in a multicast within a processor. Graph partitioner based load balancers provide good quality partitions while minimizing the communication overhead, but at a high load balancing cost. Therefore, these expensive, high-quality graph partitioning algorithms are used infrequently while cheaper refinement based algorithms are used more often.

5 Handling Imbalance in Distributed and Shared Memory

Several trends in high-performance computing are converging to drive applications and systems software to rely on multi-threading in each node’s shared memory, rather than running an independent process on each CPU core. Increasing per-chip concurrency creates pressure on system memory, system software, and application design. The general abandonment of specialized OS kernels [45, 46] in favor of general-purpose Linux has rolled back past efforts to reduce system noise [47]. Finally, CPU heterogeneity [1] and increasing application sophistication both increase load imbalance and unpredictability. In this thesis, we present a combination of the Charm++ and Adaptive MPI distributed programming models with a new parallel loop and concurrent task constructs that addresses many of these challenging trends with a low-overhead and locality-conscious design.

The number of cores and threads in each chip is increasing rapidly. Within each node, increased hardware parallelism entails reduced per-core/thread memory capacity and bandwidth. Over entire parallel systems, treating each core as an independent unit forces communication libraries to consume more memory and pushes collective algorithms further toward asymptotic scaling limits. Applications that wish to use each core independently must be structured to expose a correspondingly large and growing degree of parallelism. General whole-job load balancing mechanisms must then address the increased scale of both systems and applications. Thus, the prospect of grouping many cores together as multi-threaded units mitigates many threats to continued performance scaling.

Many parallel applications no longer work in a regime where work and data can

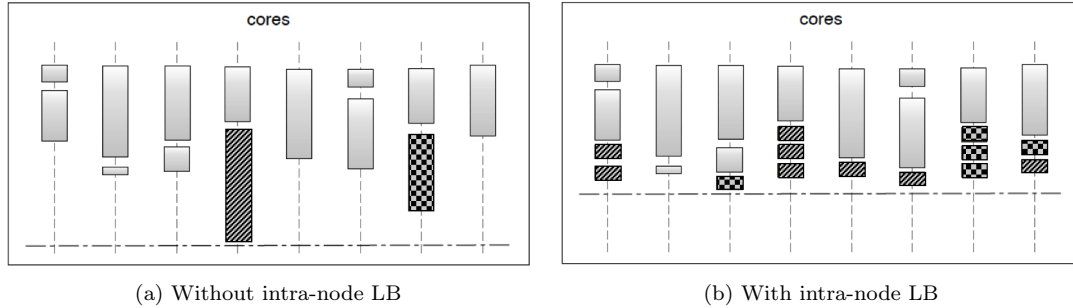


Figure 5.1: The potential benefits of intra-node work sharing on reducing load imbalance

be neatly divided into uniform chunks distributed to each processor. This trend encompasses unstructured computations, data-dependent iterative methods, variable resolution, multi-physics simulations, multi-phase execution, and many other developments that trade reduced total work or increased accuracy for more complicated and less predictable execution. Even applications that do offer simple structured decompositions are made imbalanced by hardware heterogeneity. Load balancing in various forms can be applied to aid these applications, but it too must be scalable, which often means coarsening the problem to the node level to avoid considering an excessive number of cores. Discrete units of work assignment, heuristic algorithms, and unpredictable processor performance also prevent perfect uniformity. Supplementary within-node balancing can help make up for these short-falls, as illustrated in Figure 5.1.

Even with a very balanced work assignment across nodes and individual cores, execution may not proceed at a perfectly uniform pace. Network contention can delay some messages more than others. System noise from OS processes can also non-uniformly interfere with execution [47], with hard to predict knock-on effects [48]. Dynamic work redistribution can greatly help in mitigating these effects [49].

All of these pressures lead to a conclusion that multiple cores within each node must share data and work to sustain continued scalability in problem size and per-

formance. At the same time, any sharing mechanism ideally should not compromise data locality or introduce excessive new bottlenecks or overheads. To address these desires, we introduce a design that combines the Charm++ and Adaptive MPI distributed programming models with both standard OpenMP and new parallel loop and concurrent task constructs. Charm++ intermittently performs coarse load balancing in terms of objects that encapsulate associated work and data together and assigns them to particular cores with good balance among nodes. These objects then adaptively share work with other cores in the same process, exposing fine-grained tasks only to the extent that otherwise idle cores are available to help execute them. Thus, our design ensures locality and low and proportionate scheduling overhead. We demonstrate this design’s effectiveness through the improved performance and scalability of several applications run on large supercomputers.

The contributions of this chapter are:

- An approach that combines infrequent global load balancing with shared-memory task parallelism to handle transient and persistent load imbalance.
- Efficient implementation of dynamic scheduling of fine-grained tasks which uses an adaptive schedule based on the state of the system.
- Improved performance by using the integrated runtime system on three different applications. We show improvements of 2X on ChaNGa on 128K cores and more than 3X on NAMD at 2048 cores. We also show the benefit on an MPI application, Kripke, using Adaptive MPI.

5.1 Related Work

Per-chip core and thread counts are steadily increasing in HPC systems. The trend toward increasing core/thread counts will accelerate with the increased deployment

of Knight’s Landing-generation Intel Xeon Phi hardware with several dozen cores per chip as primary processors rather than as accelerators (e.g., in NERSC’s Cori, LANL’s Trinity, and ANL’s Theta). This trend has driven scalability challenges and opportunities for increased efficiency arising from multiple cores sharing access to common memory. The ubiquitous MPI has correspondingly evolved in usage and implementation to work well in this setting [50, 51, 52, 53, 54, 55], leading to explicit support for shared memory in the MPI-3 standard. Charm++ has followed a similar progression, as described in Section 5.2.

The process-per-core model of pure MPI has not been universally sufficient. Applications may have limitations in the scalability of their parallel algorithms and data structures, or may present insufficient parallelism in their mode of work decomposition among MPI processes. Communication that could be avoided in shared memory is also an undesirable overhead. This has led to the rise of hybrid ‘MPI+X’ programming. OpenMP is the most prevalent shared-memory programming model paired with MPI, with extensive work studying its implementation and impact (e.g., [56, 57, 58]). The ‘MPI+X’ hybrid model has been increasingly used with other shared memory programming models to handle within node parallelism [56, 59, 60]. Similar work has been done with Charm++ as the distributed substrate, combined with both OpenMP¹ and the bespoke ‘CkLoop’ loop-multithreading mechanism [61], both of which we extend in the present work.

The MPI+X model on its own has been shown to improve load balance within each node [62]. We combine a periodic measurement-based inter-node load balancing scheme to attain approximate uniformity, with dynamic shared-memory execution to smooth out residual imbalances. A recent set of papers by V. Kale, Gropp, et al. have explored the hybrid model in more detail. They mix static and dynamic scheduling of work among cores on a node to improve the tradeoffs among overhead,

¹Unpublished work by Osman Sarood, 2011-2012

locality, and load imbalance [63, 64, 65]. They also show that these techniques can be used to reduce the impact of system noise [49]. Our work carries these ideas further, by adaptively tuning the level of dynamic scheduling to match its potential utility, thus pushing overhead lower.

Projects to more tightly integrate various shared and distributed memory models have also arisen, with aims to improve scheduling and locality further. OmpSs introduced concurrent tasks on top of OpenMP, with data dependences satisfied by MPI communication operations and coordinated by its runtime system. Recent versions of MPC bind an implementation of MPI that supports multiple ranks in each OS process [66] to multi-threading via POSIX threads [67], OpenMP [68], and Intel TBB. This work moves in a similar direction, by directly scheduling execution of various shared-memory tasks to run on normal Charm++ worker threads, overlaid on the work and data mappings generated by Charm++’s distributed memory load balancing infrastructure.

The approach of work-stealing task scheduling has been used in Cilk [3], Intel TBB [69], OpenMP 3.0 [70] and Habanero [71]. The randomized work-stealing used in Cilk can result in loss of locality. TBB has a mechanism to bind each loop iteration to the same worker thread that previously executed that iteration, thereby favoring temporal cache-reuse. The Habanero runtime system has an adaptive locality-aware work-stealing scheduler [72] to increase temporal data reuse.

5.2 Charm++ programming model for shared memory

Charm++ is a parallel programming system which is based on an asynchronous message driven execution model. Each application’s data and computations are encapsulated in entities called *chares*, which are C++ objects. An application

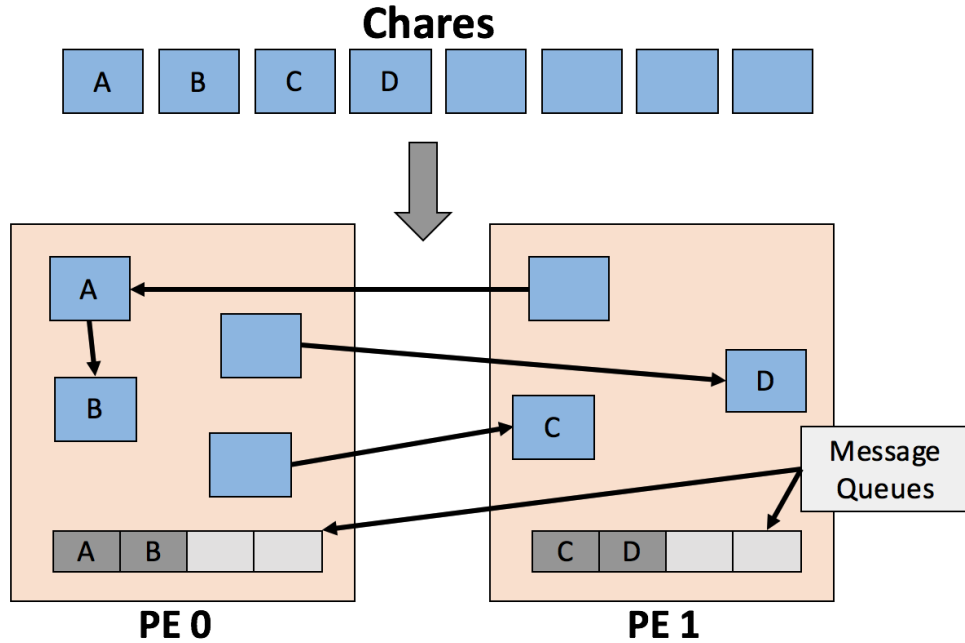


Figure 5.2: Charm++ Parallel Programming System

written in Charm++ is over-decomposed into these objects. Chares interact via asynchronous method invocations and a method on a chore is executed when a message is received for it. Chore objects are assigned to a core by the runtime system. Typically there are many more objects than the number of cores, which is known as over-decomposition. This encapsulation of data and its computation into a chore, each of which is mapped to a specific core, inherently promotes data locality.

In the message driven execution model of Charm++, the runtime system actively probes for incoming messages. On receiving a message, it identifies the corresponding *chore* that is targeted by the incoming message and schedules it. Figure 5.2 shows the overdecomposition where multiple *chares* are assigned to a PE and communicating via messages.

The SMP mode of Charm++ takes advantage of multi-core shared memory processors [61]. In this mode, a Charm++ OS process is called an SMP node and each SMP node launches multiple threads each called a PE. In a typical configuration the number of threads launched by the Charm++ process is equal to the number

of cores or hardware threads on a node. A PE is mapped to a separate core or a hardware thread. We use core, hardware thread and PE interchangeably. These threads (PEs) have CPU affinity, i.e., each PE is bound to a specific core and the operating system is not allowed to migrate the thread to another core. Each PE has a separate message queue and the scheduler on the PE picks up messages from the queue and handles it. Within an SMP node, data is shared between the PEs via pointers. Utilizing the shared memory multi-core processor in this way has many benefits. In SMP mode, intra-node communication is implemented via a single copy, rather than the double copy scheme used between nodes. It also significantly reduces the memory footprint of the program by eliminating the memory needed for intra-node communication channels and buffers. Since all PEs within an SMP node share a memory address space there needs to be only one copy of read-only data structures. Running multiple threads in a single process enables work sharing without explicit inter-process data transfer.

5.3 Overview

The challenge, as outlined in Section 5, is to balance load across PEs while managing locality. A pure task model with randomized work stealing, or a pure dynamic schedule in OpenMP, sacrifices locality significantly to an extent that often nullifies the benefits of dynamic load balancing [63, 65]. Dynamic load balancing strategies are used to balance the load and redistribute the work at runtime. These load balancing strategies can incur significant overhead due to the cost of computing a new assignment and the consequent data movement. If done less frequently, the overhead is reduced and locality is maintained, but dynamically emerging load imbalance may last longer before being corrected. With an increasing number of cores within a node, intra-node load balancing will become an effective way to reduce load

imbalance.

The approach we propose is to utilize a relatively infrequent periodic assignment of work to cores based on load measurement, combined with user assisted creation of potential tasks from the work assigned to each core that the runtime can choose to make available to other cores. The idea is to utilize the idle cycles on other cores on a node to execute tasks. We also need to make sure we do not incur task creation overhead when tasks are not needed. Figure 5.1 shows a schematic diagram of such a scenario where most of the computations are executed on the core they are assigned to, but the load imbalance towards the end triggers the dynamic creation of fine-grained tasks, which are distributed across different cores.

We support this approach with two methods for users to create potential tasks. The first method is a task abstraction that we have added to Charm++. The second one, which builds on it, is an integration of OpenMP with Charm++, such that each object can create potential tasks via OpenMP parallel loop constructs. Both of these are capable of creating potential tasks that can be used for dynamically utilizing all cores to restore balance. We also develop multiple runtime scheduling strategies for managing these potential tasks.

In the following sections we describe our approach in detail. We first discuss the periodic load balancing in Section 5.4. Then we describe the task model in Charm++ in Section 5.5. Finally, we showcase the application performance improvements achieved by using the new integrated runtime system in Section 5.6.

5.4 Persistence Based Load balancing

Many HPC applications execute the simulation in a series of time-steps or iterations until convergence is achieved. As a result, consecutive iterations have a similar computation and communication pattern. For such applications, a heuristic called

principle of persistence [6] holds, which says that the communication pattern and computation load of the recent past is a good indicator of near future. We use this to predict the load of future iterations; The predictions are used by the load balancing strategies to make the global decisions. We work with Charm++ because of its support for dynamic load balancing. As mentioned in the earlier Section 5.2, in Charm++ the data and its computation is encapsulated into a chare object which resides on a specific PE. A PE here refers to a processing element such as a core or a hardware thread. This naturally promotes locality. Load balancing aims to provide an assignment of these objects to PEs to reduce the load imbalance. The Charm++ load balancing framework provides a mechanism to collect the load and communication statistics of each chare object and the processor in a distributed database. These statistics are used by the load balancing strategies to generate a chare-to-core mapping at run time.

Charm++ contains a suite of load balancing strategies that balances load between PEs. For the purpose of this work, we use a two-level load balancing strategy for one of the applications, ChaNGa: the load is first balanced across nodes and then balanced within each node, both by assigning chares to PEs. This ensures that the load is distributed evenly among the nodes; it is also distributed evenly among cores of the node to the extent that the load predictions hold. The other load balancing strategy used in this chapter is a hierarchical load balancer. In this hierarchical strategy, the processors are divided into groups organized in a hierarchical tree fashion. At each level of the hierarchy, the root performs the load balancing strategy over the children in its sub-tree. The residual load imbalance that results in spite of this periodic balancing can be handled by the fine-grained intra node task balancing strategies described below.

5.5 Handling Residual and Transient Load Imbalance with Charm++ Task Model

5.5.1 Task API

We support two methods of task creation. One is using an API in Charm++ to support loop parallelization. The second one is creation of tasks via OpenMP's parallel loop construct which is built on top of the integrated run-time system. The following API is provided to the programmer to expose loop parallelism.

```
ParallelFor(funcptr, int argc, void* argv,  
            int start, int end, int step,  
            int redOp, void *redBuf,  
            callback*, int sync)
```

The *funcptr* is the pointer to the function that executes the chunk of work on any core within the node. We support a limited number of reduction operations. If *sync* is set then it does not return control until all the chunks are done executing. If *sync* is not set, then the control returns as soon as all the tasks have been picked by any of the cores. If a *callback* is set, then it is invoked once all the chunks of work are completed.

5.5.2 Task Generation and Scheduling

A straightforward way to schedule *parallel-for* tasks is to statically assign equal chunks of work to all the cores within the node as done by OpenMP's `static` schedule. This is not suitable for our case where worker threads may be busy with their own computation. If other cores are busy with their computation work, then they won't pick up the statically assigned task to execute. This will result in the

delay in completion of the *parallel-for* loop and wastage of CPU cycles at the caller. Alternatively, one could create all the chunks and push them into a common task queue from which other threads will pick work. This could have high overhead of task creation and contention at the shared task queue. We use a separate task queue for each PE which is described in detail in Section 5.5.3.

We explore other task generation and scheduling strategies many of which involve work-stealing such as done in Cilk [3].

Recursive ParallelFor Task Generation

Algorithm 3 describes the algorithm for recursive ParallelFor. In this mode, one task descriptor is created with all the information about the task. Typically the task descriptor contains the object pointer, function pointer, total number of chunks and an atomic variable to keep track of the number of finished chunks. In recursive ParallelFor task generation, the loop iterations are split into two halves (similar to the Cilk recursive spawn). A task message is created for one of the halves. This task message contain the iteration range and a pointer to the common task descriptor. The worker pushes the task message into the task queue and calls the function recursively for the other half. If the iteration range is within the chunksize, then the task is executed. The thief steals the task from the head of the queue. This ensures that a large fraction of the work is stolen. The thief will then generate more tasks, which are added to its task queue and starts working on chunks.

Broadcast Task Message

We have a single task descriptor with information about the task. A message containing a pointer to the task descriptor is sent to all the PEs within the node via a broadcast tree. Whenever the scheduler on a PE picks up the message, it repeatedly and atomically increments a variable to get the next chunk to work on and executes

Algorithm 3 Recursive Splitting

Input:

low - Lower Index of the Task Array
high - Higher Index of the Task Array
mid - Middle Index of the Task Array
taskDesc - Task Descriptor
chunkSize - Chunk Size

```
1: function RECURSIVESPLIT(low, high, taskDesc)
2:   size = high - low
3:   if (size < chunkSize) then
4:     executeTask(low, high, taskDesc)
5:     return 0
6:   else
7:     Task tPushed = new task(mid, high, taskDesc)
8:     Push (tPushed)
9:     RECURSIVESPLIT(low, mid, taskDesc)
10:    Task tPopped = Pop()
11:    if (tPopped = NULL) then ▷ If Pushed task is stolen
12:      return 0
13:    else ▷ If Pushed task is not stolen
14:      RECURSIVESPLIT(mid, high, taskDesc)
15:    end if
16:  end if
17: end function
```

that chunk of work, until there are no chunks left to schedule.

Only When Idle

A PE incur unnecessary overhead due to task creation and queue contention when there are no idle PEs who can steal and execute some of their tasks. We use an atomic counter to keep track of the number of idle PEs within the node. Any PE trying to generate fine-grained tasks can use this information to decide number of tasks to generate thereby adaptively controlling the number of tasks generated depending on the state of the system.

History

We utilize the *principle of persistence* to further reduce the overhead of task creation. Each PE keeps a history of the fraction of tasks that was locally executed. This

information is used to decide the number tasks to be generated and pushed to the task queue to enable work sharing.

5.5.3 Task Queue

To support tasks, we created a task queue on each PE, which is distinct from the normal message queue. The messages in the message queue are meant for that specific PE, whereas the tasks in the task queue can be distributed across different PEs on a node. The scheduler on the PE polls both the local task queue and the message queue for messages. We chose not to have a centralized task queue at the node level because then we lose locality information and there could be potential contention for the centralized queue. We have a separate task queue on each PE, which is a single producer multiple consumer queue for the fine-grained tasks. Whenever a PE becomes idle, it randomly chooses a PE and picks tasks from that PE's task queue. This is similar to Cilk's workstealing [3], except that our scheduler also polls other queues, including a PE-specific message queue for messages to chores assigned to that PE by the periodic load balancer.

The task queue is implemented using the Chase-Lev [73] non-blocking algorithm. The task queue is a double-ended queue. A `push(t)` call enqueues a task at the tail of the queue. A `pop()` call dequeues a task from the tail of the queue. A `steal()` call dequeues from the head of the queue. The queue is a cyclic array of task pointers with non-wrapping head and tail indices. A worker does a `push(t)` by adding the task at the tail of the queue and increments `T`, the tail pointer. A worker does a `pop()` by decrementing `T`. If it detects that there could be a conflict, then it uses compare and swap (CAS) to handle the conflict. A thief reads `H` and `T` and uses CAS to atomically increment `H` and obtains the task.

The task descriptor contains details about the task such as the object pointer, function pointer, parameters and an atomic variable. The message enqueued into the

task queue contains range parameters and a pointer to the common task descriptor. To avoid the overhead of creation of messages and task descriptors, we keep a pool of task messages and descriptors, which are reused.

5.6 Application Study

We study the performance benefits of our new integrated runtime system that combines the Charm++ distributed memory model with the task model on two production scientific simulation codes, ChaNGa and NAMD, as well as its use in an MPI+OpenMP proxy application, Kripke. We compare the performance of these codes with and without the task model integrated. We show the performance of ChaNGa on Blue Waters and NAMD on Blue Waters and Blue Gene Q. For all the applications, we picked the scheduling strategy that performed the best. For the Charm++ *ParallelFor*, we use the *recursive task generation* scheme and for OpenMP we use the *history* scheme. Both of them were used in conjunction with the *when idle* strategy.

Blue Waters is a Cray XE/XK hybrid machine hosted by NCSA consisting of AMD 6276 Interlagos processors located at the National Center for Supercomputing Applications (NCSA). It has 22,640 Cray XE nodes and 4,228 Cray XK nodes that include NVIDIA GPUs. On the XE nodes there are two AMD Interlagos 6276 processors processors and each processor has 8 Bulldozer cores. Each Bulldozer Core compute unit has 16 integer cores and 8 floating point cores. Our benchmarks are run entirely on the CPU-only XE nodes. Vesta, which is a Blue Gene Q installation located at Argonne National Laboratory (ANL), has 2048 nodes of 1600 MHz PowerPC A2 cores. Each node has 16 PowerPC A2 cores available to applications with 4 hardware threads per core.

5.6.1 ChaNGa

ChaNGa is an N-body cosmology simulation application implemented in Charm++. ChaNGa has been used in cosmology research to model the impact of a dwarf galaxy on the Milky Way [74], study the role of Warm Dark Matter in dwarf galaxy formation [75] and model the intracluster gas properties in merging galaxy clusters. ChaNGa uses adaptive time scales for force evaluation at multiple scales. A wide variation in mass densities results in particles having dynamical times that vary by a large factor. The irregular distribution of particles in the simulation space as well as having multiple scales creates severe load imbalance. Performing frequent load balancing by object reassignment has unacceptable overhead due to strategy time and data movement. In addition, for clustered datasets, it is often the case at the trailing end of the gravity calculation that some of the PEs are idle while others are busy. For our experiments we use a challenging dataset, *cosmo25*, which is a highly clustered 2 billion particle dark matter simulation. In this multi-stepping run of the *cosmo25* dataset, 16 substeps constitute a big step.

A time-line view created with the Projections tool [76] is shown in Figure 5.3a. We pick only a subset of cores within an SMP node for one of the substeps to showcase the load imbalance problem. The colored bars indicate that the PE is busy with computation work and the white shows idle time. We can see clearly that there is severe load imbalance. We use the task parallelization in conjunction with the node aware load balancer to handle this load imbalance. With the intra-node task parallelization, we are able to handle the load imbalance and improve the performance of this substep significantly. In Figure 5.3b we can see the impact of this in the reduction of load imbalance, wait time and iteration time before the barrier.

At the point where the application creates fine-grained tasks, it queries the adap-

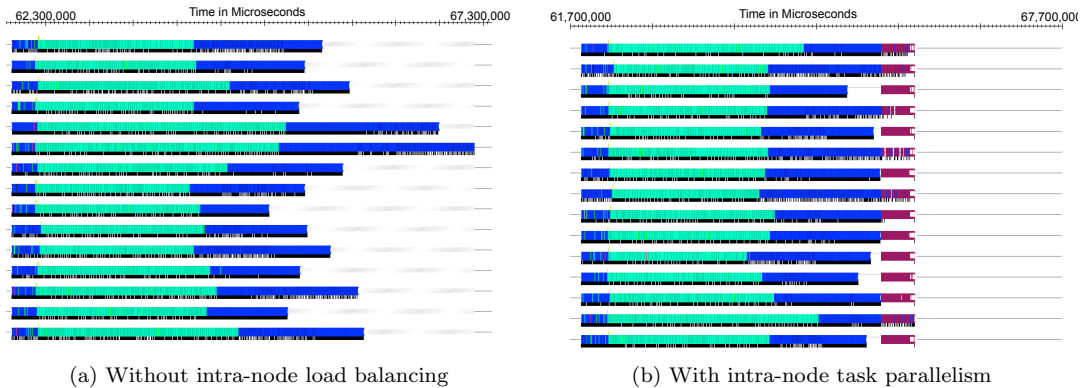


Figure 5.3: Time line profile of ChaNGa for all the PEs (rows) on a SMP process for the 128K cores run. White shows idle time and colored bars indicate busy time. Fine-grained task parallelism achieves better distribution of work among PEs. The total time per step reduces from 5.0 seconds to 4.2 seconds.

tive runtime system to find out whether it is beneficial to create tasks. The runtime system monitors the state of the PEs on a node and when there are sufficient idle PEs, it considers it as beneficial to create tasks. This prevents incurring unnecessary overhead of task creation when there is no potential benefit to it because other PEs are already busy. The chare object uses OpenMP or the Charm++ `par` for construct to create tasks out of the unfinished buckets which gets distributed among other idle cores.

Figure 5.4 compares the strong scaling performance of the original version of ChaNGa with the improved one using intra-node fine-grain tasks. At the scale of 131,072 cores, both *ParallelFor* and OpenMP give more than 2X speedup.

5.6.2 NAMD

NAMD [77] is a molecular dynamics application designed for the simulation of large biomolecular systems. Its primary focus is on all-atoms simulation methods using empirical force fields with a femtosecond time step resolution. Typical NAMD simulations include all-atom models of proteins, lipids, and/or nucleic acids as well as explicit solvent (water and ions) and range in size from 10,000 to 10,000,000

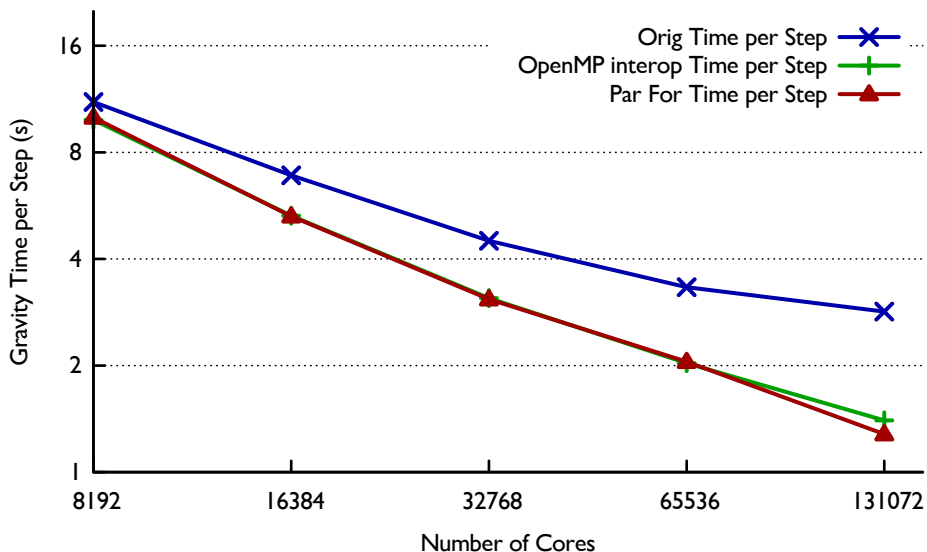


Figure 5.4: ChaNGa strong-scaling performance on Blue Waters Cray XE6 system, using Charm++ alone, with integrated OpenMP, and with the *ParallelFor* extension. Both intra-node balancing mechanisms give more than 2X speedup at 128K cores.

atoms. NAMD played an instrumental role in a recent study resolving the atomic level structure of the HIV Capsid. A recipient of the Gordon Bell Award, NAMD is based on Charm++ parallel objects and scales to hundreds of cores for typical simulations and beyond 500,000 cores for the largest simulations.

For the experiments shown here, we use the Colvar module. Colvar stands for *Collective Variables*. Colvars are used to reduce the great number of degrees of freedom present in molecular dynamics simulations into a few parameters which can either be analyzed individually, or manipulated in order to alter the dynamics in a controlled manner. In NAMD, we use the colvar module to perform energy minimization runs and determine the time taken for each step. We use a hierarchical load balancing strategy to infrequently to address the load imbalance problem. For the load imbalance arising within the node, we use our intra-node task parallelization to distribute the computation on idle PEs within a node.

Figure 5.5 compares the performance of the original version of NAMD running

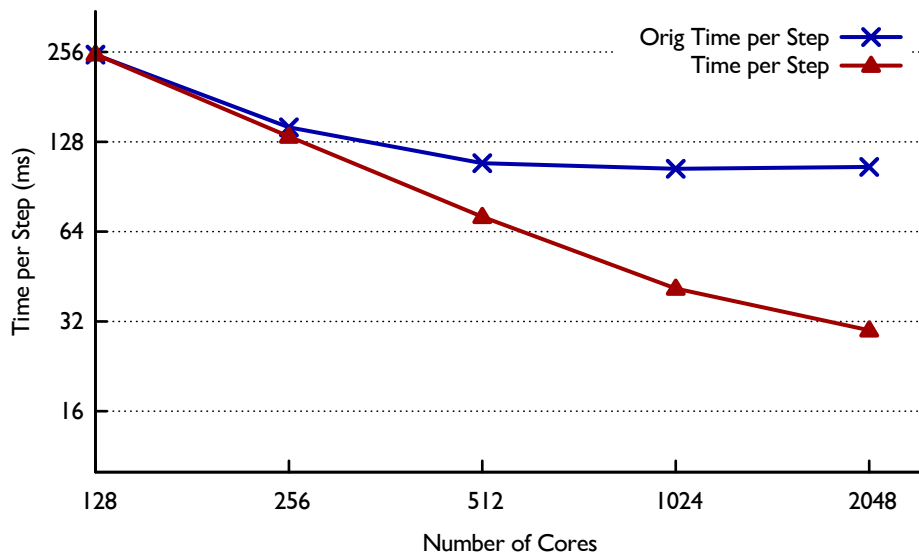


Figure 5.5: Strong scaling results comparing the performance of original Charm++ with the new integrated task model for NAMD’s colvar benchmark on IBM Bluegene/Q.

colvar module with the improved version using intra-node fine-grain tasks. At the scale of 2048 cores it gives a speedup of approximately 3.5X.

5.6.3 Kripke

Kripke [78] is an LLNL proxy application for parallel deterministic transport codes. It is written using MPI and, optionally, OpenMP for parallelism. Kripke implements the key computation and communication aspects of a production transport simulation application. Such codes are used to deterministically simulate the flux of neutral particles within a volume of interest. Kripke implements parallel sweeps through a 3D domain. The domain is decomposed in spatial zones, and subdomains are distributed to MPI ranks.

Parallel sweeps are vital communication kernels for the performance of deterministic transport codes. A sweep is a sequential traversal through a domain. Because of the sequential dependencies through the domain, and because the domain is de-

composed spatially, scaling sweeps efficiently is challenging. Thus, Kripke pipelines successive sweeps over the different energy groups and directions in the problem to attain higher efficiency. In addition to the sweep, a reduction is performed every iteration to test the global particle count for convergence. While Kripke does not actually check for convergence (it instead runs for a fixed number of iterations), the reduction keeps the communication pattern faithful to production transport codes. The reduction acts as a barrier, preventing ranks that own subdomains in the center of the domain from advancing until all ranks have finished all sweeps.

Adaptive MPI (AMPI [79]) is an implementation of the MPI standard written on top of Charm++. It provides the high-level features of Charm++, such as over-decomposition, dynamic load balancing, and automatic fault tolerance, to pre-existing MPI applications. It does so by implementing MPI ranks as lightweight, migratable user-level threads, which are encapsulated in chares. The runtime system schedules and load balances AMPI ranks the same way it does chares in Charm++ programs. MPI applications with no mutable global/static variables, such as Kripke, need only be compiled using AMPI's compiler wrappers instead of MPI's to run on AMPI.

Our implementation of the GNU OpenMP runtime can be used with AMPI + OpenMP programs the same way it is with Charm++ applications. This allows users to run an AMPI code on a node with N PEs using two modes: (a) 1 or a few AMPI ranks per node with OpenMP threads within each rank or (b) N or more AMPI ranks per node with each rank using up to N OpenMP threads, without actually oversubscribing the physical resources on the system. Our results show the benefits of this approach for applications such as Kripke which have transient load imbalances within iterations but little to no load imbalance that persists across iterations.

All of the tests below were performed on Blue Waters, using 32 cores per node.

We use the default input parameters for Kripke version 1.1, meaning we run with 4096 zones per core in 1 set, 32 groups in 2 sets, and 96 directions in 8 sets. The data is laid out in the default DGZ nesting. Note that no changes are necessary to the source code of Kripke to run it on AMPI and our implementation of OpenMP, and that all of the computational kernels use OpenMP `parallel` for loops. We show weak scaling in the number of zones, with the number of groups and directions held constant.

Figure 5.6 shows the time per iteration of Kripke using MPI, MPI+OpenMP, AMPI, and AMPI+OpenMP with two different configurations. The parenthetical in MPI+OpenMP (1) and others identifies how many ranks were launched per node. Thus, MPI+OpenMP (1) signifies the use of 1 rank per node with 32 OpenMP threads per rank, and MPI+OpenMP (16) means 16 ranks were launched per node with 2 OpenMP threads per rank. AMPI+OpenMP are similarly presented, but since our OpenMP implementation allows scheduling OpenMP threads along with AMPI ranks without resource contention, we always specify 32 OpenMP threads per rank. Consequently, AMPI+OpenMP (32) means 32 ranks were launched per node with 32 OpenMP threads per rank. In addition to MPI-only, AMPI-only, and both with one process and 32-way threading, we show the best performing combination of rank and thread counts for each.

Kripke's parallel sweeps benefit from the finer-grained pipeline parallelism that decomposing into more MPI ranks offers. On the other hand, the computational kernels benefit from OpenMP threading. Since sweep dependencies translate to idle times within a node while each wavefront passes through the domain, within-node parallelism can be also be used to balance the load across the idle threads at a given time. The combination of 32 ranks and up to 32-way threading per rank performs the best. It gives the runtime the most freedom to schedule work across all available cores on a node while still decomposing the sweep pipeline into small pipeline stages

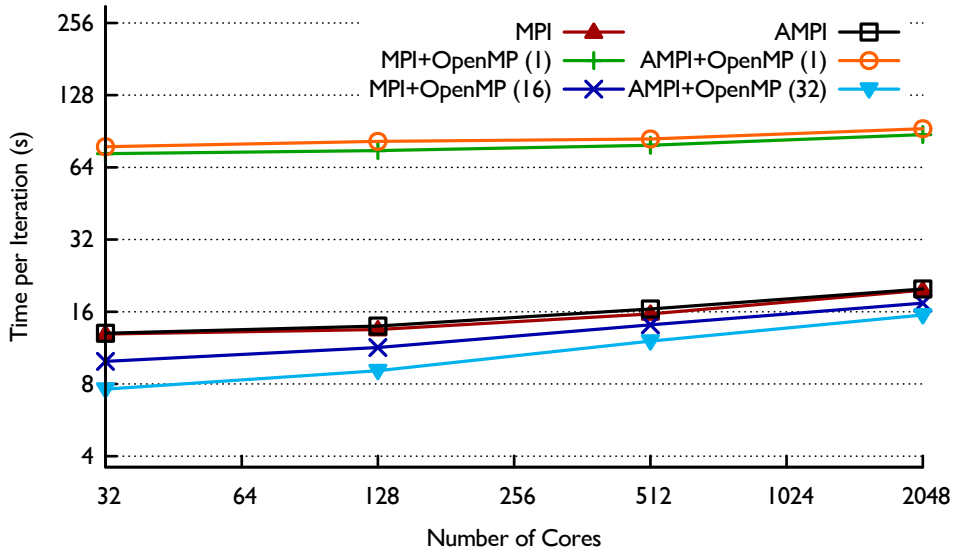


Figure 5.6: Weak scaling Kripke with 4096 spatial zones per core on Blue Waters, the time per iteration is shown for MPI and AMPI with and without OpenMP. Numbers in parentheses indicate how many ranks were used per node.

and ensures that each thread has its own work to schedule in addition to stealing others' work when idle.

5.7 Conclusion

The recent trend of rapid increase in the number of cores per chip has resulted in vast amounts of on-node parallelism. Not only the number of cores per node is increasing substantially but also the cores are becoming heterogeneous. The high variability in the performance of the hardware components introduces imbalance due to heterogeneity. Applications are also becoming more complex resulting in dynamic load imbalance. Load imbalance can result in loss of performance and decrease in system utilization. We address the challenge of balancing load across cores while maintaining locality and low overhead. In this paper, we proposed a new integrated runtime system that combines the Charm++ distributed programming model with concurrent tasks to handle load imbalance. It utilizes a relatively infrequent periodic

assignment of work to cores based on load measurement, in combination with user created tasks to handle both the persistent and transient load imbalance. OpenMP integration with Charm++ was built on top of this framework. It enables objects to create potential tasks via OpenMP's parallel loop construct. This contribution is not specific to Charm++; it is also available to MPI applications through integration with Adaptive MPI. The benefits of using this integrated runtime system is shown on three different applications. We show improvements of 2X on ChaNGa on 128K cores and more than 3X on NAMD at 2,048 cores. In these applications, benefit naturally increase with high core counts, when one is nearer to the limit of strong scaling. We also show the benefit on an MPI application, Kripke, in a weak-scaling experiments on up to 2,048 cores using Adaptive MPI.

The task generation scheme we used currently admits a relatively flat set of tasks generated by parallel loops. A possible future extension is to admit tasks with dependences, similar systems like OmpSs [59] or PaRSEC [80]. These will also create opportunities for runtime scheduling based on the knowledge of dependencies and cache or scratchpad availability of data.

6 Meta Balancer for LB Period

Modern parallel applications running on large clusters often involve simulations of dynamic and complex systems [81, 82]. 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 [83], 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 [81], FEM [84] and climate simulation, the problem is broken into smaller data and work units that execute on 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 or Charm++ [85]), 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 techniques 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, a 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 [39], ChaNGa [8] and Fractography3D [9].

Performing load balancing entails overheads that include 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 thesis, 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, 86], 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 chapter 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 Section 6.1, we provide a background on the load balancing framework in Charm++, followed by a description of Meta-Balancer in Section 6.2. Thereafter, results on using Meta-Balancer with two real world applications are presented in Section 6.3. Finally, previous work is presented in Section 6.4 followed by conclusion and future work in Section 6.5.

6.1 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. A 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].

6.1.1 Charm++ and its Load Balancing Framework

Charm++ is a parallel programming model that 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. First, it instruments the application code at a very fine-grained level and provides vital statistics for load balancing. Second, 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 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 a load balancing strategy is the responsibility of the programmer. The load balancing strategies are plugins in Charm++.

Algorithm 4 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 4, we present the iterative component of a typical Charm++ program. In Charm++, execution proceeds when functions are invoked on chares by the RTS on receiving messages for them. An application run begins with the RTS invoking appropriate functions (*ResumeWork* in our example) targetted at the chare. Most function calls (such as *call ResumeWork*) results in a message send to the 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 processors and chares

is sent either to a central processor (if using a centralized strategy) or to a set of processors (if using a hybrid strategy) [13]. 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 when they are invoked by the RTS.

6.2 Meta-Balancer

Meta-Balancer is designed as a framework that, 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 the Charm++ load balancing framework (Section 6.1.1). Meta-Balancer relies on a heuristic known as *the principle of persistence* described in Section 6. 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, a decision making module for the ideal LB period and consensus of LB period.

6.2.1 Meta-Balancer Statistics Collection

Meta-Balancer collects load information about the running application to determine if load balancing is needed. Using the same *AtSync* interface as described in Section 6.1.1, 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 6.1. However, the application needs to call *AtSync* not when the application thinks the load balancing is needed, but everytime it is possible to balance, for example at the end of every iteration.

Since Meta-Balancer requires frequent aggregation of the 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 good 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.

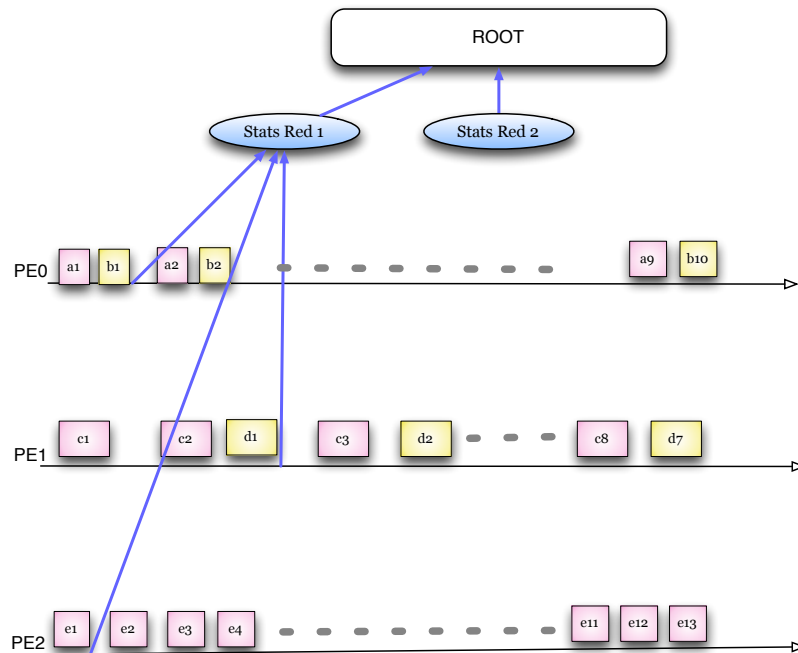


Figure 6.1: Periodic Statistics Collection

6.2.2 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

$$\zeta = \frac{L_{max}}{L_{avg}} - 1 \quad (6.1)$$

where L_{max} is the load on the most loaded processor and L_{avg} is the average load of all 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 require load balancing as it may not be profitable due to the overhead in the 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. The goal is to minimize the total execution time. This can be a challenging problem since we need to model the effectiveness of the 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:

$$L_{avg} = at + l_a \quad (6.2)$$

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:

$$L_{max} = mt + l_m \quad (6.3)$$

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:

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

Note that $\frac{\gamma}{\tau}$ represents the number of times 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.

$$\begin{aligned} \frac{d}{d\tau} (\Gamma) &= \gamma \times \left(\frac{m}{2} - \frac{\Delta}{\tau^2} \right) = 0 \\ \tau &= \sqrt{\frac{2\Delta}{m}} \end{aligned} \quad (6.4)$$

Eq 6.4 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 6.2. The ideal LB period is calculated and continuously refined by Meta-Balancer, using Eq 6.4, 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.

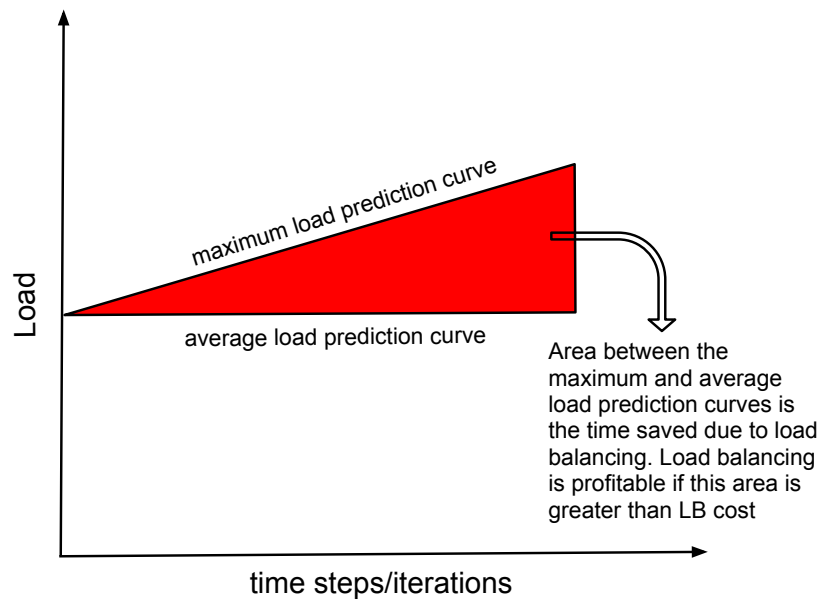


Figure 6.2: Ideal Load Balancing Period

6.2.3 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 step, the execution resumes on

each chore. 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 chore a , which receives the notification of load balancing at iteration i before it reaches iteration i . When this chore arrives at iteration i , it waits for the load balancing to be done. Consider another chore 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 chores waiting for the load balancing to be done. This scenario is possible for applications that have no explicit global synchronization at each iteration, because chores 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 chores, load balancing will never happen in this scenario as chore b missed the iteration i , causing the application to hang.

To avoid such a scenario, all chores need to reach consensus on the iteration number that chores can reach to enter the load balancing stage. Since the chores can be in different iterations, we use the scheme shown in Figure 6.3 to obtain the consensus.

First, 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 chore that resides on it and prevents any chore from going beyond that. It contributes its local iteration number via a reduction to find the maximum iteration number any chore 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

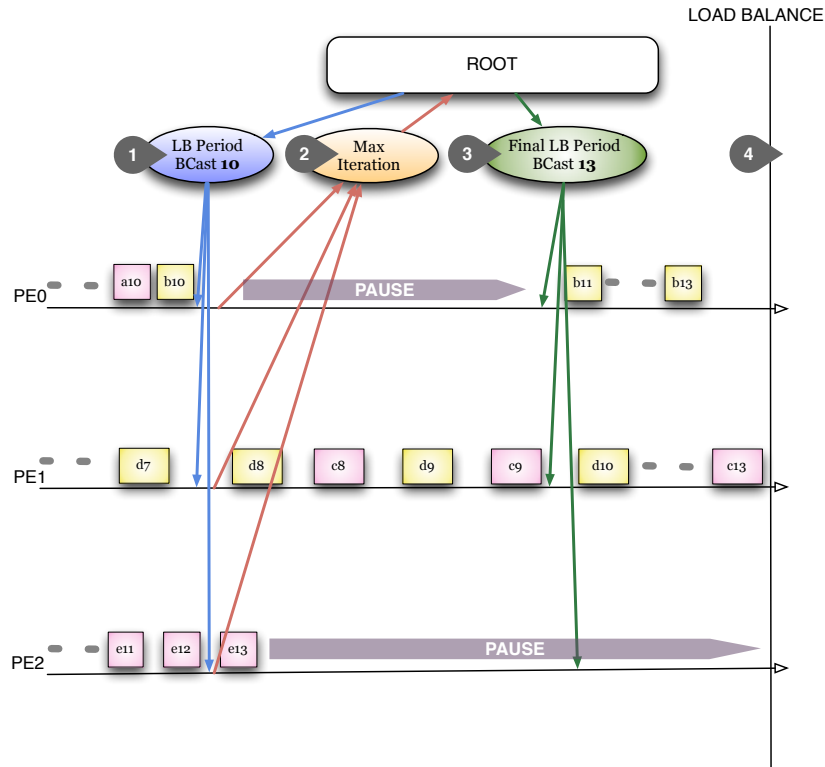


Figure 6.3: Three Step Consensus Mechanism in Meta-Balancer

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 the RTS on each processor blocks a chare which has reached its local tentative LB period.

6.2.4 Implementation

In this section we describe the implementation of Meta-Balancer and its interaction with the application and the Charm++ RTS. As mentioned earlier, a typical application can be depicted using Algorithm 4. Periodically, the application invokes *AtSync*, which passes the control to the Charm++ RTS for potentially invoking the load balancer. The functionality of the Charm++ RTS for a chare is shown in

Algorithm 5. 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 the next iteration is initiated for the chare.

The Meta-Balancer code on each processor is presented in Algorithm 6. 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 7, it finds the ideal LB period and follows the consensus mechanism described in Section 6.2.3 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 the next iteration on the chares to progress locally to the iteration chosen for the next load balancing step.

Algorithm 5 Charm RTS on each Chare

```

1: when AtSync invoked
2: update chare load in Meta-Balancer
3: if (reached LB period) then
4:   perform load balancing
5: else if (reached tentative LB period) then
6:   wait for final LB period
7: else
8:   call ResumeWork
9: end if

```

```

1: when received final LB period
2: if (curr_iter == finalLBperiod) then
3:   perform load balancing
4: else
5:   call ResumeWork
6: end if

```

Algorithm 6 Meta-Balancer on every Processor

- 1: when received **chare load**
 - 2: **if** (all chares have registered their load for an iteration) **then**
 - 3: contribute to reduction for statistics collection
 - 4: **end if**
-

- 1: when received **tentative LB period**
 - 2: find maximum iteration number of chares
 - 3: contribute to reduction for maximum iteration number
-

Algorithm 7 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 $\max\{\text{tentative LB period, maximum iteration}\}$
 - 3: inform final LB period to all processors
-

6.3 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 equipped with Cray's new high performance Gemini network.

6.3.1 LeanMD

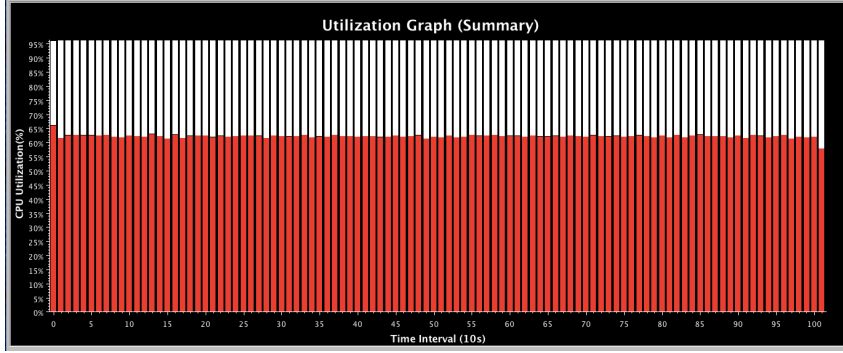
LeanMD is a molecular dynamics simulation program written in Charm++. It simulates the behavior of atoms based on the Lennard-Jones potential, which is an

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 [39], an application widely used by biophysicists, which won the Gordon Bell award.

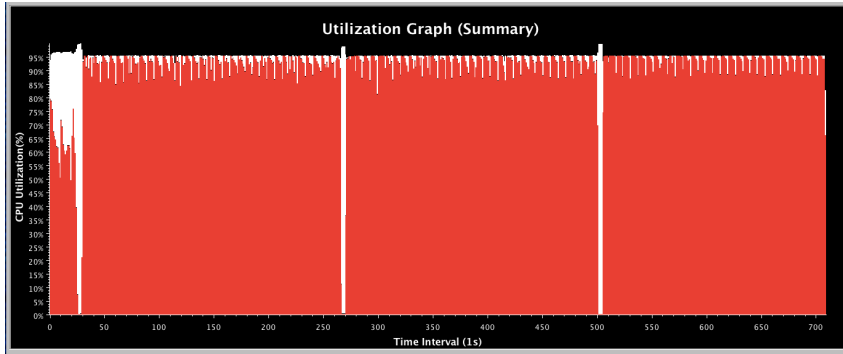
The force calculation in Lennard-Jones dynamics is done within a cutoff-radius, r_c for every atom. The three-dimensional (3D) simulation space consisting of atoms is divided into cells of dimensions that are equal to the sum of the cutoff distance, r_c and a margin. In each iteration, force calculations 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 *compute* objects, the cells perform the force integration and update various properties of their atoms – acceleration, velocity and positions. Load imbalance in LeandMD is due to the variation in the number of atoms that reside in a cell. The load on *compute* objects 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.

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, \dots , 4096) without any load balancing. The processor utilization graph for running LeandMD on 256 cores without load balancing is shown in Figure 6.4a. 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



(a) No Load Balancing



(b) Meta-Balancer

Figure 6.4: Processor Utilization of LeanMD on 256 cores

periods (10, 20, \dots , 700), leading to the result shown in Figure 6.5. 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 that gives the best performance.

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 6.1: LeanMD Application Time on Jaguar

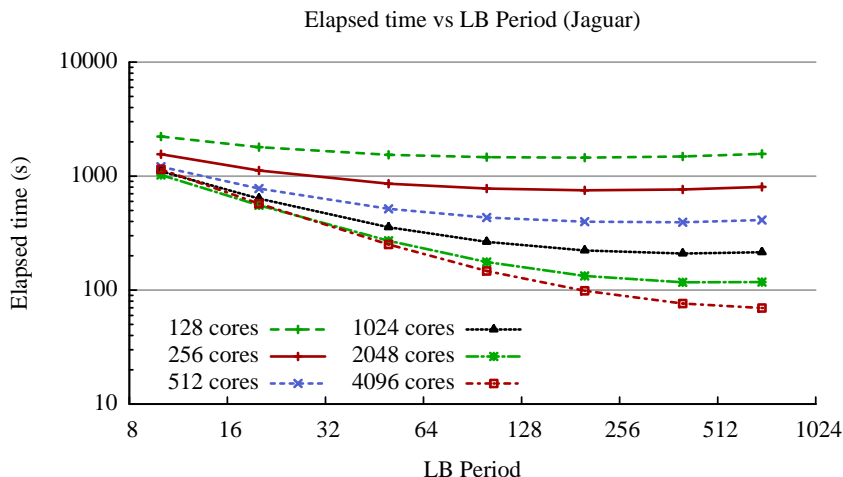


Figure 6.5: Variation in LB Period for LeanMD on Jaguar

In Figure 6.4b, we present processor utilization for the case where LeanMD is run with Meta-Balancer on 256 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 6.1. 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 the Ranger cluster to simulate a 300,000 atom system for 2000 time steps. Figure 6.6 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 pe-

riod at which the best performance is obtained also changes with the problem being simulated and the system being used to execute the application. In Table 6.2, 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.

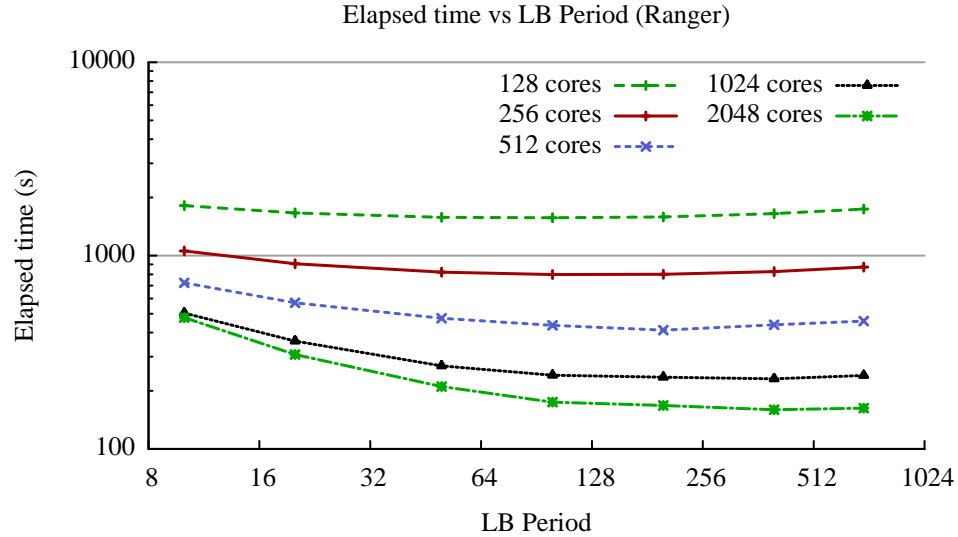


Figure 6.6: Variation in LB Period for LeanMD on Ranger

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 6.2: LeanMD Application Time on Ranger

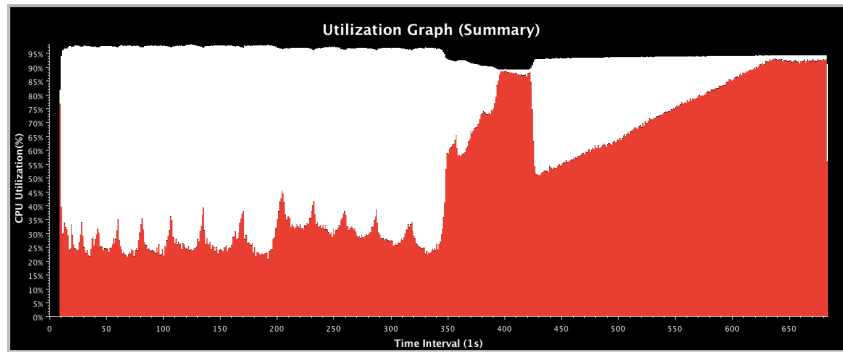
6.3.2 Fractography3D

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 a Charm++ based FEM framework [84].

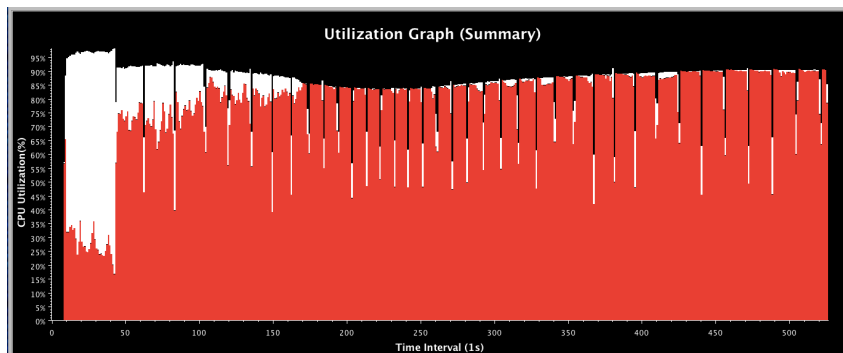
In Fractography3D, the framework discretizes a 3-D volume into tetrahedras. Typically, the number of elements is very large, and they are grouped into a number of chunks distributed across processors. During the simulation, each tetrahedral element is considered to have one of two material properties: 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 a 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 applying an external force to 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 of 64, 128, \dots , 1024. Figure 6.7a shows the processor utilization graph generated when Fractography3D 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

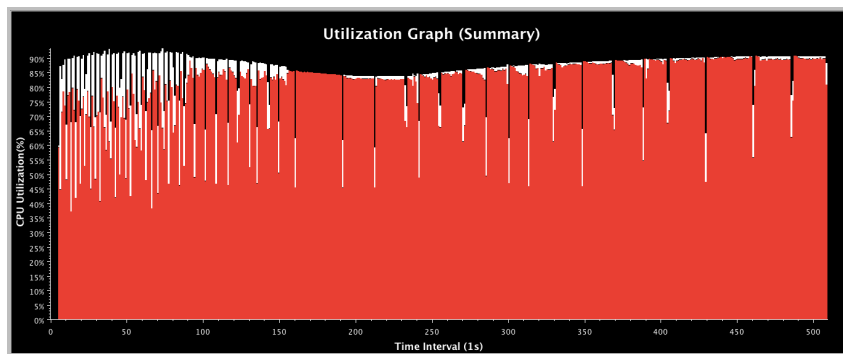
found in processor utilization on other core counts as well.



(a) No Load Balancing



(b) Periodic Load Balancing (300 iterations)



(c) Meta-Balancer

Figure 6.7: 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, \dots , 7000) to find the period that gives the best performance. Figure 6.8 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 var-

ied. If the load balancing is done very frequently, the overheads of load balancing overshoot the gains of load balancing, which results in bad performance. On the other hand, if load balancing is done very infrequently, the load imbalance in the system reduces the gains achieved by load balancing. However, for intermediate periods, such as 300 iterations, best performance is obtained. In Figure 6.7b, 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 6.7b is the 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 6.7c shows the processor utilization graph generated when Fractography3D is run on 64 cores with Meta-Balancer. It can be 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. This shows that a single static frequency is insufficient.

In Table 6.3, 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

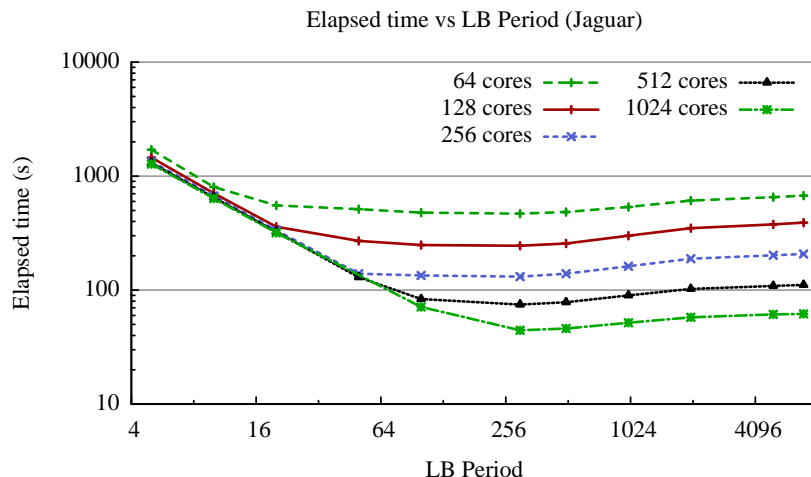


Figure 6.8: Variation in LB Period for Fractography3D on Jaguar

the 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 matches 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 completed as fast as possible to extract the intended scientific results.

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 6.3: Fractography3D Application Time

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

tography3D 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, only a 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.

6.4 Previous Work

Dynamic load balancing strategies have been studied extensively in the past [31, 32]. 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 [83], Chombo [87], and Charm++ [10]. Similar schemes have also been proposed and used in MPI applications [88, 89]. This thesis proposes concepts that build upon these existing frameworks in order to make decisions related to load balancing to get good performance.

As described in [32], 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 a load imbalance factor $\phi(t)$ is proposed in [32], 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. [90]. 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 to decide the acceptable granularity of each of these factors for several inputs. Our work requires no input from the user and hence results in complete automation.

Techniques for automation of load balancing related decisions are also presented in a recent work by Pearce et al. [86]. The primary focus of this work is on the selection of a load balancing strategy based on the 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.

6.5 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 thesis, 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 that 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 that 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.

7 Meta Balancer for LB Strategy

Many applications require dynamic load balancing to achieve good performance and increase their scalability. Different applications need to use different load balancing strategies. The Charm++ load balancing framework contains more than a dozen load balancers. Each of these load balancers is suitable for certain applications. Using a load balancer that is not well suited for an application will result in loss of performance. For example, in the case of a communication intensive application, using a strategy that just balances the load without considering the communication pattern will result in bad performance. Typically, the application behavior depends on the machine characteristics and the problem being simulated. As a result, choosing the load balancing strategy that gives the best performance becomes difficult. Most commonly, the application programmer decides which load balancer to use and when to do load balancing based on some educated guess. That load balancer is then used for the entire run of the application, which may result in suboptimal solutions.

In this thesis, we propose the Meta-Balancer framework that monitors the system and application characteristics to decide the load balancing strategy. This runtime system component continuously monitors the system and, based on the observed characteristics, automatically chooses different load balancing strategies at runtime. This enables multiple load balancers to be used at different times in the application in an adaptive manner. The main contributions of this chapter are

- Identification of different features or statistics that are used to automate the decision

- Automatic collection of these statistics in an asynchronous manner
- A decision tree to identify the load balancing strategy
- Use of machine learning to identify the load balancing strategy

7.1 Meta-Balancer

Our Meta-Balancer framework monitors the application and system characteristics by automatically collecting statistics about it. It then chooses the load balancing strategy based on the observed characteristics. It is implemented as a part of the Charm++ runtime system. Meta-Balancer consists of three major components, namely, asynchronous statistics collection, construction of the features and decision making module for the ideal load balancing strategy.

7.1.1 Meta-Balancer Statistics Collection

Meta-Balancer collects load and communication information about the application frequently. These statistics are collected at the iteration boundary. The load balancing framework in Charm++ automatically collects the load and communication at each PE and stores it in a distributed manner. Meta-Balancer uses some of the information stored in the load balancing framework and gathers these statistics via an asynchronous reductions, as shown in Figure 6.1. Table 7.1 shows the different statistics that are collected.

7.1.2 Meta-Balancer Feature Construction

The raw statistics that are collected need to be converted into meaningful features based on which the load balancing strategy can be chosen. If `Max_PELoad` will vary from one application to another and therefore new features that give a relative

Statistics	Description
Num_PEs	Number of PEs in the system
Num_Ovld_PEs	Number of overloaded PEs in the system
Avg_PE_Load	Average load of the PEs in the system
Min_PE_Load	Minimum load of the PEs in the system
Max_PE_Load	Maximum load of the PEs in the system
PE_Load_STD	Standard deviation of the PE loads
PE_Load_Skewness	Skewness of the PE loads
PE_Load_Kurtosis	Kurtosis of the PE loads
Avg_PE_BGLoad	Average background load of PEs
Avg_Utilization	Average utilization of the PEs
Min_Utilization	Minimum utilization of the PEs
Max_Utilization	Maximum utilization of the PEs
Num_Objs_Per_PE	Number of objects per PE
Avg_Obj_Load	Average object load
Min_Obj_Load	Minimum object load
Max_Obj_Load	Maximum object load
Total_Msgs	Total number of messages transferred
Total_Bytes	Total amount of bytes transferred
Total_Ext_Msgs	Total number of messages transferred outside a PE
Total_Ext_Bytes	Total amount of bytes transferred outside a PE
Avg_Comm_Nghbor	Average number of communicating neighbors per object
Avg_Hops	Average number of hops per message
Avg_Hopbytes	Average number of hop bytes per message
Alpha	Cost of latency per message
Beta	Per link 1/bandwidth
Rate_Max_Load	Rate of change of maximum load of PEs across iterations
Rate_Avg_Load	Rate of change of average load of PEs across iterations
Migration_Size	Average amount of data per object to be transferred during migration

Table 7.1: Meta-Balancer strategy selection statistics

measure instead of absolute values need to be constructed out of the statistics.

Table 7.2 shows the different features that are derived from the statistics in Table 7.1.

7.1.3 Load Balancing Strategy Selection

There are a number of load balancing strategies in Charm++. These strategies handle load imbalance for different application characteristics. When no load balancing is required, we refer to it as NoLB. Some of these strategies are:

- **GreedyLB:** A centralized strategy that uses greedy heuristic to assign heaviest tasks onto least loaded processors iteratively. This strategy does not take

into consideration the current assignment of tasks to processors. It also doesn't consider the communication pattern. It has a high overhead because the strategy is done serially. GreedyLB is most suitable for applications which has high compute load imbalance but is not affected by communication.

- **RefineLB:** A centralized strategy that carries out load balancing by incrementally transferring the load away from the overloaded processors. It takes into account the current assignment of work units onto processors and therefore reduces the number of work units migrated. Since this strategy is done sequentially and tries to optimize for the best possible assignment for a given threshold, it has a high strategy cost.
- **MetisLB:** A centralized strategy that passes the load information and the communication graph to METIS, a graph partitioning library, and uses the recursive graph bipartitioning algorithm in it for load balancing. This strategy takes into account the communication pattern and tries to balance load while minimizing the edge cut. MetisLB is most suitable for communication intensive applications.
- **ScotchLB:** A centralized strategy that uses SCOTCH graph partitioning library to make load balancing decisions. This strategy takes both communication and computation load to perform the mapping. ScotchLB is suitable for applications that is affected by communication as well as the computation load imbalance.
- **HierarchicalLB:** A hierarchical strategy [13] in which processors are divided into independent groups and groups are organized in a hierarchical manner. At each level of the hierarchy, the root node performs the load balancing for the processors in its sub-tree. This strategy can use different load balancing

algorithms at different levels.

- **GrapevineLB/DistributedLB**: GrapevineLB, also referenced as DistributedLB in this chapter, is a distributed load balancing strategy that uses gossip based information propagation to obtain partial information about the global state of the system. Using this information it does randomized work transfer. Details are in Chapter 3.

7.1.4 Load Balancing Strategy Selection Using Decision Tree

Figure 7.1 shows the decision tree for choosing the different load balancing strategies. If the average system utilization is low, then it indicates that there is a load imbalance problem and a suitable load balancing strategy needs to be called. If there is a higher communication cost in comparison to the computation load, then a communication aware load balancers, such as MetisLB and ScotchLB, is more suitable. But if the computation load imbalance is more dominant, then one of the different computation balancers, such as GreedyLB, RefineLB, DistributedLB, HierarchicalLB, needs to be called. If the load varies very frequently, then load balancing strategy will need to be called frequently. But if the load does not vary frequently, then strategies that give good balance of load is chosen even though they tend to have higher load balancing cost. In this case centralized strategy that have a complete view of the system tend to perform better than the distributed or hierarchical strategies. But if the estimated overhead of the centralized strategy is more than the benefit, then strategies with low overheads, such as DistributedLB and HierarchicalLB, are used.

This runtime system based automatic selection of strategy can perform load balancing adaptively and choose different load balancing strategies at different stages

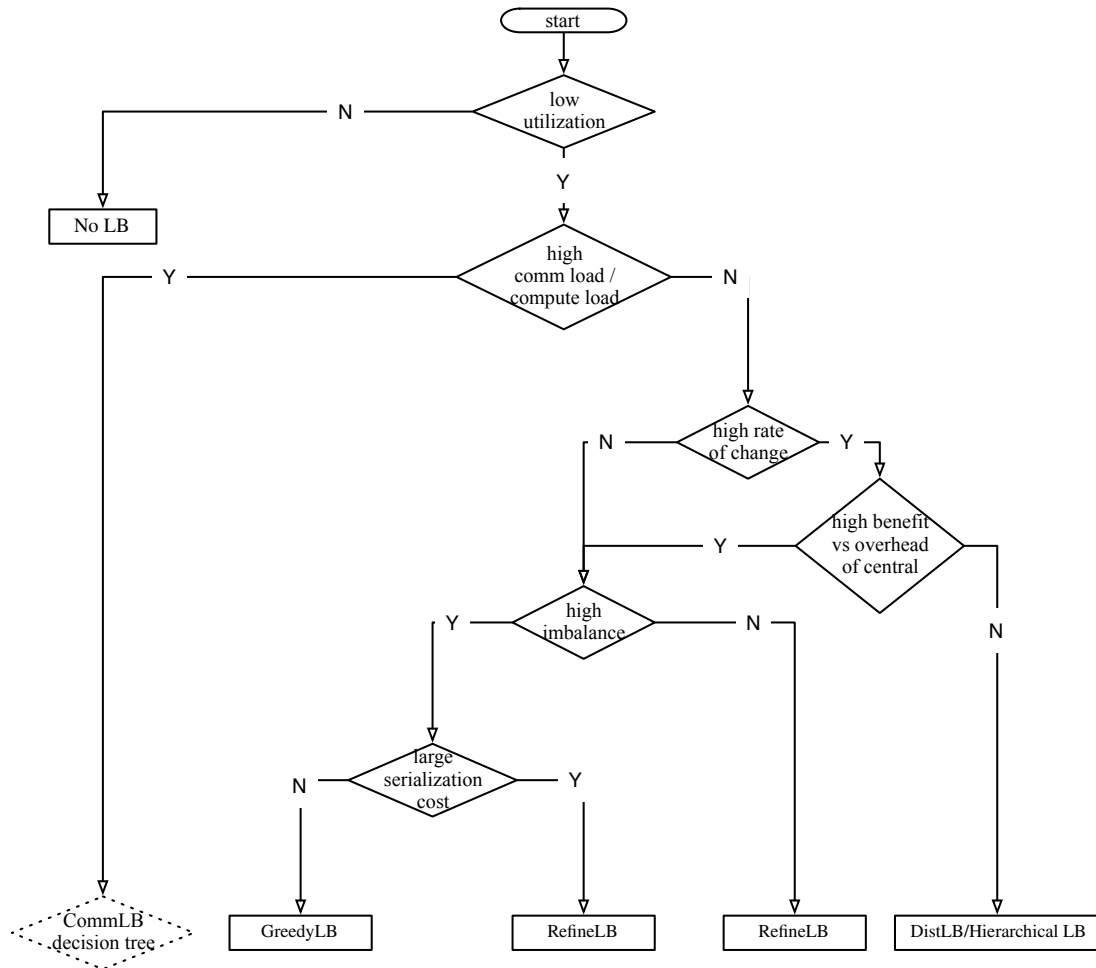


Figure 7.1: Decision tree for choosing the load balancing strategy.

of the application run depending on the runtime characteristics of the application.

7.1.5 Load Balancing Strategy Selection Using Machine Learning

Machine learning techniques have been used to do pattern recognition and have been employed in a variety of tasks such as classification, data mining, computer vision etc. These techniques operate by building a model from a sample input set and learn to predict the outcome. In this work, a supervised random forest technique is used to predict the load balancing strategy for a particular run. In supervised learning

algorithms, the tool is presented with example inputs and their desired outputs.

In our case, we used a benchmark, called *lbttest*, to generate the training set for the random forest. This benchmark is flexible and can generate various scenarios representative of a real application. It supports different communication pattern, such as 3D mesh, 3D mesh and random. It can be also vary the bytes communicated and the amount of computation load. We ran this benchmark with different parameters to generate different characteristics. For each parameter option, all the load balancers were run to identify the best performing load balancing strategy. For the training set, the derived features shown in Table 7.2 are used as input to the machine learning algorithm. The desired output for the training is the best performing load balancing strategy.

7.2 Experimental Results

The benchmark used for the strategy selection is *lbttest*. We collected more than 200 samples to generate the training set for the machine learning model. We divided the samples into training and test set and the learning algorithm was given the training set and evaluated on the test set. All the derived statistics in table 7.2 were given as features and expected outcome is the best performing load balancing strategy.

Figure 7.2 shows the performance of the machine learning model in predicting the expected outcome. We can see that it is able to achieve an accuracy of 88%. The confusion matrix shows how the expected outcome varied. In this figure it shows that predictions for RefineLB was misclassified and HybridLB and ScotchLB were chosen instead.

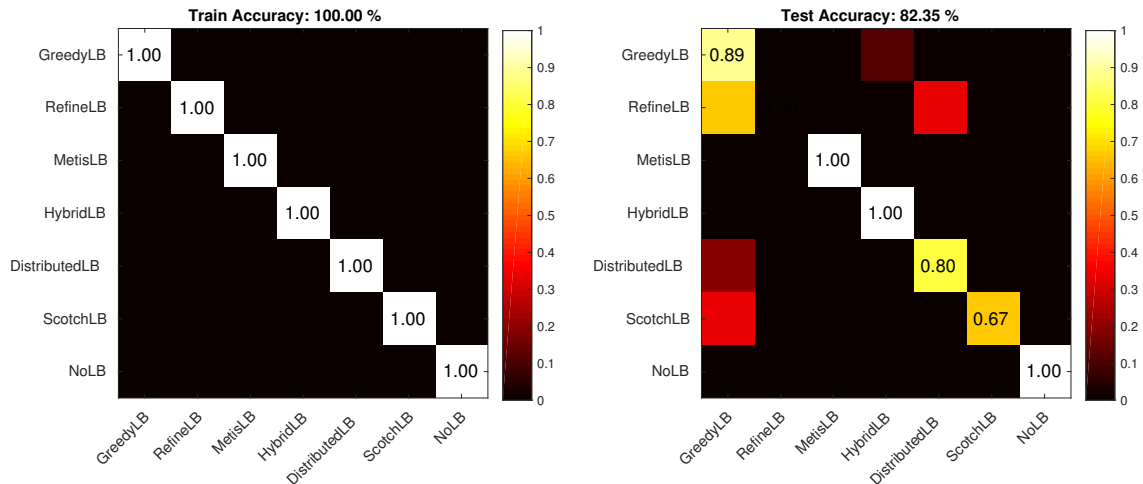


Figure 7.2: Accuracies on the training and test set along with confusion matrix.

7.3 Conclusion

Load imbalance is a very important factor affecting the performance of various applications. Application programmers have to deal with the complexity of choosing load balancing strategy and period under dynamic conditions. This may result in suboptimal solutions. In this chapter, we presented a runtime system module that automatically predicts a suitable load balancing strategy to employ based on the runtime characteristics of the application as well as the system. We selected and collected a number of statistics in an asynchronous manner. Using these statistics a set of features were constructed based on which a decision tree was proposed. A random forest machine learning technique was used to predict the load balancer based on those features. We were able to achieve a good accuracy of 80% with the model that was trained.

Features	Description	Derived
Num.PEs	Total number of PEs in the system	
Ovld.PEs.Percent	Percentage of overloaded PEs w.r.t total PEs	$\frac{Ovld_PEs}{Num_PEs}$
PE.Load.Imb	PE load imbalance	$\frac{Max_PE_Load}{Avg_PE_Load}$
PE.Load.RSD	Relative standard deviation of PE load	$\frac{PE_Load_STD}{Avg_PE_Load}$
PE.Load.Skewness	Skewness of the PE loads	
PE.Load.Kurtosis	Kurtosis of the PE loads	
PE.BGLoad.Percent	Background load in percentage w.r.t average PE load	$\frac{Avg_PE_BGLoad}{Avg_PE_Load}$
LB.Gain	Expected improvement in time with LB	$Max_PE_Load - Avg_PE_Load$
Avg.Utilization	Average utilization of the PEs	
Min.Utilization	Minimum utilization of the PEs	
Max.Utilization	Maximum utilization of the PEs	
Num.Obj.Per.PE	Number of objects per PE	
Avg.Obj.Load	Average object load	
Min.Obj.Load	Minimum object load	
Max.Obj.Load	Maximum object load	
Total.Msgs	Total number of messages transferred	
Total.Bytes	Total amount of bytes transferred	
Ext.Msgs.Percent	Percentage of messages transferred outside a PE	$\frac{Total_Ext_Msgs}{Total_Msgs}$
Ext.Bytes.Percent	Percentage of bytes transferred outside a PE	$\frac{Total_Ext_Bytes}{Total_Bytes}$
Internal.Msgs.Percent	Percentage of messages transferred within a PE	$\frac{Total_Int_Msgs}{Total_Msgs}$
Internal.Bytes.Percent	Percentage of bytes transferred within a PE	$\frac{Total_Int_Bytes}{Total_Bytes}$
Avg.Comm.Nghbor	Average number of communicating neighbors per object	
Avg.Hops	Average number of hops per message	
Avg.Hopbytes	Average number of hop bytes per message	
Alpha	Cost of latency per message	
Beta	Per link 1/bandwidth	
Comm.Cost.vs.Compute.Cost		$\frac{AlphasTotal_Msgs + BetasTotal_Bytes}{Avg_PE_Load * Num_PEs}$
Rel.Rate.Max.Load	Relative rate of change of maximum load of PEs	$\frac{Rate_Max_Load}{Max_Load}$
Rel.Rate.Avg.Load	Relative rate of change of maximum load of PEs	$\frac{Rate_Avg_Load}{Avg_Load}$
Migration.Overhead	Migration cost	$Alpha + Beta * Migration_Size$
Overhead.vs.Benefit.Central	Overhead vs benefit of using centralized strategy	

Table 7.2: Meta-Balancer strategy selection features

8 Meta Balancer for Thermal Variation

With the move towards exascale, power and energy consumption have become important issues in high performance computing. Recent studies show that HPC systems are drawing enormous amounts of electrical power. The increase in the number of cores and clock speed results in heat generation and increase in core temperatures. This makes the hardware more vulnerable to both transient and permanent faults. Therefore, cooling is necessary to prevent overheating of the chip. However, cooling also takes large amounts of energy. A study done in 2004 shows that 40% to 50% of the energy consumed by a data center is spent in running the computer room at a low temperature [91]. In order to reduce the cooling energy, the computer room air conditioning (CRAC) temperature can be set at a higher value. But this will result in high ambient temperature and possible overheating of the cores. To avoid overheating, modern day microprocessors are equipped with an on-chip temperature sensor and mechanisms to control the dynamic voltage and frequency using DVFS. Dynamic voltage and frequency scaling, DVFS, is commonly used to reduce power and the amount of heat generated by the chip by adjusting the frequency of the microprocessor. Running a processor at a lower frequency reduces the amount of heat generated and conserves power. Therefore, setting a high CRAC temperature and controlling the chip temperature using DVFS can be a possible solution to reduce the cooling energy, which accounts for a significant part of the power consumption.

However, using DVFS to control temperature has its drawbacks. Reducing the frequency may incur a timing penalty. Since the processors may overheat at different times, they may be running at different frequencies. The timing penalty is not

just due to the lower frequency but also due to the load imbalance created by the different processor speeds. In HPC applications, where there is an interdependence of tasks across processors, if one processor is slowed down, the entire application may consequently be slowed down. Even if there are no such dependency, there will be load imbalance between the processors. As a result, decreasing the frequency will result in degradation of performance and increase in the total execution time. In order to minimize the timing penalty, load balancing can be employed to improve the system utilization. This technique has been shown to be effective in reducing the cooling energy [92, 93].

In a recent work [92], a temperature-aware dynamic load balancing strategy was proposed which controls the chip temperature using DVFS and uses load balancing to reduce that timing penalty. This scheme performs periodic temperature checks, applies DVFS on cores that are hotter or colder than the threshold temperature and invokes the load balancer. This approach puts the burden on the application programmer to specify the period to control the temperature and invoke the load balancer. If the user performs frequent temperature checks and load balancing, it may lead to loss of performance due to overhead. But if the user specifies long interval to check and load balance, then the temperature of the core may exceed the specified temperature threshold leading to overheating. Moreover, invoking a load balancer also incurs overhead. Thus, if the user invokes the load balancer frequently, then the overhead of load balancing may exceed the benefit. But if the load balancer is invoked infrequently, then it may result in loss of performance due to load imbalance. Putting the burden on the user to specify an ideal temperature check and load balancing period may be inefficient.

In this work, we propose a framework, MetaTempController, which will automatically control the temperature of cores and perform load balancing without any support from the user. In this framework, which will be a part of the adaptive run-

time system, the runtime system will monitor the application characteristics and the core temperatures asynchronously. To minimize the cooling energy we increase the CRAC temperature, use DVFS to limit the processor temperature and perform load balancing automatically based on the information collected by the runtime system. This work extends the cool load balancer approach [92] and builds upon on the concept of an automated load balancing framework [40].

The key contributions of this chapter are:

- We introduce a generic technique that can be used to automatically control the temperature of the processors and avoid hot-spots.
- We demonstrate that our dynamic technique has less timing penalty and can be used with a wide range of applications having different characteristics.
- We present an implementation of our concept as `MetaTempController` in Charm++ runtime system which executes in the background and is transparent to the application programmer.

8.1 Background

Our approach to saving cooling energy involves setting a high CRAC temperature value. But to prevent overheating and formation of hot spots, we use DVFS to control the temperature of each chip. In order to efficiently control the temperature and minimize the timing penalty, we rely on an adaptive runtime system with the capability for load balancing. We chose the CHARM++ parallel programming system for this purpose.

8.1.1 Charm++ and its Load Balancing Framework

CHARM++ [85] is a message driven parallel programming model that has parallel entities called objects or chares. Chares form the basic unit of computation. A programmer divides the computation into chares, which are distributed among processors by the runtime system. It hinges on the idea of over-decomposition, i.e., dividing the problem into more work units than the total number of processors in the system. In turn, this over-decomposition improves the performance by overlapping communication and computation. Each of these tasks or chares is a migratable C++ object that can reside on any processor and can be migrated to any processor. This migratable nature of chares provides the capability for load balancing. When there is an imbalance of load, migrating the objects from overloaded processors to underloaded processors helps achieve balance and improve the performance of the application. The CHARM++ runtime system records the computation load and the communication pattern of these chares and use this information for load balancing. The load balancing framework in CHARM++ is based on a heuristic known as the *principle of persistence* [6], which states that the recent past is a good indication of the future. CHARM++ provides the application programmer with a suite of load balancers and the capability to add new custom load balancing strategies. These load balancers can be easily plugged in to the application at runtime. The key advantage of this approach is that it is application independent.

8.1.2 Temperature Control using DVFS

Dynamic voltage and frequency scaling (DVFS) is a widely used technique to automatically adjust the frequency of a processor either to conserve power or to reduce the amount of heat generated. Several manufacturers have developed processors capable of global dynamic frequency and voltage scaling.

Algorithms using DVFS have shown dramatic energy savings while providing the necessary peak computation power in general-purpose systems [94]. Fine-grained DVFS has emerged as a popular way for designers to exploit growing transistor budgets [95] in chip-multiprocessors (CMPs). The decrease in temperature allows the system to decrease the power dedicated for cooling or, if possible, to be turned off entirely increasing the overall system power savings.

However, reducing the frequency level slows down the computation. Ideally, DVFS techniques are used to manage the frequency and/or voltage so as to provide the minimum speed the processor needs to manage its workload while maintaining computational time constraints or throughput constraints and thereby reducing its energy consumption [96].

8.2 Related Work

Minimizing energy consumption has become an important subject for research in HPC. Cooling energy optimizations have been primarily addressed for data centers [97, 98]. In general, these techniques involve placing the most heat generating jobs in the coolest areas of the data center. This particular solution cannot be applied to our current work because different tasks in an HPC application behave very similarly and thus consume the same amount of energy and produce the same amount of heat and they often cover a large fraction of the machine. Another approach to reducing total energy consumption presented in [99] limits the temperature of the cores by turning the different nodes on and off as needed. This solution is problematic when applied to HPC because of the high interdependence between tasks, and the time penalty in execution time it would incur.

In HPC, controlling CPU frequency and voltage to reduce the energy have been studied before. For example, a previous work showed significant energy savings by

using DVFS to change the frequency of the cores during the communication phase of an MPI application [100]. The major drawback of this approach is the time penalty incurred in the execution time of the application. Another interesting work proposed in [101] creates a schedule for when DVFS should be run for a particular HPC application. The schedule tries minimize the timing penalty for a given power limit. In [102] a kernel-level DVFS governor is proposed that tries to determine an optimal frequency for a particular workload.

The closest work to the present work is the ‘Cool’ load balancer by Sarood Et al. [92]. In that work, an approach was proposed for saving cooling energy by constraining core temperature while minimizing the associated timing penalty using task migration. It uses DVFS and a temperature-aware load balancer to achieve this task. Although this scheme has shown substantial energy reduction for HPC applications at the cost of some modest timing penalty in the computation time, it relies on the user to specify a fixed period for temperature check and load balancing. Our approach, which is a part of the run time system, will automatically and dynamically perform the task of checking the temperature and load balancing without any input from the user.

8.3 Limitations of Periodic Approach

Although the recent work proposed by Sarood [92] is successful in reducing the cooling energy significantly, it has certain short comings. In this scheme, a temperature-aware load balancing strategy is proposed, which is invoked periodically at the user specified interval. At the specified period, a global barrier is enforced and temperature-aware load balancing is performed at the central location. As a part of the load balancing framework, the temperature of each processor is checked and if it exceeds the pre-set threshold, the frequency of that processor is reduced. If the

Algorithm 8 Periodic temperature-aware dynamic load balancing

Input: P - Set of processors T - Temperature threshold $Temp_{p_i}$ - Temperature of processor p_i At user specified period p

```
1: Enforce a global barrier
2: for all  $p_i \in P$  do
3:   if  $Temp_{p_i} > T$  then
4:     Decrease the frequency of  $p_i$ 
5:   else
6:     if  $Temp_{p_i} < T$  then
7:       Increase the frequency of  $p_i$ 
8:     end if
9:   end if
10: end for
11: Invoke the load balancer
```

temperature is below the threshold, then the frequency is increased. An adjustment of frequencies can result in load imbalance and to handle that, the load balancer is invoked. This scheme is depicted in Algorithm 8.

In this section, we will highlight the drawbacks of this scheme. Notice that the temperature check is triggered periodically every p seconds, where p is specified by the user. After the global barrier, DVFS is used to limit the temperature of cores and load balancing is performed to reduce the timing penalty due to load imbalance. Here, the application programmer has the responsibility of identifying the period for temperature checks and load balancing. This becomes increasingly a burden as the period is application and system dependent. This not only puts the burden on the application programmer, but also may result in not being able to control the temperature in a dynamic environment. Processors tend to have higher temperatures in computation intensive applications, while some applications with lower system utilization generate less heat. This indicates that the ideal temperature check and load balancing period is application dependent. Further, invoking the load balancer also incurs overhead. If the load balancing cost exceeds the benefit,

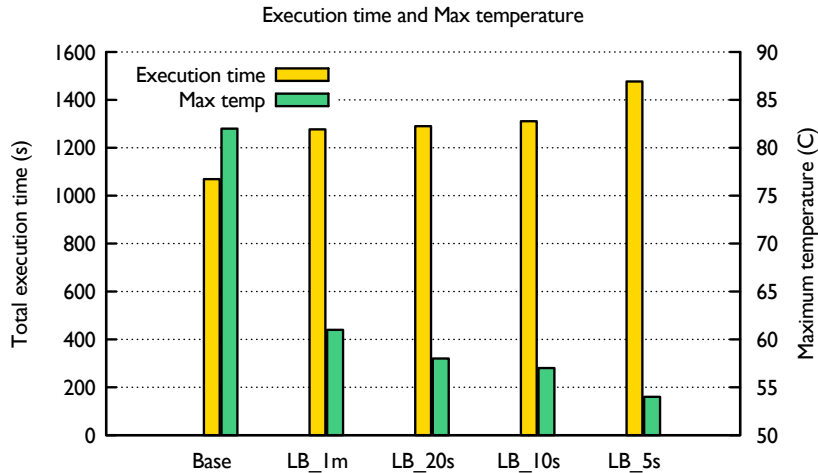


Figure 8.1: Comparison for maximum temperature and timing penalty for various user specified period

it results in increasing the total execution time.

Figure 8.1 shows the maximum temperature and timing penalty using this algorithm with different user specified periods for a run of *wave2D* on 128 cores. The CRAC is set to 24°F and the threshold temperature is 50°C. Details of the application and the experimental setup are described in Section 8.5. If the temperature check is performed frequently, the overhead due to barriers and load balancing may increase the timing penalty, whereas if the temperature check is performed infrequently, it could result in overheating of cores. Leaving it to the application programmer to manually identify the period in a dynamic application is inefficient.

8.4 MetaTempController

The MetaTempController framework is implemented as a part of the CHARM++ adaptive runtime system. The generic idea of this framework is to let the runtime system monitor the system temperature and application characteristics, and based on the collected information, make decisions to adjust the frequencies or invoke the load balancer. We choose to implement this framework in CHARM++, how-

ever, it is possible to implement this approach in any other programming model. MetaTempController consists of two major components, namely, automatic temperature controller and automatic load balancer.

8.4.1 Temperature Control

If the CRAC temperature is increased to reduce the cooling energy, it may result in an overheating of the processors. To ensure that the processors are not over heated and hot spots are not created, the temperature of the chip needs to be controlled. Temperature control plays an important part in reducing the cooling energy. In order to control the temperature effectively, MetaTempController collects the temperature information for each core in a distributed fashion. Temperature measurements for all the cores on a chip is collected frequently and decisions to control the temperature are made. Note that in this scheme, the temperature control is done independently on each processor, whereas in [92] there is a global barrier. Since the computer hardware in the cluster does not allow frequency changes of a single core, DVFS is applied to the entire chip. Further, the hardware has discrete voltage and frequency levels built into it, called the 'P-states'. The chip frequencies can be set only to those discrete operating points. Whenever the temperature of a core exceeds the specified threshold, the MetaTempController identifies this and triggers mechanism to limit the temperature. It uses DVFS to lower the frequency by one step (increase P-state by one level). Running the processor at a lower frequency reduces the amount of heat generated and helps reducing the machine and cooling energy. However, if all the cores on a chip have temperatures below the specified threshold, then the frequency of the chip is increased by one step. Since the temperature statistics are collected in a distributed manner without enforcing a barrier, this scheme incurs very little overhead.

8.4.2 Load Balancing

Even though DVFS limits the processor temperature and eliminates hot spots, it incurs a timing penalty. This timing penalty can occur due to: 1) processors operating at lower frequency or 2) load imbalance due to different processor speeds. In order to reduce the timing penalty, load balancing needs to be performed. However, performing load balancing entails overheads that include the time spent on collecting load balancing statistics, finding a new mapping and migrating the objects based on the mapping. Since the load balancer incurs overhead, it becomes necessary to determine whether invoking the load balancer is profitable. If the load balancer is invoked too frequently, the overhead of load balancing may exceed the benefit and result in increased execution time. A common practice is to invoke the load balancer periodically at a period specified by the user. However, this prevents load balancing from adapting to the dynamic application and system characteristics. MetaTempController relies on the concept of an automated load balancing framework [40]. This framework collects a minimum set of load balancing statistics in an asynchronous manner via a reduction tree. Once the aggregate information is available, it determines whether there is any load imbalance. If there is load imbalance, it may lead to performance loss. However, if the overhead of load balancing is more than the benefit, performing load balancing won't be beneficial. MetaTempController identifies an ideal load balancing period based on the application characteristics and the cost of load balancing.

8.5 Results

In this section, we present an evaluation of the effectiveness of MetaTempController and compare it with other schemes using three applications *wave2D*, *leanMD* and *kNeighbor*. We show that MetaTempController is able to constrain the core tem-

perature to a specified threshold, invoke the load balancer whenever beneficial and extract the best performance for the application automatically at run time.

8.5.1 Experimental Setup

The experiments were run on a cluster with 160 cores (40 nodes). Each node of the cluster is a single socket Dell T5500 machine with a quad-core Intel Xeon E5520 chip. The Intel Xeon E5520 chip supports seven different frequencies ranging from 1.6GHz to 2.53GHz through Intel’s Turbo Boost Technology. The *cpufreq* module which is available in Ubuntu 10.04 allows us to step up or down the frequency by 0.13GHz in each step. A frequency shift from one level to another takes a processor a few microseconds. For all our runs, we use 128 cores out of the 160 cores.

For all the experiments, the computer room air conditioning temperature was set to 74° F and the threshold temperature was fixed at 50° C. These are independent variables and each can effect the power reduction greatly. The effect of those to power reduction is discussed in detail in an earlier work [92].

8.5.2 Applications

wave2D is a computation-intensive finite differencing application. It is implemented using a 2-D mesh structure. Our runs execute 25,000 iterations with a mesh of size 128×16 .

leanMD is a molecular dynamics application written in Charm++ that simulates the behavior of atoms based on the Lennard-Jones potential. The computations performed in this program are similar to the force calculation in NAMD [81]. The simulation is in a three-dimensional space consisting of atoms which are divided into cells. In each iteration, force calculations are done for all pairs of nearby atoms. Once the force calculation is performed, the cells update the acceleration, velocity and position of the atoms within their space. We benchmark *leanMD* on a system

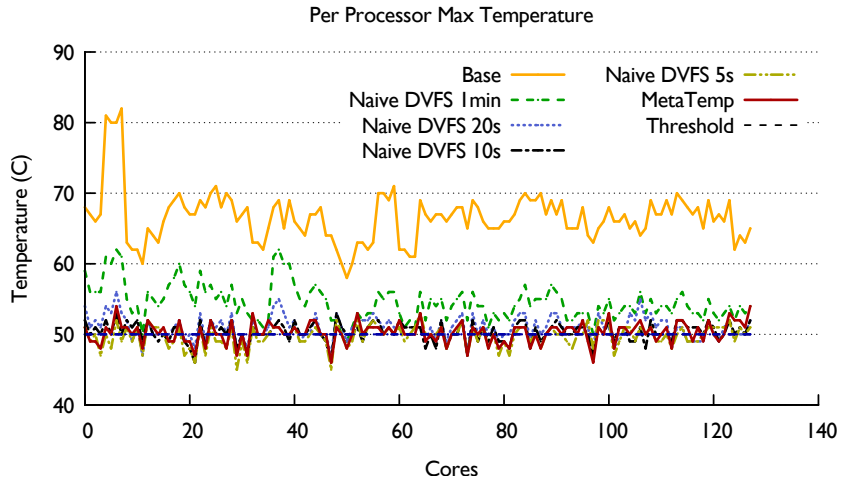


Figure 8.2: Maximum Temperature of the Processors for *wave2D*

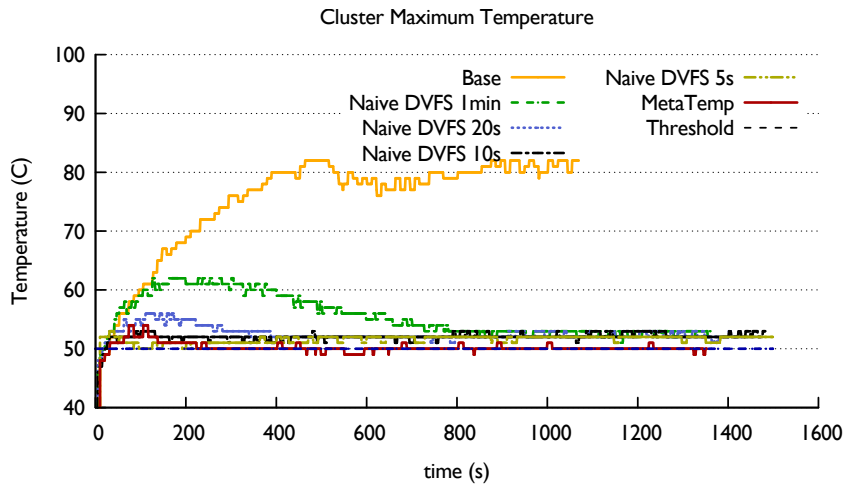


Figure 8.3: Maximum Temperature of the Processors Over Time for *wave2D* of 128,000 atoms for 500 iterations.

kNeighbor is a micro-benchmark with a near-neighbor communication pattern. In this benchmark, each object exchanges 16KB sized messages with a fixed set of fourteen neighbors in every iteration. We evaluate this benchmark by executing 25,000 iterations.

All the above applications do not have any inherent load imbalance. Thus, any imbalance that occurs is a result of changes to processor frequencies.

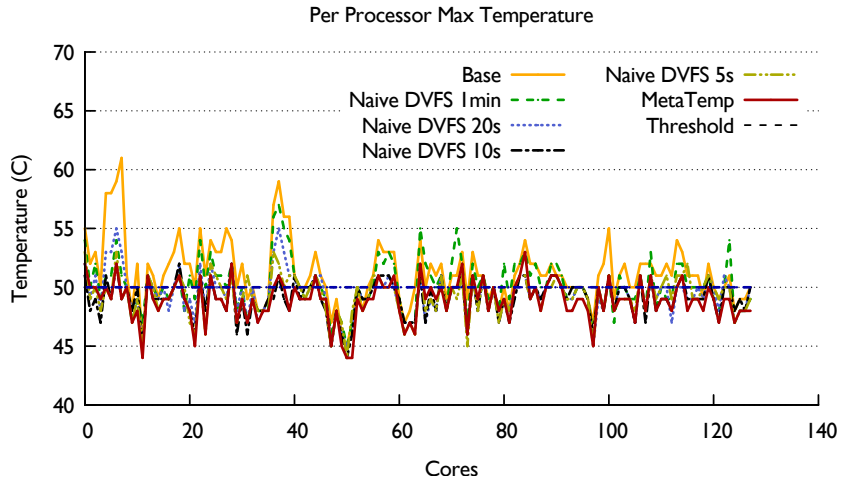


Figure 8.4: Maximum Temperature of the Processors for kNeighbor

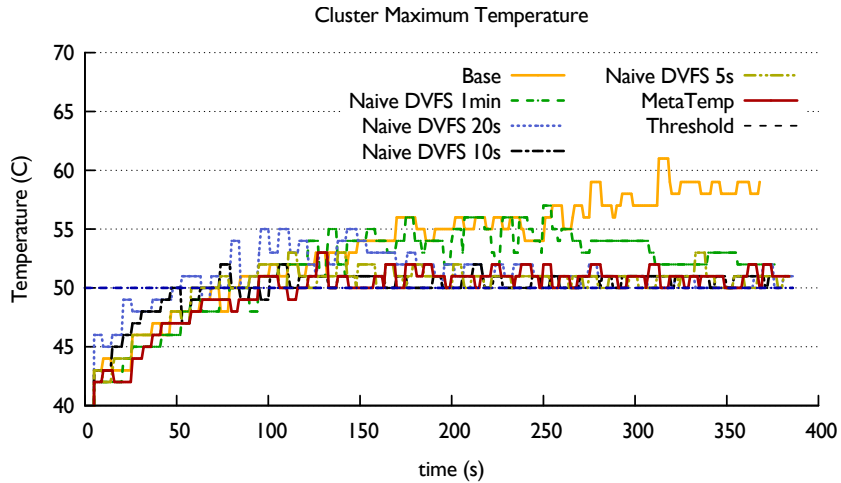


Figure 8.5: Maximum Temperature of the Processors Over Time for kNeighbor

8.5.3 Experimental Results

We use the following metrics to evaluate the effectiveness and behavior of MetaTempController: 1) Temperature Control, 2) Timing Penalty 3), Frequency, 4) Overhead, 5) Power and Energy

Temperature Control

wave2D: *wave2D* being computation intensive benchmark, results in an increase in core temperature and hot-spots. Figure 8.3 shows that for a run of *wave2D* without

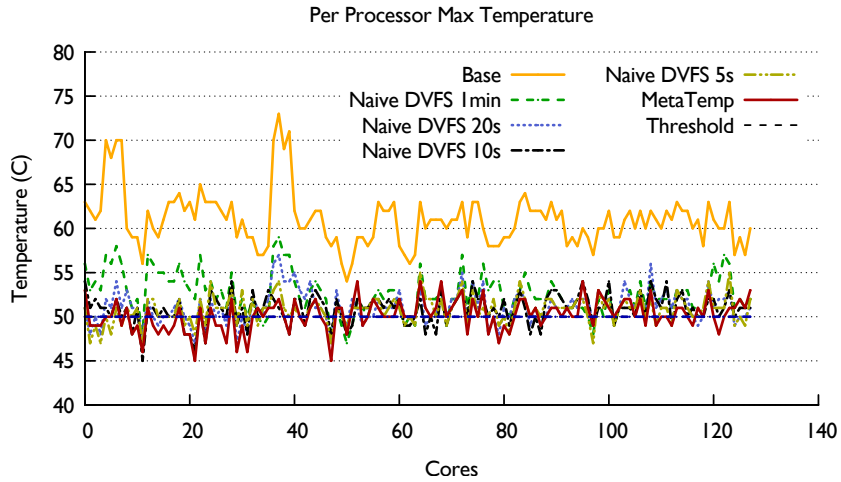


Figure 8.6: Maximum core temperature for the entire run for leanMD. This indicates region of hot-spots.

any temperature control, the maximum temperature on any core reaches 82°C . Figure 8.2 indicates that some of the cores are hot-spots. Core temperatures are checked periodically and DVFS is used to keep the temperature of a core within the threshold of 50°C . Figure 8.3 shows the maximum temperature of any core over time for various temperature check period. A period of 1 min is able to bring the maximum temperature down to 62°C but it is insufficient to keep the temperature within the threshold. Temperature check with 20s period is able to reduce the temperature further but it is still above the threshold. For this application, a periodicity of 5 seconds is necessary to ensure that the maximum temperature of any core is within the threshold. MetaTempController is able to automatically control the temperature using DVFS and keep it within the threshold.

kNeighbor: Unlike *wave2D* or *leanMD*, *kNeighbor* is a communication intensive benchmark because of which the temperature of the cores reaches a maximum of 61°C without any temperature control as shown in Figure 8.5. Again Figure 8.4 indicates the formation of hot-spots. A periodicity of 1 min for temperature check is not sufficient to keep the temperature within threshold of 50°C . Whereas a periodicity of 10 or 5 seconds controls the temperature. MetaTempController successfully

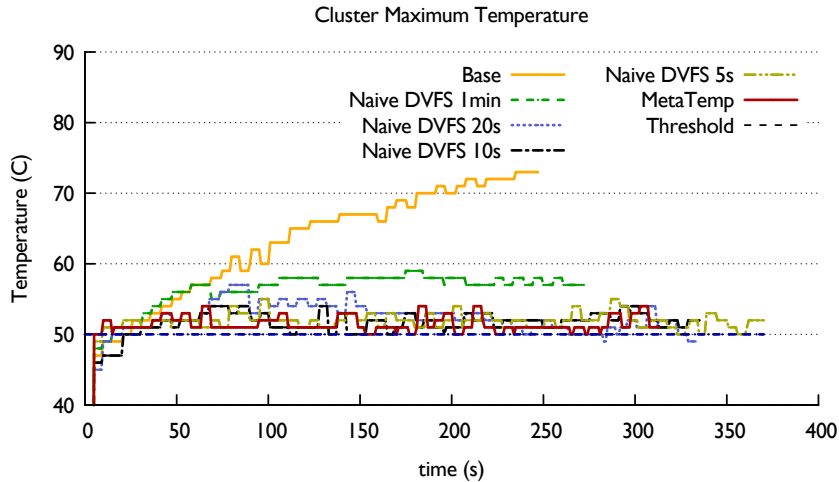


Figure 8.7: Maximum temperature on any core over time for leanMD. Without any control, temperature reaches 73° C and MetaTempController keeps it within threshold.

controls the temperature to within the specified threshold of 50° C. The key thing to note here is that the ideal period to control the temperature is application dependent. For *wave2D* the ideal period was 5 seconds whereas for *kNeighbor* it is 10 seconds. MetaTempController automatically adjusts the temperature without any support from the user.

leanMD: Figure 8.7 shows the maximum temperature for any core in the system for the entire run of *leanMD* using various periodicity for temperature control. It can be seen that for the run of *leanMD* without any temperature control, the maximum temperature goes up to 73° C. This is above the threshold of 50° C. Figure 8.6 indicates that there are few hot-spots created resulting in high temperature. A periodicity of 1 min is able to control the temperature to a certain extend, but still causes the temperature to reach 59° C. This indicates that periodicity of 1 min is not frequent enough to keep the temperature within the threshold. For *leanMD*, a periodicity of 10 seconds is required to ensure that the maximum temperature of any core in the system is within the threshold. We find that, MetaTempController is successful in keeping the temperature within the threshold of 50° C.

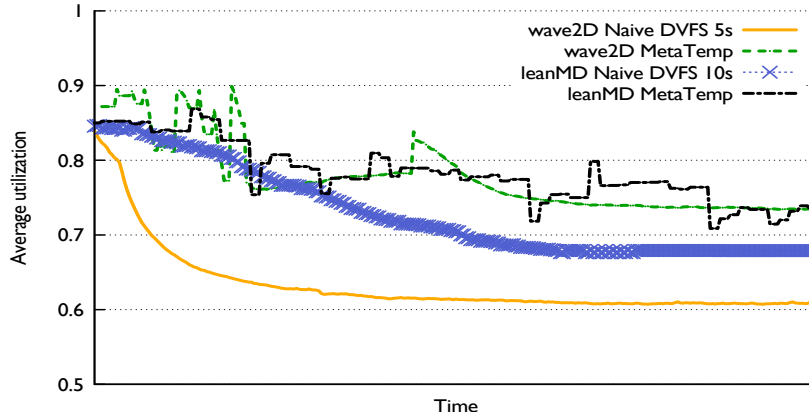


Figure 8.8: Average utilization over time with and without MetaTempController

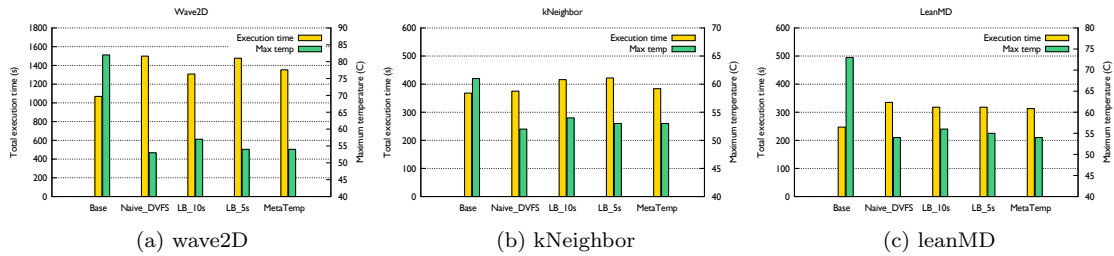


Figure 8.9: Execution time and temperature for different strategies

Timing Penalty

wave2D: Using DVFS to control temperature results in load imbalance which leads to low system utilization. Figure 8.8 shows the average system utilization when the temperature is controlled. The system utilization drops from 89% to 60% during the run. The frequency of the cores that are hot-spots are reduced, which results in load imbalance. Figure 8.8 shows the average system utilization when load balancing is performed. It can be seen that the load balancer is successful in improving the utilization and attains a minimum utilization of 73%.

Load balancing incurs overhead that includes the time for finding a new assignment of objects to processors as well as the time for migration. Figure 8.9 compares various schemes including no temperature control, temperature control without load balancing, periodic load balancing and MetaTempController. In the

no temperature control case, the total execution time is 1069 seconds, but the core temperature reaches 82° C. Controlling the temperature using DVFS keeps the temperature within the threshold, but the execution time increases by 40% to 1499 seconds. Performing load balancing frequently incurs overhead that may overshoot the gains from load balancing. A periodic load balancer with a period of 5 seconds has an execution time of 1477 seconds and therefore does not provide much benefit. A period of 10 seconds is insufficient to keep the temperature within threshold and causes temperature to rise till 57° C. MetaTempController successfully controls the temperature and removes hot-spots using DVFS and also reduces the timing penalty by 10%.

kNeighbor: *kNeighbor* being communication intensive, its characteristics is different from *wave2D* or *leanMD*. Figure 8.9 shows the maximum temperature and the total execution time for various schemes including no temperature control, temperature control, periodic load balancing and MetaTempController. Without any temperature control, the execution time is 368 seconds and the maximum temperature is 61° C. It can be seen that controlling the temperature with DVFS results in a slowdown of only 4%. This indicates that there is no significant load imbalance. Therefore, performing load balancing very often will not yield any benefit and instead will incur more overhead. Figure 8.9 shows that the periodic load balancer incurs more overhead and increases the total execution time by 13% in comparison to the no temperature control run and 8% to the temperature control run. MetaTempController automatically calls the load balancer only if the benefit of load balancing exceeds the overhead. It identifies that load balancing does not improve and hence invokes load balancing only once. The timing penalty of MetaTempController is 4% over the no temperature control run. Thus MetaTempController is able to automatically control the temperature within the threshold as well as minimize the timing penalty depending on the application characteristics.

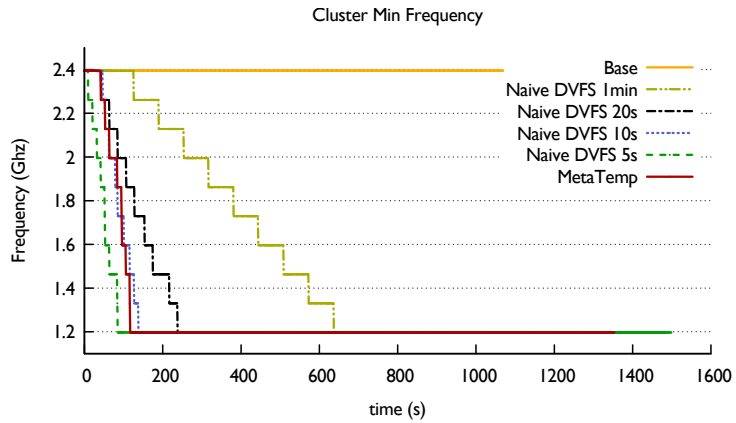
leanMD: In order to control the temperature, the frequency of the chip is adjusted using DVFS. Decreasing the frequency results in load imbalance which leads to lower system utilization. Figure 8.8 shows the average utilization of the system when the temperature is controlled using DVFS. In the beginning of the run, the average utilization is 85% and reduces to 67%. This is due to the load imbalance created as a result of the reduction in the frequency of the cores which are hot-spots. Figure 8.8 shows the average utilization using the load balancer. The average utilization of the system improves in comparison to no load balancer and attains a minimum utilization of 73%.

In Figure 8.9, without any temperature control, the total execution time is 247 seconds, but the maximum temperature reaches 73° C. Performing temperature control without any load balancing results in a total execution time of 335 seconds and a slowdown of 35%. Periodic load balancing reduces the timing penalty by 5% with a total execution time of 318 seconds. MetaTempController automatically performs temperature control and load balancing leading to an execution time of 313 seconds. This shows that MetaTempController is able to keep the temperature within the threshold as well as reduce the timing penalty automatically without any user support.

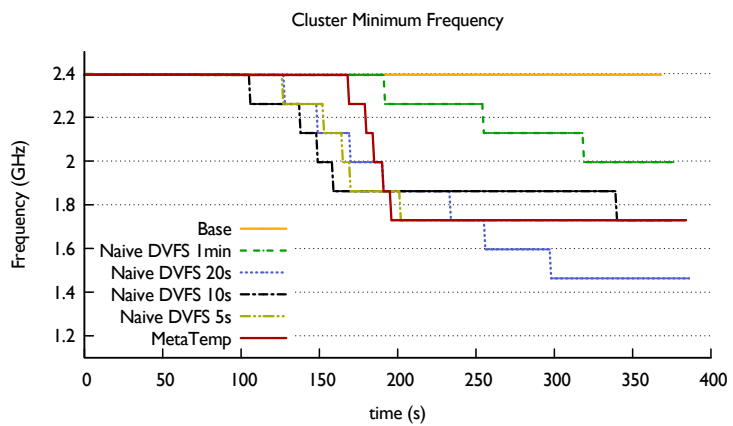
Frequency

Without load balancing, the processor with the slowest frequency dictates the total execution time of the application. In the timing penalty section we have shown that when the temperature check period decreases, total execution time increases. The reason for this behavior can be seen in Figure 8.10. Frequent temperature checks causes to reach the minimum frequency at a faster rate and the stable minimum frequency to be lower.

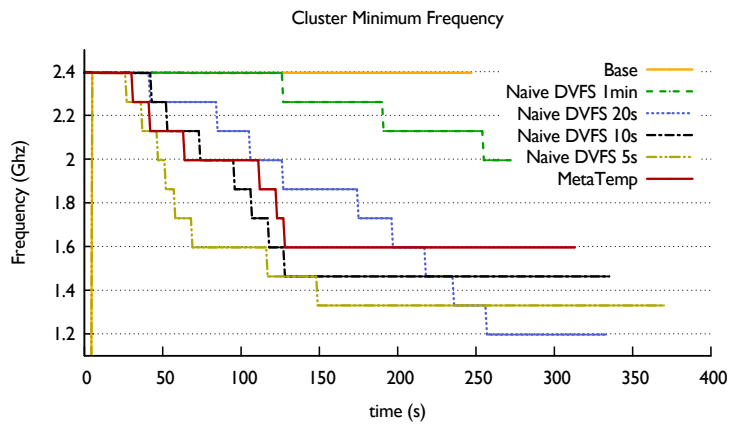
Load balancing enables MetaTempController to maintain a higher frequency



(a) wave2D



(b) kNeighbor



(c) leanMD

Figure 8.10: Minimum frequency of the processors over time compared to naive DVFS in all of the applications as the Figure 8.10 shows. It removes the work from the overloaded processors so that they do not heat up that

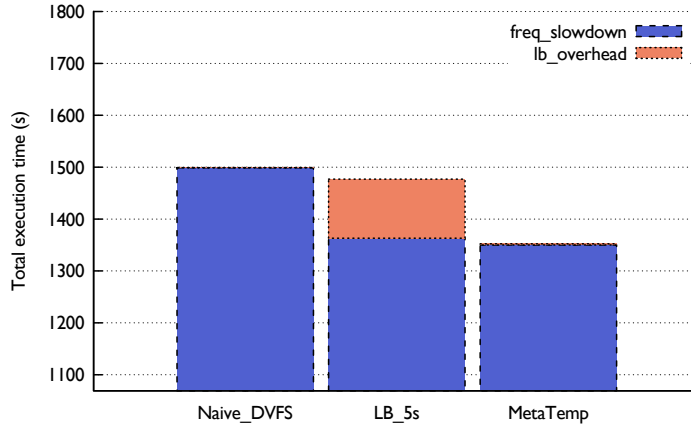
much and need to decrease the frequency. Without load balancing the processor with the slowest frequency dictates the total execution time of the application. In the timing penalty section we have shown that when the temperature check period decreases, total execution time increases. The reason for this behaviour can be seen in Figure 8.10. Frequent temperature checks causes the system to reach the minimum frequency at a faster rate and the stable minimum frequency to be lower.

Overhead

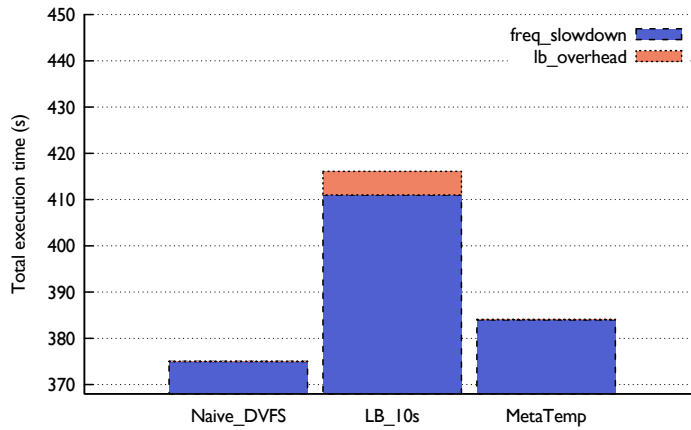
Figure 8.11 shows the slowdown caused by the load balancing and temperature check. The starting point of the y-axis is the execution time of the plain run. MetaTempController has less slowdown caused by frequency decreases compared to both naive DVFS and periodic load balancing as it has a higher stable frequency as stated in the previous section. Moreover, MetaTempController has a negligible load balancing overhead. The reason for this is its smart load balancing strategy. kNeighbor represents an exceptional case. Because it is highly communication bound, naive DVFS does not cause a significant overhead. The processors do not heat up and exceed the threshold temperature, and thus frequency decrease and load balancing is not needed. MetaTempController understands this and only does load balancing two times in the beginning of the application. On the other hand, periodic temperature check strategy continues load balancing until the end, which is the main reason for the significant difference between MetaTempController and the periodic approach. MetaTempController has more severe frequency slowdown than naive DVFS, but this is a cost worth paying for the universality.

Power and Energy

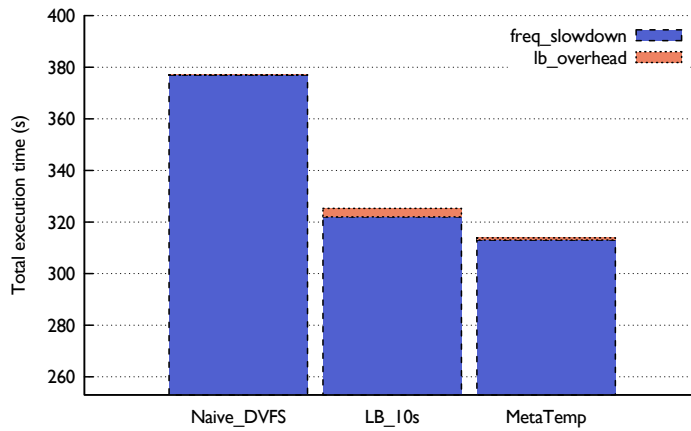
In this section, we evaluate the ability of MetaTempController to reduce energy consumption in comparison to the naive DVFS scheme and periodic temperature-aware



(a) wave2D



(b) kNeighbor



(c) leanMD

Figure 8.11: Execution time breakdown into temp check and lb overhead load balancer. We also reduce the timing penalty for applications while limiting the core temperatures to the specified thresholds. While we get savings from cool-

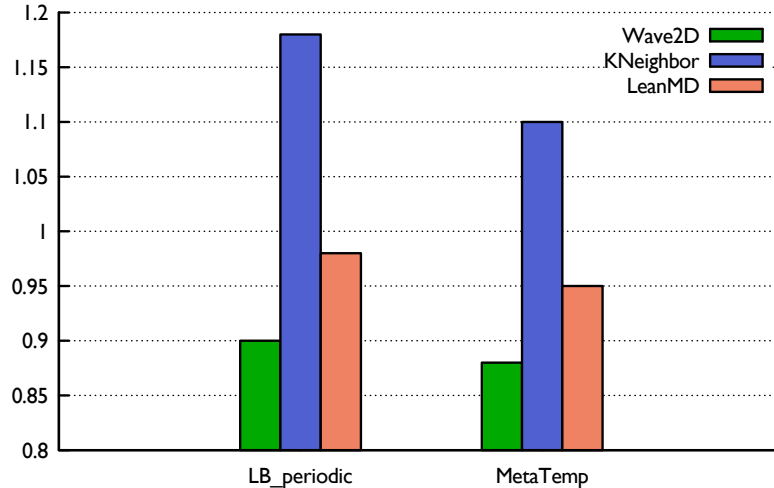


Figure 8.12: Machine energy normalized according to naive DVFS

ing energy, we also manage to improve the machine energy. Figure 8.12 shows the normalized machine energy for periodic strategy and MetaTempController with respect to the naive DVFS. Machine energy is calculated as the product of the average power and the execution time. For *leanMD* and *wave2D*, the hand tuned periodic temperature-aware load balancer and MetaTempController are able to reduce the machine energy in comparison to the naive DVFS scheme. For *leanMD*, periodic scheme gives 2% whereas MetaTempController gives 5% reduction in machine energy. We see a much higher reduction for *wave2D*, where the periodic scheme provides 9% and MetaTempController provides 12% reduction in machine energy. Since *kNeighbor* is a communication intensive benchmark, naive DVFS results in an only slight increase in the total execution time. We also saw in Section 8.5.3 that invoking the load balancer frequently increased the execution time. This results in increasing the machine energy for the periodic scheme by 15.5%. Since MetaTempController automatically identifies this, it invokes load balancing less frequently. Therefore, even though it increases the machine energy it is not as bad as the periodic scheme.

8.6 Conclusion

Increase in power demands and total energy consumption in HPC has become an important issue in the construction and maintenance of machines. In this work, we extended a previous approach on temperature-aware load balancing and our work on an automated load balancing framework to implement an automatic control system for reducing the cooling energy. We introduced MetaTempController, which automatically controls the temperature using DVFS and performs load balancing to minimize the overhead without any input from the user. We demonstrated the effectiveness of MetaTempController using three applications. We showed that MetaTempController is able to successfully limit the temperature and reduce the timing penalty in comparison to the existing schemes.

References

- [1] Bilge Acun, Phil Miller, and Laxmikant V. Kalé. Variation among processors under Turbo Boost in HPC systems. In *International Conference on Supercomputing (ICS)*. ACM, 2016.
- [2] Yow-Jian Lin and Vipin Kumar. And-parallel execution of logic programs on a shared-memory multiprocessor. *J. Log. Program.*, 10(1/2/3&4):155–178, 1991.
- [3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP'95*, pages 207–216, Santa Barbara, California, July 1995. MIT.
- [4] Yanhua Sun, Gengbin Zheng, Pritish Jetley, and Laxmikant V. Kale. An Adaptive Framework for Large-scale State Space Search. In *Proceedings of Workshop on Large-Scale Parallel Processing (LSPP) in IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2011*, Anchorage, Alaska, May 2011.
- [5] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.
- [6] Laxmikant V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.
- [7] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. Overcoming scaling challenges in biomolecular simulations across multiple platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.
- [8] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Massively parallel cosmological simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

- [9] Sandhya Mangala, Terry Wilmarth, Sayantan Chakravorty, Niles Choudhury, Laxmikant V. Kale, and Philippe H. Geubelle. Parallel adaptive simulations of dynamic fracture events. *Engineering with Computers*, 24:341–358, December 2007.
- [10] Gengbin Zheng. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.
- [11] George Cybenko. Dynamic Load Balancing for Distributed Memory Multiprocessors. *Journal of parallel and distributed computing*, 7(2):279–301, 1989.
- [12] Ishfaq Ahmad and Arif Ghafoor. A semi distributed task allocation strategy for large hypercube supercomputers. In *Proceedings of the 1990 conference on Supercomputing*, pages 898–907, New York, NY, 1990. IEEE Computer Society Press.
- [13] Gengbin Zheng, Abhinav Bhatele, Esteban Meneses, and Laxmikant V. Kale. Periodic Hierarchical Load Balancing for Large Supercomputers. *International Journal of High Performance Computing Applications (IJHPCA)*, March 2011.
- [14] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. Work stealing and persistence-based load balancers for iterative overdistributed applications. In *HPDC*, 2012.
- [15] W.O. Kermack and A.G. McKendrick. Contributions to the mathematical theory of epidemics. ii. the problem of endemicity. *Proceedings of the Royal society of London. Series A*, 138(834):55–83, 1932.
- [16] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.
- [17] George Karypis and Vipin Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Proc. of the 8th SIAM conference on Parallel Processing for Scientific Computing*, 1997.
- [18] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy, and L.A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *IPDPS*, pages 1–11. IEEE, 2007.
- [19] Cdric Chevalier, Francois 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 *In Proceedings of Euro-Par 2006, LNCS 4128:243252*, pages 243–252, 2006.

- [20] B. Hendrickson and R. Leland. The Chaco user's guide. Technical Report SAND 93-2339, Sandia National Laboratories, Albuquerque, NM, October 1993.
- [21] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.
- [22] S. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. Saltz. Runtime and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing 1994*, November 1994.
- [23] Yaun-Chien Chow and Walter H. Kohler. Models for dynamic load balancing in homogeneous multiple processor systems. In *IEEE Transactions on Computers*, volume c-36, pages 667–679, May 1982.
- [24] L. M. Ni and Kai Hwang. Optimal load balancing in a multiple processor system with many job classes. In *IEEE Trans. on Software Eng.*, volume SE-11, 1985.
- [25] Chengzhong Xu, Francis C. M. Lau, and Ralf Diekmann. Decentralized remapping of data parallel applications in distributed memory multiprocessors. *Concurrency - Practice and Experience*, 9(12):1351–1376, 1997.
- [26] Jacques E. Boillat. Load balancing and poisson equation in a graph. *Concurrency: Practice and Experience*, 2(4):289–313, 1990.
- [27] YF Hu and RJ Blake. An optimal dynamic load balancing algorithm. Technical report, Daresbury Laboratory, 1995.
- [28] David Peleg and Eli Upfal. The token distribution problem. *SIAM Journal on Computing*, 18(2):229–243, 1989.
- [29] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive load balancing policies for dynamic applications. In *IEEE Concurrency*, pages 7(1):22–31, 1999.
- [30] Anna Ha'c and Xiaowei Jin. Dynamic load balancing in distributed system using a decentralized algorithm. In *Proc. of 7-th Intl. Conf. on Distributed Computing Systems*, pages 170–177, April 1987.
- [31] L. V. Kalé. Comparing the performance of two dynamic load distribution methods. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 8–11, St. Charles, IL, August 1988.
- [32] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. In *IEEE Transactions on Parallel and Distributed Systems*, volume 4, pages 979–993, September 1993.

- [33] Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *Software Engineering, IEEE Transactions on*, (1):32–38, 1987.
- [34] Bruce Hendrickson and Karen Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184(2):485–500, 2000.
- [35] K.P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.
- [36] Michael Short. Improved inequalities for the poisson and binomial distribution and upper tail quantile functions. *ISRN Probability and Statistics*, 2013, 2013.
- [37] F. Mattern. Algorithms for distributed termination detection. *Distributed computing*, 2(3):161–175, 1987.
- [38] Akhil Langer, Jonathan Lifflander, Phil Miller, Kuo-Chuan Pan, , Laxmikant V. Kale, and Paul Ricker. Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement. In *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2012)*, New York, USA, October 2012.
- [39] Chao Mei, Yanhua Sun, Gengbin Zheng, Eric J. Bohm, Laxmikant V. Kalé, James C. Phillips, and Chris Harrison. 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*, Seattle, WA, November 2011.
- [40] Harshitha Menon, Nikhil Jain, Gengbin Zheng, and Laxmikant V. Kalé. Automated load balancing invocation based on application characteristics. In *IEEE Cluster 12*, Beijing, China, September 2012.
- [41] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):101–117, 1994.
- [42] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 28, New York, NY, USA, 1995. ACM.
- [43] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *BELL System Technical Journal*, pages 291–307, February 1970.
- [44] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. 19th Design Automation Conference*, pages 175–181, 1982.

- [45] Edi Shmueli, George Almasi, Jose Brunheroto, Jose Castanos, Gabor Dozsa, Sameer Kumar, and Derek Lieber. Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L. In *Proceedings of the 22Nd Annual International Conference on Supercomputing, ICS '08*, pages 165–174, New York, NY, USA, 2008. ACM.
- [46] Suzanne M Kelly and Ron Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Annual Technical Conference*, pages 16–19. Citeseer, 2005.
- [47] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *ACM/IEEE SC2003*, Phoenix, Arizona, November 10–16, 2003.
- [48] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.
- [49] Vivek Kale, Abhinav Bhatele, and William D. Gropp. Weighted locality sensitive scheduling for mitigating noise on multicore clusters. In *18th annual IEEE International Conference on High Performance Computing (HiPC 2011)*, December 2011.
- [50] Erik Demaine. A threads-only MPI implementation for the development of parallel programs. In *In: Proceedings of the 11th International Symposium on High Performance Computing Systems*, pages 153–163, 1997.
- [51] Kai Shen, Hong Tang, and Tao Yang. Adaptive two-level thread management for fast MPI execution on shared memory machines. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, SC '99*, New York, NY, USA, 1999. ACM.
- [52] Hong Tang, Kai Shen, and Tao Yang. Program transformation and runtime support for threaded MPI execution on shared-memory machines. *ACM Trans. Program. Lang. Syst.*, 22(4):673–700, July 2000.
- [53] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian W. Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. Leveraging MPI's one-sided communication interface for shared-memory programming. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface, EuroMPI'12*, pages 132–141, Berlin, Heidelberg, 2012. Springer-Verlag.

- [54] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. MPI+MPI: a new hybrid approach to parallel programming with MPI plus shared memory. *Computing*, 95(12):1121–1136, 2013.
- [55] Andrew Friedley, Greg Bronevetsky, Torsten Hoefler, and Andrew Lumsdaine. Hybrid MPI: efficient message passing for multi-core systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 18. ACM, 2013.
- [56] Lorna Smith and Mark Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9(2,3):83–98, August 2001.
- [57] Eduard Ayguade, Marc Gonzalez, Xavier Martorell, and Gabriele Jost. Employing nested OpenMP for the parallelization of multi-zone computational fluid dynamics applications. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 6. IEEE, 2004.
- [58] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP '09*, pages 427–436, Washington, DC, USA, 2009. IEEE Computer Society.
- [59] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M Badia, Eduard Ayguade, and Jesús Labarta. Productive cluster programming with ompss. In *Euro-Par 2011 Parallel Processing*, pages 555–566. Springer, 2011.
- [60] James Dinan, Pavan Balaji, Ewing Lusk, P Sadayappan, and Rajeev Thakur. Hybrid parallel programming with MPI and unified parallel C. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 177–186. ACM, 2010.
- [61] Chao Mei. *Message-driven parallel language runtime design and optimizations for multicore-based massively parallel machines*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.
- [62] Julita Corbalan, Alejandro Duran, and Jesus Labarta. Dynamic load balancing of MPI+OpenMP applications. In *Parallel Processing, 2004. ICPP 2004. International Conference on*, pages 195–202. IEEE, 2004.
- [63] Vivek Kale, Amanda Randles, and William D Gropp. Locality-optimized mixed static/dynamic scheduling for improving load balancing on SMPs. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 115. ACM, 2014.

- [64] Vivek Kale, Simplice Donfack, Laura Grigori, and William D Gropp. Lightweight scheduling for balancing the tradeoff between load balance and locality, 2014. Poster presented at SC'14.
- [65] Simplice Donfack, Laura Grigori, William D Gropp, and Vivek Kale. Hybrid static/dynamic scheduling for already optimized dense matrix factorization. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 496–507. IEEE, 2012.
- [66] Marc Prache, Patrick Carribault, and Herv Jourden. MPC-MPI: An MPI implementation reducing the overall memory consumption. In Matti Ropo, Jan Westerholm, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 16th European PVM/MPI Users Group Meeting (EuroPVM/MPI 2009)*, volume 5759 of *Lecture Notes in Computer Science*, pages 94–103. Springer Berlin Heidelberg, 2009.
- [67] Marc Pérache, Hervé Jourden, and Raymond Namyst. Mpc: A unified parallel runtime for clusters of numa machines. In *European Conference on Parallel Processing*, pages 78–88. Springer, 2008.
- [68] Patrick Carribault, Marc Prache, and Herv Jourden. Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC. In Mitsuhsa Sato, Toshihiro Hanawa, MatthiasS. Miller, Barbara M. Chapman, and BronisR. de Supinski, editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, Proceedings of the 6th International Workshop on OpenMP (IWOMP 2010)*, volume 6132 of *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin Heidelberg, 2010.
- [69] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.
- [70] OpenMP ARB. OpenMP application program interface version 3.0. In *The OpenMP Forum, Tech. Rep*, 2008.
- [71] Rajkishore Barik, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşirlar, Yonghong Yan, et al. The Habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 735–736. ACM, 2009.
- [72] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *ACM Sigplan Notices*, volume 45, pages 341–342. ACM, 2010.
- [73] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28. ACM, 2005.

- [74] C. W. Purcell, J. S. Bullock, E. J. Tollerud, M. Rocha, and S. Chakrabarti. The Sagittarius impact as an architect of spirality and outer rings in the Milky Way. *Nature*, 477:301–303, September 2011.
- [75] J.-h. Kim, T. Abel, O. Agertz, G. L. Bryan, D. Ceverino, C. Christensen, C. Conroy, A. Dekel, N. Y. Gnedin, N. J. Goldbaum, J. Guedes, O. Hahn, A. Hobbs, P. F. Hopkins, C. B. Hummels, F. Iannuzzi, D. Keres, A. Klypin, A. V. Kravtsov, M. R. Krumholz, M. Kuhlen, S. N. Leitner, P. Madau, L. Mayer, C. E. Moody, K. Nagamine, M. L. Norman, J. Onorbe, B. W. O’Shea, A. Pillepich, J. R. Primack, T. Quinn, J. I. Read, B. E. Robertson, M. Rocha, D. H. Rudd, S. Shen, B. D. Smith, A. S. Szalay, R. Teyssier, R. Thompson, K. Todoroki, M. J. Turk, J. W. Wadsley, J. H. Wise, A. Zolotov, and f. t. AGORA Collaboration²⁹. The AGORA High-resolution Galaxy Simulations Comparison Project. *The Astrophysical Journal*, 210:14, January 2014.
- [76] L.V. Kalé and Amitabh Sinha. Projections : A scalable performance tool. In *Parallel Systems Fair, International Parallel Processing Symposium*, pages 108–114, April 1993.
- [77] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.
- [78] A. J. Kunen, T. S. Bailey, and P. N. Brown. KRIPKE - a massively parallel transport mini-app. 2015.
- [79] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Toton, Lukasz Wesolowski, and Laxmikant Kale. Parallel programming with migratable objects: Charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC ’14*, pages 647–658, Piscataway, NJ, USA, 2014. IEEE Press.
- [80] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.
- [81] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, MD, September 2002.
- [82] Greg Weirs, Vikram Dwarkadas, Tomek Plewa, Chris Tomkins, and Mark Marr-Lyon. Validating the Flash code: vortex-dominated flows. In *Astrophysics and Space Science*, volume 298, pages 341–346. Springer, 2005.

- [83] 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(2–3):133–152, 2005.
- [84] Orion Lawlor, Sayantan Chakravorty, Terry Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng, and Laxmikant Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235, September 2006.
- [85] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOP-SLA '93*, pages 91–108. ACM Press, September 1993.
- [86] Olga Pearce, Todd Gamblin, Bronis R. de Supinski, Martin Schulz, and Nancy M. Amato. Quantifying the effectiveness of load balance algorithms. In *26th ACM international conference on Supercomputing, ICS '12*, pages 185–194, 2012.
- [87] Chombo Software Package for AMR Applications. <http://seesar.lbl.gov/anag/chombo>.
- [88] Julita Corbalán, Alejandro Duran, and Jesús Labarta. Dynamic load balancing of mpi+openmp applications. In *ICPP*, pages 195–202, 2004.
- [89] Ioana Banicescu and Susan Flynn Hummel. Balancing processor loads and exploiting data locality in n-body simulations. In *In Proceedings of Supercomputing95 (CD-ROM, 1995*.
- [90] Bruce S. Siegell and Peter A. Steenkiste. Automatic selection of load balancing parameters using compile-time and run-time information, 1996.
- [91] R. Sawyer. Calculating total power requirements for data centers. *White Paper, American Power Conversion*, 2004.
- [92] Osman Sarood, Phil Miller, Ehsan Totoni, and L. V. Kale. ‘Cool’ Load Balancing for High Performance Computing Data Centers. In *IEEE Transactions on Computer - SI (Energy Efficient Computing)*, September 2012.
- [93] Osman Sarood and Laxmikant V. Kalé. A ‘cool’ load balancer for parallel applications. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.
- [94] P. Pillai Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

- [95] S. Herbert. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. *International Symposium on Low Power Electronics and Design*, 2007.
- [96] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *Proceedings of the conference on Design, automation and test in Europe-Volume 1*, page 10004. IEEE Computer Society, 2004.
- [97] Cullen Bash and George Forman. Cool job allocation: measuring the power savings of placing jobs at cooling-efficient locations in the data center. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–6. USENIX Association, 2007.
- [98] Lizhe Wang, Gregor von Laszewski, Jai Dayal, and Thomas R Furlani. Thermal aware workload scheduling with backfilling for green data centers. In *Performance Computing and Communications Conference (IPCCC), 2009 IEEE 28th International*, pages 289–296. IEEE, 2009.
- [99] Shen Li, Hieu Le, Nam Pham, Jin Heo, and Tarek Abdelzaher. Joint optimization of computing and cooling energy: Analytic model and a machine room case study. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 396–405. IEEE, 2012.
- [100] Min Yeol Lim, Vincent W Freeh, and David K Lowenthal. Adaptive, transparent cpu scaling algorithms leveraging inter-node mpi communication regions. *Parallel Computing*, 37(10):667–683, 2011.
- [101] Robert Springer, David K Lowenthal, Barry Rountree, and Vincent W Freeh. Minimizing execution time in mpi programs on an energy-constrained, power-scalable cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 230–238. ACM, 2006.
- [102] S Huang and W Feng. Energy-efficient cluster computing via accurate workload characterization. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 68–75. IEEE Computer Society, 2009.