Feedback Guided Load Balancing in Distributed Memory Environments

---

Masters Dissertation

Robert Edward Starr
s1131111

MSc. High Performance Computing

EPCC
James Clerk Maxwell Building
The University of Edinburgh
Edinburgh
United Kingdom

Dissertation Supervisor:
Dr. J. Mark Bull

August 24, 2012

Acknowledgements

First and foremost I would like to thank God for the time, talents, and myriad of other opportunities I have been blessed with both before coming to this M.Sc. program, and during my year of studies in Edinburgh.

It would be remiss of me to not also mention a hearty thanks to Dr. J. Mark Bull, my project supervisor. His patience with me throughout the course of this project, as well as his guidance and help in understanding the nature of this project and the Feedback Guided Load Balancing model, have contributed immensely to the work that is in this final paper; I would not have successfully completed this dissertation without his guidance and help. He has been a wonderful mentor through the course of this dissertation and it has been a distinct pleasure to work for/with him over these last few months.

In addition, thanks are due to the Edinburgh Parallel Computing Centre (EPCC) at the University of Edinburgh and the excellent staff. From the lecturers who provided great knowledge and information about High Performance Computing, to the rest of the staff who have helped make our lives as students easier this year, they have all played a role that I am thankful for and probably do not fully appreciate. It has been a great experience doing my Masters degree under their tutelage and I highly doubt I could have gone anywhere else and received the quality of education provided by them. It was also great to work on the awesome supercomputer that *HECToR* is, which EPCC helps manage.

I would also like to thank my family, wonderful girlfriend Kelsey Lounsbury, and many great friends for all the support they have given me throughout this year and during this dissertation program. They have been instrumental in keeping me focused on the task at hand and ensuring that the doldrums did not take hold permanently during my studies. In addition, special thanks to my parents (Ken and Sharon Starr), my girlfriend, and two great friends (Anthony Anziano and Kaleb Boultinghouse) for agreeing to read through my dissertation draft and ensure there were no horrendous spelling or grammar issues.

I also owe a huge vote of thanks to my financial sponsors throughout the year: Rotary International, Mortar Board National Senior Honor Society, and the Scottish Government. Each played a vital role in allowing me to stay in Edinburgh for a year to pursue my Masters degree through their generous scholarships. The Rotary links especially have allowed me to meet some very gracious people this year who have helped support me throughout the year. I am especially thankful for Rotarian Arthur Thornton, his wife Gusia, and Rotarian Lyle Millage for all their kind words and help throughout the year. I only hope I have been as great a representation of the ideals of Rotary, Mortar Board, and the Scottish government as they themselves are.

In closing, special thanks to all the other students working on this M.Sc. in HPC who have provided close camaraderie throughout this program. It was fantastic working side by side with them all and sharing the experience. Best wishes in future pursuits!

Abstract

Any code that parallelizes computational work is required to come up with a method to load balance the work across the available processors. Most codes of this nature use some heuristic to decide how to perform this load balancing. In essence, they must have a (cheap) method of coming up with a number to act as a proxy for dividing up the actual computational work. For some applications that use these codes, that heuristic works very well; for others, the performance achieved is less optimal.

This project implements a dynamic load balancing algorithm for use in application problems executed on a distributed memory architecture. The idea is relatively simple: during simulation, record load data during computation time on each processor when each round of the program loop executes. Use the local load data and total load data to approximate the actual workload across each processor. Take that approximation against the total workload and redistribute the load across processors accordingly to achieve a better balance for the next run. Repeat the prior steps through every round of the program loop. If the workload is reasonable, this method converges to an optimal distribution of work, leading to a well load-balanced program execution.

This project implements and benchmarks such a dynamic load balancing algorithm in MPI to analyze the performance in a distributed memory scenario and the effect that such dynamic load balancing has. Benchmarking this algorithm involves using computational work for a variety of different workload distributions to see how the algorithm handles different imbalances in workload that occur during program initialization and/or execution. The computational work used in benchmarking can be obtained by either using real application data to simulate the computational work, or by writing test functions to fake real computational work. Given that obtaining real application data to fit all test cases is difficult, test methods are used to fill the scenarios to simulate.

These benchmarked results are then analyzed to determine the overall performance of the new algorithm given the overall increased availability of processors to execute over. In general, the results of the performance tests have shown that the new algorithm converges to an optimal solution in a minimal number of load balancing steps, dependent upon the load characteristics of the problem. In addition, the time taken for each load balance step, and the time for the total execution of the problem are reasonably low and do not vary much, even when increasing the total problem size or scaling over more processors.

Those results are important in showing the viability of the algorithm for use in real problem situations. The ability to scale over a larger number of processors without letting the increased communication negatively affect performance show the algorithm to be a possible replacement for existing shared and distributed memory dynamic load balancing techniques among certain classes of problems.

Contents

1 Introduction **1**

1.1 Load Balancing Models . . . 2
1.1.1 Diffusion Model . . . 2
1.1.2 Wave Propagation Model . . . 3
1.1.3 Feedback Guided Load Balancing . . . 4
1.2 Static vs. Dynamic Load Balancing . . . 4
1.3 Machine Architecture and its effect on Load Balancing . . . 6
1.4 Chapter Summary and Work to Follow . . . 7

2 Feedback Guided Load Balancing **9**

2.1 Mathematical Model of Feedback Guided Load Balancing . . . 10
2.2 Algorithm Implementation . . . 15
2.2.1 Language and Library Underpinnings . . . 15
2.2.2 MPI_Scan and MPI_Allreduce . . . 16
2.2.3 Convergence Check Implementation . . . 17
2.2.4 Load Balance Step Implementation . . . 19
2.2.5 Algorithm Complexity . . . 21
2.2.6 Issues Encountered during Development . . . 22
2.2.7 Current *FGLB* Algorithm vs. Algorithm for use with Real Applications . . . 23
2.3 Chapter Conclusion . . . 24

3 Algorithm Testing Environment **25**

3.1 *HECToR*: UK National Supercomputing Service . . . 25
3.1.1 Hardware Environment . . . 25
3.1.2 Software Environment . . . 28
3.2 Parameters File for *FGLB* Program Execution . . . 29
3.3 Synthetic Load Data for Simulation . . . 30
3.4 Functions Used to generate Synthetic Load Data . . . 31
3.4.1 Function 1: Linearly Increasing Load . . . 32
3.4.2 Function 2: Single-Sided Load . . . 32
3.4.3 Function 3: Sine Function-modeled Load . . . 33
3.4.4 Function 4: Sine Function Load with pseudo-random load spikes 34
3.4.5 Function 5: Sine Function-modeled Load for Data testing . . . 36
3.5 Chapter Conclusion . . . 37

4 Results & Performance Evaluation **39**
4.1 Processor Scaling Results 39
4.1.1 Quality of Load Balance Solution 40
4.1.2 Load Balance Steps to Convergence 42
4.1.3 Simulation Execution Time 43
4.1.4 Processor Scaling Conclusions 44
4.2 Iteration Scaling Results 44
4.2.1 Percentage Load Difference 44
4.2.2 Load Balance Steps to Convergence 45
4.2.3 Simulation Execution Time 46
4.2.4 Iteration Scaling Conclusions 46
4.3 Execution Time of the Load Balancing Step 47
4.4 Non-Convergent Load Balancing 48
4.5 Data Simulation 50
4.6 Summary of Results 52

5 Conclusions **55**
5.1 Possible Future Work 56

A Complete *FGLB* Algorithm *MPI* Pseudocode **63**

B Sample Makefile used on *HECToR* **67**

C Sample PBS Script used on *HECToR* **71**

D Tables of Raw Data from Output of Test Runs on *HECToR* **73**
D.1 Results from Processor Scaling Tests 73
D.2 Results from Iterations Scaling Tests 76
D.3 Results from Non-Convergent Load Balancing Tests 78
D.4 Results from Data Simulation Tests 79

List of Figures

2.1 Initial workload distribution over problem iteration domain 10
2.2 Calculating the new boundaries and workload for the problem iteration domain 11
2.3 Final workload distribution over problem iteration domain 13
2.4 Comparison of average time versus increasing number of processors for 10000 operations of MPI_Scan 17

2.5 Comparison of average time versus increasing number of processors for 10000 operations of MPI_Allreduce 18

3.1 Representation of *HECToR* interconnect network between compute cores 26
3.2 Representation of CC-NUMA memory hierarchy as present on *HECToR* 27
3.3 Representation of Gemini interconnect as present on *HECToR* 27
3.4 Simulated load profile of Function 1 over 100,000 functional iterations . 31
3.5 Simulated load profile of Function 2 over 100,000 functional iterations with 1,024 executing processors 32
3.6 Simulated load profile of Function 3 over 100,000 functional iterations . 33
3.7 Simulated load profile of Function 4 over 100,000 functional iterations . 34
3.8 Simulated load profile of Function 4 zoomed into iteration 0 - 200 to better see the load profile and random spikes 35
3.9 Simulated load profile of Function 5 for data testing over 5,000 functional iterations 36

4.1 Load difference vs. increasing number of processors over 500,000 iterations 40
4.2 Closer look at Function 1,3,4: Load difference vs. increasing number of processors 41
4.3 Load balance steps vs. increasing number of processors over 500,000 iterations 42
4.4 Simulation execution time vs. increasing number of processors over 500,000 iterations 43
4.5 Load difference vs. increasing number of iterations executed on 1,024 processors 45
4.6 Load balance steps vs. increasing number of iterations executed on 1,024 processors 46
4.7 Simulation execution time vs. increasing number of iterations executed on 1,024 processors 47
4.8 Load Balance step execution time vs. load balance step executed on 3,072 processors and 500,000 iterations 48
4.9 Percentage load difference vs. load balance step for 500,000 iterations executed on 8 & 16 processors 49
4.10 Percentage load difference vs. load balance step for 500,000 iterations executed on 32 & 64 processors 50
4.11 Simulation execution time vs. increasing size of data for 5,000 iterations executed on 256 processors 51
4.12 Simulation execution time vs. increasing size of data for 5,000 iterations executed on 1,024 processors 52

Listings

2.1 High level *FGLB* algorithm pseudocode 15
2.2 *loadBalance* method call tree 19
3.1 Example parameters dat file used to set simulation variables for the *FGLB* algorithm 30
A.1 Detailed *MPI* pseudocode: main function 63
A.2 Detailed *MPI* pseudocode: convergenceCheck and loadBalance functions 64
A.3 Detailed *MPI* pseudocode: computeNewBounds function 64
A.4 Detailed *MPI* pseudocode: exchangeBounds function 65
A.5 Detailed *MPI* pseudocode: exchangeData function 66
B.1 Makefile to compile *FGLB* algorithm on *HECToR* 67
C.1 PBS Script used to submit *FGLB* algorithm to backend of *HECToR* . . . 71

List of Tables

D.1 Results of Function 1 on 500,000 Iterations (Time in seconds) 73
D.2 Results of Function 2 on 500,000 Iterations (Time in seconds) 74
D.3 Results of Function 3 on 500,000 Iterations (Time in seconds) 74
D.4 Results of Function 4 on 500,000 Iterations (Time in seconds) 75
D.5 Results of Function 1 on 1,024 Procs (Time in seconds) 76
D.6 Results of Function 2 on 1,024 Procs (Time in seconds) 76
D.7 Results of Function 3 on 1,024 Procs (Time in seconds) 77
D.8 Results of Function 4 on 1,024 Procs (Time in seconds) 77
D.9 Results of Function 3 Load Difference vs. LB Iteration on 500,000 Iters 78
D.10 Results of Function 4 Load Difference vs. LB Iteration on 500,000 Iters 78
D.11 Results on 256 Processors and 5,000 Iterations (Time in seconds) 79
D.12 Results on 1,024 Processors and 5,000 Iterations (Time in seconds) . . . 79

Chapter 1

Introduction

High Performance Computing is used by a variety of fields ranging from bioinformatics and chemicals science to aerospace research and petroleum engineering. Each field uses HPC, and the Supercomputers that serve as the backbone for HPC, for differing reasons, but the fundamental principle linking them all together is the attempt to get either faster performance of their code, or the ability to run their code over a larger dataset than can be achieved with normal computing. Indeed, it is the striving to get better and better performance that is driving the current push towards the development of an Exascale machine and software that can be effectively parallelized on such a machine.

However, as with all problems in computing, there exists certain problems that cannot take best advantage of parallel performance in practice, sometimes yielding results that are worse than the serial counterparts, or just are not as parallel-efficient as they should be. For some problems, there is no easy fix for those issues, no obvious paradigms that can be used to increase the efficiency. But, for others, the technique of load balancing provides a way of increasing the efficiency and performance of the problem.

Load balancing is basically a means of allocating the "load" of a problem over the available resources in as even of a way as possible to get the best performance out of the problem. Dynamic load balancing carries this idea further by using some functional metric (such as run-time, data size, etc.) to measure the performance of a certain problem and then allocate that problem as evenly as possible over the available processors using that metric. This metric is measured during steps of the program execution to determine if the load balance is indeed getting more optimal and the program efficiency is increasing, as it should be.

The rest of this report looks at Feedback Guided Load Balancing, a type of dynamic load balancing that aims to decrease the number of load balancing iterations needed to achieve an optimal balance by dealing with piecewise constant approximations of the problem load, and then moving all of the determined non-optimal load to under loaded processors. This is determined using an algorithm that computes where the optimal load

boundaries should lay, and shifting the boundaries as needed. This particular approach is based on a method of feedback dynamic loop scheduling developed by Dr. J. Mark Bull ([4]).

The novelty of this approach is the fact that such an algorithm has not been used in practice on distributed memory machines, where the communications cost can sometimes overwhelm the benefits to be achieved by the load balance solution. The overarching purpose of this paper is to analyze the load balancing solution that Feedback Guided Load Balancing presents and whether the increased communications cost that comes from distributed memory scenarios is worth the increased scalability such machines allow for and whether the algorithm can indeed converge to optimal load balancing solutions over a reasonable number of iterations. First though, a few classifications of load balancing need to be looked at, specifically how they relate to the algorithm presented here.

1.1 Load Balancing Models

Load balancing algorithms follow some model that determines the overall approach to load balancing that is implemented in an algorithm for a specific architecture. The easiest, and most commonly implemented, model is the "Diffusion Model", which is looked at in more detail below. In addition, some attention is given to the "Wave Propagation Model", which was presented in Constantinos Christofi's 2011 EPCC Masters Dissertation titled "Feedback Guided Load Balancing in a Distributed Memory Environment" ([6]) before moving on to a short overview of Feedback Guided Load Balancing, the model this paper is focused on.

1.1.1 Diffusion Model

Diffusion methods allow for dynamic load balancing by getting data from nearest neighbors about their load and then shifting excess load off or receiving extra load if under loaded; the work shifted to each of its neighbors is proportional to the calculated load difference between them. Functionally, this occurs by measuring the load difference between any two processors and iterating over the problem until some convergence test is passed, usually the load difference being smaller than a specified value. The name of the model is taken from the view of the way the balancing occurs, effectively, the load "diffuses" through the processors until it is in a balanced state ([11] Pg. 209-210, [9] Pg. 24-25).

Diffusion methods are easy to implement because only nearest neighbor communications and an adequate convergence are needed to achieve a working diffusion based load balancing algorithm. In such algorithms, complexity goes to $O(1)$ in terms of communications cost, because that cost is fixed between neighbors during each iteration of the

problem. Communications tend to be the most computationally and memory intensive bits of a code, so having this cost fixed is a good argument for diffusion methods. However, there are still a few problems with diffusion methods. Firstly, such an algorithm will typically converge to an optimal balancing solution, but, due to the ability to only move data to the nearest neighbors, it can converge slowly over workloads to an optimal balance ([6] Pg. 6-8). Secondly, a highly imbalanced workload will require an even higher number of iterations, which, despite the low communications cost, would make the algorithm too expensive to use ([11] Pg. 209-210).

In ideal cases, diffusion methods yield a balanced load after a small number of load balancing steps. But, for cases where this does not occur, a better method of load balancing is needed. There are numerous enhancements to diffusion methods presented in various papers ([11], [13], [8]), including the "Wave Propagation Model" that will be looked at below, however, while most of the enhancements decrease the number of load balance steps needed to reach convergence, they are not new algorithmic approaches to the load balancing problem, which is needed to eliminate the slow convergence.

1.1.2 Wave Propagation Model

This model falls under the classification of a Diffusion Model, but with improvements aimed at decreasing the total number of iterations required to reach an optimal load balance. It is essentially a fast-start version of an equivalent diffusion algorithm, whereby more load is moved in the early load balancing steps so that an optimal load balance can be reached in less steps in the ideal case. A high precision timer is used to measure work as it progresses, then the load balance step is executed based on the time each processor required to execute its workload. Each processor communicates with its two nearest neighbors and either offloads or accepts more load depending on how loaded it is compared to the nearest neighboring processors. If a processor needs to offload data, it is offloaded evenly between the left and right neighbors so that these neighboring processors can then move the excess load on to their neighboring processors, etc. until an adequate balance is achieved.

To keep communication costs low, the implementation described in [6] uses the *SHMEM* library with the C programming language in a library called "DLBLib". The *SHMEM* library helps hide the overt communications between each processor on distributed memory machines and instead acts more like a shared memory multiprocessor with simple "puts" and "gets" for memory access. Using this underpinning, and keeping the communication to nearest neighbors only, helps maintain communication on order $O(1)$ like other diffusion algorithms ([6] Pg. 13-16).

The fast-start behavior of this algorithm does help it achieve an optimal load balance in fewer steps than a naïve implementation of the diffusion model under certain load conditions. However, in cases where the load is heavy on the ends and light in the middle, this implementation could take longer to reach an optimal balance than a naïve

implementation because of the localized communications that are used. This localization means balance is determined without a complete grasp of the overall load across the full scope of the problem. Also, given the way that load is moved to both the left and right neighbors, there are cases where some of the load that was just offloaded one load balancing step before, will be moved back to the processor that had offloaded it. This increases overall costs of the execution and affects the performance of the algorithm in general.

The “Wave Propagation Model” is a useful algorithm for certain classifications of load problems and an improvement over a simple implementation of the generic diffusion method, but there are useful performance benefits that can be gained by shifting away from the diffusion method entirely.

1.1.3 Feedback Guided Load Balancing

Feedback Guided Load Balancing (or *FGLB* for short) is a novel approach being applied to dynamic load balancing on distributed memory systems. Whereas the diffusion model slowly migrates load to nearest neighbors based on the calculated load difference between, this model uses two global communication steps to determine the total load over the problem, and the prefix sum for all loads to the left of the processors. These values are used to determine the optimal load boundaries for the problem, which are then communicated to each processor via point-to-point communication. This model is more expensive in terms of communication than diffusion methods; with the trade-off being that the number of steps needed to reach optimal balance is smaller for certain classifications of problems.

An expanded look at the mathematics behind FGLB and the implementation characteristics of the algorithm is provided in Chapter 2 “Feedback Guided Load Balancing”, while the viability tests and results are presented in Chapter 4 “Results & Performance Evaluation”.

1.2 Static vs. Dynamic Load Balancing

The load balancing models above make reference to “dynamic” load balancing. It is useful here to clarify the difference between such dynamic load balancing and its counterpart, static load balancing for future reference. There can easily be some confusion over what is meant by the terms “static” and “dynamic” load balancing since the terms are used to apply to two different applications within load balancing. The first context has to do with how load balancing determining is approached (which will henceforth be referred to as “static execution” and “dynamic execution”), while the second context has to do with whether the load changes during execution and how the algorithm responds to that (henceforth referred to as “static load” and “dynamic load”). Both contexts are

examined in more detail below.

In the first context, “static execution” is used to refer to an algorithm that determines the best load balance before a problem is executed and then runs that load balance solution until completion, with no changes to the load balance occurring during execution. In that sense, static execution load balancing is best used for problems whereby the computational requirements do not change during the measurement of the calculation and problem execution. Therefore, once some load is assigned to a processor, it will run there until all the execution is completed. This type of load balancing uses much less communication than dynamic execution solutions since the load is only determined once, before the start of execution, which means that the computational overhead is much less. However, because static execution load balancing needs the load information and communication behavior of the problem prior to its execution to achieve a good load balance solution, it cannot stand up to any changes in the load during problem execution and cannot shift the load if it is not optimally balanced ([12] Pg. 1-2, [24] Pg. 127).

“Dynamic execution” load balancing describes a load balancing algorithm that uses some metric to measure program execution during computation time, and that metric determines the best load balance for the given problem during the next iteration. The idea is that the individual fraction of the total load each processor needs becomes apparent during execution, and then it can be assigned; as computation progresses, differing processors will wind up with different amounts of work. So, rather than having a solution before execution that does not change at all during execution like the static execution method described above, this dynamic execution method allows the problem to dictate the load balance as the execution progresses. Though, this load balance method can take up a large percentage of overall computational time and negatively affect performance since there is a load balance step that needs to be executed each iteration. However, there are numerous cases by which it can be proven that the prior load estimates used by static execution methods will not achieve a most optimal load balance solution; in such cases, a dynamic execution load balancing solution is needed ([7] Pg. 279-280, [11] Pg. 209).

In the second context, “static load” load balancing refers to problems where the total load of the system does not change during or in between program execution. Whatever the load is at the start of the program execution remains fixed throughout. In such cases, it effectively models a problem in a closed system, whereby no data is either generated or lost during the work loop. Many classes of problems where load balancing is needed fall under this classification.

“Dynamic load” load balancing refers to the opposite, where the total load of the system can change during program situation. Depending on the problem being load balanced, it can either generate more load to be balanced or get rid of some of the load that was being balanced. For such problems, the “dynamic load” methods need to be able to

actively cope with any changes in the load across the entire problem. If the problem has many small changes during execution, that will require load balancing during each iteration, and it possible that such problems will never reach an optimal load balance due to the shifting load.

The algorithm described in Chapter 2 "Feedback Guided Load Balancing" deals entirely with cases where the total load remains fixed ("static load") and a functional metric is used to determine the most optimal load balance ("dynamic execution") throughout program execution. It was important, for the purposed of testing the *FGLB* algorithm, that the load not change during program execution, so that the performance of the algorithm in terms of determined load balance solution could be accurately measured and analyzed without having to deal with cases where the load was changing during execution, thus static load methods were used. It was also deemed an important focus for the *FGLB* algorithm to work with dynamic execution load balancing since it is more useful in a larger variety of applications that cannot determine an optimal load balance solution beforehand as static execution methods require.

1.3 Machine Architecture and its effect on Load Balancing

Another important facet relative to the model and algorithm that is presented and benchmarked in this report is machine architecture in relation to memory. High Performance Computing is marked by two main types of architecture in this sense: shared memory and distributed memory machines.

Shared memory machines are noted by every processor having access to all the available physical memory (RAM) for the purposes of computation and execution. This allows any processor within the machine to access any of the memory within the machine during execution, with specific software paradigms controlling actual access to ensure no memory is accidentally overwritten or race conditions exist. This makes shared memory machines ideal for load balancing algorithms, since an over-loaded processor only needs to tell an under-loaded processor where to shift execution to in order to achieve a better load balance; no data needs to be moved to the local memory of another processor because all of the memory is shared by all of the processors. However, a distinct problem with shared memory machines is the level of scalability. Shared memory machine architectures plateau at under approximately 100 processors and the same amount of memory (in GB). This limits their available performance for certain applications with larger data requirements ([17] Pg. 125-126).

Distributed memory machines are noted for each processor having some local physical memory, while the physical memory from other processors is separated. In order to access memory from other processors, explicit "messages" are sent between processors

whereby the data is read from the sending processor, sent to the receiving processor, and then written to the receiving processors local memory. This exchange is obviously more performance intensive and has a higher latency than shared memory “exchanges” because of the read, communication, and write aspect. However, distributed memory machines allow for more processor scaling by chaining these processors and their local memory sets together with some communications network. Yet, best performance for load balancing on such machines can only be obtained by direct management of the communication patterns, which means that the software is quite harder to develop ([7] Pg. 280-282, [18] Pg. 1-3).

There have been numerous dynamic load balancing algorithms developed for both shared and dynamic memory hierarchies, including examples shown in [7], [17], and [18]. The more recent hybrid approaches to machine architecture (notably CC-NUMA architectures) and Partitioned Global Address Space (PGAS) programming models have partially changed the approach to dynamic load balancing, however the fundamental trade-off remains communications cost vs. processor scalability. For that reason, *FGLB* focuses on dynamic load balancing on a distributed memory environment to see if the increased processor scalability afforded by such machines represents better performance in the face of increased communication costs.

1.4 Chapter Summary and Work to Follow

The introduction has carefully laid out the background of load balancing with some discussion of two other load balancing models. It has also provided some background of static vs. dynamic and shared memory vs. distributed memory load balancing. Feedback Guided Load Balancing is a dynamic load balancing model designed to run on distributed memory environments with aims towards decreasing the amount of communication needed for each load balancing step by dealing with optimal load boundaries. This approach is taken because of the impact that such a viable algorithm will have on dynamic load balancing techniques. Moving forward, in Chapter 2 “Feedback Guided Load Balancing” some further attention is given to the mathematical basis behind the model, as well as an overview of the implementation and algorithmic complexity. Then, in Chapter 3 “Algorithm Testing Environment”, the testing enviroment is explained while in Chapter 4 “Results & Performance Evaluation”, the performance results are highlighted and analyzed with a view of whether *FGLB* is indeed a viable alternative approach to dynamic load balancing. Finally, in Chapter 5, the conclusions of the research and this report are presented.

Chapter 2

Feedback Guided Load Balancing

As noted in the introduction, *FGLB* is a novel approach being applied to dynamic load balancing on distributed memory systems. Rather than following the status quo diffusion model and its variants that calculate load differences between nearest neighbors only, and thus take more steps to reach an optimal balance, the model takes an approach looking at the load from a perspective of its functional iteration boundaries and optimal boundaries derived from piecewise constant approximations of the load.

If the load is thought to lie within a set of iteration boundaries, then the load can be represented by some mathematical function approximating a curve. Each processor then owns and operates over a distinct subsection of those iterations to which the curve belongs; a usual approximation being total iterations / total processors. Taking each subsection as separate, a piecewise constant approximation of the load can then be taken for each processor so that each processor has a distinct fraction of the total load. Using this piecewise constant approximation, it then stands that there exists optimal iteration boundaries whereby these piecewise approximations can be equated between each processor to achieve an optimal load balance for the problem.

If the above model is implemented naïvely, then global communication will be needed to determine the total load between all processors. Each processor will then need to determine its fraction of the total load and communicate that fraction to all the other processors to determine which processors need to offload data, and which processors need to receive more load. Then the new boundaries will need to be communicated to each processor to ensure that the whole iteration space is assigned where it needs to be among all the processors. That naïve approach is very costly in terms of communication due to all the global communication that would need to take place; in essence on order $O(P^2)$ communications would need to be carried out, where P is the total number of processors. That behavior would severely limit scalability and make it unviable as an alternative to the diffusion model.

However, there is a better way to implement the above model to make it more al-

gorithmically efficient by making use of some mathematical properties related to the piecewise constant approximation of the load on each processor. That is highlighted in more detail below.

2.1 Mathematical Model of Feedback Guided Load Balancing

Coming at the problem from a more mathematically informed approach yields a better solution. This particular approach is based on a method of feedback dynamic loop scheduling developed by Dr. J. Mark Bull ([4]). Using the piecewise constant approximation of the problem load on each processor seen in the naïve approach above (a pictorial representation of such an approximation can be seen in Figure 2.1), certain definitions and statements can be made:

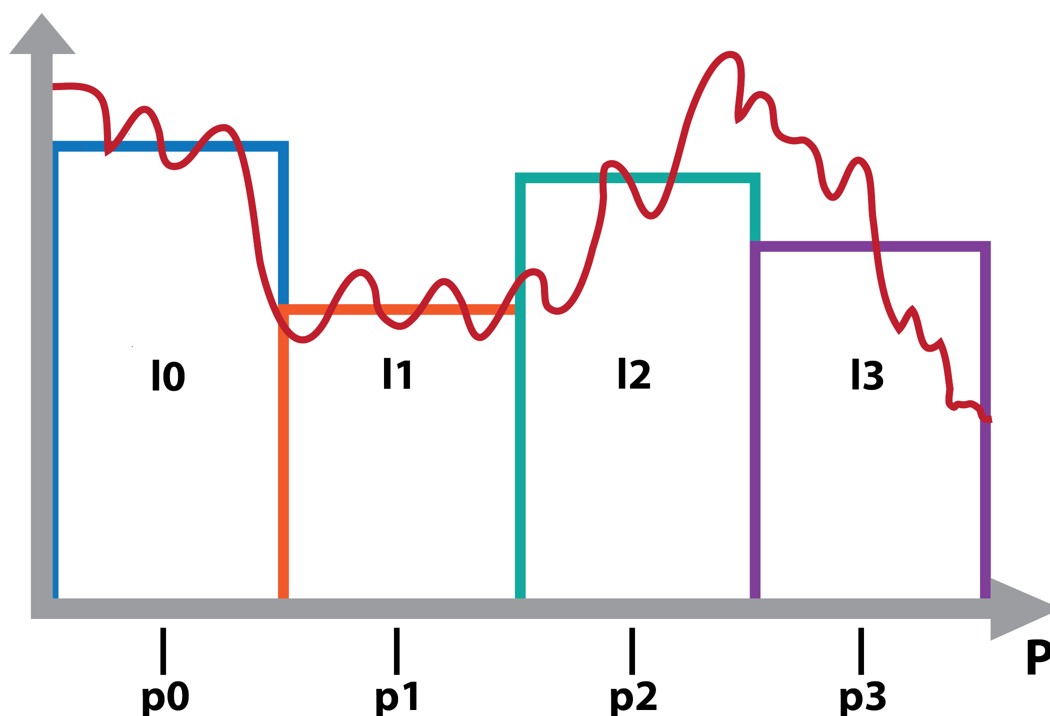

Figure 2.1: Initial workload distribution over problem iteration domain

Definition 1 Let P represent the total number of processors

Definition 2 Let L represent the total load over all processors

Definition 3 Let p_i, where i ranges from 0 to $P - 1$ processors, represent an individual processor

Definition 4 Let l_i represent the load on processor p_i

Definition 5 Let ${}_l b_i$ represent the lower iteration boundary that processor p_i has

Definition 6 Let ${}_h b_i$ represent the upper iteration boundary that processor p_i has

Statement 1 $\frac{L}{P}$ is equal to the optimal load for each processor

Statement 2 In an optimized solution, there are distinct values of k that correspond to each processor $p_i = p_k$ according to the relation $k\frac{L}{P}$

Statement 2 is derived from Statement 1 as it stands to reason that there are distinct boundary points for each processor based on the relation presented in Statement 1. For instance, p_0 has boundaries $0\frac{L}{P}$ and $1\frac{L}{P}$, p_1 has boundaries $1\frac{L}{P}$ and $2\frac{L}{P}$, etc. Thus, Statement 2 can be derived from Statement 1.

Using the above definitions and statements, an equation can be constructed to represent what values of k lie within the lower (${}_l b_i$) and upper (${}_h b_i$) boundaries of processor

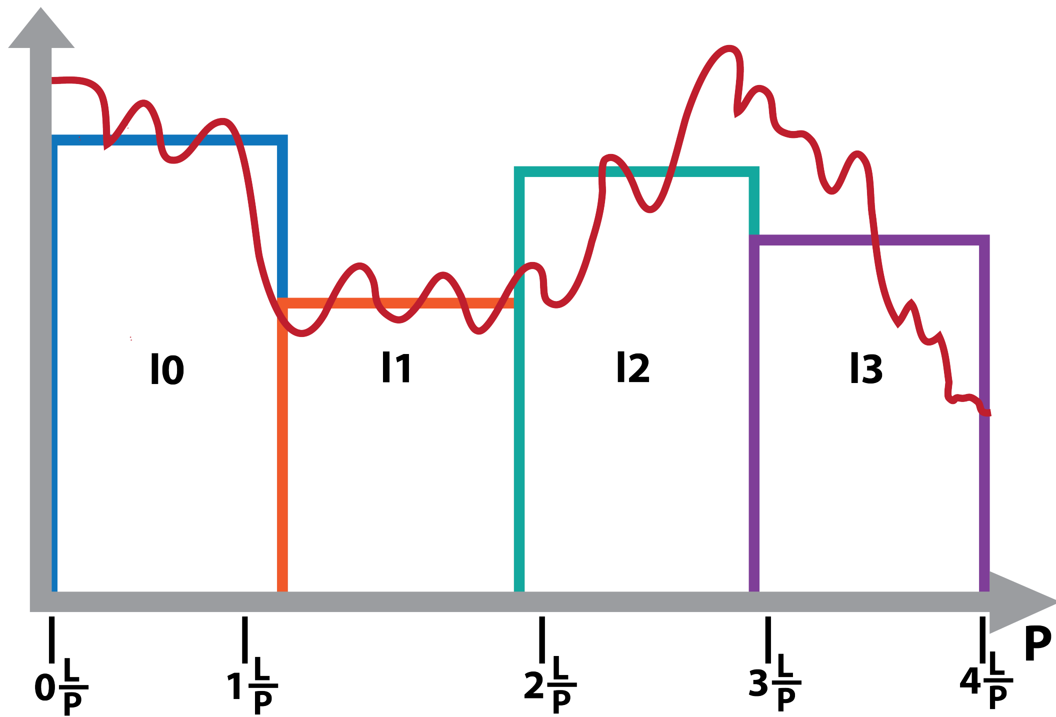

Figure 2.2: Calculating the new boundaries and workload for the problem iteration domain

p_i according to its local load, l_i. Equation (2.1) shows the mathematical relation of k to each processor p_i and its local load l_i. These k values dictate what processor should have that optimal boundary to achieve a more optimal load balance. A pictorial representation of the new boundary calculation is shown in Figure 2.2.

$$\forall p_j \in j = 0, \ldots, P-1 : \sum_{i=0}^{j-1} l_i < k\frac{L}{P} < \sum_{i=0}^{j} l_i \tag{2.1}$$

An aside here that becomes important during implementation is the fact that $\sum_{i=0}^{j} l_i$ is equivalent to a parallel scan of the local loads across all processors, so that p_0 has l for p_0 (l_i for $i = 0$); p_1 has l for p_0 and p_1 (l_i for $i = 0$,1); p_2 has l for p_0, p_1, and p_2 (l_i for $i = 0$,. . . ,2), etc. Also, $\sum_{i=0}^{j-1} l_i$ is simply $\sum_{i=0}^{j} l_i - l_j$.

Given that each processor can look at its current fraction of the total iterations and see what processors have optimal boundaries that lie in that fraction, it then becomes an algebraic exercise to determine what the actual iterations boundaries are for each of the k values corresponding to a processor. Since each workload is being approximated using a piecewise constant, new boundaries can be computed such that the low iteration boundary of p_k is equal to Equation (2.2), and the high iteration boundary of p_{k-1} is equal to Equation (2.3).

Statement 3 p_k has new ${}_lb_k$ according to Equation (2.2)

$${}_lb_k = \left\lceil \left(\left(k\frac{L}{P} - \sum_{i=0}^{k-1} l_i \right) / \left(\frac{\sum_{i=0}^{k} l_i}{{}_hb_i - {}_lb_i} \right) \right) + {}_lb_i \right\rceil \tag{2.2}$$

Statement 4 p_{k-1} has new ${}_hb_{k-1}$ according to Equation (2.3)

$${}_hb_{k-1} = \left\lfloor \left(\left(k\frac{L}{P} - \sum_{i=0}^{k-1} l_i \right) / \left(\frac{\sum_{i=0}^{k} l_i}{{}_hb_i - {}_lb_i} \right) \right) + {}_lb_i \right\rfloor \tag{2.3}$$

Each processor can use both equations to determine what processors have optimal boundaries within its current iteration space, as well as what those optimal boundary points are. Point-to-point communication can then be used to exchange boundaries among all processors so that each processor ends up with its new optimal boundaries. A pictorial representation of the new optimal load distribution is shown in Figure 2.3.

In addition to calculating and sending the new boundaries, any data associated with those boundary shifts will also need to be communicated as well. The question then becomes how to determine what data gets sent to what processor. Using Equation (2.1), the distinct k values can be determined and boundaries determined according to Statement 3 and Statement 4. For a processor p_i with iteration boundaries between ${}_lb_i$ and ${}_hb_i$, it can have between 0 and $P-1$ distinct k values within its iteration boundaries. A processor with 0 k values sends all of its data to one other processor. A processor

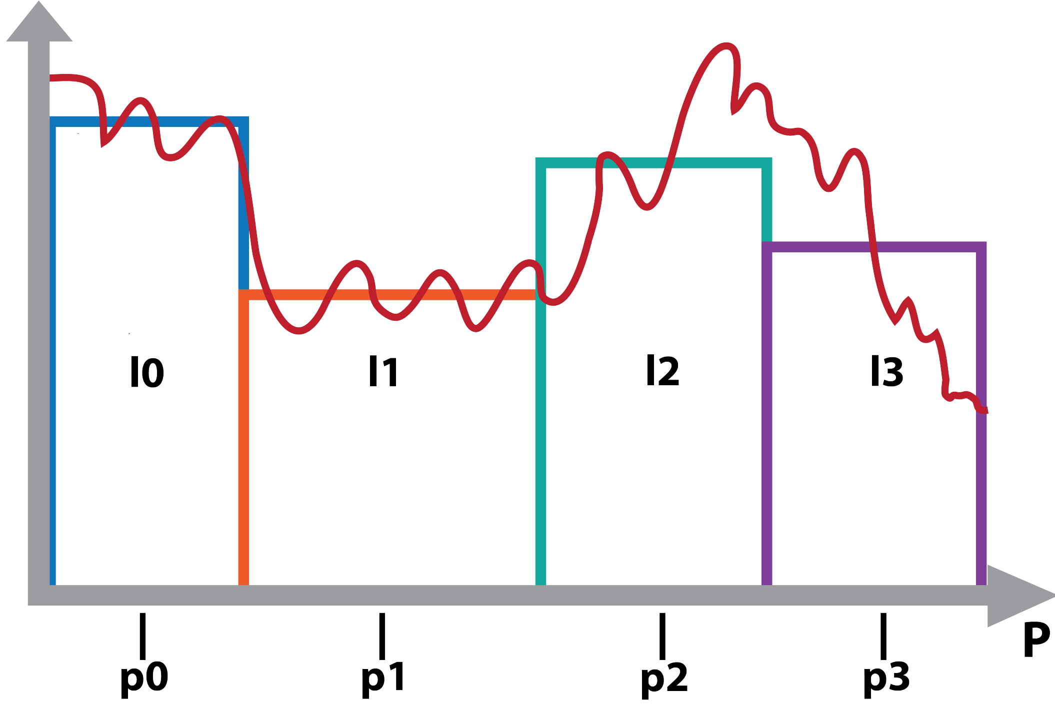

Figure 2.3: Final workload distribution over problem iteration domain

with 1 or more k values contains the optimal lower boundary point $({}_l b_k)$ for p_k according to Statement 3 and the optimal upper boundary point $({}_h b_{k-1})$ for p_{k-1} according to Statement 4. It can then be reasoned that every processor p_i needs to conduct a send for every k value and one additional send for the $k-1$ case in order to transmit all the data it currently has. In fact, the data communication step actually has three cases depending on the number of k values processor p_i currently has, as shown below:

Case 1 Processor p_i has 0 k values and needs to send data to 1 other processor

Case 2 Processor p_i has 1 k value and needs to send data to 2 other processors

Case 3 Processor p_i has 2 or more k values and needs to send data to $k+1$ other processors

With Case 1, the processor p_i has 0 k values, so it does not know what processor, p_k to send its data to. However, a pseudo k value (referred to as $\bar{k}$) can be determined according to Equation (2.4), which is simply a modified form of Equation (2.1) so that processor p_i knows what processor needs the data it is currently holding. After the $\bar{k}$ value is determined, processor p_i can send all of the data it is holding to $p_{\bar{k}}$.

$$p_{\bar{k}}, \bar{k} = \forall p_j \in j = 0, \ldots, P-1 : \left(\left(\sum_{i=0}^{j-1} l_i\right) / \left(\frac{L}{P}\right)\right) - 1 \qquad (2.4)$$

With Case 2, processor p_i has 1 k value between its lower and upper iteration boundaries ($_lb_i$ and $_hb_i$), and thus has the optimal lower boundary point ($_lb_k$) for p_k according to Statement 3 and the optimal upper boundary point ($_hb_{k-1}$) for p_{k-1} according to Statement 4. It can then be deduced that p_k receives new data associated with iterations between $_lb_k$ (the result of Equation (2.2)) and $_hb_i$, while p_{k-1} receives new data associated with iterations between $_lb_i$ and $_hb_{k-1}$ (the result of Equation (2.3)). The calculated data can then be sent by processor p_i to the two separate processors, p_k and p_{k-1}.

In the last case, Case 3, the calculated data to send closely follows the form of Case 2, but with some modification to allow the middle data to be sent to the right processor.

Definition 7 Let K represent the total number of k values processor p_i has

Definition 8 Let k_n, between 1 and K, represent each individual k value processor p_i has

For the two end cases, the calculated data follows the same pattern of results as Case 2. The lower end is not k_1 due to the fact that every processor p_i has to make $K + 1$ sends to get rid of all the data. The lower end is $k_1 - 1$ and $p_{k_1 - 1}$ (treated as k_0 and p_{k_0}), equivalent to p_{k-1} from Case 2; while the upper end is k_K and p_{k_K}, equivalent to p_k from Case 2. p_{k_0} receives data associated with iterations between $_lb_i$ and $_hb_{k_0}$ (using Equation (2.3) with appropriate k value substitution). Meanwhile, p_{k_K} receives data associated with iterations between $_lb_{k_K}$ (using Equation (2.2) with appropriate k value substitution) and $_hb_i$. That covers both end cases for a processor p_i with any number of k values 2 or greater and also takes care of the required $K + 1$ sends.

Then, for k_n for n between 1 and $K - 1$, the data to be sent can be calculated so that each processor p_{k_n} receives data associated with iterations between $_lb_{k_n}$ and $_hb_{k_n}$ (using Equation (2.2) and Equation (2.3) respectively with appropriate k value substitution). Those boundaries are, by nature, a subset of the total iterations of processor p_i between $_lb_i$ and $_hb_i$. That takes care of all the data that needs to be sent to all the processors p_k, that processor p_i has k values for.

Calculating the data to be moved and then ensuring that the correct data is sent to each processor p_k that needs it is more complicated than the calculations and sends needed to transmit the new optimal boundaries. But the three cases above cover all the cases that need to be worried when sending the iteration data about during use of the Feedback Guided Load Balancing model.

In the end, this model allows for more work to be done locally (the k and boundary calculations are done in place on each processor) with an eye towards minimizing the global communication of the algorithm. Such a minimization is the only way to make the algorithm feasible for implementation on distributed memory systems. Now that the mathematical basis for the model is established, an algorithm can be developed

and implemented that takes advantage of the communications minimization from the mathematical model.

2.2 Algorithm Implementation

There were some specific considerations to take into account when developing an implementation of the model into a specific algorithm that could be run on distributed memory machines. Most notably were which programming language to develop the source in and what library to use to facilitate communication between the processors on a distributed memory machine.

2.2.1 Language and Library Underpinnings

Since distributed memory machines are naturally suited to a "message-passing" model, the "Message Passing Interface" (*MPI*) library was a natural fit for the purpose of implementing this model. The *C* programming language has a history of great support with the (*MPI*) library, so that was also a natural fit to move forward with in development. But, before any actual development could take place, a high level view of the program flow needed to be established as a backbone to the implementation. Thus the pseudocode presented in Listing 2.1 "High level *FGLB* algorithm pseudocode" was developed for that purpose.

```
//Initialize Variables & MPI

//Establish initial data for functional work load

//Distribute problem iteration onto available processors with equal
//bounding to start            (total iterations / total processors)

while( load balance is not converged ){

    //Record load data per processor while work is occurring

    //Gather parallel prefix sums of local data on each processor
    //and the total load across all processors

    //Run loadBalance method to load balance the program
    loadBalance(){

        //Calculate new boundaries and what processor those
        //boundaries belong to based on k values, then send new
        //boundary data to the appropriate processor

        //Receive the new boundaries from the respective processor
```

```
24
            //Calculate any data that needs to be moved and send it to
26          //the needed processor based on k values

28          //Receive the sent data and assign it into the data array
        }
30  }

32  // Output any data and then finalize MPI and close the program
```

Listing 2.1: High level *FGLB* algorithm pseudocode

The pseudocode is mostly self-explanatory, but some care should be given to Line 12 and 21, which is where most of the communication cost comes from in the algorithm implementation. Line 12 needs a parallel scan in order to get the results needed for Equation (2.1) and an all-to-all communication to determine the total load over all the processors. Line 21 corresponds to the load balance step, which is where the k calculation takes place and the point-to-point communication to communicate the new iteration boundaries and any data that needs to be moved.

2.2.2 MPI_Scan and MPI_Allreduce

One thing the pseudocode brought to the forefront was the importance of using parallel prefix algorithms to achieve the result needed from Equation (2.1). Thus, MPI_Scan and MPI_Allreduce became the linchpins to an efficient implementation of the algorithm. If the performance requirements of both were not efficient over larger scaling of processors and iteration space, then the entire algorithm would be rendered obsolete in comparison to diffusion methods. Further research needed to be done on parallel prefix algorithms ([2, 10, 20, 21]) and the performance characteristics of MPI_Scan ([14, 19, 25]) to ensure that would not be the case for this implementation.

One important consideration with regard to prefix algorithms is that they are limited by the number of individual elements that each processor has to compute; having to operate over a greater number of elements than existing processors tend to have worse performance than operating an equal number of elements and processors and not tend to $\log(n)$ behavior in execution ([10] Pg. 35). However, for this algorithm, each processor is only dealing with a single value equal to the total of the local load that it is operating over, therefore n remains equal to the value of P. So that issue would not present itself in this algorithm implementation since performance will still tend to $O(\log(n))$ = $O(\log(P))$ in terms of communication.

To ensure that mathematical relationship held during functional execution, some performance tests of MPI_Scan and MPI_Allreduce were carried out on *HECToR*, a distributed memory machine that was used for algorithm performance testing as well (more on *HECToR* hardware and software in Chapter 3 “Algorithm Testing Environment”). As

can be seen in Figure 2.4 for MPI_Scan and Figure 2.5 for MPI_Allreduce, the performance of each closely follows behavior consistent with that of $\log(P)$. Both tests were conducted using an average time from 10,000 calculations of the respective algorithm. Each calculation either Scanned or Allreduced over a single integer value initialized to the processors rank ranging from a minimal 2 processors up to 256 processors. This allowed for a sense of the scaling of each over an increasing number of processors, which the graphs show that there is better consistent performance with increasing number of processors according to the $O(\log(P))$ behavior.

It was therefore reasonable to conclude that both MPI_Scan and MPI_Allreduce were suitable methods to use in implementing this algorithm in the manner presented in the

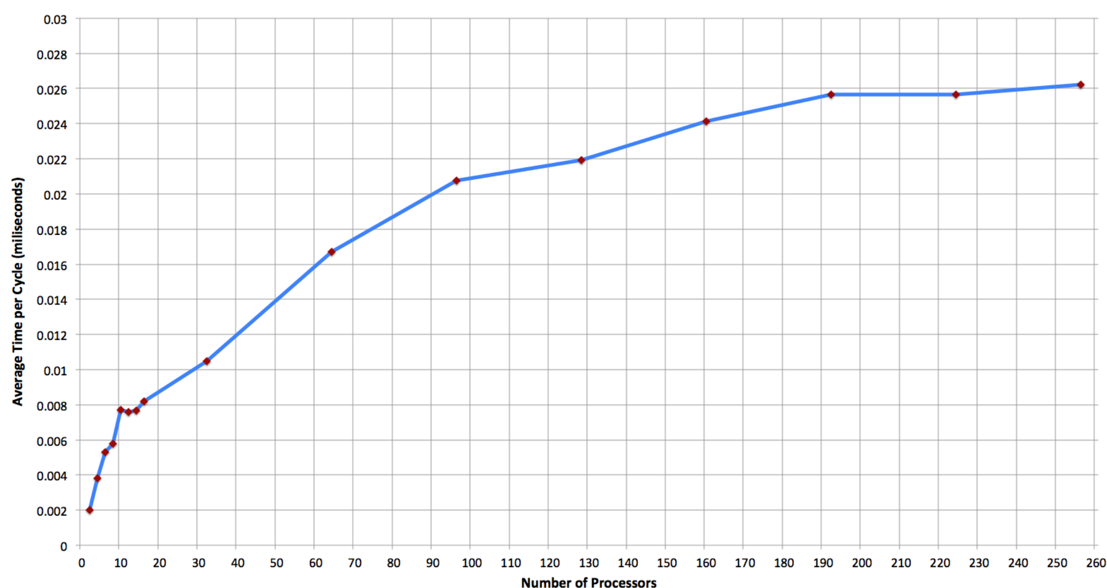

Figure 2.4: Comparison of average time versus increasing number of processors for 10000 operations of MPI_Scan

pseudocode. Using this parallel scan and the mathematical model developed above, the communications complexity drops from $O(P)$ in the naïve algorithm to $O(\log(P))$ in general, which makes it more feasible as an alternative to the Diffusion algorithm. Communications cost and convergence rate still end up being the trade off between this model and diffusion models however, which Chapter 4 “Results & Performance Evaluation” analyzes.

2.2.3 Convergence Check Implementation

An important consideration for any load balancing algorithm is how to handle convergence testing and when to stop load balancing. In the best case, load balancing would stop when each processor has a load equal to the optimal load per processor, $\frac{L}{P}$ as shown in Statement 1. In most executions however, it is not possible for every processor (or

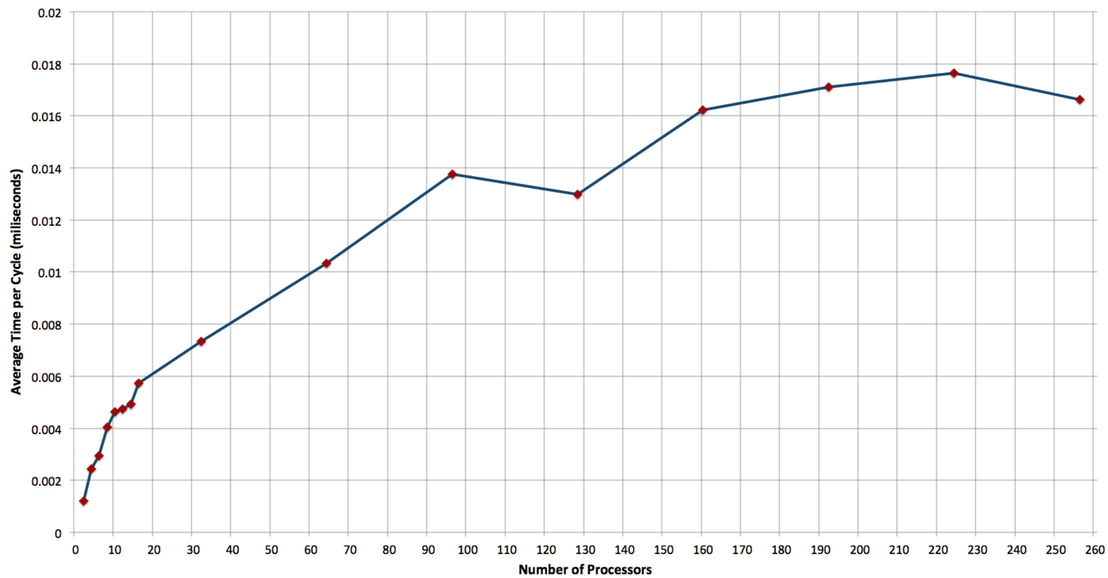

Figure 2.5: Comparison of average time versus increasing number of processors for 10000 operations of MPI_Allreduce

any processor for that matter) to reach that optimal value, which leads back to the question of when is it best to stop load balancing; is the trade off between further load balancing iterations and getting closer to the optimal value worth it?

For this implementation, a simple metric was used to determine when to stop load balancing. Using MPI_Allreduce and the "MPI_MAX" operation, the value corresponding to the load value from the processor with the maximum load of all processors is returned. A convergence value is then calculated according to Equation (2.5). That convergence value is then compared to the convergence value that was calculated during the last load balance loop. If the values are equal or if the new convergence value is worse than the prior convergence value, then load balancing stops, otherwise load balance continues in the next loop.

$$\text{convergence value} = max_{i=0,P-1}(l_i) - \frac{L}{P} \tag{2.5}$$

The reason this convergence check was chosen for the *FGLB* algorithm was because it is proportional to the execution time as it relates to the difference between the optimal load balance and the current load balance of the problem. $\frac{L}{P}$ represents the optimal load that each processor should have, and therefore the optimal time that the algorithm would take to run if the system were perfectly load balanced. $max_{i=0,P-1}(l_i)$ represents the worse balanced processor within the system, which is the processor that should take the longest time to execute, and is thus the performance bottleneck. The difference between the two represent how close the current load balance is to the optimal load

balance, and is thus a good, simple convergence check for the *FGLB* algorithm.

This simple convergence metric does a fair job of allowing the load balance to continue until a most optimum balance is reached. Yet there are some cases where this metric has a false positive and stops load balancing too early (as seen in the section "Non-Convergent Load Balancing" within Chapter 4) or continues load balancing even though only nominal gains towards the optimal balance are being made. In such cases, a more optimized convergence check could be implemented to minimalize those occurrences (such work is discussed in the "Possible Future Work" section within Chapter 5). But, for the purposes of algorithm benchmarking and analysis as presented in this paper, the above implemented convergence check performs adequately.

2.2.4 Load Balance Step Implementation

It is useful here to go into a little more depth about the implementation details of the fundamental method for this algorithm, the *loadBalance* method. The relation between the *loadBalance* method and the rest of the pseudocode can be seen on Line 16 of the High level *FGLB* algorithm pseudocode. The method obviously follows the mathematical model laid out above, but the implementation has some specific differences required to ensure that *FGLB* could run in an *MPI* implementation. A call tree for the method is presented in Listing 2.2 to highlight what methods exist within the *loadBalance* method, as those methods will be discussed in more detail below.

```
    //Run loadBalance method to load balance the program
 2  loadBalance(){
        //Calculate new boundaries and what processor those
 4      //boundaries belong to based on k values
        computeNewBounds()
 6
        //Send new boundary data to the appropriate processor
 8      //Receive the new boundaries from the respective processor
        exchangeBounds()
10
        //Take care of any data that needs to be moved
12      loadBalanceData(){

14          //Calculate data that needs to be moved based on k values
            computeNewData()
16
            //Send data the needed processor based on k values
18          //Receive the sent data and assign it in data array
            exchangeData()
20      }
    }
```

Listing 2.2: *loadBalance* method call tree

As the "*loadBalance* method call tree" shows, the first method within *loadBalance* is the *computeNewBounds* method which functionally implements Equation (2.1) so that each processor can determine what optimal iteration boundaries lie within its current iteration fraction. The result from the MPI_Scan completed before moving into the *loadBalance* method is used to determine the k values by execution of a *C for* loop operating according to Equation (2.6) (which is simply Equation (2.1) rewritten). Each k value that falls in between those two values is stored and used to calculate the new optimal boundaries for each k value according to Equation (2.2) and (2.3). There are cases where the last processor ($p =$ commSize $- 1$ in *MPI* terms) calculates a k value higher than the number of processors currently in execution. That is illegal and must be handled so that that k value is not stored, which would cause a fault later in execution.

$$\frac{\sum_{i=0}^{j-1} l_i}{L/P} < k < \frac{\sum_{i=0}^{j} l_i}{L/P} \tag{2.6}$$

Within this method there also has to be special handling of cases where a processor does not have a k value within its current iteration fraction such that it assigns all of its iterations to p_k where k is equal to Equation (2.4), for the purpose of moving data later in the *loadBalance* method. The purpose of that equation was discussed further earlier in the chapter while discussing the mathematical basis for the Feedback Guided Load Balancing model.

The *exchangeBounds* method is then used to communicate the new boundaries calculated to each processor (k value) determined. Since a processor could be sending a number of messages ranging anywhere from 0 to $P - 1$, non-blocking *MPI* sends needed to be used. Once the processors that do have optimal boundaries within their current iterations send those values, all the processors can then receive the values. Receiving processors do not know what processor is sending their new boundaries, so each processor must use *MPI_Status* values to keep track of the sending processors. In addition, tags must be used to keep the new lower and upper boundary messages separate from each other. Lastly, $p =$ Rank 0 does not receive a lower boundary and $p =$ Rank commSize $- 1$ does not receive an upper boundary because they lie on the extreme ends of the total number of work iterations for a given problem. So care needs to be taken to ensure they do not post *MPI_Recv*s for their respective end points and that their *MPI_Status* values reflect themselves for their respective lower and upper boundaries.

The last method to look at is the *loadBalanceData* method, which was not actually used during most of the simulation testing, as it was not needed for all the cases being tested. Some results from simulations including it were needed though to show the viability of the *FGLB* algorithm in a real problem scenario. The method contains two sub methods, one to calculate the amount of data to send to each processor and one to actually send that data. In both cases, the sending processor does a number of sends equal to $K + 1$ since each optimal boundary point requires some data to be sent on

each side of the boundary according, which was discussed before when talking about the mathematical model of Feedback Guided Load Balancing. After the amount of data to send is calculated and sent using non-blocking *MPI* sends, and then received by the receiving processor, the actual data can be sent also using non-blocking *MPI* sends. Then it becomes a trivial matter to copy the received data into the receiving processors data array.

The *loadBalance* method is the workhorse of the entire *FGLB* algorithm implementation and is where most of the time is spent during execution. Despite some special cases that needed to be handled to ensure proper operation, the algorithm closely follows the flow of execution described by the mathematical model and the pseudocode presented above and stands as a good implementation of Feedback Guided Load Balancing.

2.2.5 Algorithm Complexity

Though both MPI_Scan and MPI_Allreduce are both used heavily in the algorithm and form the backbone of it, that is not the only communication that takes place and effects performance of the algorithm. It is beneficial to do a more thorough analysis of the model as a whole and the complexity as it relates to performance characteristics. Below are listed the main communications that occur during algorithm execution and the complexity as it relates to execution time:

- MPI_Scan to gather parallel sums of local load data - $O(\log(P))$
- MPI_Allreduce to determine total load of problem - $O(\log(P))$
- MPI_Allreduce to determine maximum load of from all processors - $O(\log(P))$
- Point-to-Point communication to send new lower boundaries - in the worst case this tends to $O(P)$ if a single processor has all the boundaries within its own load, in the best case it tends to $O(1)$ since each processor only has to send one message
- Point-to-Point communication to send new upper boundaries - in the worst case this tends to $O(P)$ if a single processor has all the boundaries within its own load, in the best case it tends to $O(1)$ since each processor only has to send one message
- Point-to-Point communication to send data size that will be sent - in the worst case this tends to $O(P)$ if a single processor has all the boundaries within its own load, in the best case it tends to $O(1)$ since each processor only has to send one message
- Point-to-Point communication to send new data - in the worst case this tends to $O(P-1)$ if a single processor has all the boundaries within its own load, in the best case it tends to $O(1)$ since each processor only has to send one message

Though the method required to move new data is not utilized in all the tests of the algorithm, it is important to a real application using the algorithm and is thus included in the complexity listed above. Therefore, in the worst case, overall algorithm complexity tends toward $O(4P + 3\log(P))$ in detail, though in true "Big-O" notation the algorithm has a communications complexity on order $O(P)$ per iteration of the load balancing algorithm since the $\log(P)$ terms are miniscule compares to the P terms from point-to-point communications. In the best case, the overall algorithm complexity tends toward $O(4 + 3\log(P))$ in detail; in true "Big-O" notation complexity on order $O(\log(P))$.

The worst case occurs if all the load starts on a single processor since that processor then has to handle all of the communication. This behavior is mostly seen at the start of the load balancing execution in cases where the load is poorly balanced. As execution progresses however, this poor load balance situation should gradually moves towards a more optimal load balance, which then means the complexity of the algorithm gets better. Thus, the *FGLB* algorithm should actually gets closer to the best case scenario as execution progresses.

Even in the worst case scenario, the algorithm has a complexity on order $O(P)$, which is equivalent to the naïve implementation of the Feedback Guided Load Balancing model, as is to be expected. However, since the complexity decreases towards the best case scenario as execution progresses, the true complexity is on order $O(\log(P))$, which is much better behavior than the worst case scenario. This algorithm complexity is not as good as most diffusion methods that tend to $O(1)$, however, taking into account the fact more processors on a given problem is likely to help decrease the number of load balance iterations needed to achieve an optimal balance means that the *FGLB* algorithm could still have equal or better performance than diffusion methods. Thus the results presented in Chapter 4 "Results & Performance Evaluation" become very important to determining the long-term viability of this algorithm as a replacement for current diffusion methods on distributed memory machines.

2.2.6 Issues Encountered during Development

The development from a mathematical model to a working implementation did not always progress smoothly and there were some problems encountered along the way, especially when trying to implement the data transfer portion of the implementation consisting of three functions: *loadBalanceData*, *computeData*, and *exchangeData*.

The biggest issue in regards to the data transfer, which was unable to be solved during the course of this implementation, is that the program will produce a segmentation fault if the size of data being modeled gets too large. For a simulation, the user sets the number of problem iterations to operate over as well as the number of integers (data size) that each iteration should get as a means of simulating real data associated

with problem iterations. If the number of problem iterations were large (over 10,000 iterations), then the program would produce a segmentation fault if the data size got too large. The same thing occurred if the number of processors was increased with a medium number of iterations and data size. A trade-off between the number of problem iterations and the data size was thus present in the code. Despite best efforts, the reason for the segmentation fault could not be traced, which limited the simulation with data in the results. It seems that the fault might be some sort of memory limitation related to the hardware that the program was running on, but that is unproven thus far.

Another problem that manifested itself in the course of development was the manner in which the load data was being stored. Since the code uses synthetic data (which is discussed further in section "Synthetic Load Data for Simulation" within Chapter 3) to simulate the load, rather than determining it from whatever problem is being load balanced, that load data needs to be saved per problem iteration and in between each load balance step. To handle that initially, an array of integers was set up to hold the synthetic load data during the load balance step. It became quickly apparent that, in order to scale up to approximately 1,000,000 or more problem iterations, that array would need to be changed to *C long* type, otherwise there would be overflow issues when determining the local load on a processors and total load over all processors. Once that change was made however, there were no overflow issues and the program ran as expected.

Despite the issues highlighted above, most of the development proceeded as well as can be expected for any software development project. Little bugs and issues cropped up here and there, but were all dealt with as needed to ensure a working implementation was finalized whereby performance testing could be completed.

2.2.7 Current *FGLB* Algorithm vs. Algorithm for use with Real Applications

The *FGLB* algorithm as it currently stands is a good implementation of the mathematical model presented earlier in the chapter. But, the algorithm currently, is really designed as a simulation of the Feedback Guided Load Balancing model. Any further development of the code for use with real applications and real application data would require some changes to the code, especially in the *problemInterface* methods. In addition, the *parameters* method and means of holding "problem iterations" and "problem data" (both problems which were highlighted above in the "Issues Encountered during Development" section) would not be needed because that would all be taken from whatever real application is being load balanced.

2.3 Chapter Conclusion

This chapter has provided an overview of the mathematical model for Feedback Guided Load Balancing as well as some background information on the *MPI* implementation for distributed memory machines. As the “Algorithm Complexity” section above shows, the complexity of the algorithms tends to $O(P)$, which gives an initial indication that the proposed algorithm may be more efficient than current diffusion methods since more load can be moved per iteration, potentially leading to faster convergence, while keeping the communications costs low. In the next chapter, Chapter 3 “Algorithm Testing Environment” the testing environment is discussed before moving on to Chapter 4 “Results & Performance Evaluation” and a discussion of the performance results obtained from this implementation of the *FGLB* model.

Chapter 3

Algorithm Testing Environment

Now that the mathematical model for Feedback Guided Load Balancing and an algorithmic background has been established, some description of the testing environment is necessary. This chapter will first discuss the hardware and software behind *HECToR*, the UK National Supercomputing Service and distributed memory machine that served as the test bed for the *FGLB* algorithm, then move into an overview of the specific testing techniques employed within the algorithm and the means of producing synthetic load data to test the algorithm on.

3.1 *HECToR*: UK National Supercomputing Service

HECToR (High End Computing Terascale Resource) is the UK National Supercomputing Service funded by various UK Research councils (notably the Biotechnology and Biological Sciences Research Council (BBSRC), Engineering and Physical Sciences Research Council [EPSRC], and Natural Environment Research Council [NERC]) and managed by the Edinburgh Parallel Computing Centre (EPCC) based at the University of Edinburgh in Scotland. The machine is used by a variety of organizations and departments on projects ranging from chemistry and materials science to genetic modeling and weather forecasting. Currently ranked 32 in the June 2012 list of top 500 supercomputers in the world ([22]), *HECToR* has provided consistent performance over its years of operation and remains a vital research tool for High Performance Computing users and researchers across the United Kingdom and Europe ([16]).

3.1.1 Hardware Environment

HECToR began as a base Cray XT4 system with dual core nodes in late 2007, going through various hardware upgrades and improvements to reach the now "Phase 3" upgrade that was completed in December 2011. It is now a Cray XE6 system spread over 30 cabinets with a total of 704 compute blades. Each compute blade consists of

4 dual-nodes, with each dual-node containing two 16-core, 2.3 GHz *AMD* Interlagos Opteron processors operating over the same shared memory. This gives *HECToR* a total of 90,112 processing cores and peak theoretical performance of approximately 1 petaflop ([16]).

In addition to the processor capacity, each dual-node also has 32GB of main memory (RAM) that both 16-core processors can access. Meaning that each processor core has approximately 1GB of dedicated memory assigned to it when all the processors on a compute node are in use. The memory hierarchy within a node follows a shared memory architecture (all 32 cores sharing all 32GB of memory), while in between nodes it follows the standard distributed memory architecture. Figure 3.1 (image taken from [3] Pg. 32) shows a pictorial representation of the relationship between the "shared memory" style compute nodes and the system interconnect used to connect the nodes together into a further "distributed memory" machine (P for processors, M for mem-

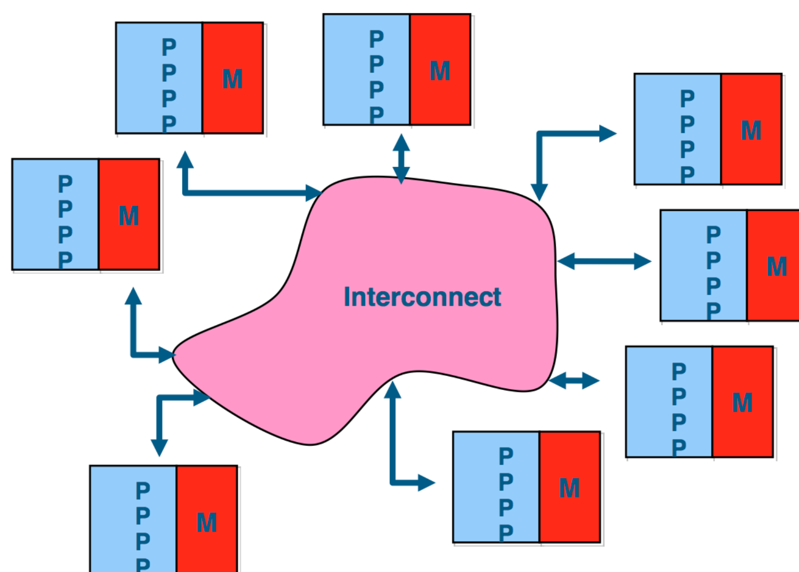

Figure 3.1: Representation of *HECToR* interconnect network between compute cores

ory). This leads to the possibility of running hybrid codes to capitalize on the best of both programming paradigms, but, for the purposes of testing the *FGLB* algorithm, it is viewed simply as a distributed memory machine and the "message-passing" model of parallel programming is used ([3]).

Another important feature on *HECToR* is the manner of memory implementation within a compute node. In a compute node, the shared memory falls under a type of non-uniform memory access (CC-NUMA) whereby there are a series of buses connecting individual cores on the chip to their successive levels of cache and eventually to the actual physical memory (RAM). Thus certain cores are closer to each other in terms of memory access than other cores within the same compute node. This has consequences when a problem is being run on nodes that need to access certain data that a neighboring

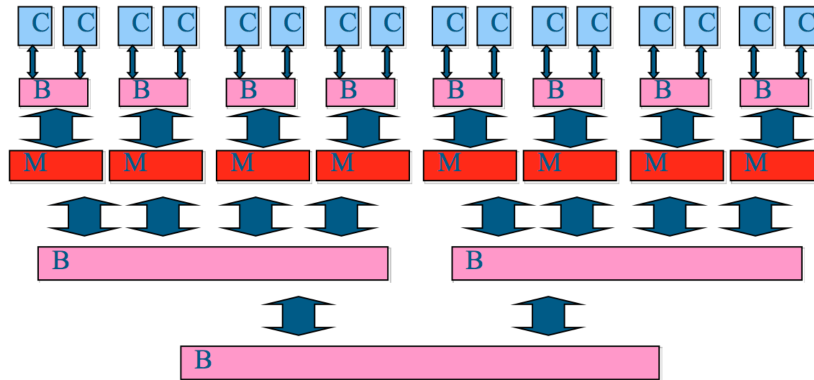

Figure 3.2: Representation of CC-NUMA memory hierarchy as present on *HECToR*

processor has locality towards because it will take longer to access data from a core farther away from the calling processor due to this bus hierarchy. It therefore becomes immensely important to design software that keeps data locality in mind to minimize the access times for that data. A pictorial representation of a CC-NUMA memory hierarchy is presented in Figure 3.2 (C for cores, B for bus, M for memory) ([3] Pg. 33).

It is also necessary to mention a bit about the interconnect that *HECToR* has installed. Interconnects serve as the communications backbone for any supercomputer or machine used for HPC. In the earliest supercomputers, interconnects were simply built out of commodity networking routers and switches. But, as the need for increased band-

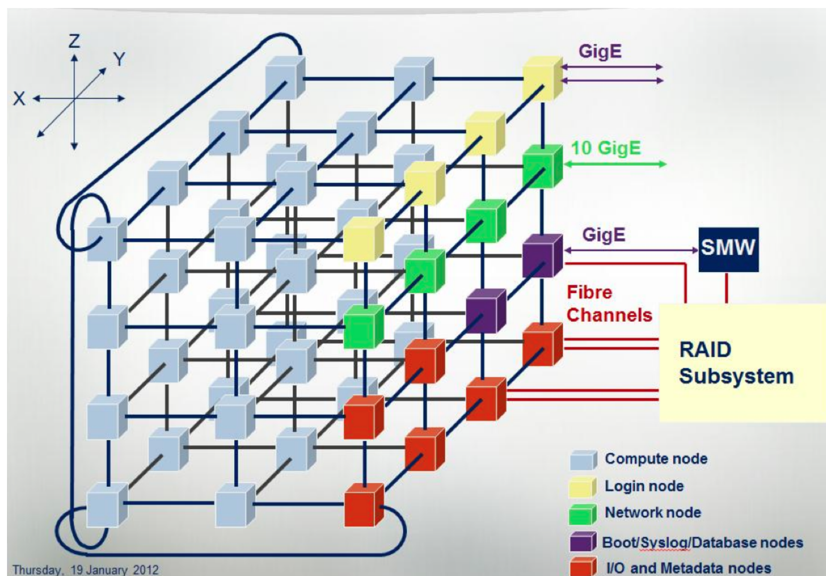

Figure 3.3: Representation of Gemini interconnect as present on *HECToR*

width and decreased latency grew, manufacturers designed specialty interconnects that focused purely on throughput between nodes. Since “Phase 2”, *HECToR* has had the Cray Gemini Interconnect, which is Crays latest specialty interconnect for its supercomputers. Gemini provides 3 times improvement in latency and 100 times improvement in

throughput over its predecessor interconnect with the ability to scale to approximately 1,000,000 or more cores. This high performance interconnect allows *HECToR* the ability to facilitate faster computing and memory access over a larger number of cores than its prior versions, which comes in quite handy for benchmarking such algorithms as *FGLB*. The Gemini interconnect as it functions in *HECToR* can be seen in Figure 3.3 ([3] Pg. 15,17).

Lastly, emphasis should also be placed on the file system that *HECToR* implements. The Lustre file system is implemented as a parallel file system to allow more concurrent access and better integration with parallel codes to allow for faster access to disk in order to reduce the bottleneck that hard disk access is normally for such HPC systems. Though Lustre is equally hardware and software innovation and implementation, it is the physical hardware on which data is read and written that concerns the user and Lustre provides a means of supporting larger and faster execution of parallel codes through its parallel file system implementation. Though the *FGLB* algorithm benchmarked and analyzed in this report does not deal with disk access much, in a *FGLB* implementation used with real application data disk access would be hugely important ([3]).

Based on the information provided above, it can be seen that *HECToR* provides the necessary hardware support that is needed for the testing of an algorithm such as *FGLB*. The hardware implemented and maintained on the machine provide a perfect test bed to put the *FGLB* algorithm through the paces and determine whether the algorithm implementation is truly suited to be a replacement to existing diffusion methods.

3.1.2 Software Environment

Similar to the hardware environment for *HECToR*, the software environment is also configured and optimized for best parallel performance for HPC programs. Besides the software that is used to facilitate the Lustre parallel file system mentioned above, there is also the operating system of the machine, compilers, development tools, and library support software to consider. *HECToR* is a large machine used by varied agencies, and thus must have varied support for many development platforms that users may require.

First and foremost is the operating system used. Given the bent towards optimal parallel performance of the machine, it stands to reason that no commodity operating system would satisfy the requirements. Indeed, *HECToR* uses the Cray Linux Environment (CLE, and version 4.0 specifically on the current phase), which is essentially a version of SuSe Linux stripped down and focused on parallel performance by reducing memory usage within the system and other general operating system effects. *HECToR* is split between “compute nodes” and “service nodes”. The service nodes that average users have access to run a full copy of SuSe Linux to provide all the necessary tool support. The compute nodes on the other hand, the essential backend of the system, run the stripped down version to increase their overall performance. This OS dichotomy allows the system of scale to approximately 500,000 or more nodes while still maintaining and

enhancing overall parallel performance ([3] Pg. 7-9).

HECToR also supports a couple of compiler suites including the default Cray compilers, as well as the performance-optimized *PGI* compilers and the widely-used GNU compilers. Wrappers to the generic compilers are supplied (for instance *cc* corresponds to *C* compilers and *CC* corresponds to *C++* compilers). Access to the specific compiler is controlling via modules and need to be "loaded" in order to be changed. For this project, the default Cray compilers were used as it was deemed to provide the best optimized support on Cray hardware. Makefiles, via the generic *Make* application were used to actually compile the *FGLB* algorithm into a proper executable. The Makefile used for *HECToR* is provided in Listing B.1 in appendix chapter B ([3] Pg. 21-24).

Access to the backend nodes for the purposes of running parallel jobs is controlled via the "Parallel Batch System", PBS for short. PBS is an environment used to handle scheduling and execution of parallel jobs on HPC machines. The idea is to have the backend system isolated from the user while still being able to implement their requests automatically. For the *HECToR* system, any jobs to run need to be moved into the "work" directory for that user, and then scripts used to submit the executable to be run. The scripts have to follow a specified format (an example of which is seen in Listing C.1 in appendix chapter C) that determines the number of processors to run the executable over, the amount of time to give it, what budget to access, what executable to run (via the "aprun" command), etc. Scripts are then submitted using the "qsub" command, which then accesses the PBS scheduler to determine the queue needed based on the number of processors and amount of time requested. This allows access to the backend of the system while keeping it protected and separate from user meddling and interaction ([3] Pg. 25-31).

HECToR also has support for a number of different utilities and tools designed to aid the user and programmer when debugging and developing their parallel codes. These tools were not used during the development of the *FGLB* algorithm, but the overall software support on *HECToR* was very useful to the development and testing of the algorithm.

3.2 Parameters File for *FGLB* Program Execution

It was a necessity that the *FGLB* be tested over varying load situations, number of executing processors, and number of functional problem iterations. These could have simply been hard-coded into the source code and then changed as needed and recompiled to get the specific results. But a more elegant solution was to implement a method to read in the needed information from some text file. Therefore, a parameters struct and method were implemented in *C* whereby all the needed variables for the simulation could be read in from a dat file. This allows the simulation to be modified numerous times without recompiling the code, which is very helpful on a machine like *HECToR*

where backend access is so limited.

For the *FGLB* implementation, four variables are needed to setup the simulation successfully:

1 The function used to simulate load data (*FunctionSelection*, ranging from 1 to 7 and discussed in more detail below)

2 Functional iteration boundaries to run simulation over (*IterationBoundaries*, ranging from 1 to maximum *C* integer)

3 Whether to run with data simulated or not (*DataSimulation*, 0 for no or 1 for yes)

4 Number of integers per iteration when simulating data (*DataSimSize*, ranging from 0 to maximum *C* integer)

These values can be changed as needed to accommodate any needed characteristics while running the *FGLB* simulation. An example of a parameter file used in this project is presented in Listing 3.1.

```
FunctionSelection 1
IterationBoundaries 500000
DataSimulation 1
DataSimSize 10
```

Listing 3.1: Example parameters dat file used to set simulation variables for the *FGLB* algorithm

3.3 Synthetic Load Data for Simulation

A matter of key importance to testing the *FGLB* algorithm was what means to gather the problem data to test the algorithm on. Real application data is always preferred, since it would then give an indication of performance in relation to a real application. However, the problem is getting real application data that models the situations that need to be tested for the performance assessment of the code. Another problem with real application data is that making changes to the data profile on such data becomes a lot harder, meaning if the need to alter the data profile in any way to better model a situation ever arose; there would be no easy way to do that.

Therefore, it was decided that simulating the load data would be the best approach to take. Using various functions that corresponded to various load profiles (using the parameters file mentioned above), load values can be returned for every functional problem iteration according to the dictating function. These load values are then used in

place of whatever functional metric would be used in real application codes for the purpose of load balancing. This allows for easier control of the data profile for the purposes of *FGLB* load balancing and performance testing.

Definition 9 Let m represent an individual iteration within the problem

Definition 10 Let M represent the total number of iterations in the problem

Definition 11 Let l_m, between 0 and $M-1$, represent the load on an iteration

3.4 Functions Used to generate Synthetic Load Data

The functions used to generate the synthetic load data needed to cover a few different scenarios for the purposes for performance testing of the *FGLB* algorithm to develop a complete picture of the behavior of the algorithm with difference extreme scenarios. To that end, four different functions were designed and tested to see how the algorithm behaved in the difference scenarios. The different functions and an overview of the load profile generated by each is presented in more detail below while the results from the tests with each function are presented in sections “Processor Scaling Results” and “Iteration Scaling Results” of Chapter 4.

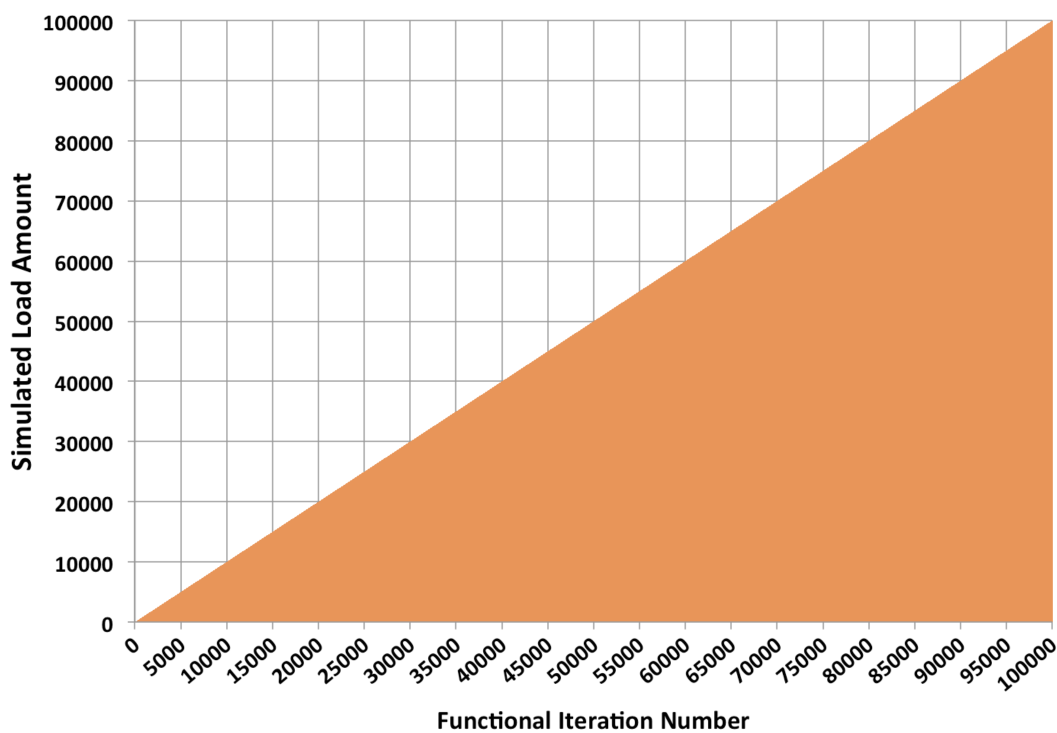

Figure 3.4: Simulated load profile of Function 1 over 100,000 functional iterations

3.4.1 Function 1: Linearly Increasing Load

The first, and most easily implemented, function was based on a linearly increasing load fixed to be proportional with the growing number of functional iterations. For each iteration, the load data was simply equivalent to the number of the iteration that it was being determined for ($l_m = m$). That equates to the $y = x$ function in mathematical graph notation. A visual representation of this function is presented in Figure 3.4 over 100,000 functional iterations. Function 1 was designed to test how the algorithm executes over linearly increasing load amounts and to ensure that adequate balancing was being achieved on what should be an "easy" load balancing problem for such an algorithm. In a perfectly balanced situation, each successive executing processor will have a smaller fraction of the total number of functional iterations due to the linearly increasing load.

3.4.2 Function 2: Single-Sided Load

Another important metric for the *FGLB* algorithm was how it performs when all the load is on a single processor. Function 2 was designed to simulate load data that only existed on the first processor (Rank 0 in *MPI* terms), while all other processors started

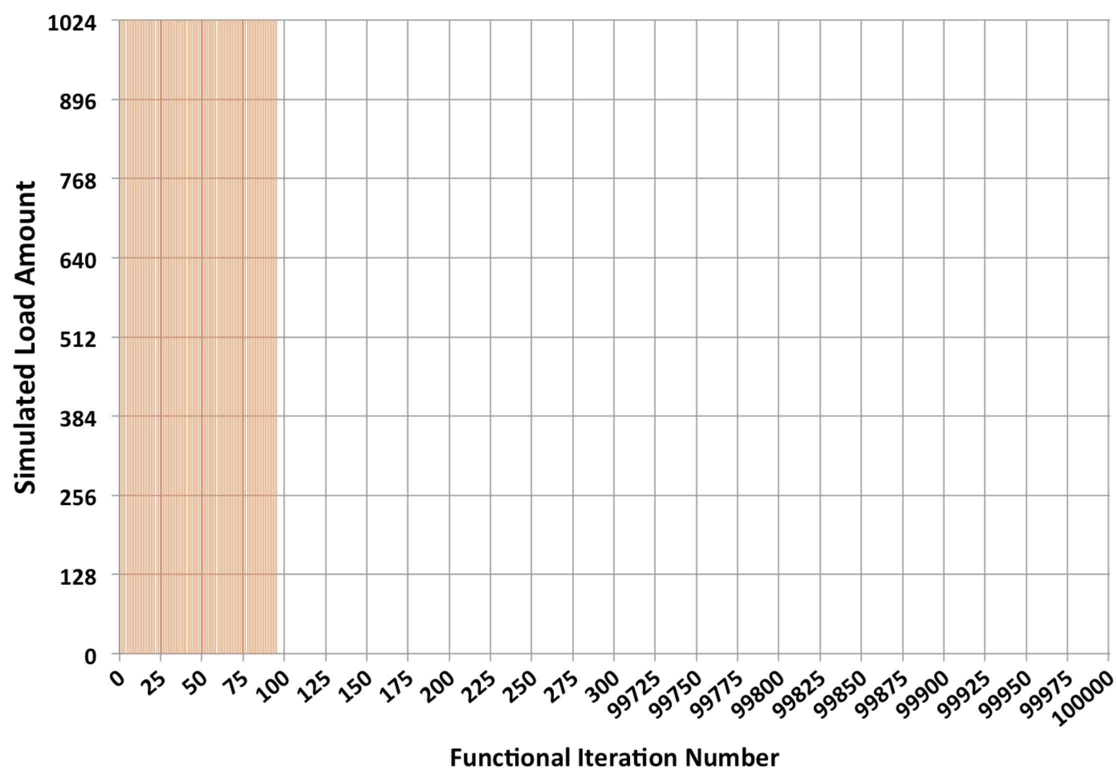

Figure 3.5: Simulated load profile of Function 2 over 100,000 functional iterations with 1,024 executing processors

with loads equal to 0. This was obviously dependent upon the total number of functional iterations as well. For example, with 100,000 iterations and 1,024 executing processors, each processor starts with $\frac{100{,}000}{1{,}024} = \approx 98$ iterations. Therefore Rank 0 would have load data (with a value instantiated to the total number of processors, $l_m = P$) on iterations 0 through 97. This example is visually represented in Figure 3.5 (note that it jumps from iteration 300 to iteration 99,700 to better see the load profile). If the number of functional iterations or the number of executing processors were changed, then the number of iterations with load data in them would also change. Function 2 was designed to test whether the *FGLB* algorithm could effectively deal with situations where all the load started on a minimum number of processors, a situation which usually causes problems with diffusion methods due to the number of load balancing iterations needed to eventually move all of that load from one processor down to all the others.

3.4.3 Function 3: Sine Function-modeled Load

It was also important to see how the algorithm would operate over a "reasonable" workload that had higher and lower periods of load over its functional iterations. To model that sort of behavior, a *sine* function was employed according to Equation (3.1). For this Function (and that equation) each functional iteration number is treated as a degree and the returned value of the *sine* of that degree was the simulated load value.

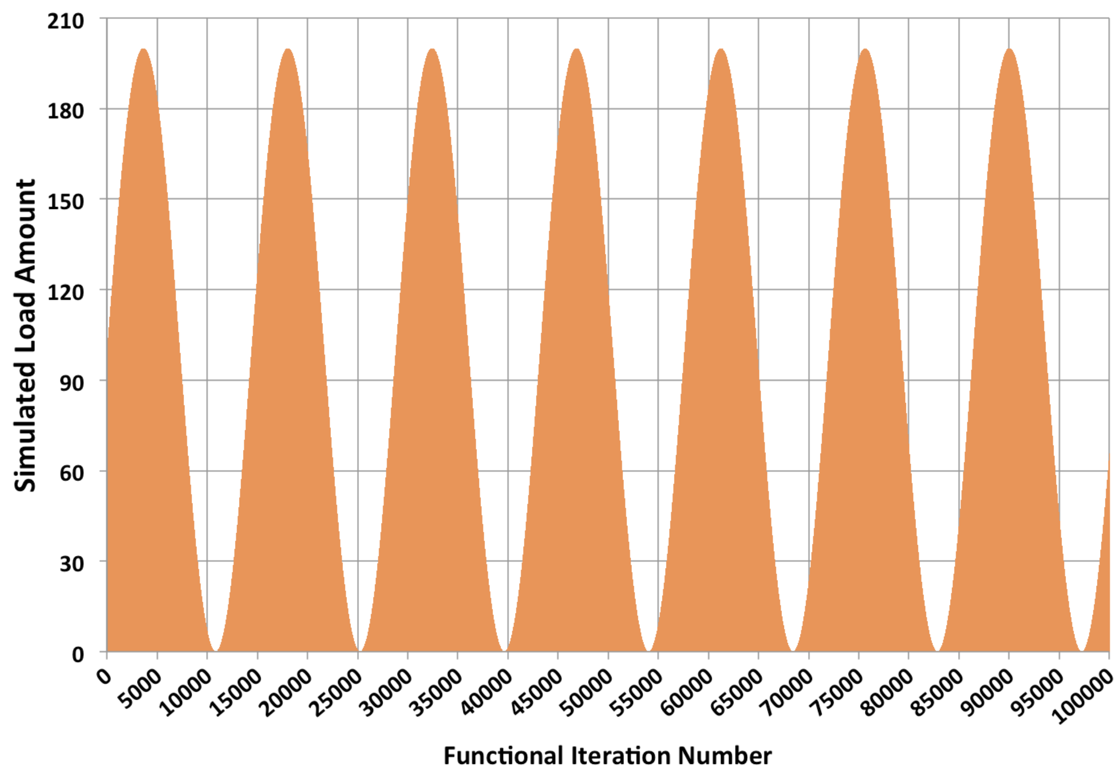

Figure 3.6: Simulated load profile of Function 3 over 100,000 functional iterations

$$l_m = \left(100 \cdot \sin\left(\frac{m \cdot \pi}{180} \cdot \frac{1}{40}\right)\right) + 100 \tag{3.1}$$

However, as the equation shows, the period of the function needed to be lengthened (to $\frac{1}{40}$, or one cycle of the sine wave every 14,400 degrees, equal to every 14,400 iterations) in order to provide a better load profile over increasing number of iterations. In addition, the amplitude of the *sine* function was increased one hundred fold and altered shifted up by 100 units as well. Therefore, the synthetic load values (l_m) fall between 0 and 200 as the function runs its course. These changes allow for a load profile that scales well over increasing number of iterations and gives enough simulated load for the *FGLB* algorithm to conduct efficient load balancing activities. A visual representation of Function 3 over 100,000 iterations is presented in Figure 3.6.

3.4.4 Function 4: Sine Function Load with pseudo-random load spikes

A function also needed to be designed to test how the *FGLB* algorithm would handle random fluctuations in load data that some real applications can have. To that end the

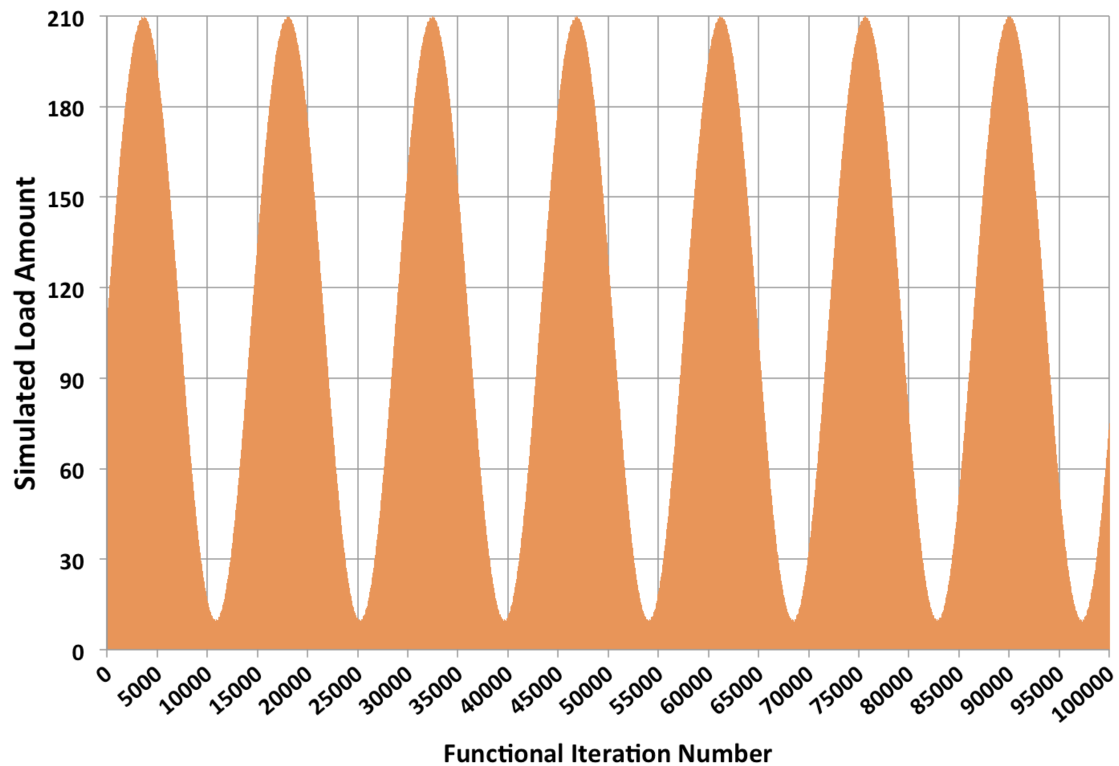

Figure 3.7: Simulated load profile of Function 4 over 100,000 functional iterations

sine function from Function 3 was employed with a pseudo-random generation of extra load data. Two 5 digit prime numbers ($a = 11003$, $b = 10007$) were used with each functional iteration number to generate a “random” number (r) falling between 0 and 1 according to Equation (3.2).

$$r = \frac{(m \cdot a) \mod b}{b} \tag{3.2}$$

If r was lesser than 0.50, then the load value would simply be equal to Equation (3.1). If r was 0.50 or greater, then the load value would be equal to Equation (3.1) + Equation (3.3).

$$r = \frac{(m \cdot a) \mod b}{{}^{b}/_{10}} \tag{3.3}$$

This meant, on a macro level, that load values for Function 4 would be between 10 and 210 depending on which iteration was being looked at, as can be seen in Figure 3.7 for 100,000 iterations.

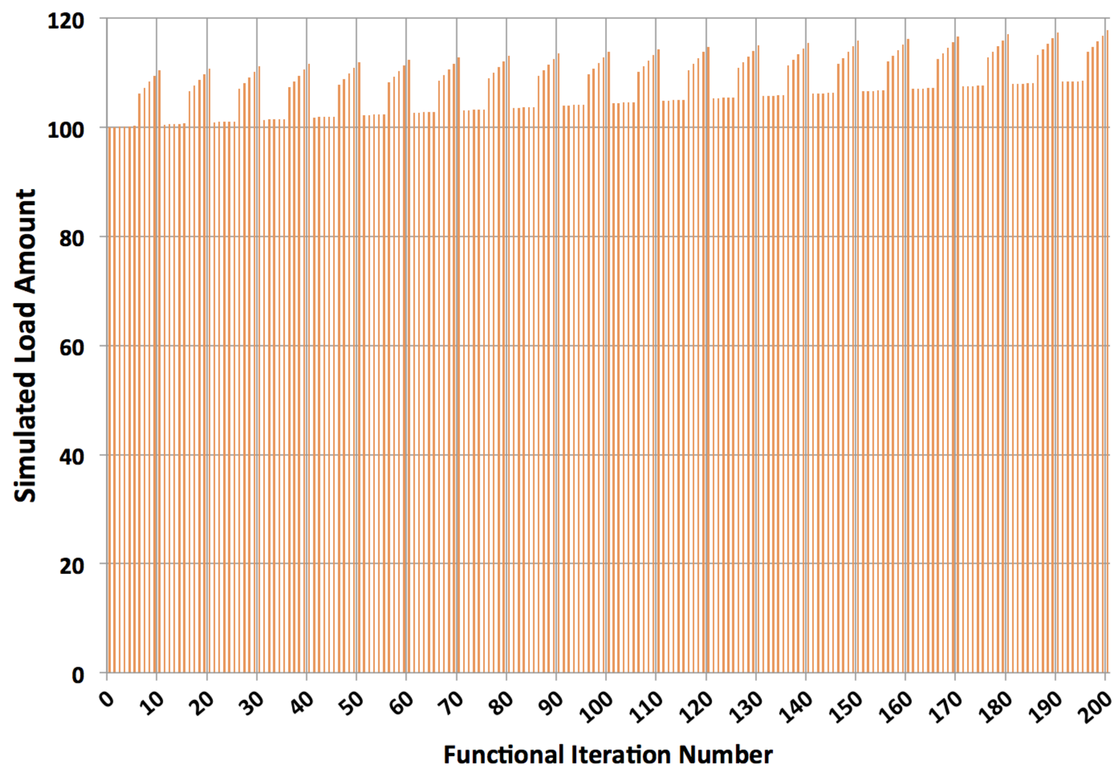

Figure 3.8: Simulated load profile of Function 4 zoomed into iteration 0 - 200 to better see the load profile and random spikes

On the surface it seems as if Function 4 is simply Function 3 with an altered amplitude, but, upon closer inspection of the load profile over the functional iterations (as can be in Figure 3.8 for iterations 0 to 200 within 100,000), a distinct pattern of increased load every five iterations can be detected following the natural curve of the *sine* function from Function 3. Over 100,000 iterations, that pattern blends into the background, but, for load balancing algorithms, still needs to be handled to ensure an optimal balance. Due to the small changes that occur frequently between the different iterations, it can be quite tough for some load balancing programs to converge to an optimal solution. Thus it is a good problem to test the *FGLB* algorithm against.

3.4.5 Function 5: Sine Function-modeled Load for Data testing

Due to the segmentation faults that were occurring when running higher number of iterations while the data simulation was enabled (explained in more detail in section "Issues Encountered during Development" of Chapter 2), Function 3 had to be altered into a new Function 5 to allow for testing of the data simulation within the *FGLB* algorithm. Equation (3.4) shows the new function used for the data simulation, which is almost exactly the same as Function 3, but the period is now shorter ($\frac{1}{30}$ instead of $\frac{1}{40}$).

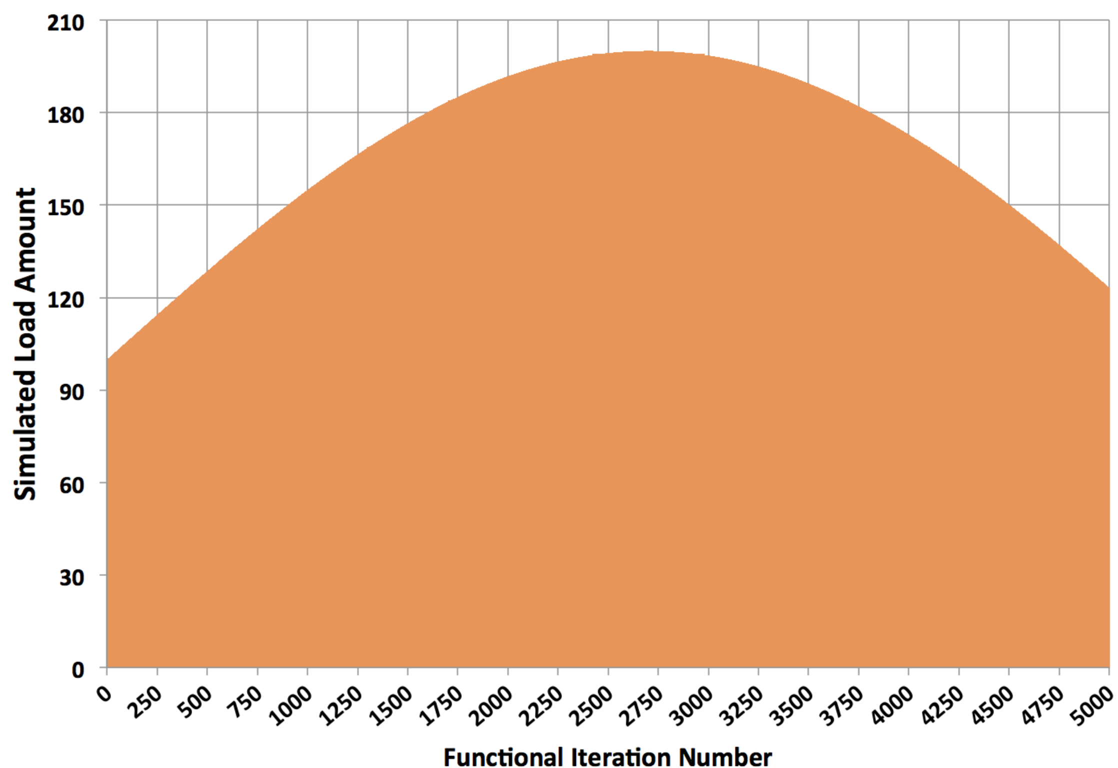

Figure 3.9: Simulated load profile of Function 5 for data testing over 5,000 functional iterations

$$l_m = \left(100 \cdot \sin\left(\frac{m \cdot \pi}{180} \cdot \frac{1}{30}\right)\right) + 100 \tag{3.4}$$

In addition the number of total functional iterations was restricted to 5,000. This gives the load profile a much different look than Function 3 over 100,000 iterations (the load profile of Function 5 can be seen in Figure 3.9). That allowed for testing of the *FGLB* with data simulation enabled to proceed without any further segmentation faults or other issues. The results from those tests can be seen in section "Data Simulation" of Chapter 4.

3.5 Chapter Conclusion

Now that the functions used to generate the synthetic load data have been explained, and the background information about the hardware and software of *HECToR* has been highlighted, discussion of the results from each test and the overall performance of the *FGLB* algorithm can be carried out in the next chapter, Chapter 4 "Results & Performance Evaluation".

Chapter 4

Results & Performance Evaluation

Now that all the background about the mathematical model, algorithm implementation, *HECToR* machine specifics, and means of testing the algorithm have been discussed, the results of the tests can be presented and analyzed. As stated at the outset of this paper, the fundamental question behind this algorithm implementation on a distributed memory system is whether the increased cost of communication is worth the trade-off for potential faster convergence than traditional diffusion methods. It is with that view that the data to follow is presented and analyzed, as well as to determine the overall performance of the algorithm.

4.1 Processor Scaling Results

One of the most important aspects of this algorithm was the ability to scale over an increasing number of processors and ensure that doing so did not negatively affect the quality of the load balancing solution. Distributed memory machines offer an advantage over shared memory machines in that they can scale to a larger number of processors more easily. It was thus important to see whether the *FGLB* algorithm could also scale well with an increasing number of processors as the mathematical model indicated it could. To accurately test processor scaling, each function from section "Functions Used to generate Synthetic Load Data" of Chapter 3 was executed on 500,000 functional iterations ranging from 64 to 4,096 executing processors. Less than 64 processors was deemed to not be worth exploring since that would then fall under the realm of shared memory machines and the maximum processor scaling that can be achieved with such machines. For each run of the simulation, the program was allowed to run until convergence was reached, or until 25 total load balancing iterations occurred.

4.1.1 Quality of Load Balance Solution

The results from this test are related to how the quality of the load balance solution changes when the number of executing processors changes. If increasing the number of processors causes a worse load balance, then the algorithm is not suitable for us. The metric used to determined how well balanced a particular problem was is the same metric that the convergence test used, Equation (2.5). The difference between the maximum load on all processors and the optimal load of the problem provide a good metric of how well balanced a particular solution is, which is the same reason that was employed as the convergence test for the algorithm. Obviously, the smaller the difference between the max and optimal load values, the better the final load balance solution is.

From Figure 4.1, we see that all functions except Function 2, converged to a very small difference between the max load and the optimal balance. Function 2 is the single sided synthetic load data; for 500,000 iterations, it ranges between 122 iterations with load data ($\frac{500,000}{4,096}$) to 7,812 iterations ($\frac{500,000}{64}$). As the number of processors increases, there are less iterations with load data, which leads to the issue of too many executing processors for the amount of load data available to work over. Thus the load balance gets progressively worse over the increasing number of processors. At its best (64 processors and 7,812 iterations with data), Function 2 has a load difference of 1.2×10^{-4}, which is not as good as the algorithm is able to do with the other functions.

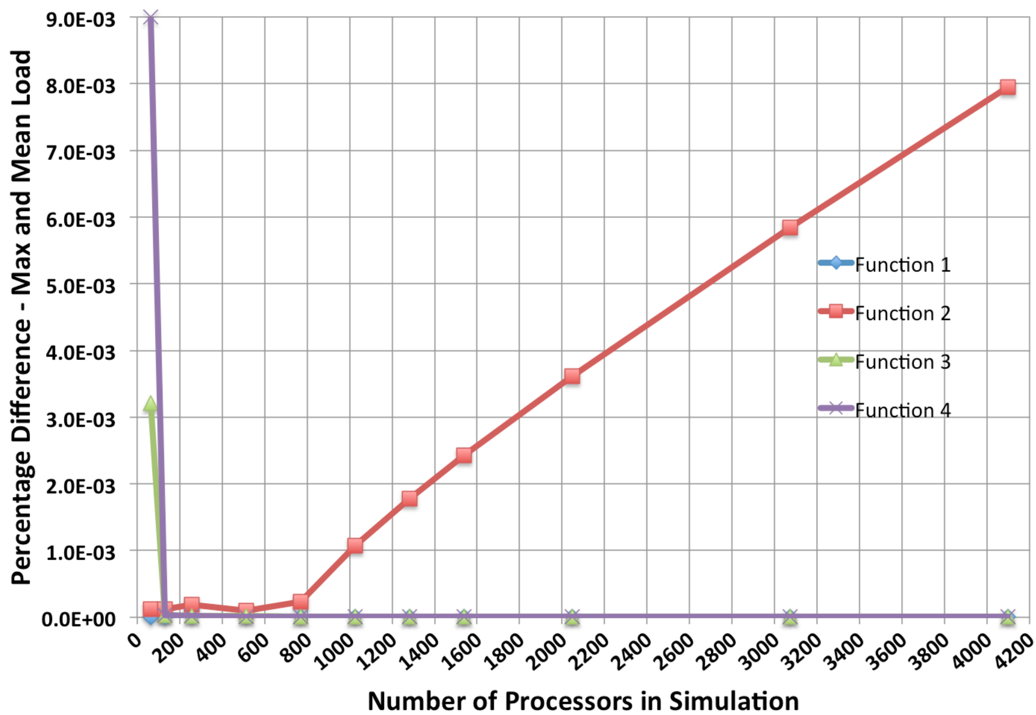

Figure 4.1: Load difference vs. increasing number of processors over 500,000 iterations

However, if you look at it in more detail, $\frac{7{,}812 \text{ iterations}}{64 \text{ processors}} = 122.0625$. So in an optimal solution, each processor would get 122.0625 iterations, but you cannot split whole iterations. So, instead, each processor gets 122 iterations, leaving 4 iterations left over ($.0625 \cdot 64$), meaning, at the most optimal load, the processor with maximum load will have 123 iterations, or 1.575% of the load ($\frac{123}{7{,}812}$). When the optimal load fraction (1.563% of the load, $\frac{122.0625}{7{,}812}$) is subtracted from that, the result is 0.012% load difference, or 1.2×10^{-4}, which is the same quality of solution that he algorithm load balanced to. So, while the load difference is worse than the other functions, it is in fact the most optimal load balance that can be achieved for Function 2.

Figure 4.2 looks at Function 1, 3, and 4 in closer detail. From the results, it can be seen that all other function achieve a load difference of about 4.0×10^{-6} for problems using greater than about 1,000 executing processors. For less than 1,000 executing processors, the linearly increasing load (Function 1) still maintains an optimal load balance in the same range, though the two *sine* based functions (Function 3 and 4) have far worse load balances for smaller numbers of processors.

In general, the load balance results show that, for an increasing number of processors, the *FGLB* algorithm maintains an efficient quality of load balance solution for the given problem, and, except for Function 2, it actually produces a more optimal load balance solution as the number of processors increases. Function 2 still shows that the *FGLB*

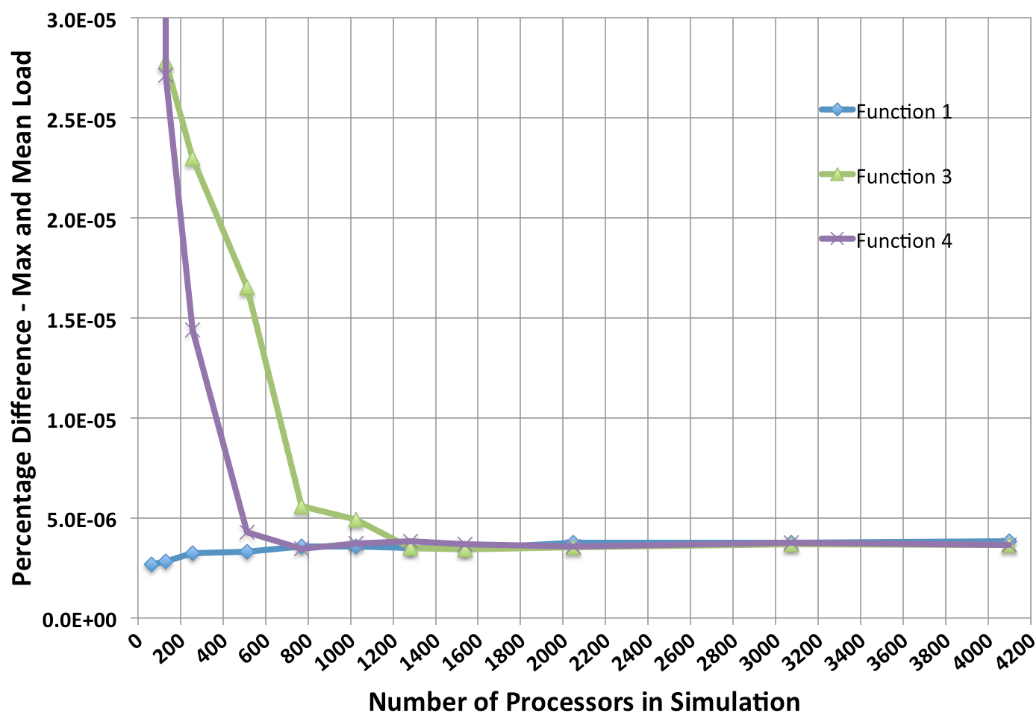

Figure 4.2: Closer look at Function 1,3,4: Load difference vs. increasing number of processors

algorithm still achieves the most optimal load balance available for that problem, even though the the problem tends to be inefficient in terms of load balancing. The results show that the algorithm can maintain a good quality of load balance solutions while increasing the number of executing processors, a very useful property for the algorithm to have.

4.1.2 Load Balance Steps to Convergence

The other important part in relation to the load balancing solution is the number of steps it took to reach the most optimal load balance the *FGLB* algorithm could (which was just discussed in "Quality of Load Balance Solution"). The results of this test are presented in Figure 4.3.

Similar to the Quality of Load Balance Solution above, problems using above 1,000 executing processors show fast convergence for all functions, though there is an outlier with Function 3 and 1,280 processors. Meanwhile, less than about 1,000 executing processors creates some problems for Function 3 and Function 4 especially. Function 1 and 2 behave as expected, reaching convergence quickly because of the ease of the problem (for Function 1) and the overall lack of data (for Function 2).

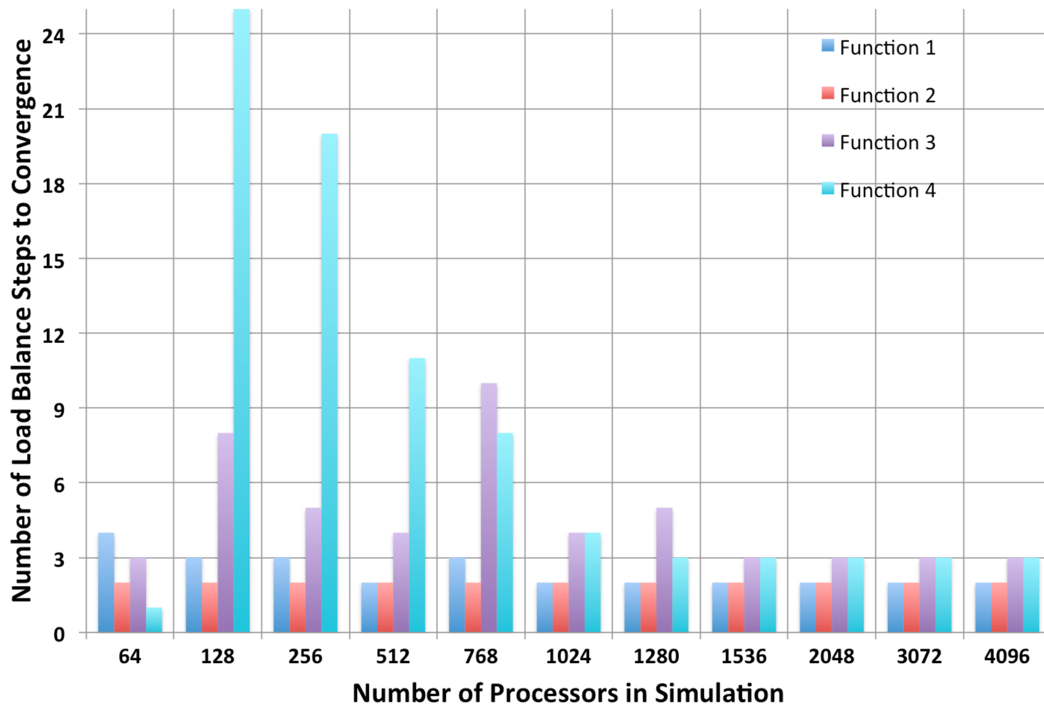

Figure 4.3: Load balance steps vs. increasing number of processors over 500,000 iterations

Given the nature of Function 3 as a *sine* function with approximately 35 periods within 500,000 iterations, and Function 4 as the same but with same random extra data bits, it seems that the *FGLB* algorithm has some issue reaching convergence quickly with a smaller number of executing processors. However, looking at the eventual convergence value reached, it may be one of the classes of problems that would benefit from a more optimized convergence test. Such a test would stop load balancing sooner when the load balance percentage difference falls under a certain threshold. The important part is that, for all the test functions, the optimal load balance still manages to converge in a reasonable number of iterations when increasing the number of processors.

4.1.3 Simulation Execution Time

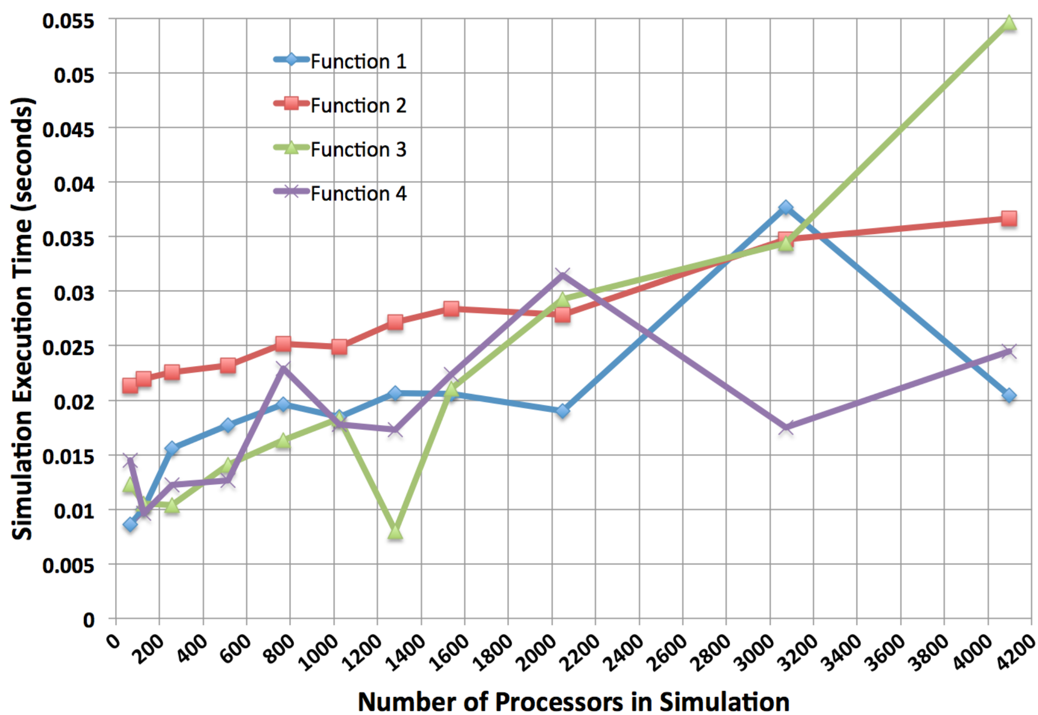

Figure 4.4: Simulation execution time vs. increasing number of processors over 500,000 iterations

The last piece of the processor scaling puzzle is the overall execution time for the simulation, which is presented in Figure 4.4. The time measured is the total time for the simulation to execute, whether that takes 2 load balancing iterations or 6 load balancing iterations; each program runs for a total of 25 "work iterations" (in a real application, work would be done each iteration). As is expected, the general trend is increasing execution time when the total number of executing processors increases. However, the increasing behavior is less than linear, on average, for each function over the increasing processors, which is a good sign that the increased communications that come with an increased number of processors is not wreaking havoc on the overall execution performance of the *FGLB* algorithm. Yet there are still some outliers and other generally

odd trends in the execution time (for instance, Function 1 at 3,072 processors) that cannot be easily explained by simply attributing it to increasing the number of processors. Therefore, a look at the execution time of each load balance step to find out why those performance oddities exist is undertaken in section "Simulation Execution Time".

4.1.4 Processor Scaling Conclusions

Taking into account the results from all three parts of the processor scaling test, it can be deduced that, when enough load data is available, increasing the number of processors can be justified in terms of execution time and number of load balancing iterations, and that doing so will not lead to an inefficient load balance solution. The sub linear behavior of the execution time vs. increasing number of processors shows that the algorithm benefits from the underlying performance of the *MPI* all-to-all communications and the minimization of point-to-point communications that occurs as the load gets more balanced. In addition, it does seem that there is some benefit derived generally from the algorithm model in that less load balance steps are needed despite the increased communications. There are still other tests to consider, but, thus far, the algorithm performance looks promising.

4.2 Iteration Scaling Results

Another important aspect to look at with regards to *FGLB* algorithm performance was how the algorithm stood up against increasing total functional iterations. The hope was that the algorithm would show nearly constant performance no matter the number of iterations, as the mathematical model showed it should. For these tests, each function from section "Functions Used to generate Synthetic Load Data" of Chapter 3 was executed on 1,024 processors and functional iterations ranging from 50,000 to 1,000,000.

4.2.1 Percentage Load Difference

Yet again, how well balanced the problem became was a starting point for this test and the metric for measuring the load balance was the same as was used in the processor scaling tests. The results are presented in Figure 4.5. In all cases, the optimal load balance gets better quickly between 50,000 and 250,000 iterations, then reaches an asymptotic limit around 2.0×10^{-6} for the remaining iterations. As such, it seems to follow behavior close to the mathematical function $y = \frac{1}{x}$. Since $\frac{1}{x}$ is related to the integral of the natural log ($\ln(x)$), it stands to reason that the overarching performance from the all-to-all communications within the algorithm implementation drive the performance of the algorithm over an increasing number of functional problem iterations. This is a very good result to see for all functions being tested.

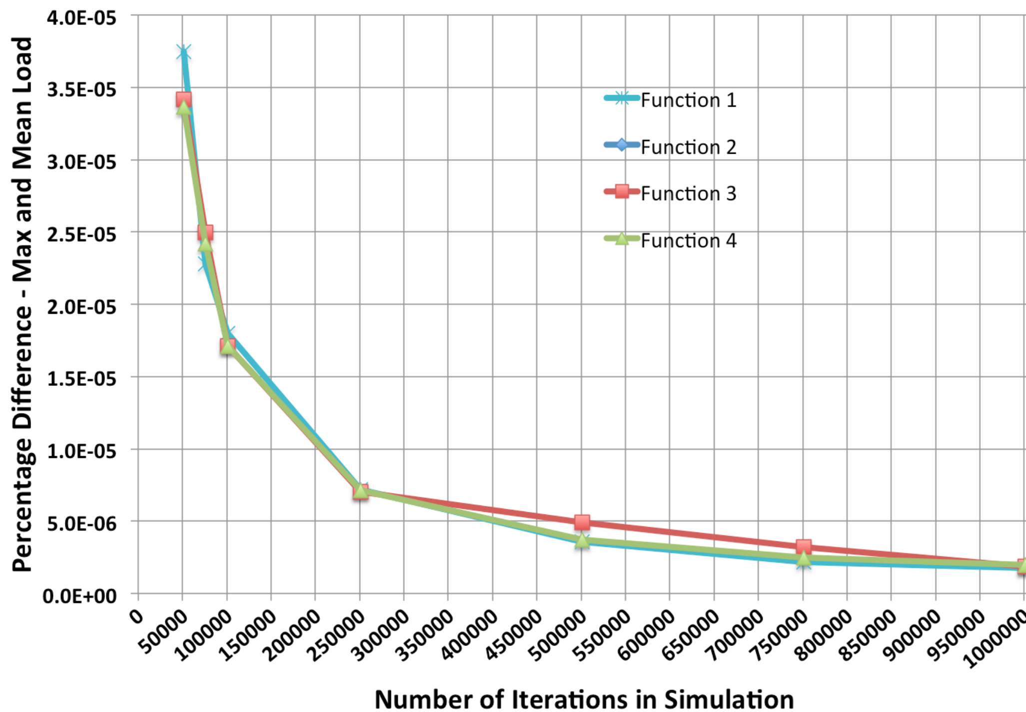

Figure 4.5: Load difference vs. increasing number of iterations executed on 1,024 processors

4.2.2 Load Balance Steps to Convergence

The performance of each function in terms of the number of load balancing steps needed to reach convergence also showed great promise. Shown ranging from 5,000 to 1,000,000 functional problem iterations, the performance of all functions for iterations less than 500,000 has kept exceptionally small and controlled, with most of the functions taking about 2 load balancing steps to reach convergence, as can be seen in Figure 4.6. For 750,000 and 1,000,000 problem iterations, Function 1 and 2 still provide the same level of performance, but Function 3 and 4 struggle, jumping to 9 and 12 load balancing iterations for the respective iteration limits.

Since there are about 52 and 69 cycles of the *sine* functions for 750,000 and 1,000,000 iterations respectively for both Function 3 and 4, it seems that 1,024 processors is simply not enough to deal with that size of problem. Given the results of the processor scaling test, it can be assumed that an increase in the number of executing processors would indeed bring that number of load balancing iterations down for those two sets of functional iterations. Despite the extremes from those two sets of iterations, Function 3 and 4 do mostly follow the same performance level as Function 1 and 2 over an increasing number of functional iterations. Once again, a very good result to see for all functions being tested.

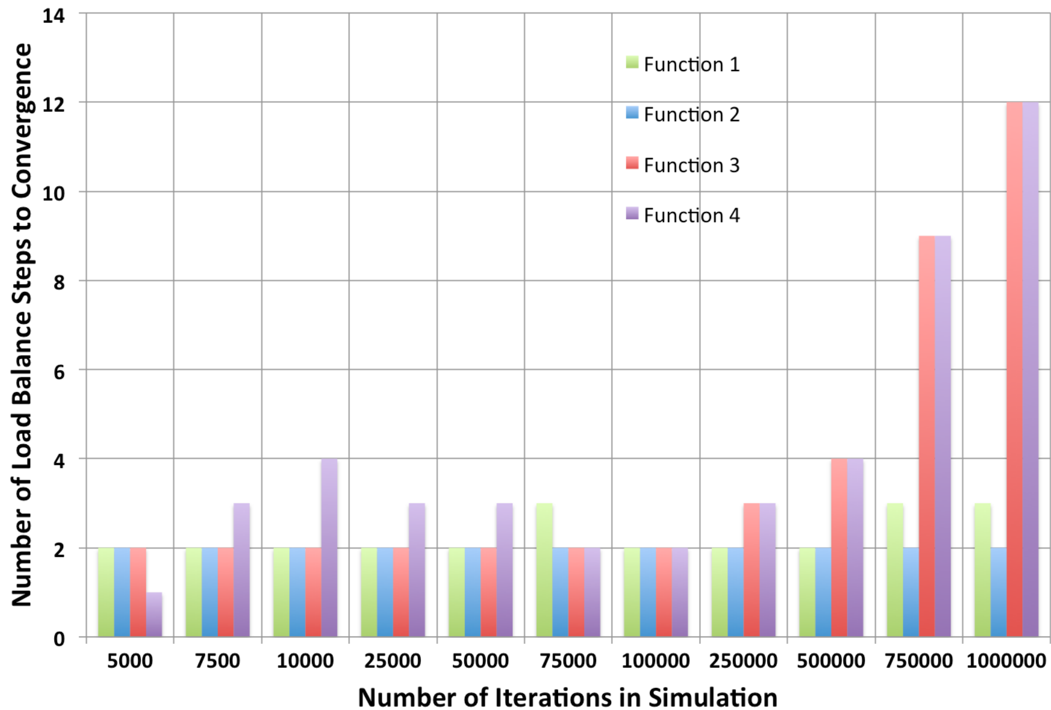

Figure 4.6: Load balance steps vs. increasing number of iterations executed on 1,024 processors

4.2.3 Simulation Execution Time

Lastly was the performance of the *FGLB* algorithm over increasing functional iterations with respect to total execution time. As with the results from the processor scaling test, the execution time increase was sub-linear for increasing number of iterations; though Function 2 did show worse overall execution time than the other three functions (nearly linear increase in execution time). The results can be seen in Figure 4.7.

4.2.4 Iteration Scaling Conclusions

Looking at the results from all three components of the Iteration scaling test, it can be deduced again that the *FGLB* algorithm performs well over an increasing number of iterations. Convergence rates for all the functions remain fairly constant across increasing iterations and the quality of the load balance solution also gets better. That, in addition to the sub-linear execution times shows that the algorithm performs very well in respect to increasing functional iterations. Like the results from the processor scaling test before it, this test gives every indication that the current *FGLB* implementation is viable as a replacement for diffusion methods for certain classes of real application data.

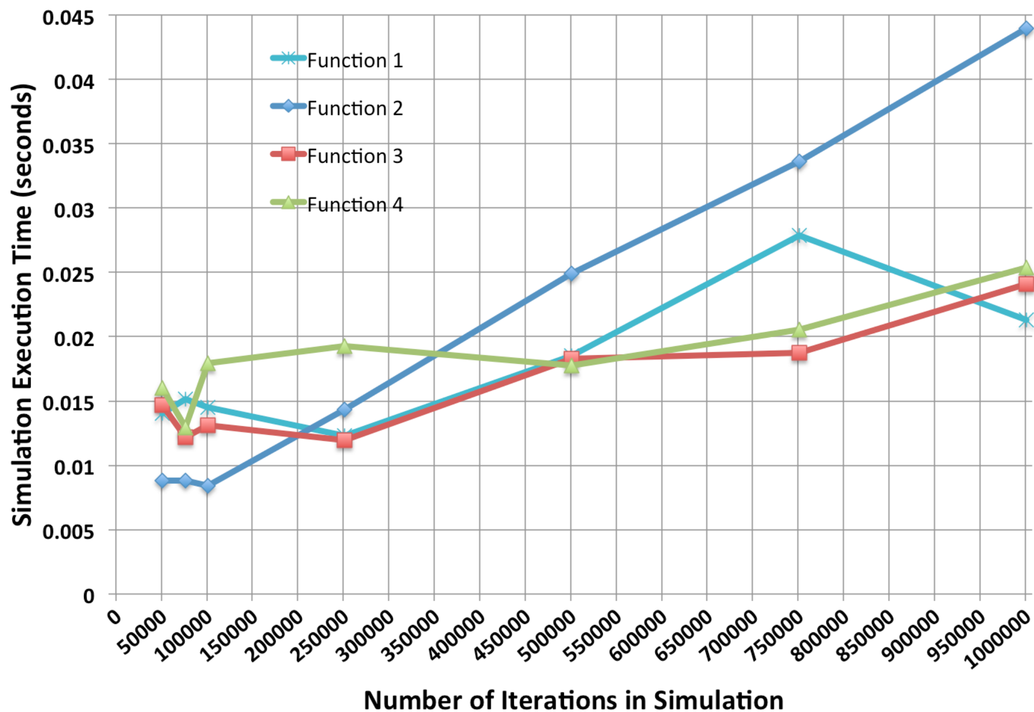

Figure 4.7: Simulation execution time vs. increasing number of iterations executed on 1,024 processors

4.3 Execution Time of the Load Balancing Step

Given that the execution time data for both the processor and iteration scaling tests, presented in Figures 4.4 and 4.7 respectively, showed some odd behavior that seemed to follow no discernible pattern, it seemed good to investigate how the execution time changed during each load balance iteration and see if there was some correlation there between the outliers and the way the load balance steps were being executed. The execution time for each load balance step for 500,000 iterations of each function on 3,072 processors is presented in Figure 4.8.

As was expected given the complexity and nature of the algorithm highlighted in the "Algorithm Complexity" section of Chapter 2, the amount of time required to execute a load balance step decreased for each subsequent load balance step. This confirms the idea that, as the load balance gets better during execution, the amount of time required for communication tends to $O(\log(P))$, thereby effectively increasing the efficiency of the algorithm. The total execution time largely depends on how poorly balanced a problem is to being with, but all the functions in the test benefit from increased communications efficiency that the algorithm displays.

However, that does not do much to satisfy the question of why certain problems seem

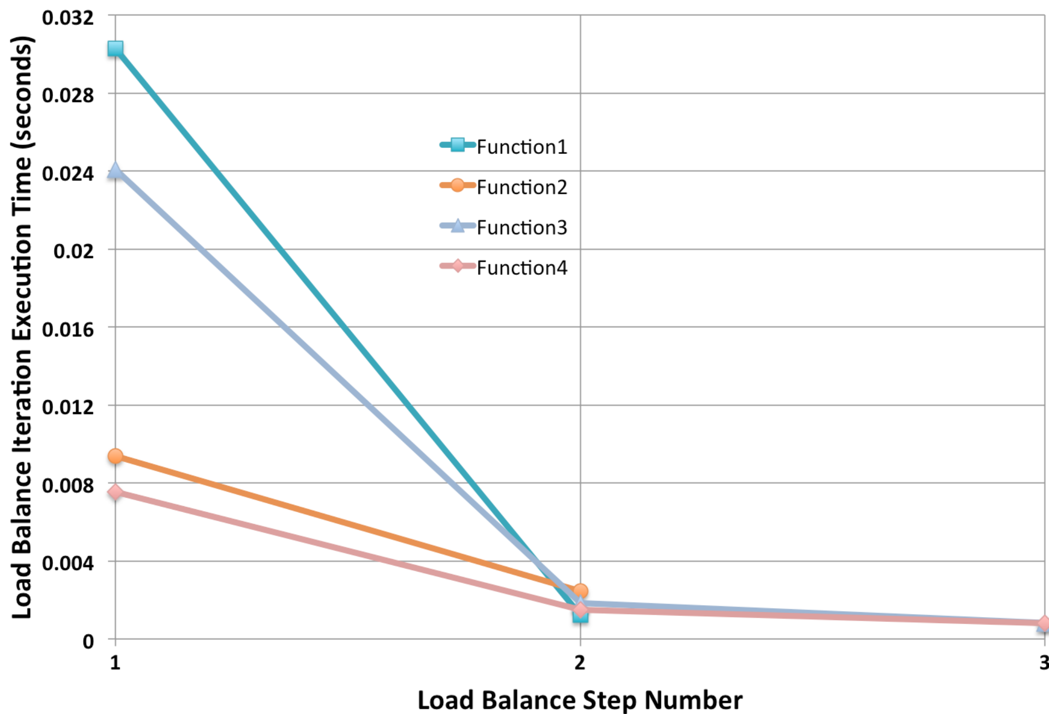

Figure 4.8: Load Balance step execution time vs. load balance step executed on 3,072 processors and 500,000 iterations

to randomly perform worse in both the processor and iteration scaling tests. Since it does not seem directly tied to load balance step performance, it could be tied to how poorly balanced the load is in certain situations (for instance, Function 1 and 3 show very poor performance comparatively during the first load balance execution shown in Figure 4.8), but then gets better during the next load balance step. Thus it stands to reason that the first load balance step dictates how well an algorithm will perform with respect to execute time, which is tied to load imbalance at the beginning of the program execution. Though more tests would be needed to confirm this behavior.

4.4 Non-Convergent Load Balancing

The tests from the "Processor Scaling Results" section above were also run on a smaller number of processors initially, where some odd behavior was noticed. For tests run on Function 3 and 4 over 8, 16, 32, and 64 executing processors, the load balance got worse at the first load balancing step. According to the convergence test being used for the *FGLB* algorithm, if the balance is worse than the prior balance, the load balance step stops executing. Thus, those problems were effectively not being load balanced due to the convergence check falsely identifying them as converged. This was likely caused by the way Function 3 and 4 vary over the iterations according to their *sine* functions.

To see if allowing those problems to run further would lead to a more optimized load

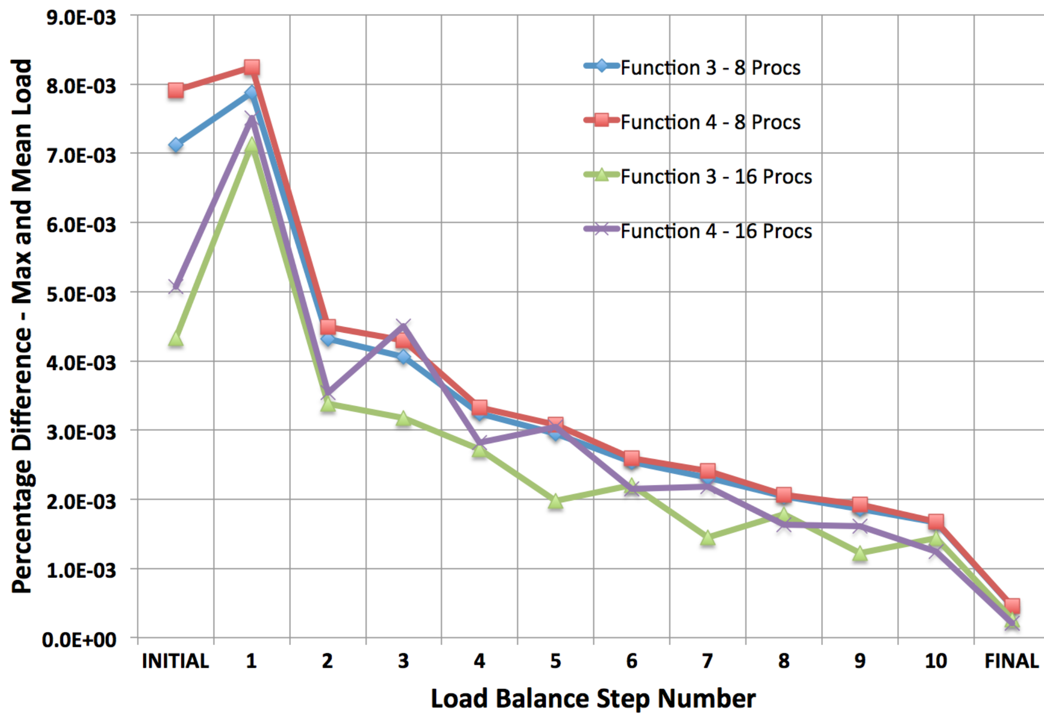

Figure 4.9: Percentage load difference vs. load balance step for 500,000 iterations executed on 8 & 16 processors

balance, another test was set up for whereby each function was allowed to load balance for the full number of work iterations and see how the load balance changed over each load balancing step. The results from 8 and 16 executing processors are presented in Figure 4.9, while the results from 32 and 64 are presented in Figure 4.10.

We see in each case, for both Function 3 and Function 4, that the load balance does get better as the program is allowed to continue executing. For 8, 16, and 32 processors, the difference between the starting load balance and the ending load balance are quite stark; performance gets remarkably better by allowing the program to run for more load balance steps. 64 processors does not have the same level of increase in optimal load balance as the other simulations, but even it does increase its optimal load balance some by allowing more load balance steps.

This test goes to show that there are still some kinks with the convergence test to work out so that fringe cases like this do not occur and stop the program from load balancing to a reasonably optimal level. It also highlights the need for there to be a cut off value for when the algorithm should stop executing load balance iterations, otherwise the algorithm could just have a pattern of going from decent load balance to worse load balance back to decent load balance in a never-ending cycle.

One way to stop that cycle while still allowing the program to run to a most optimal load balance is by having some convergence threshold, that, if reached, will stop the

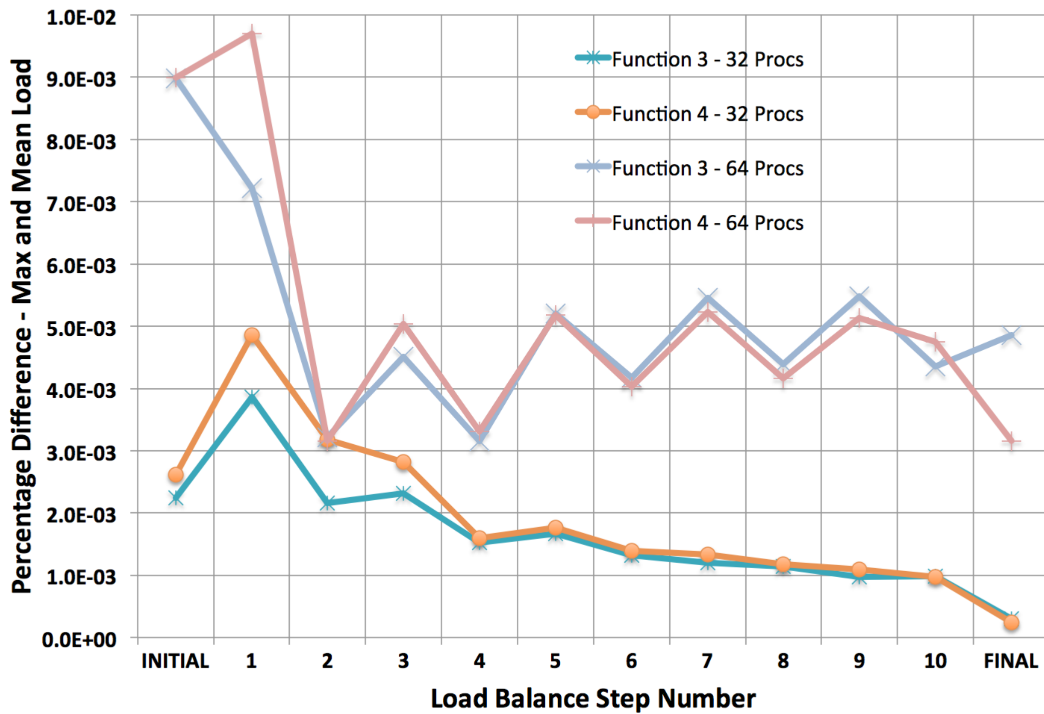

Figure 4.10: Percentage load difference vs. load balance step for 500,000 iterations executed on 32 & 64 processors

load balancing. The threshold would need to be set in such a way as to allow the *FGLB* algorithm to make a good attempt at load balancing, while preventing the solution from oscillating simply because the threshold has not be reached, though that depends upon the program being load balanced as well. To stop the oscillating behavior, a counter could be implemented such that, if it oscillates a certain number of times without any performance increase, load balancing will stop. However, implementing those ideas could take a lot of work.

A way to possibly stop the "false positives" that were being exhibited in these examples above is by not testing the first load balance. Instead, in order to allow all the balancing to progress further, let the load balance step run once before the convergence test is used for each subsequent load balance iteration. Such ideas could prevent some of the convergence test issues that were experienced during testing of the *FGLB* algorithm.

4.5 Data Simulation

Lastly, some tests and experimentation was carried out while simulating each functional iteration having a certain amount of "data" associated with it. There were some problems with the data simulation, as noted in section "Issues Encountered during Development" of Chapter 2, so it had to be run on a smaller number of iterations using a modified version of Function 3, as shown in Figure 3.9. These tests were conducted so

that between 0 and 1,000 integers were used for each functional iteration to see how an increased amount of data would affect the performance of the *FGLB* algorithm. The test was conducted on 256 and 1,024 executing processors, with 5,000 functional iterations.

Figure 4.11 shows the results for the total program execution times for the data simulation run on 256 processes. The bottom axis is logarithmically scaled to get a better view of the execution time for each respective data size. The problem converged in two

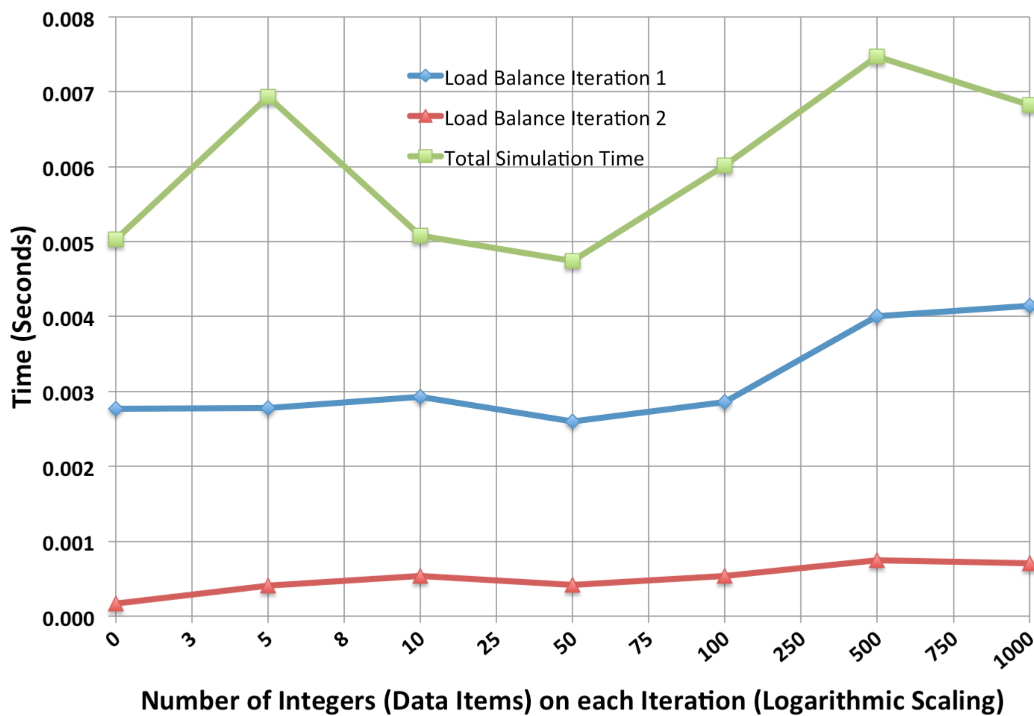

Figure 4.11: Simulation execution time vs. increasing size of data for 5,000 iterations executed on 256 processors

load balancing steps, so data was only moved twice during execution. We see that the first load balance iteration took over 3 times the amount of time that the second load balance iteration took to execute. This might seem odd at first, but given that the second load balance should require a lot less data to be moved due to the way the algorithm minimizes communication during the subsequent load balance steps, it makes sense

Figure 4.12 shows the results for the total program execution times for the data simulation run on 1,024 processes. The timing results closely follow the results from the test with 256 processes, though the simulation on 1,024 processors obviously takes longer because it is having to communicate between a lot more executing processors when trying to move all of its data.

Both tests also show that the execution time increases sub-linearly for an increasing amount of simulated data. This is due to the manner in which data is communicated

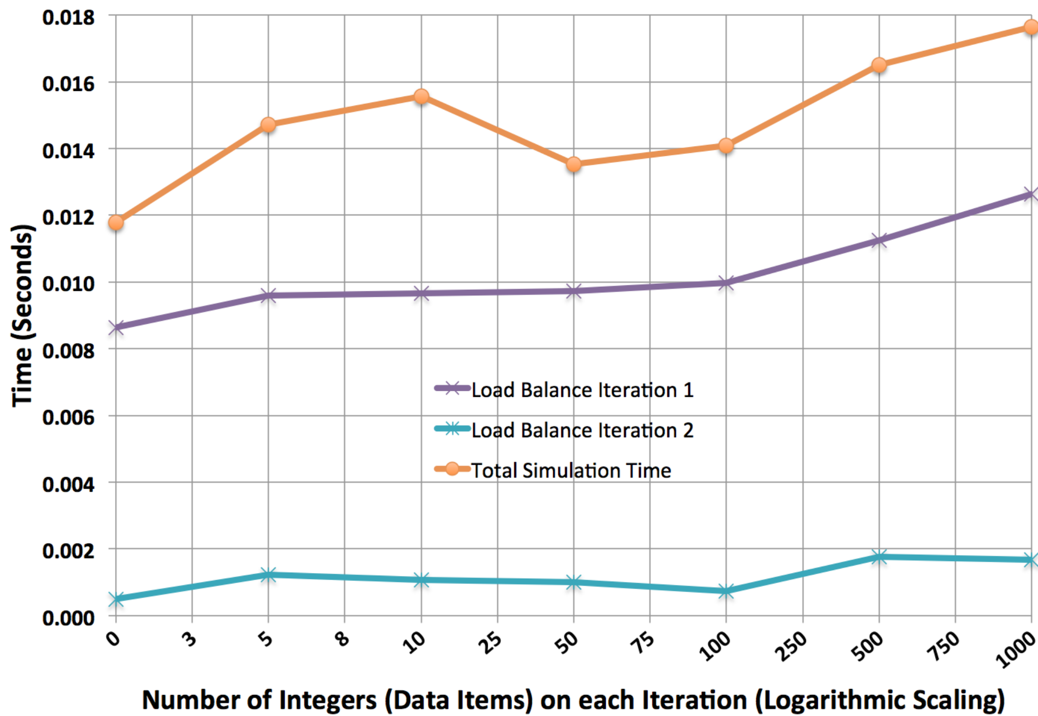

Figure 4.12: Simulation execution time vs. increasing size of data for 5,000 iterations executed on 1,024 processors

in full chunks, rather than in smaller chunks that would require more messages. This minimization of messages keeps the costs at the same level as the standard boundary sends within the algorithm. Thus, as the algorithm determines a more optimal balance of the problem as execution progresses, the data transmission benefits from the same algorithmic complexity that tends to $O(\log(P))$ in the best cases. As long as the maximum send buffer for *MPI* is not exceeded in practice, this behavior should continue for any number of data items associated with each iteration.

4.6 Summary of Results

As the results from the processor and iteration scaling tests show, the *FGLB* shows promising performance in terms the number of load balance iterations needed to reach a best optimal convergence value and its overall execution time for such performance. In addition, the quality of the load balancing solution actually tends to increase and get better as more processors or iterations are added to a given problem. All this behavior leads to a conclusion that the extra communication costs required by the algorithm to function on a distributed memory machine is kept to a minimum by means of the algorithm implementation. The reasons for this are explained in the section on "Algorithm Complexity" in Chapter 2. In addition, since the number of load balancing iterations to reach the best optimal convergence is also kept to a minimum, as well as the execution time, it can be deemed that the algorithm performs quite well, on order $O(\log(P))$ as

determined. Thus, it does seem to be a good replacement for some existing diffusion methods.

In addition to the above tests, tests were also conducted to see how data transfer would proceed, and how the convergence test affected the way convergence occurred in some problems. Though the code is by no means a finished, slick API for all real applications to come in and use as a load balancing algorithm, it does provide enough functionality to adequately test the algorithm implementation and show that Feedback Guided Load Balancing model is a viable means of load balancing problems on distributed memory machines.

Chapter 5

Conclusions

This paper has been used to present Feedback Guided Load Balancing, a method of dynamic execution load balancing designed to be used on distributed memory machines to take advantage of the available scaling such machines allow for.

Chapter 1 “Introduction” introduced some key concepts for load balancing algorithms, including the differences between Static vs. Dynamic Load Balancing, Machine Architecture and its effect on Load Balancing, and a few different Load Balancing Models, including the Diffusion Model (one of the most common types of load balancing implemented on a variety of machines).

Chapter 2 “Feedback Guided Load Balancing” was then used to spend some time discussing the Mathematical Model of Feedback Guided Load Balancing and the basic performance such a model should exhibit in practice, as well as details of the exact algorithm implementation in *MPI* and the way the model had to be altered to fit the execution requirements in *MPI*.

Then, in Chapter 3 “Algorithm Testing Environment”, time was spent talking about the hardware and software of *HECToR*, the machine used for the purposes of testing the *FGLB* algorithm implementation before moving into some detail on the functions used to generate the synthetic load data that the algorithm used to determine how to load balance the given problem.

Lastly, Chapter 4 “Results & Performance Evaluation” presented all the data and graphs collected throughout the course of testing the algorithm and analyzed the results to give a final opinion about how the algorithm functions and how effective it is at load balancing the based on the synthetic load data.

The whole point of developing this algorithm and implementing it was to see if the extra communications costs that normally occur with distributed memory machines would bog down the algorithm and make it all but unusable. The aim specifically is the devel-

opment of an algorithm that can possibly replace existing diffusion methods and take advantage of the greater scaling available on distributed memory machines.

Such diffusion methods tend to be order $O(1)$ in time complexity, since each executing processor only has to send data to its neighboring processors. The Feedback Guided Load Balancing model, and the specific *MPI* algorithm implementation, tends to be order $O(\log(P))$ in the best case scenario, and $O(P)$ in the worst. The worst case is usually seen at the start of the load balancing execution and when load is very unbalanced. As execution progresses though, and the load balance gets better, time complexity drops to $O(\log(P))$ due to the way the *FGLB* algorithm uses the calculated optimal load boundaries and point-to-point sends.

Which means, while the *FGLB* algorithm can never reach the time complexity of diffusion methods, it does has an optimal complexity that is sub-linear, and continues to show performance increases when scaling over a larger number of processors. That is a very important characteristic for an algorithm designed to run on distributed memory systems, as it will be better able to utilize the increased availability of processor and memory hardware such machines allow. With more tests, especially against specific diffusion method-based algorithms, it may show that Feedback Guided Load Balancing provides a more efficient implementation of load balancing for specific classes of problems and applications.

The results prove that the *FGLB* algorithm performs well over increasing processors and iterations, in fact, the quality of the load balance solution actually gets better with increasing processors and iterations. That result, combined with the reasonable execution time performance and the low number of load balance iterations needed to reach an optimal load balance, means the *FGLB* algorithm does perform as expected and describes a model of dynamic execution load balancing that provides good scaling results and good performance over the available processors and memory on distributed memory systems such as *HECToR*.

5.1 Possible Future Work

As always with any project of this magnitude, there is some work that simply does not get accomplished due to the time and scope restrictions. This dissertation was designed to explore the Feedback Guided Load Balancing model and design a working implementation of it to be tested on distributed memory machines. At the core, the code and this paper present such a working implementation and the results obtained from testing the algorithm. Yet there is always further work that can be done on any algorithm, and especially one as experimental as this algorithm was.

First, it would be very useful and interesting to test the *FGLB* algorithm on real application data to determine how the algorithm performs in such situations. Any attempt to

use this algorithm for real application data would require some reworking of the source code to achieve such a thing. The current source code is not fully implemented as a nice library plugin for any manner of real applications. It was designed to test the algorithm using synthetic load data. Though there are some current shells of what could be easily changed to an interface and, eventually, a full API to whatever problem is being load balanced, there would still need to be some work done to get that fully implemented. A next interesting project for this algorithm could indeed be taking the source and making the necessary changes to make it a full API for real load balance applications, and then testing how the algorithm works when used with real applications.

Also, further work needs to be carried out to see what exactly is the cause of the segmentation faults that are occurring when the size of the data simulation increases. Any real application needs the ability to transfer associated data for each functional iteration when load balancing is carried out. The framework is there and theoretically it should all be working. If it is the case that all available memory on *HECToR* is being used up, leading to the segmentation faults, then an implementation for real applications that does not need to store large arrays of synthetically generated load may solve the issue. Or it may be something else entirely that went undetected during the course of the development of the current *FGLB* algorithm. Either way, it is a problem that needs to be addressed going forward with this algorithm and implementation.

Another interesting project that could be undertaken is implementing this algorithm on a Partitioned Global Address Space (PGAS) language such as Unified Parallel C (*UPC*). It was originally a goal of this project to implement a second version of the algorithm in UPC for the purpose of performance comparisons. However, it soon became clear that goal was too ambitious for the scope of the project and work on the *UPC* version needed to be abandoned. *UPC*, and other PGAS languages like it, use software paradigms to present distributed memory machines as shared memory machines to the programmer by taking care of the underlying communications needed. Therefore the programmer only needs to deal with the code as if designing it on a shared memory machine. Though care does need to be taken with “data affinity” and how the data is decomposed to maximize memory access on data that is close to the calling processor ([23]). It would be great to compare the performance between the existing *FGLB* algorithm implemented with the “message-passing model” that is *MPI* and the Feedback Guided Load Balancing model implemented in a PGAS language to see if there are any performance benefits to be gained by letting the language handle the communications implicitly rather than letting the programmer handle them explicitly.

It would also be encouraged to test the *FGLB* algorithm against other load balancing algorithms, especially diffusion method-based algorithms, to see how the performance of *FGLB* measures up against other algorithms. According to theoretical performance and the results obtained from the tests undertaken in this paper, it does seem that *FGLB* is a viable algorithm for use in load balancing applications and could potentially perform better than existing diffusion methods, but having concrete tests and results of whether

or not that is true would be very useful and give a better idea of whether the *FGLB* algorithm is worth future progress and development. Originally, the idea was to incorporate some tests against other algorithms in this paper, but there was unfortunately not enough project time to accomplish that goal.

Lastly, it would be useful to conduct some work into better understanding the performance aspects of the algorithm through profiling and other performance measurement tools. Though the mathematical model of the algorithm and the results from the tests presented in this paper do give a good overview of the performance of the algorithm, it would be good to delve more into the underlying hardware and software performance of the algorithm, which could, in turn, yield some optimizations in the way the code is implementation to further increase the performance of the algorithm. That would then increase the usability and viability of the algorithm when compared against existing load balancing solutions.

Bibliography

[1] Orestis Agathokleous. Dynamic loop nesting in shared memory programming. Research dissertation, EPCC - The University of Edinburgh, Edinburgh, Scotland, UK, August 2011. http://www.epcc.ed.ac.uk/wp-content/uploads/2011/11/OrestisAgathokleous.pdf.

[2] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, November 1990. http://www.cs.cmu.edu/~blelloch/papers/Ble93.pdf.

[3] Elena Breitmoser, Joachim Hein, and Christopher Johnson. Advanced parallel programming lecture 2 - hector. Internet Release: EPCC - The University of Edinburgh, February 2012. https://www2.epcc.ed.ac.uk/msc/students/courses1112/APP/Slides/L02-Hector.pdf.

[4] Dr. J. Mark Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *In: Euro-Par'98 Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 377–382. Euro-Par, Springer-Verlag, September 1998. http://www.epcc.ed.ac.uk/~markb/docs/europar98.ps.gz.

[5] Sébastien Chauvin, Proshanta Saha, François Cantonnet, Smita Annareddy, and Tarek El-Ghazawi. *UPC Manual*. High Performance Computing Laboratory - The George Washington University, Washington, DC 20052, USA.

[6] Constantinos Christofi. Feedback guided load balancing in a distributed memory environment. Research dissertation, EPCC - The University of Edinburgh, Edinburgh, Scotland, UK, August 2011. http://www.epcc.ed.ac.uk/wp-content/uploads/2011/11/ConstantinosChristofi.pdf.

[7] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, 1989. http://www.dartmouth.edu/~gvc/Cybenko_JPDP.pdf.

[8] 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. *Applied Numerical Mathematics*,

52(2-3):133–152, February 2005. http://j.teresco.org/research/publications/adapt03/adapt03.pdf.

[9] Ralf Diekmann, Burkhard Monien, and Robert Preis. Load balancing strategies for distributed memory machines. In *Parallel and Distributed Processing for Computational Mechanics*, pages 124–157, D-33102 Paderborn, Germany, 1999. Department of Computer Science, University of Paderborn. http://faculty.cs.byu.edu/~snell/Classes/CS584/papers/DMP_Lastvert_Report.pdf.

[10] Alan Edelman. Applied parallel computing lecture 3 - parallel prefix, mathematics 18.337, computer science 6.338, sma 5505. Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, Spring 2004. http://ocw.mit.edu/courses/mathematics/18-337j-applied-parallel-computing-sma-5505-spring-2005/lecture-notes/chapter_3.pdf.

[11] G. Horton. A multi-level diffusion method for dynamic load balancing. *Parallel Computing*, 19(2):209–218, February 1993. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.159.2598&rep=rep1&type=pdf.

[12] Stephen W. Keckler. The importance of locality in scheduling and load balancing for multiprocessors. Technical Report MIT Concurrent VLSI Architecture Memo 61, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, February 1994. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.5244&rep=rep1&type=pdf.

[13] E. Luque, A. Ripoll, A. Cortés, and T. Margalef. A distributed diffusion method for dynamic load balancing on parallel computers. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pages 43–50, 08193-Bellaterra, Barcelona, Spain, 1995. Departament d`Informàtica, Universitat Autònoma of Barcelona. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.24.8988&rep=rep1&type=pdf.

[14] Michael McCool. Parallel pattern 8: Scan. Internet Release - Intel Software Blog, September 2009. http://software.intel.com/en-us/blogs/2009/09/15/parallel-pattern-8-scan/.

[15] John T. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, September 1994. http://www.dcs.gla.ac.uk/publications/PAPERS//7060/parscan-PPL94.ps.

[16] EPCC The University of Edinburgh. Hector: Uk national supercomputing service. Internet Release: EPCC - The University of Edinburgh, August 2012. http://www.hector.ac.uk.

[17] Stephen Olivier and Jan Prins. Scalable dynamic load balancing using upc. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 123–131, Washington, DC, USA, 2008. IEEE Computer Society. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.145.263&rep=rep1&type=pdf.

[18] Stephen Olivier, Jan Prins, James Dinan, Gerald Sabin, P. Sadayappan, and Chau-Wen Tseng. Dynamic load balancing of unbalanced computations using message passing. In *Proceedings of the 6th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, Washington, DC, USA, 2007. IEEE Computer Society. http://www.cs.unc.edu/~olivier/pmeo07.pdf.

[19] Peter Sanders and Jesper Larsson Traff. Parallel prefix (scan) algorithms for mpi. In *Proceedings of the 13th European PVM/MPI User's Group conference on Recent advances in parallel virtual machine and message passing interface*, EuroPVM/MPI'06, pages 49–57, Berlin, Heidelberg, 2006. Springer-Verlag. http://algo2.iti.kit.edu/sanders/papers/scan.pdf.

[20] Shubhabrata Sengupta, Mark Harris, Michael Garland, and John D. Owens. Efficient parallel scan algorithms for many-core gpus. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, *Scientific Computing with Multicore and Accelerators*, Chapman & Hall/CRC Computational Science, chapter 19, pages 413–442. Taylor & Francis, January 2011. http://www.taylorandfrancis.com/books/details/9781439825365/.

[21] Shubhabrata Sengupta, Aaron E. Lefohn, and John D. Owens. A work-efficient step-efficient prefix-sum algorithm. *Communications of the ACM*, pages 1–2, 2006. http://www.idav.ucdavis.edu/func/return_pdf?pub_id=894.

[22] Top500 Supercomputer Sites. Top500 list - june 2012. Internet Release: TOP500.Org, August 2012. http://www.top500.org/list/2012/06/100.

[23] Tim Stitt Ph.D. An introduction to the partitioned global address space (pgas) programming model. Internet Release: Connexions, Rice University, March 2010. http://cnx.org/content/m20649/latest/.

[24] Toufik Taibi1, Abdelouahab Abid, and Engku F. E. Azahan. A comparison of dynamic load balancing algorithms. *J.J.Appl. Sci: Natural Sciences*, 9(2):125–132, 2007. http://www.asu.edu.jo/TestWeb/userfiles/file/natural_pdf/Volume-9-2007/Number-2/A%20Comparison%20of%20Dynamic%20Load%20Balancing%20Algorithms.pdf.

[25] Hao Zhu, David Goodell, William Gropp, and Rajeev Thakur. Hierarchical collectives in mpich2. Technical Report ANL/MCS-P1622-0509, Argonne National Laboratory, Argonne, IL 60439, USA, May 2009. http://www.mcs.anl.gov/uploads/cels/papers/P1622.pdf.

Appendix A

Complete *FGLB* Algorithm *MPI* Pseudocode

```
   Initialize Variables (data[], localLoad, partialLoad, etc.) & MPI
 2
   //Establish initial data for functional work load
 4 data[] gatherData( from source file )

 6 //Distribute problem iteration onto available processors with equal
   //bounding to start          (total iterations / total processors)
 8
   while(work not finished){
10
       //Record load data per processor while work is occurring
12     doWork(data[], localLoad)

14     //To gather partial sums and total load across all processors
       MPI_Scan(localLoad, partialLoad, count, datatype, MPI_SUM,
16         MPI_COMM_WORLD)
       MPI_Allreduce(localLoad, totalLoad, count, datatype, MPI_SUM,
18         MPI_COMM_WORLD)

20     //If load has changed then check for convergence
       if( localLoad or totalLoad has changed between iterations ){
22
           //Check if load balancing has converged
24         covergenceCheck(localLoad, totalLoad)
       }
26
       //If load balancing has not converged to optimal balance, then
28     //run loadBalance again
       if( ! converged ){
30         //Function to repartition boundaries and data based on
           //current load balance
32         loadBalance()
       }
34 }
```

```
36  outputData( data[] )
    MPI_Finalize()
```

Listing A.1: Detailed *MPI* pseudocode: main function

```
38  //Check if load balancing has converged
    covergenceCheck(localLoad, totalLoad){
40
        //To find maximum load from all processors
42      MPI_Allreduce(localLoad, maxLoad, count, datatype, MPI_MAX,
            MPI_COMM_WORLD)
44
        if( loadDiff < ( maxLoad - ( totalLoad / processors ) ) ){
46          return true; //Has converged
        } else {
48          loadDiff = maxLoad - ( totalLoad / processors )
            return false; //Has not converged
50      }
    }
52
    //Function to repartition boundaries and data based on current
54  //load balance
    loadBalance(){
56
        //Function to compute the new boundaries based on current
58      //load balance
        computeNewBounds()
60
        //Function to send and receive the new boundaries based on
62      //current load balance using non-blocking sends
        exchangeBounds()
64
        //Function to send and receive any data moved when boundaries
66      //were changed using non-blocking sends
        exchangeData()
68  }
```

Listing A.2: Detailed *MPI* pseudocode: convergenceCheck and loadBalance functions

```
    //Function to compute new boundaries based on current load balance
70  computeNewBounds(){

72      //Each processor determines optimal load from that iteration
        optimalLoad = totalLoad / processors
74      //Total of load from each processor before current processor
        priorLoad = partialLoad - localLoad
76
```

```
    //Determine whether optimal load for another processor lies
78  //within current processor boundaries
    for( k = ( floor(priorLoad / optimalLoad) );
80       k < ( ceil(partialLoad / optimalLoad) ); ++k){

82            loadPoints[boundsInside] = k
              ++boundsInside
84  }

86  //Determine new boundaries for each processor current processor
    //has optimal load from
88  for( i = 0; i < boundsInside; ++i){
         //0 index of hiBound = which processor high boundary is for
90       highBound[0][i] = ( loadPoints[i] - 1 )
         //0 index of lowBound = which processor low boundary is for
92       lowBound[0][i] = ( loadPoints[i] )
         //1 index of highBound sets high boundary
94       highBound[1][i] = ((((loadPoints[i] * optimalLoad) -
              priorLoad) / ((localLoad) / (highBound - lowBound)))
96            + lowBound )
         //1 index of lowBound sets low boundary
98       lowBound[1][i] = highBound[1][i]
    }
100 }
```

Listing A.3: Detailed *MPI* pseudocode: computeNewBounds function

```
    //Function to send and receive the new boundaries based on current
102 //load balance using non-blocking sends
    exchangeBounds(){
104
    //Sends each set of new boundaries
106 for( i = 0; i < boundsInside; ++i){
         MPI_Isend( highBound[1][i], 1, MPI_INT, highBound[0][i],
108           highTag, comm, request )
         MPI_Isend( lowBound[1][i], 1, MPI_INT, lowBound[0][i],
110           lowTag, comm, request )
    }
112
    //Receive new highBound
114 MPI_Recv( highBound, 1, MPI_INT, MPI_ANY_SOURCE, highTag, comm,
         highStatus)
116 //Receive new lowBound
    MPI_Recv( lowBound, 1, MPI_INT, MPI_ANY_SOURCE, lowTag, comm,
118      lowStatus)

120 //Wait until all sends and receives have completed
    MPI_Wait( request, status)
122 }
```

Listing A.4: Detailed *MPI* pseudocode: exchangeBounds function

```
    //Function to send and receive any data moved when boundaries were
124 //changed using non-blocking receives
    exchangeData(){
126
        numberReceives = 1+(highStatus.MPI_SOURCE-lowStatus.MPI_SOURCE)
128
        //Sends each set of new data sizes
130     for( i = 0; i < boundsInside; ++i){
            MPI_Isend( newDataSize, 1, MPI_INT, highBound[0][i], tag,
132             comm, request )
        }
134     //Receive new data size
        for( i = 0; i < numberReceives; ++i){
136         MPI_Recv( sizeToSwap, 1,  MPI_INT, MPI_ANY_SOURCE, tag,
                comm, status )
138     }
        MPI_Wait( request, status)
140
        //Sends each new data
142     for( i = 0; i < boundsInside; ++i){
            MPI_Isend( oldData, newDataSize, MPI_DATA, highBound[0][i],
144             tag, comm, request )
        }
146     //Receive new data
        for( i = 0; i < numberReceives; ++i){
148         MPI_Recv( newData, sizeToSwap,  MPI_DATA, MPI_ANY_SOURCE,
                tag, comm, status )
150     }
        MPI_Wait( request, status)
152 }
```

Listing A.5: Detailed *MPI* pseudocode: exchangeData function

Appendix B

Sample Makefile used on *HECToR*

```
#
# Makefile for Hector System
#
# Feedback Guided Load Balancing in Distributed Memory Environments
# MPI Implementation
# Version 1.0
#
# M.Sc. High Performance Computing - Dissertation
# 2012-08-11
#
# A makefile for the Dissertation project used to compile the main
# program executable
#
##########################################################################
# MACROS
##########################################################################

# Sets the name of the executable to be built. Change this as needed
EXEFILE = fglb

# Sets phony flags to ensure these targets always run
.PHONY : clean cleanoutput cleantgz cleandocs cleanall dist docs

# Sets compiler, library, and c flags
CC = cc
OFLAGS = -O3#
DFLAGS = #-g
LFLAGS = -lm#

# List of all non-source files that are part of the distribution
AUXFILES = Makefile_H Makefile_N data docs output scripts README

# This is a list of all directories containing project code
SRCDIRS = source

# This is a list of all directories containing object code
BUILDDIRS = build
```

```
38
   # All source and header files within the project & test directories
40 SRCFILES = $(shell find $(SRCDIRS) -type f -name "*.c")
   HDRFILES = $(shell find $(SRCDIRS) -type f -name "*.h")
42
   # Sets object files based on project and test .c files
44 OBJFILES := $(patsubst $(SRCDIRS)/%.c,$(BUILDDIRS)/%.o,$(SRCFILES))

46 # Sets all files and folders in the project
   ALLFILES = $(AUXFILES) $(SRCDIRS) $(BUILDDIRS)
48
   ######################################################################
50 # TARGETS
   ######################################################################
52
   # Default target to build simple executable of the imageProcessor
54 all: $(EXEFILE)

56 $(EXEFILE): $(OBJFILES)
        $(CC) $(DFLAGS) $(OFLAGS) $(LFLAGS) $(OBJFILES) -o $@
58
   $(OBJFILES) : $(SRCFILES) $(HDRFILES)
60      $(CC) $(DFLAGS) $(OFLAGS) -c
                    $(patsubst $(BUILDDIRS)/%.o, $(SRCDIRS)/%.c, $@) -o $@
62


64 # Creates the API documents with doxygen based on source code
   docs:
66      @doxygen data/doxygen.config


68
   # Create a tgz file of all the project code and files
70 dist:
        @tar -czpf $(EXEFILE).tgz $(ALLFILES)
72


74 # Cleans all object files, executables, & tilde files in the project
   clean:
76      @find . -type f -name "*.o" -exec $(RM) '{}' \;
        @find . -type f -name "$(EXEFILE)" -exec $(RM) '{}' \;
78      @find . -type f -name "*~" -exec $(RM) '{}' \;

80 # Cleans all object files, executables, & tilde files in the project
   cleanoutput:
82      @find . -type f -name "*.out" -exec $(RM) '{}' \;

84 # Cleans any tgz files in the project
   cleantgz:
86      @find . -type f -name "*.tgz" -exec $(RM) '{}' \;
        @find . -type f -name "*.tar" -exec $(RM) '{}' \;
88
   # Cleans all the files in the docs folder of the project
90 cleandocs:
        @find docs -type f -name "*.*" -exec $(RM) '{}' \;
```

```
92      @find docs -type f -name "installdox" -exec $(RM) '{}' \;
        @find docs -type f -name "Makefile" -exec $(RM) '{}' \;
94      @$(RM) -rf docs/html/search;
        @$(RM) -rf docs/html;
96      @$(RM) -rf docs/latex;

98  # Cleans everything using all the cleans above
    cleanall: clean cleanoutput cleantgz cleandocs
100
    # This line required by Make - Do not delete!
```

Listing B.1: Makefile to compile *FGLB* algorithm on *HECToR*

Appendix C

Sample PBS Script used on *HECToR*

```
#!/bin/bash --login
#
#PBS -N FGLBDataSim
#PBS -l walltime=00:10:00
#PBS -l mppwidth=256
#PBS -l mppnppn=32
#PBS -A d34
#PBS -j oe
#
# Change to the directory that the job is submitted from
cd $PBS_O_WORKDIR
#
for t in 256
do
#
for p in 0 5 10 50 100 500 1000
do
echo "Running MPI program FGLB on " $t " processes"
echo "------------------------------------------------------------"
aprun -n $t -N 32 ./fglb data/dataSim/parameters$p.dat
echo
done
#
done
```

Listing C.1: PBS Script used to submit *FGLB* algorithm to backend of *HECToR*

Appendix D

Tables of Raw Data from Output of Test Runs on *HECToR*

D.1 Results from Processor Scaling Tests

Table D.1: Results of Function 1 on 500,000 Iterations (Time in seconds)

Total Processors	LB Steps to Convergence	Mean Load	Max Load	Load Difference	Total Execution Time
2	9	5.000E-01	5.000E-01	1.519E-06	0.0299380
8	7	1.250E-01	1.250E-01	1.894E-06	0.0150760
16	5	6.250E-02	6.250E-02	3.618E-06	0.0099220
32	5	3.125E-02	3.125E-02	2.107E-06	0.0094540
64	4	1.563E-02	1.563E-02	2.661E-06	0.0085890
128	3	7.813E-03	7.815E-03	2.843E-06	0.0100490
256	3	3.906E-03	3.909E-03	3.249E-06	0.0155570
512	2	1.953E-03	1.956E-03	3.330E-06	0.0176830
768	3	1.302E-03	1.306E-03	3.565E-06	0.0196300
1024	2	9.766E-04	9.801E-04	3.557E-06	0.0184910
1280	2	7.813E-04	7.848E-04	3.511E-06	0.0206210
1536	2	6.510E-04	6.545E-04	3.449E-06	0.0205490
2048	2	4.883E-04	4.921E-04	3.780E-06	0.0190330
3072	2	3.255E-04	3.293E-04	3.768E-06	0.0376920
4096	2	2.441E-04	2.480E-04	3.844E-06	0.0204730

Table D.2: Results of Function 2 on 500,000 Iterations (Time in seconds)

Total Processors	**LB Steps to Convergence**	**Mean Load**	**Max Load**	**Load Difference**	**Total Execution Time**
2	2	5.000E-01	5.000E-01	0.000E+00	0.0205150
8	2	1.250E-01	1.250E-01	8.000E-06	0.2238230
16	2	6.250E-02	6.253E-02	2.800E-05	0.2396840
32	2	3.125E-02	3.130E-02	4.600E-05	0.2498940
64	2	1.563E-02	1.575E-02	1.200E-04	0.0213070
128	2	7.813E-03	7.937E-03	1.240E-04	0.0219740
256	2	3.906E-03	4.096E-03	1.900E-04	0.0225930
512	2	1.953E-03	2.049E-03	9.606E-05	0.0231690
768	2	1.302E-03	1.536E-03	2.340E-04	0.0251980
1024	2	9.766E-04	2.049E-03	1.073E-03	0.0249200
1280	2	7.812E-04	2.564E-03	1.783E-03	0.0271430
1536	2	6.510E-04	3.077E-03	2.426E-03	0.0283840
2048	2	4.883E-04	4.098E-03	3.610E-03	0.0278580
3072	2	3.255E-04	6.173E-03	5.847E-03	0.0347810
4096	2	2.441E-04	8.197E-03	7.953E-03	0.0366510

Table D.3: Results of Function 3 on 500,000 Iterations (Time in seconds)

Total Processors	**LB Steps to Convergence**	**Mean Load**	**Max Load**	**Load Difference**	**Total Execution Time**
2	25	5.000E-01	5.017E-01	1.729E-03	0.2147620
8	1	1.250E-01	1.321E-01	7.129E-03	0.0602750
16	1	6.250E-02	6.683E-02	4.327E-03	0.0309630
32	1	3.125E-02	3.350E-02	2.248E-03	0.0170070
64	3	1.563E-02	1.884E-02	3.214E-03	0.0122690
128	8	7.813E-03	7.840E-03	2.782E-05	0.0105560
256	5	3.906E-03	3.929E-03	2.299E-05	0.0104080
512	4	1.953E-03	1.970E-03	1.651E-05	0.0141080
768	10	1.302E-03	1.308E-03	5.603E-06	0.0163070
1024	4	9.766E-04	9.815E-04	4.927E-06	0.0183080
1280	5	7.813E-04	7.848E-04	3.513E-06	0.0079640
1536	3	6.510E-04	6.545E-04	3.431E-06	0.0210420
2048	3	4.883E-04	4.918E-04	3.533E-06	0.0292660
3072	3	3.255E-04	3.292E-04	3.675E-06	0.0344360
4096	3	2.441E-04	2.478E-04	3.666E-06	0.0546660

Table D.4: Results of Function 4 on 500,000 Iterations (Time in seconds)

Total Processors	**LB Steps to Convergence**	**Mean Load**	**Max Load**	**Load Difference**	**Total Execution Time**
2	25	5.000E-01	5.007E-01	7.378E-04	0.2796390
8	1	1.250E-01	1.329E-01	7.908E-03	0.0876220
16	1	6.250E-02	6.757E-02	5.073E-03	0.0445610
32	1	3.125E-02	3.387E-02	2.620E-03	0.0230610
64	1	1.563E-02	2.462E-02	8.998E-03	0.0145100
128	25	7.813E-03	7.840E-03	2.714E-05	0.0096070
256	20	3.906E-03	3.921E-03	1.439E-05	0.0122390
512	11	1.953E-03	1.957E-03	4.279E-06	0.0126340
768	8	1.302E-03	1.306E-03	3.447E-06	0.0229260
1024	4	9.766E-04	9.803E-04	3.735E-06	0.0177620
1280	3	7.812E-04	7.851E-04	3.849E-06	0.0172930
1536	3	6.510E-04	6.547E-04	3.691E-06	0.0223400
2048	3	4.883E-04	4.919E-04	3.570E-06	0.0314690
3072	3	3.255E-04	3.293E-04	3.744E-06	0.0174700
4096	3	2.441E-04	2.478E-04	3.635E-06	0.0244640

D.2 Results from Iterations Scaling Tests

Table D.5: Results of Function 1 on 1,024 Procs (Time in seconds)

Total Iterations	LB Steps to Convergence	Mean Load	Max Load	Load Difference	Total Execution Time
5000	2	9.766E-04	1.279E-03	3.026E-04	0.0162490
7500	2	9.766E-04	1.210E-03	2.336E-04	0.0137090
10000	2	9.766E-04	1.157E-03	1.802E-04	0.0130240
25000	2	9.766E-04	1.044E-03	6.788E-05	0.0130130
50000	2	9.766E-04	1.014E-03	3.747E-05	0.0140200
75000	3	9.766E-04	9.993E-04	2.278E-05	0.0151730
100000	2	9.766E-04	9.945E-04	1.797E-05	0.0144950
250000	2	9.766E-04	9.838E-04	7.197E-06	0.0122920
500000	2	9.766E-04	9.801E-04	3.557E-06	0.0184910
750000	3	9.766E-04	9.788E-04	2.197E-06	0.0278970
1000000	3	9.766E-04	9.783E-04	1.761E-06	0.0213230

Table D.6: Results of Function 2 on 1,024 Procs (Time in seconds)

Total Iterations	LB Steps to Convergence	Mean Load	Max Load	Load Difference	Total Execution Time
5000	2	9.766E-04	2.500E-01	2.490E-01	0.0141670
7500	2	9.766E-04	1.429E-01	1.419E-01	0.0097520
10000	2	9.766E-04	1.111E-01	1.101E-01	0.0100000
25000	2	9.766E-04	4.167E-02	4.069E-02	0.0109550
50000	2	9.766E-04	2.083E-02	1.986E-02	0.0088290
75000	2	9.766E-04	1.370E-02	1.272E-02	0.0088270
100000	2	9.766E-04	1.031E-02	9.333E-03	0.0084130
250000	2	9.766E-04	4.098E-03	3.122E-03	0.0143420
500000	2	9.766E-04	2.049E-03	1.073E-03	0.0249200
750000	2	9.766E-04	1.366E-03	3.896E-04	0.0335970
1000000	2	9.766E-04	1.025E-03	4.803E-05	0.0439330

Table D.7: Results of Function 3 on 1,024 Procs (Time in seconds)

Total Iterations	**LB Steps to Convergence**	**Mean Load**	**Max Load**	**Load Difference**	**Total Execution Time**
5000	2	9.766E-04	1.160E-03	1.832E-04	0.0263040
7500	2	9.766E-04	1.133E-03	1.569E-04	0.0145440
10000	2	9.766E-04	1.099E-03	1.229E-04	0.0232330
25000	2	9.766E-04	1.037E-03	6.050E-05	0.0156260
50000	2	9.766E-04	1.011E-03	3.417E-05	0.0146800
75000	2	9.766E-04	1.002E-03	2.497E-05	0.0121560
100000	2	9.766E-04	9.937E-04	1.713E-05	0.0131040
250000	3	9.766E-04	9.836E-04	7.029E-06	0.0119480
500000	4	9.766E-04	9.815E-04	4.927E-06	0.0183080
750000	9	9.766E-04	9.798E-04	3.216E-06	0.0187740
1000000	12	9.766E-04	9.784E-04	1.862E-06	0.0241070

Table D.8: Results of Function 4 on 1,024 Procs (Time in seconds)

Total Iterations	**LB Steps to Convergence**	**Mean Load**	**Max Load**	**Load Difference**	**Total Execution Time**
5000	1	9.766E-04	1.173E-03	1.966E-04	0.0139010
7500	3	9.766E-04	1.128E-03	1.514E-04	0.0164300
10000	4	9.766E-04	1.110E-03	1.330E-04	0.0186970
25000	3	9.766E-04	1.035E-03	5.881E-05	0.0236950
50000	3	9.766E-04	1.010E-03	3.363E-05	0.0160330
75000	2	9.766E-04	1.001E-03	2.420E-05	0.0129580
100000	2	9.766E-04	9.937E-04	1.713E-05	0.0179390
250000	3	9.766E-04	9.837E-04	7.131E-06	0.0192980
500000	4	9.766E-04	9.803E-04	3.735E-06	0.0177620
750000	9	9.766E-04	9.791E-04	2.498E-06	0.0205220
1000000	12	9.766E-04	9.785E-04	1.974E-06	0.0253470

D.3 Results from Non-Convergent Load Balancing Tests

Table D.9: Results of Function 3 Load Difference vs. LB Iteration on 500,000 Iters

LB Step Iteration	**8 Processors**	**16 Processors**	**32 Processors**	**64 Processors**
INITIAL	7.129E-03	4.327E-03	2.248E-03	8.976E-03
1	7.879E-03	7.132E-03	3.869E-03	7.219E-03
2	4.320E-03	3.380E-03	2.157E-03	3.214E-03
3	4.056E-03	3.177E-03	2.317E-03	4.510E-03
4	3.244E-03	2.724E-03	1.523E-03	3.152E-03
5	2.952E-03	1.982E-03	1.666E-03	5.201E-03
6	2.543E-03	2.206E-03	1.321E-03	4.179E-03
7	2.309E-03	1.449E-03	1.208E-03	5.460E-03
8	2.042E-03	1.783E-03	1.145E-03	4.388E-03
9	1.861E-03	1.224E-03	9.723E-04	5.482E-03
10	1.664E-03	1.444E-03	9.899E-04	4.354E-03
FINAL	4.501E-04	2.682E-04	3.016E-04	4.858E-03

Table D.10: Results of Function 4 Load Difference vs. LB Iteration on 500,000 Iters

LB Step Iteration	**8 Processors**	**16 Processors**	**32 Processors**	**64 Processors**
INITIAL	7.908E-03	5.073E-03	2.620E-03	8.998E-03
1	8.250E-03	7.512E-03	4.859E-03	9.703E-03
2	4.490E-03	3.542E-03	3.180E-03	3.161E-03
3	4.299E-03	4.502E-03	2.814E-03	5.036E-03
4	3.330E-03	2.822E-03	1.603E-03	3.312E-03
5	3.081E-03	3.046E-03	1.768E-03	5.178E-03
6	2.590E-03	2.151E-03	1.395E-03	4.036E-03
7	2.407E-03	2.179E-03	1.330E-03	5.229E-03
8	2.068E-03	1.631E-03	1.174E-03	4.169E-03
9	1.927E-03	1.611E-03	1.099E-03	5.130E-03
10	1.680E-03	1.240E-03	9.757E-04	4.755E-03
FINAL	4.563E-04	2.092E-04	2.451E-04	3.161E-03

D.4 Results from Data Simulation Tests

Table D.11: Results on 256 Processors and 5,000 Iterations (Time in seconds)

Data Size (# of Integers)	**LB Iteration 1 Time**	**LB Iteration 2 Time**	**Total Simulation Time**
0	0.0027700	0.0001650	0.0050300
5	0.0027740	0.0004070	0.0069340
10	0.0029250	0.0005400	0.0050760
50	0.0025980	0.0004200	0.0047380
100	0.0028540	0.0005350	0.0060210
500	0.0040010	0.0007480	0.0074700
1000	0.0041480	0.0007030	0.0068250

Table D.12: Results on 1,024 Processors and 5,000 Iterations (Time in seconds)

Data Size (# of Integers)	**LB Iteration 1 Time**	**LB Iteration 2 Time**	**Total Simulation Time**
0	0.0086390	0.0005010	0.0117870
5	0.0095830	0.0012240	0.0147190
10	0.0096510	0.0010640	0.0155680
50	0.0097170	0.0009990	0.0135200
100	0.0099750	0.0007390	0.0140810
500	0.0112560	0.0017590	0.0165030
1000	0.0126260	0.0016810	0.0176530