Feedback guided load balancing in a distributed memory environment

Constantinos Christofi

August 18, 2011

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2011
Abstract

Feedback guided dynamic load balancing is the process of equilibrating the load of all the processors of a system during execution time based on information gathered from them. It provides better execution times by better utilizing the systems resources.

In this work we have suggested a new dynamic load balancing model, named Wave Propagation Model, that is based on the diffusion balancing method. The suggested model was implemented in a C language library, the DLBLib, which offers to the programmer the ability to add dynamic load balancing to his applications with minimum effort.

Simulations and benchmarks executed using the DLBLib showed that the development of this library can be beneficial for parallel applications as it achieves good performance with minimum cost. It will allow programmers to easily load balance their applications whilst not adding complexity to their codes.
Contents

1 Introduction 1
  1.1 Dynamic Load Balancing basics 3
  1.2 Aims and Objectives 3
  1.3 Organisation of the Dissertation 4

2 Background theory 5
  2.1 Load Balancing Models 5
     2.1.1 The Gradient Model 5
     2.1.2 The Diffusion Model 6
     2.1.3 The Dimension Exchange Model 8
     2.1.4 The Hierarchical Balancing Model 9
  2.2 Load Balancing Models Comparison 10
     2.2.1 Balancing Initiation 10
     2.2.2 Balancing Domains 11
     2.2.3 Degree of Knowledge Required 11
     2.2.4 Aging of Information 12
     2.2.5 Overhead Distribution and Complexity 12

3 The Wave Propagation Model 13
  3.1 Requirements for the Wave Propagation Model 13
  3.2 Illustration of the model 14
  3.3 Wave Propagation Algorithm 15
  3.4 Wave Propagation Model Vs Former Models 16

4 Implementation of Dynamic Load Balancing Library (DLBLib) 17
  4.1 DLBLib Interface 18
     4.1.1 DLBL_Init 18
     4.1.2 DLBL_Start_load_execution 18
     4.1.3 DLBL_End_load_execution 19
     4.1.4 DLBLMalloc 19
     4.1.5 DLBL_Finalize 20
     4.1.6 DLBL_Finalize 20
     4.1.7 DLBL_Finalize 21
     4.1.8 DLBL_Finalize 21
     4.1.9 DLBL_Finalize 22
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Comparison Analysis of Balancing Models</td>
<td>10</td>
</tr>
<tr>
<td>4.1</td>
<td>Partial summary of DLBLib interface.</td>
<td>21</td>
</tr>
<tr>
<td>6.1</td>
<td>Overview of the tests conducted for the Static Total Load scenario.</td>
<td>37</td>
</tr>
<tr>
<td>6.2</td>
<td>Overview of the tests conducted for evaluating the Scalability of DLBLib for 4096 byte load unit.</td>
<td>44</td>
</tr>
<tr>
<td>6.3</td>
<td>Overview of the tests conducted for evaluating the Scalability of DLBLib for 409600 byte load unit.</td>
<td>45</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Load Balancing Process ........................................ 2
2.1 Gradient Surface ............................................... 6
2.2 Sender Initiated Diffusion .................................... 7
2.3 Dimension Exchange Model ................................... 8
2.4 Hierarchical Balancing Model ................................. 9
3.1 The behavior of the Wave Propagation Model ............... 15
4.1 A pseudocode example of user defined pack function. ... 25
4.2 A pseudocode example of user defined unpack function. .. 26
4.3 The typical layout of a DLBLib program. ..................... 27
4.4 Example application Dynamically load balanced using DLBLib. ... 28
6.1 Processor Load over Iterations in 8 Processor System ... 34
6.2 Execution Time over Iteration ................................. 36
6.3 Load Balancing Process in 8 Processor System ............. 38
6.4 Load Balancing Process in 128 Processor System .......... 39
6.5 Load Balancing Process in 512 Processor System .......... 40
6.6 Load Balancing Process in a dynamic load environment. ... 41
6.7 Load Balancing Process in worst case scenario 1. ........ 42
6.8 Load Balancing Process in worst case scenario 2. ......... 43
6.9 Load Balancing Cost over System Size for a small dataset. ... 45
6.10 Load Balancing Cost over System Size for a large dataset. ... 46
Acknowledgements

I would like to seize the opportunity to thank all the lectures of the MSc in High Performance Computing program for their assistance over the year. Many of the topics covered by the courses during the last year at EPCC have helped me towards the completion of this project.

I would like to thank my parents Christakis and Niki Christofi for their continued support throughout my education.

I would like to thank my flatmates and friends Andreas Iacovou and Charis Ioannou for offering their time to help me during the whole year. Also a special thanks to my fellow classmates for providing me with good excuses for breaks during the course of this project.

Finally, i would like to siencerely thank my project supervisor Dr Lawrence Mitchell for his support and guidance. He showed confidence in me from the start and helped me throughout the project.
Chapter 1

Introduction

Advances in hardware and software techniques have led to increased interest in the use of massively parallel systems for real-time and large scale scientific and commercial applications[30]. One of the key issues in such systems is the design and development of efficient techniques that can effectively distribute the load of these applications evenly across the multiple processors of a parallel system [3]. When an application is divided into a number of processes that will run in parallel each processor performs a certain amount of work associated with one process. There are the cases where some processors finish their work faster than others either because they have less work to do or because they perform faster than others. As a result they become idle waiting for heavier loaded or slower processors to finish execution. In order to achieve maximum performance from the system all the processors should be operating on equal amounts of work that will lead to small idle time and minimize the total execution time[4]. The effort to assign the work evenly across the available processors is called load balancing.

In applications with constant workloads, such as dense linear systems solvers, it is possible for the designer of the application to estimate the work load of the problem and distribute it among the available processors during the implementation phase. Such a load balancing is called static load balancing[1]. By contrast other applications, such as adaptive finite element codes, have varying workloads that can not be compute prior to the execution of the application. As the execution of these applications evolves the load of the processors tends to vary. The techniques for balancing these variations in the load of the processors during execution time is called dynamic load balancing[1].

Recent increases in the numbers of processors available in the parallel computing systems demand the use of scalable dynamic load balancing algorithms. Thus this work concentrates on the design of a distributed algorithm. We avoid centralized methods due to the fact that the scheduler, of these methods, often forms a bottleneck in systems with large number of processors. Many approaches have already been studied and general methods of load balancing are given in [1]. At a higher level we can distinguished between global and local load balancing algorithms. With a global load balancing method
one processor can exchange load with any other processor in the system. The load is sent through the routing network of the system. Several global distributed methods have already been implemented but they all have the drawback that as the number of processors increases the global communication becomes a bottleneck[18]. On the other hand the local approaches seem more suitable for massively parallel systems with tens of thousands of processors. By using a local load balancing algorithm a processor can exchange load only with its directly connected neighbors. These methods although they reduce the overhead of the global communications have the drawback that they need more steps in order to converge to a solution[24] [1]. A successful load balancing algorithm must be able to have global knowledge, for the load on all the processors, but limit the load exchange between directly connected neighbors. The load must be exchanged asynchronously in such a way that can balance all the processors in a relative small number of steps. This way the load balancer will be able to minimize the total execution time of the application by introducing the less overheads possible.

For the load balancing process we assume that all the load in the system can be represented by distinct load units. Each load unit consists of the data that have to be processed by a processor. Load units can migrate between processors and the final result is not affected by that migration.

For the application process we assume that the total load of the system is varying over time. A processor's load can increase or decrease dynamically during the execution time without following a specific pattern.
1.1 Dynamic Load Balancing basics

From an application developer’s perspective, dynamic load balancing is pretty straightforward. The optimal load distribution cannot always be calculated prior to executing the application, so a dynamic load balancer can offer you the ability to spread the work amongst several processors and keep them evenly loaded throughout the whole execution.

To accomplish this simple task, we need a few things. First off, we need to know the total amount of work available in the application at a given time as well as the fraction of that work that is assigned to each of the processors available in the system. This information will help us to classify all the processors, based on the amount of the work that they are assigned with, as overloaded or underloaded. Next, we need a load balancing algorithm. This algorithm describes how the available work is going to be redistributed to the processors of the system in a way that will result in an optimal distribution. There are various models for dynamic load balancing in a multiprocessor environment. The most common of them are described in Chapter 2. Finally, we need a mechanism for moving the work between the processors. For efficient load balancing we need that mechanism, especially in a distributed memory environment, to have low latency (low set-up cost) and high bandwidth. In the absence of a data moving mechanism with these characteristics the load balancer can be a bottleneck and even introduce additional overheads to the total execution time of the application.

1.2 Aims and Objectives

The purpose of this work is to address the problem of the dynamic load balancing in distributed memory environments by suggesting a new load balancing model suitable of solving the problem efficiently and developing a library that implements the model in order to simplify its use in real scientific codes. The objectives of this work are presented by the following tasks:

- Suggestion of a feedback guided dynamic load balancing model that can be used efficiently in distributed memory environments.

- Development of a library with interface for the C programming language which implements the suggested model. The library should make the task of the dynamic load balancing of scientific codes, existing or new, trivial.

- Benchmark the dynamic load balancing library in order to verify its correctness and its performance with various application types and finally test its scalability.
1.3 Organisation of the Dissertation

Chapter 2 will review some of the basic load balancing models and provide some background theory about them. A set of matrices for evaluating and classifying the balancing models is also discussed in this chapter. The proposed model for addressing the dynamic load balancing problem is introduced and explained in Chapter 3. The implementation of the proposed model into a load balancing library called DLBLib is covered by Chapter 4. The same chapter also includes a sample application load balanced using the DLBLib library. In Chapter 5 we describe the benchmarking application developed for testing the correctness and efficiency of the DLBLib library. We also include a brief description of HECToR, UK national high performance service, used for running the experiments. The results themselves are presented and discussed in detail in Chapter 6. This work concludes with its findings and conclusions in Chapter 7 before outlining potential future work in Chapter 8.
Chapter 2

Background theory

The problem of Load Balancing dates back to the first parallel computer. Ever since, there is a constant need for more powerful hardware that will minimize the execution time of our applications [3]. Physical and economical factors restrict the design and manufacture of faster and faster computers. This leads us to the need to find ways to maximize the performance of the available hardware. One way, is to better utilize the hardware in order to minimize the time that is idle and subsequently increase its performance.

The aim of dynamic load balancing is to minimize the execution time of a parallel program by better controlling the use of computational and communicational resources. All parallel programs can be represented as tasks (load units) that can be executed in parallel by any one of the available processors of the system. The final outcome of the program must not be affected by the execution sequence of the tasks. That is the idea on which the dynamic load balancing is based on. We can migrate some of these load units, from heavily loaded processors to under utilized processors, in an effort to minimize the total execution time of the parallel program.

In the next section we present and describe some of the most known models used for dynamic load balancing. In section 2.2 we explain the basic features that characterize and categorize the load balancing models presented in section 2.1.

2.1 Load Balancing Models

2.1.1 The Gradient Model

Lin and Keller in [2] suggested a method of dynamic load balancing based on the Gradient Model which transfers excessive load from heavy loaded processors to nearby idle processors. The Gradient Model is distributed and fully asynchronous.
Firstly, every processor determines its load condition as light thus it wishes to receive more load, heavy thus it wishes to unload some of its load and moderate thus it does not wishes to sent or receive any load. Then each processor determines its proximity value using the following template: The proximity of a light processor is set to zero, the proximity of a processor neighboring a light processor in one, the proximity of the neighbor’s neighbor is two etc. These processor proximity values are later used to construct a Gradient Surface (Shown in Figure 2.1) which serves as a minimum distance “map” from heavy loaded to light loaded processors. During a load balancing iteration excessive load from heavily loaded processors migrates to the nearest neighbor with the least proximity value. After each iteration of the method every processor redefines its load condition, proximity value and a new Gradient Surface is constructed from the new values. Global load balance is achieved by successive iterations of the model until all the processors are moderately loaded.

2.1.2 The Diffusion Model

An other well known and studied load balancing model is the so-called diffusion load balancing. The model is described and evaluated by Cybenko in [3] and Boillat in [6]. Diffusion load balancing is a synchronised algorithm which works in rounds (i.e iterations). The basic idea is that in each round every processor balances its load with all or some of it’s neighbors[19]. This neighborly balancing occurs independently (pair-wise) between the processors without the need of a master/coordinator processor. Global load balancing is achieved by a number of subsequent rounds of the algorithm and result in an even distribution of the load among the processors.

Over the years the algorithm was studied for both static, where each processor was

![Gradient Surface](Image)
assigned with an initial load units and the objective is to evenly distribute them among all the processors as fast as possible, and dynamic, where the total number of load units varies over time, scenarios. A number of different load migration policies were studied for the number of load units that a processor is allowed to transfer to a neighbor. The scenario with the assumption that only one load unit is allowed to be forwarded by a processor to each one of it’s neighbors [10, 11, 12, 13] and the scenario where only a constant number of load units can be passed by each processor [14] are referred as token distribution problems. In addition the algorithm was studied with the scenario that the load units can be split arbitrary between processors [15, 16, 17].

Among the various diffusion strategies available the ones widely used are Sender Initiated Diffusion and the Receiver Initiated Diffusion[4]. These two techniques are presented and described in the following sections.

**Sender Initiated Diffusion (SID)**

SID is a local technique where a highly loaded processor acts independently and “unloads” some of its excess load units to its underloaded neighbors. Each processor is responsible of sending a load update message to its immediate neighbors informing them about its assigned load. Every time a processor receives a load update message from a neighbor, it sends some of its excess load units to that processor only if its load is under a specific preset threshold. In this way each processor’s load information is limited only within its own domain (i.e. itself and its immediate neighbors). The global balance is achieved by the employment of the algorithm to overlapping balancing domains. The application of the model is shown in Figure 2.2.

![Sender Initiated Diffusion](image.png)

**Figure 2.2: Sender Initiated Diffusion**

In this example of the SID algorithm excess load from the red processor is sent to neighboring underloaded processors.
Receiver Initiated Diffusion (RID)

RID is a strategy that can be viewed as a converse approach with SID. In RID it is the receiver (underloaded) processors that initialises the balancing as opposed to SID. Every processor that its load is below a preset threshold sends to all its immediate neighbors a load request message. When a neighbor receives such a message it will fulfill the request with some load units (up to half its own load). With RID approach as with SID the load information required by a processor is limited in its balancing domain and the global balancing is achieved by overlapping the balancing domains of all the processors. As opposed to SID with RID the majority of the balancing overhead is assigned to the underloaded processors. This characteristic of the algorithm removes some of the additional overhead from the highly loaded processors reducing the effect of the balancing to the total execution time.

2.1.3 The Dimension Exchange Model

The Dimension Exchange Model is a fully synchronous local load balancing approach described in [3],[4],[5] and [19]. The model was originally designed for systems with hypercube topologies but it can be applied to other topologies too with some modifications. The model works by “folding” an N processor hypercube into logN dimensions. The balancing is performed at one of the logN dimensions at a time. All the processor pairs in the first dimension check in-order their load and balance it between themselves[18]. Then all the processors balance, in-order, their load between themselves in the second dimension and so on. The process is repeated to all the dimensions of the hypercube until every processor has balanced its load with each of its neighbors. Global system load balancing is ensured by the synchronous in-order balancing of each dimension. An example of a cube topology balanced with the Dimension Exchange Model is shown in Figure 2.3.

![Figure 2.3: Dimension Exchange Model](image)

*In the Dimension Exchange Model the processors balance in order in each dimension. In the cube of the example the balancing is performed in each dimension (a), (b) and (c) in the order shown due to synchronisation requirements.*
2.1.4 The Hierarchical Balancing Model

Another well known asynchronous load balancing strategy is the Hierarchical Balancing Method. HBM organises the multiprocessor system into a hierarchical balancing domain. This hierarchy is usually a binary tree and the balancing occurs at the different levels of the tree distributing the responsibility of the balancing to all the nodes. Each non-leaf processor is responsible for the balancing of the two sub-trees that have that processor as root. All the processors of the system send load information, about their lower level processors, upwards in the tree. If a processor detects load imbalance between its two sub-trees that is greater that a preset threshold then the two balancing sub-domains (left and right) are considered imbalanced. The root processor of the imbalanced sub-domains is responsible to notify the processors at the lower levels of the overloaded tree about the amount of the load units that each one has to move to the underloaded sub-domain. Every one of the overloaded processors has to move the designated amount of load to its matching underloaded processor. The pairs of the matching processors in the two sub-domains are predetermined.

The hierarchical scheme of this models (shown in Figure 2.4) allows it to distribute the load balancing responsibilities and overheads to all the processors of the system. This reduces the amount of communications required for the application of the algorithm compared with non-hierarchical models. Also it provides us with the ability to apply different imbalance thresholds to each of the hierarchy levels to further minimize the amount of load migration. These characteristics make the HBM a suitable balancing model for large systems[30] [4].

![Hierarchical Balancing Model Diagram](image)

Figure 2.4: Hierarchical Balancing Model
Hierarchical organization of an eight processor system arrange in a ring topology. The intermediate processors in the tree are the processors responsible to manage the balancing of their sub-domains.
2.2 Load Balancing Models Comparison

The majority of the load balancing models available today are defined by a set of attributes that makes them suitable for specific types of applications or specific types of architectures. For this work we choose a set of five attributes to classify the models examined in the previous sections of the chapter. These attributes are presented and explained in the next sections. Table 2.1 shows a comparison of the presented models based on these attributes.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>GM</th>
<th>SID</th>
<th>RID</th>
<th>HBM</th>
<th>DEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiation</td>
<td>Receiver</td>
<td>Sender</td>
<td>Receiver</td>
<td>Designated</td>
<td>Designated</td>
</tr>
<tr>
<td>Balancing Domain</td>
<td>Variable</td>
<td>Overlapped</td>
<td>Overlapped</td>
<td>Variable</td>
<td>Variable</td>
</tr>
<tr>
<td>Degree of Knowledge</td>
<td>Global*</td>
<td>Local</td>
<td>Local</td>
<td>Global</td>
<td>Global</td>
</tr>
<tr>
<td>Aging Period</td>
<td>$O(\text{diameter}(N))$</td>
<td>$f(u, K)$</td>
<td>$f(u, K)$</td>
<td>$f(u, N)$</td>
<td>constant</td>
</tr>
<tr>
<td>Overhead Distribution</td>
<td>uniform**</td>
<td>uniform</td>
<td>uniform</td>
<td>nonuniform</td>
<td>uniform</td>
</tr>
</tbody>
</table>

$N$ is the number of processors, $u$ is the update factor and $K$ is the number of near-neighbors. $f$ is a function combining the two factors. $\text{diameter}(N)$ is the maximum number of hops between any two processors of the system.

* global knowledge of the location of underloaded processors.
** statistically uniform for a random load distribution.

The table is a reproduction from [4].

Table 2.1: Comparison Analysis of Balancing Models

2.2.1 Balancing Initiation

Balancing Initiation is an important attribute for a dynamic load balancing model as it specifies which processor is responsible of detecting the load imbalance and therefore triggering the balancing algorithm. This attribute may affect the load balancing algorithm in terms of implementation complexity, robustness, and overheads introduced. A load balancing algorithm can be sender initiated, receiver initiated or specified initiated. For a sender initiated algorithm the overloaded processor that wants to send some of its excess load away initiates the load balancing sequence. Similarly for a receiver initiated algorithm the underloaded processor that wants to be assigned with some extra load initiates the load balancing by requesting some load from the overloaded processors. Finally the balancing of a specified initiated algorithm is triggered by predefined processors that are responsible for monitoring the loads of a number of processors and when they detect a load imbalance they initiate the load balancing.

Mark Willebeek-LeMair in [4] tested the sender initiated and the receiver initiated balancing approaches in a model that assumes a fixed number of tasks (i.e. load units) present in the system at initialisation time and performance was measured as the total time required to complete all the tasks. His results show that a receiver initiated algorithm outperforms the sender initiated algorithm. The results were supported by
the theory as with the receiver initiated balancing the task of detecting the imbalance, which requires processor time, is assigned to the underloaded processors that would be idle waiting the overloaded ones to finish executing their load. In this way we minimize the overheads introduced to the total execution time.

### 2.2.2 Balancing Domains

Balancing Domains are used in load balancing algorithms to decentralize the balancing process and to decrease the complexity of the algorithm by minimizing its scope [21] [4]. A Balancing Domain is in practice the subset of the processors of the system that the load balancing algorithm is applied on at a single step. The balancing models can be divided into two categories based on their Balancing Domains. The first category is the one with the *overlapping domains* which consists of the processor initiating the balancing and a small set of surrounding processors with which it balances its load units. Global balancing is achieved through the overlap of these domains and the diffusion of excess load units through the Balancing Domains across the whole network. The other category is the one with the *variable domains* which change their shape as different processors are part of these domains at subsequent balancing iterations. Global balancing is achieved by changing the shape of the domain to accommodate larger numbers of processors at different levels of the balancing process.

Shin and Chang showed in [21] that load balancing algorithms continue to benefit from larger and larger balancing domains up to a maximum size. This size is determine by characteristics of the system like the size or the topology of it. Beyond that maximum size the algorithm shows negative performance as the size of the balancing domain increases. This shows us that in the case of the overlapping domains we have to choose wisely the number of the processors that are included in a domain.

### 2.2.3 Degree of Knowledge Required

The Degree of Knowledge Required by a load balancing model is a significant attribute of the model as it is directly connected with the accuracy of the balancing decisions the algorithm makes in order to achieve global load balancing [4]. In practice the degree of knowledge for a processor is the number of the neighboring processors that it has information about the amount of their load. All the balancing models can be categorised based on their knowledge as *global*, i.e. the algorithm has information about the load of all the processors in the system or knows the positions of all the underloaded processors, and as *local*, i.e. the algorithm’s load information is limited to a small number of neighboring processors for every processor of the system.

Although global knowledge is beneficial for the load balancing algorithm’s decisions there is also the trade off of the additional communication overheads that they are introduced in the effort to make the global knowledge available to all the processors of the
system. On the other hand if we limit the knowledge of the algorithm to the $k$ nearest neighbors we reduce significantly the amount of communication required thus reducing the implied overhead of the load balancing algorithm.

2.2.4 Aging of Information

Another attribute of a load balancing model that plays a crucial role to the degree of accuracy of its decisions is the attribute of the Aging of Information. In practice the term Aging of Information specifies the length of the delay from the time that the load information of the system and/or process was acquired to the time it is used by the algorithm for making a load balancing decision [29]. In cases where the load of the system is changing rapidly the load information is valid only for a short period of time and if the load balancer makes a decision based on aged information its effectiveness might be affected.

The load balancing models can be categorised by their degree of aging of the information into two groups. For the first category the aging is depended by the number of processors or the topology of the system. For larger balancing domains we have additional delays from the collection of the load information to the time that it is used. The aging of information becomes a negative factor for the accuracy of the algorithm as the size of the domain increases. In this category we place the models that their aging period is affected by the number of processors that are member of the balancing domain or the number of levels in a hierarchical balancing domain. We also include the models that their information aging is affected by the number of maximum hops between any two processors in the system. In the second category we place the models that their aging period is constant due to their synchronous operation.

2.2.5 Overhead Distribution and Complexity

Parallel applications use load balancing algorithms in an effort to reduce their execution time. Load balancing requires processor cycles and therefore some additional processing is added to the parallel application. It is desirable for the models used for these balancing algorithms to be able to minimize the load balancing overhead on the total execution time by having low complexity and to be able to distribute that overhead as evenly as possible across all the processors of the system. This eliminates the risks of any bottlenecks introduced by the balancing process and ensures that the balancing overheads will not increase dramatically the total execution time.

Some models show the desirable behavior by achieving a uniform overhead distribution that is independent of the number of processors in the system and affected only by the number of the neighbors of each processor. Some other schemes on the other hand are forced to a nonuniform overhead distribution by mostly their varying size balancing domains.
Chapter 3

The Wave Propagation Model

Careful investigation of the load balancing models presented in Chapter 2 revealed some of their positive and negative attributes for their use towards the fulfillment of the aims of this work. The problem of the dynamic load balancing for massive parallel systems had to be addressed. A new model for dynamic load balancing for distributed memory environments which simulates the behavior of a wave while propagating through a liquid is suggested. The model relies on the existence of a low set-up to transfer ratio communication mechanism, found in most of the High Performance Computing systems, to shift an overloaded processor’s excess load units to its two directly connected neighbors. By successive applications of the algorithm the excess load is automatically transferred to lightly loaded processors. The suggested model can be viewed as a kind of Sender Initiative Diffusion with some changes that make it unique.

In the next section, we explain some of the requirements that shaped the suggested model to its final form. These requirements are of high importance for the successful use of the model. In Section 3.2 the model is illustrated using a physical world analogy of a liquid equilibrating its surface. The balancing algorithm is presented in Section 3.3 and finally in Section 3.4 we present the basic differences of the suggested model with the previously suggested ones.

3.1 Requirements for the Wave Propagation Model

The problem that the suggested model has to solve can be divided into two tasks, load sharing and dynamic load balancing. The task of load sharing is to supply all the processors of the system with at least some load. Dynamic load balancing is the task of first equilibrating the load as even as possible among the processors and then to keep all the processors evenly loaded throughout the whole execution of the application.

The number of processors in modern HPC systems increases quickly which results to the need of scalable balancing algorithms. This need for scalability dictates us that
the proposed model must be decentralised providing the ability to distribute the algorithm’s overhead across all the processors of the system in an effort to minimize the risk of increasing the total execution time of our application due to a bottleneck. Kumar [22] showed with experiments the need of such decentralised balancing algorithms and suggested schemes for different architectures such as hypercube, mesh, and network of workstations.

The model must limit its load migration pattern at the nearest neighbors base while collecting global knowledge. Global knowledge is collected in the form of the total load of the system. Global communications tend to be expensive in time due mostly to the synchronization that they imply. The general drawback of global models, where one processor may transfer load to any other processor in the system, is that as the number of processors increase global communications tend to slow down the algorithm. Thus, for this work a local communication approach seems more applicable although it requires more iterations to equilibrate the load.

Dynamic load balancing algorithms base the precision of their load migration decisions on load information that is collected from the processors of the system. As a consequence the load information collected can become worthless for the balancer as it ages. The older the information used for a decision the more likely the load configuration has changed in the meantime. To address this problem a synchronous approach can be used for the balancing algorithm where the load migration decision are made immediately after the load information is collected and before the load configuration can change.

We assume that the load of the applications consists of independent load units that can migrate freely to any processor in the system. This load units include the data of the computation and can be executed, in any order possible without that effecting the final result of the application. We also assume that the amount of load increases and decreases dynamically in time either by the creation or destruction of load units, or by the variation in the execution time that each individual load unit require. This dynamical behavior of the load is not predictable, thus, load information must be collected in real time with the lowest cost possible. In this work we suggest the use of a high precision timing function to measure the load, in execution time, of each processor. For simplicity we assume that all load units need the same time to execute.

3.2 Illustration of the model

The proposed model can by illustrated by the physical effect that a propagating wave has on the surface of a liquid. For this illustration a flat container is filled with a homogeneous liquid. The container is such that the liquid in its balanced state has the same height, from the bottom of the container, across the whole surface. If somehow additional liquid is collected to a point of the surface, increasing its height form the bottom, a wave will be created in the attempt of the liquid to equalise it self so the height is
again the same everywhere. The equalisation happens while the wave propagates the superfluous liquid to the neighboring points of the liquid. The wave can travel several times back and forth across the whole surface of the liquid until all points have the same height again. The global equalisation is achieved by strictly local liquid movements as none of the additional liquid will move to locations with lacking liquid by skipping some points of the surface. The discrete equivalent of the model is shown in Figure 3.1. The liquid in the wave propagation model corresponds to the load of the process. The number of the liquid molecules represent the load units assigned to each processor.

Figure 3.1: The behavior of the Wave Propagation Model
The surface of the liquid is discretised into boxes, each one representing a processor, and the liquid molecules, shown with the red circles, represent the load units of the application. The balancing steps are shown by the numbers on the right.

### 3.3 Wave Propagation Algorithm

If there are some heavily loaded processors and some lightly processors in the system, the excess load from the heavily loaded ones should be transferred to the lightly loaded ones. With the wave propagation model the load is transfered implicitly to the correct
processors with any knowledge of the load of other processors following the next
algorithm. Each processor informs all the other processors about its own load. The
loads of all the processors are locally added so each processor is aware about the total
load of the system. While there are processors with higher load that the average load of
the system they shift all their excess load to their left and right processors. In the same
time they receive load from these processors if they are also highly loaded. In the case
that the excess load unit of a processor is only one then that load unit is migrating always
to the right neighbor. This is important as by keeping this consistent we can guarantee
that the excess load will reach an underloaded processor eventually. The lightly loaded
processors only receive load units from their neighbors without giving any away. By
successive application of this algorithm the load is transferred to processors with less
load than others resulting to global load balancing. If there are no more underloaded
processors in the system the load balance process is done. The total load is continuously
monitored and if a new imbalance is detected the process begins again.

3.4 Wave Propagation Model Vs Former Models

The Wave Propagation Model is based on the successful model of Sender Initiated Diff-
fusion, but with some customisations that might enhance its performance. Through the
examination of the widely used models, presented in Chapter 2, a number of desirable
properties, from other models, were merged into the diffusion model.

The main difference between the SID model and the suggested is the amount of load
units shifted in every step of the algorithm. With SID only a constant fraction of the
load of each processor can migrate to its neighbors. With the wave propagation model
a processor, with excess load, shifts in one step all the load units, exceeding the average
load of the system, to its two directly connected neighbors. This feature gives the model
a fast start behavior as in the first steps of the algorithm the changes are more drastic.
The suggested model makes no assumptions for the topology of network connecting
the processors. It relies on the underlying message passing library of the machine to
determine the left and right neighbor for every processor. The migration of the load
units is then limited to these two neighbors as opposed to former models that work on
hypercubes or torus. This benefits the portability of the implemented library as it can
work with any given network topology and also minimize the aging of the load infor-
mation obtained. Another important feature of the model is that it uses the time that
each processor needed to execute its assigned work to determine its load. This differs
from other models that use the number of processor cycles, or other dedicated hardware
counters, that are inaccurate in the case of heterogeneous systems.
Chapter 4

Implementation of Dynamic Load Balancing Library (DLBLib)

As stated in the aims and objectives of this work we developed a library that can automate the dynamic load balancing process for user applications. The library, DLBLib, consist of a C language interface through which the wave propagation model is used for the load balancing process. To use the library a user must first implement a pack and an unpack function. Through the pack function, the user must be able to pack a number of load units in a contiguous buffer, so they can be sent by the library to another process during the balancing process. Similarly through the unpack function the contiguous buffer, received from another process, must be unpacked and its data must be merged with the data of the application.

DLBLib is implemented in the C programing language. C was chosen because of its high performance, portability, and the specific characteristics it exhibits, like:

- Low-level access to memory by converting machine addresses to pointers.
- Function and data pointers supporting some kind of run-time polymorphism.
- Array indexing defined in terms of pointer arithmetics.
- Complex functionality such as I/O, string manipulation, and mathematical functions.
- A large number of compound operators, such as +=, -=, *=, ++, etc.

C is also the language of choice for the majority of scientific applications making it available in all HPC systems. DLBLib was design and implemented so it takes advantage of the SHMEM interface. SHMEM is a high efficient single sided communication library available on Cray HPC systems like HECToR[31]. SHMEM as well as HECToR are discussed in Chapter 5.

The interface of the DLBLib is presented and explained in Section 4.1. Later in Section
4.2 we present in brief a guide on how the library can be used for the dynamic load balancing of an application. Finally in Section 4.3 we present a small example application that is load balanced with DLBLib.

4.1 DLBLib Interface

Here we present the callable interface of the DLBLib. The interface is a set of functions that the user can call from inside the user application. The minimum set of these functions required for the load balancing process are also briefly described in Table 4.1.

4.1.1 DLB_Init

The DLBLib library is initiated through the call of the DLB_Init function. The function can be called only once, by all the processes of the system, and must be the first call to the DLBLib interface made by the main program. DLBLib functions called prior to DLB_Init will not execute correctly. The DLB_Init function starts up virtual processing elements, which are processes that participate in shared memory (Cray SHMEM) operations, as well as, allocates memory from the symmetric heap needed by the library. Finally it constructs one virtual ring network on which it places all the processors of the system. This also determines the left and right neighbor of each processor. Programs that use both MPI and DLBLib should call MPI_Init followed by DLB_Init. At the end of the program, DLB_Finalize should be called followed by MPI_Finalize.

SYNOPSIS

```c
void DLB_Init(int my_pe, int num_pes, int num_units)
```

INPUT PARAMETERS

`num_units`: The number of the load units available in the application at start-up.

OUTPUT PARAMETERS

`my_pe`: The id of calling the process as created by the SHMEM library.
`num_pes`: The sum of all the processes running the application.

4.1.2 DLB_Start_load_execution

The DLB_Start_load_execution function is used for starting the high precision timer used by the DLBLib to calculate the load of the calling process. The user must call the function, through his code, at the point that the load begins to execute. Calling the function at a wrong point will result in inaccurate load information.

SYNOPSIS

```c
void DLB_Start_load_execution()
```
4.1.3 DLB_End_load_execution

The DLB_End_load_execution function stops the high precision timer of the DLBLib and calculates the load of the process in execution time. After calculating the load, the function updates the global load information of the system. The user must call the function at the point of his code where the execution of the load ends. Calling the function at a wrong point will result in inaccurate load information. The call to the function implies a barrier among all calling processes.

SYNOPSIS

void DLB_End_load_execution()

INPUT PARAMETERS
none

OUTPUT PARAMETERS
none

4.1.4 DLBMalloc

The DLBMalloc function returns a pointer to a block of at least size bytes suitably aligned for any use. This space is allocated from a special part of the heap which is addresses with the same address range on all the processors (in contrast to malloc(), which allocates from the private heap). The same address is returned for the same memory block on all calling processes. The space returned is left uninitialized. The function must be used by the user during the implementation of the pack function for the dynamic allocation of the buffer containing the packed data. If malloc() is used instead the DLBLib will not function correctly.

SYNOPSIS

void* DLBMalloc(int size)

INPUT PARAMETERS
size: The minimum size of the allocated memory block.

OUTPUT PARAMETERS
none
4.1.5 DLB_Free

The DLB_Free function causes a block of memory pointed by a pointer be deallocated, that is, made available for further allocation. If the provided pointer a null pointer, no action occurs; otherwise, if the argument does not match a pointer earlier returned by the DLB_Malloc function, or if the space has already been deallocated, DLB_Free returns.

SYNOPSIS

void DLB_Free(void *ptr)

INPUT PARAMETERS

*ptr: Pointer to the memory block to be deallocated.

OUTPUT PARAMETERS

none

4.1.6 DLB_Balance

The main balancing process is performed automatically through the DLB_Balance function. When called, the function calculates the load state of the processor based on the variation from the average load and a tolerance parameter, specified by the user. After classifying the processor, as heavily loaded or not, the function calculates if and how many load units have to be moved. Then, by calling the user specified pack function, it packs the excess load units of that processor into two symmetrically allocated buffers ready to be send to the left and right neighbors. Then by invoking the SHMEM interface it exchanges the processors excess load units, packed in the buffers, with data from the left and right neighbor. At the end the function calls the user specified unpack function in order to unpack the received buffers and append the load units contained to the load of the processor. DLB_Balance can be called as often as required by the user. Each call to the function results to the application of one iteration of the balancing algorithm. Load balancing using the DLBLib is possible only after collecting load data using the DLB_Start_load_execution and DLB_End_load_execution functions.

SYNOPSIS

void DLB_Balance(void *pack, void *unpack, void *processor_data, float *tolerance)

INPUT PARAMETERS

pack: The user specified function that packs the data that need to be moved to an other process.
unpack: The user specified function that unpacks the data received from an other process.
processor_data: The data structure that holds the data of the calling process.
tolerance: The tolerance of the balancer.
**4.1.7 DLB_Finalize**

DLBLib library terminates through the call to the `DLB_Finalize` function. The function can be called only once, by all the processes of the system, and must be the last call to the DLBLib interface made by the main program. DLBLib functions called later to `DLB_Finalize` will not execute correctly. The function cleans up resources allocated for the library by a processor. Programs that use both MPI and DLBLib at the end should call the `DLB_Finalize` function first, followed by `MPI_Finalize`.

**SYNOPSIS**

```c
void DLB_Finalize()
```

**INPUT PARAMETERS**

none

**OUTPUT PARAMETERS**

none

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DLB_Init(my_pe, num_pes, num_units)</code></td>
<td>Initialises the Dynamic Load Balancing Library and allocates all the required memory for the library.</td>
</tr>
<tr>
<td><code>DLB_Start_load_execution()</code></td>
<td>Starts the load measuring timer of the library.</td>
</tr>
<tr>
<td><code>DLB_End_load_execution()</code></td>
<td>Stops the load measuring timer of the library and updates the applications total amount of load.</td>
</tr>
<tr>
<td><code>DLB_Malloc(size_in_bytes)</code></td>
<td>Allocates dynamically a buffer size <code>size_in_bytes</code> in the symmetric heap and returns its address.</td>
</tr>
<tr>
<td><code>DLB_Free(ptr)</code></td>
<td>Deallocates a memory block which was symmetrically allocated using <code>DLB_Malloc()</code>.</td>
</tr>
<tr>
<td><code>DLB_Balance(pack, unpack, load, tolerance)</code></td>
<td>Load balances all the calling processes based on the average load of the system.</td>
</tr>
<tr>
<td><code>DLB_Finalize()</code></td>
<td>Finalizes the Dynamic Load Balancing Library and deallocates all the allocated memory.</td>
</tr>
</tbody>
</table>

The minimum set of functions needed for the use of the DLBLib.

Table 4.1: Partial summary of DLBLib interface.

**4.1.8 DLB_Calculate_neighbors**

`DLB_Calculate_neighbors` function places the processors in a virtual ring topology and calculates the left and right neighbors of each processor based on that topology. This functionality is very useful because it adds an abstraction layer between the balancing algorithm and the actual topology of the processors. The function is design for internal use by the library and should not be called through the user code.
SYNOPSIS
void DLB_Calculate_neighbors(int my_pe, int num_pes)

INPUT PARAMETERS
my_pe: The id of the virtual processing element running on the calling process.
num_pes: The number of all the virtual processing elements running in the system.

OUTPUT PARAMETERS
none

4.1.9 DLB_Get_load_average

The function calculates and returns the average load of all the processes of the system. It is mainly used internally by the library but it can also be used through the user code. DLB_Get_load_average can be very useful for the user mainly for statistics.

SYNOPSIS
float DLB_Get_load_average()

INPUT PARAMETERS
none

OUTPUT PARAMETERS
average_load: The average load of the system.

4.1.10 DLB_Calculate_excess_units

The function calculates and returns the number of the excess load units of the calling process. It bases the calculation on the total load of the system (in seconds) and the load of the calling processor. For simplicity it assumes that all the load units take the same time to execute thus it assign to them the same load. After deciding on how many load units have to be discarded, in order to reduce the load of the processes to the average load of the system, it returns that number. The function is designed for internal use by the library and should not be called through the user code.

SYNOPSIS
int DLB_Calculate_excess_units(double total_load, double my_load)

INPUT PARAMETERS
total_load: The total load of the system measured in execution time.
my_load: The load of the calling process measured in execution time.

OUTPUT PARAMETERS
excess_load_units: The number of load units that have to be discarded from the calling process.
4.1.11 DLB_Timer

The DLB_Timer function implements a high precision clock. When called it returns a value in seconds of the time elapsed from some arbitrary, but consistent point. That point will remain consistent during program execution, making subsequent comparisons possible. The function can be used by the user for timing parts of its code.

SYNOPSIS

float DLB_Timer()

INPUT PARAMETERS
none

OUTPUT PARAMETERS

time: The time elapsed from a consistent point in the past measured in seconds.

4.2 DLBLib User guide

4.2.1 Introduction

The Dynamic Load Balancing Library or DLBLib is a library of functions that can be used in C programs. As its name implies the DLBLib is intended for load balancing parallel programs that run on distributed memory High Performance Computers.

DLBLib is a feedback guided dynamic load balancing library for distributed memory environments. It can be used for load balancing parallel applications in order to minimize their total execution time. With DLBLib one can easily load balance his application without having any previous experience with load balancing. It requires only the knowledge of the interface and how to use the library. The load balancing process is hidden by the library and performed automatically without any interaction with the user.

To start using the DLBLib you need to get the library code and place the header file in the same directory as the source code of your application. The library consists of the file dlblib.h which contains the definitions and function prototypes necessary for compiling a DLBLib program. This User guide is a brief tutorial instruction to the most important features of the DLBLib for C programmers. It is intended for use by programmers with some experience in C and parallel programming (message passing) but with no experience in dynamic load balancing.

4.2.2 User Defined Pack and Unpack functions

As mention before the load balancing process is hidden from the user and performed automatically by the library. This functionality requires from the user the definition
of two functions, one for packing data ready to be send to another processor, and one for unpacking the received data. The programmer must define these functions in order to perform correctly for the application being balanced while following the library’s requirements. The prototype and other requirements for the functions are explained below.

**Pack Function**

The pack functions must be capable of packing the excess load units of the processor, as calculated by the DLBLib, in a continuous symmetrically allocated buffer. This buffer can then be handled by the library for the load balancing process. The pack function must be implemented by the programmer in the same file as the application or in a file included to the main program. The prototype that should be followed is:

```c
void *pack(void *data_in, void **data_out, int num_unit,
           int *size_of_buffer)
```

A pointer to the data structure that represents the load of the process is passed through the `void *data_in` parameter. The double pointer `void **data_out` is used for returning the allocated by the pack function buffer containing the load units to be moved. The number of the load units that will be packed for sending is specified by the `int num_unit` parameter. Finally the size of the allocated buffer is returned by reference through the `int *size_of_buffer` parameter. The function must allocate a continuous symmetric buffer, using the `DLB_Malloc()` function, big enough for packing `num_unit` load units from the calling process excess load. The allocated buffer must have the same size on all the processors. So the `DLB_Malloc()` must be called with the same arguments on all processes. Then `num_unit` load units must be removed from the data structure representing the load of the process and placed in the allocated buffer. At this point the user must make any changes to the `data_in` structure to reflect the new load after the removal of the excess load units. Finally the size of the allocated buffer must be returned through the `size_of_buffer` parameter and the buffer itself through the `data_out` parameter. A pseudocode example of a pack function is demonstrated in Figure 4.1.
void *pack(void *data_in, void **data_out, int num_unit, int *size_of_buffer){
   *data_out = DLB_Malloc(size_of_returned_buffer);
   copy num_unit load units from data_in to *data_out;
   update data_in structure to reflect the remaining load;
   update size_of_buffer with the size of the allocated buffer;
   return;
}

Figure 4.1: A pseudocode example of user defined pack function.

Unpack Function

The unpack functions will be responsible of unpacking the buffer, received from other processes, and adding the contained load units to the load of the calling process. The unpack function must be implemented by the programmer in the same file as the application or in a file included to the main program. The prototype that should be followed is:

void *unpack(void *data_in, void *data_received)

The load of the calling process will be provided to the function through the parameter void *data_in, which refer to the data structure representing the load units of the process. The buffer containing the load units received from neighboring processors will be available through reference to the symmetrically allocated memory by the void *data_received parameter. The function must unpacked the received buffer and add the load units contained in it to the data structure that represents the load of the calling process (void *data_in). After that the user must made any changes required to the data structure in order to reflect the load with the addition of the received load units. Finally the buffer received must be freed using the DLB_Free() function for better memory management. A pseudocode example of an unpack function is presented in Figure 4.2
void *pack(void *data_in, void *data_received) {
    copy the load units from data_received to data_in;
    update data_in structure to reflect the new load;
    deallocate the received buffer;
    return;
}

Figure 4.2: A pseudocode example of user defined unpack function.

4.2.3 General DLBLib Programs

Every application that need to be load balanced using the DLBLib mus include the preprocessor directive:

```
#include "dlblib.h"
```

This file, "dlblib.h", contains the definitions and functions prototypes necessary for compiling a DLBLib program. Before any other DLBLib function can be called in a user code the function DLB_Init() must be called, and it should called only ones throughout the whole execution. It signals the library to perform the required setup for its correct functionality. After the application finishes using the DLBLib the function DLB_Finalize() must be called. This cleans up any pending communications and tears down the setup made for the library - e.g. it deallocates memory buffers. So a typical DLBLib has the layout of the Figure 4.3.
```c
#include "dlblib.h"

int main(int argc, char ** argv){
    
    /*No DLBLib Functions called before this point*/
    DLBLib_Init();
    
    /*Use the DLBLib for balancing the application*/
    
    DLBLib_Finalize();
    /*No DLBLib Functions called after this point*/
    
}
```

Figure 4.3: The typical layout of a DLBLib program.

### 4.3 Example application

```c
#include <stdio.h>
#include <stdlib.h>
#include "dlblib.h"

#define RUNS 10000 //maximum number of main loop iterations
#define NUM_OF_CORES 1024 //number of the processors of the system

struct data_struct{
    int load;
};

//user specified pack function
void pack(struct data_struct *my_load, struct data_struct **out_data, int num_units, int *buffer_size){
    *out_data = DLBLibMalloc(sizeof(struct data_struct));
    (**out_data).load = num_units;
    (**out_data).load = (**out_data).load - num_units;
    *buffer_size = sizeof(struct data_struct);
}

//user specified unpack function
```
void unpack(struct data_struct *my_load, struct data_struct *in_data){
    DLB_Free(in_data);
}

void main(int argc, char *argv[]){
    int my_pe = 0; //rank of process
    int num_pes = 0; //number of processes
    int i = 0; //iterator
    int iteration = 0; //iterator
    int total = 0; //total number of load units

    struct data_struct my_load; //data structure representing the load

    //calculate number of total load units
    for(i=0; i<NUM_OF_CORES; i++){
        total = total + i;
    }

    //initialize DLBLib
    DLB_Init(&my_pe, &num_pes, total);

    //set intitial load
    my_load.load = my_pe;

    //main loop of the application
    for(iteration=0;iteration<RUNS;iteration++){

        //start measuring the process load
        DLB_Start_load_execution();

        //simulate a time consuming calculation
        for(i=0; i<my_load.load; i++){
            sleep(1);
        }

        //stop measuring the process load
        DLB_End_load_execution();

        //apply one balancing iteration
        DLB_Balance(pack, unpack, &my_load, 0.0);
    }

    //finalize DLBLib
    shmem_finalize();
}

Figure 4.4: Example application Dynamically load balanced using DLBLib.
Chapter 5

Experimental Design

In this Chapter we present the design of the experimental procedure followed throughout this work as well as the system on which the experiments took place. In Section 5.1 we describe the benchmarking application that was developed for testing the correctness and efficiency of the DLBLib. The HECToR system, on which the experiments were executed, is briefly described in Section 5.2 and finally in Section 5.3 we present the shared memory library on top of which we had implemented the DLBLib.

5.1 Benchmark Application

For the fulfillment of the last aim of the project a benchmarking application was developed. The benchmark was designed so it can test the DLBLib for its correctness, efficiency and scalability. It can simulate an application, with random or seeded load distribution across the processors, which can have customizations over a number of features. The features that can be tested through the benchmark include the dynamic or static total load of the system, the size of the exchange informations (load units) and the number of processors of the system. The different features are described below.

5.1.1 Static Vs Dynamic load

The load of the system actually refers to the amount of work that all the processors of the system have to execute. In some types of applications or problems that amount of work is not the same across all the iterations as load is created or destroyed as the execution evolves. This behavior can cause a load balanced application to become imbalanced.

In order to test the reaction of the proposed model and library the benchmarking application can simulate both static and dynamic loads. When setup for static load the application will generate an initial amount of load which will stay the same until the
end of the simulation. In the case of the dynamic load the application’s initial amount of load will vary over time following a random behavior. The frequency of the changes of the load can be controlled by the benchmarking application.

5.1.2 Random Vs Balanced initial state

In a balanced initial state all the processors of the system have more or less the same amount of load to execute. This can not guarantee that the application will remain load balanced as the total load can be dynamically varying. In a random initial state the total load of the system is distributed to the processors randomly creating an uneven distribution.

The benchmark implemented has the ability to simulate both random and balanced initial states of the system. This functionality of the benchmark tests the library’s behavior in these two scenarios. In the scenario with the random initial load distribution a different amount of load is created for each of the processors of the system. The creation of the load is controlled by a random generator function which can be seeded in order to provide the option to repeat the same distribution in the future. In the balanced distribution scenario the random generator function is used to create an initial load that is then copied to all the processors of the system. This results to an optimal balanced state for the system.

5.1.3 Amount of Migrating Data

The amount of the migrating data depends of the characteristics of the application to be load balanced. It represents the data that each processor has to send and receive to and from its left and right neighbor measured in bytes. This characteristic is crucial for the suitability of the proposed load balancing model for data intensive applications. It also tests the SHMEM interface which is the communication library chosen for the communications of the DLBLib.

The implemented benchmark can be setup in such a way that it simulates various amount of migrated data. When set to small data sizes it can simulate applications that operate on small data sets. On the other hand applications with large data sets can be tested with the benchmark set to larger data sizes. By setting the data size to very small values we can also test the delay of the library. The delay is an indication of the cost of the communication setup required for the load balancing process.

5.1.4 Scalability

The scalability refers to the ability of the library to continue to perform well as the number of the processors in the system increases. In particular we are interested in the cost
of the balancer for larger processor numbers. This cost is measured in the overheads introduced to the application by the balancer. These overheads include the additional time that the library needs in order to perform the balancing - e.g. Time for communications, synchronisation, etc.

In order to test the scalability of the proposed model the benchmarking application was designed with the ability to change the size of the system at runtime. This can help test the library for various system sizes. In theory the library can scale well as the number of communications required by each processor is constant for all the sizes of the system. To verify that the test application has the ability to simulate from small systems, with tens of processors, to larger systems, with tens of thousands of processors.

5.2 Hardware - HECToR

The HPC system that was used for the experiments testing the load balancing library DLBLib was HECToR. HECToR (High End Computing Terascale Resources) is UK’s academic national supercomputer service. The service is a Cray XE6 system and is currently in its Phase 2b with a peak performance of over 360 teraflops. More information about the hardware of HECToR are given in the next Section. In Section 5.2.2 we discuss the operating and file systems currently available on the service. Finally in Section 5.2.3 we present the Gemini interconnect which is currently install of HECToR.

5.2.1 Basic Hardware overview

HECToR[26] currently in its Phase 2b is housed in 20 cabinets which hold a total of 464 compute blades. Four compute nodes are included in each one of the compute blades with each node having two 12-core AMD Opteron 2.1GHz Magny Cours processors. The compute system totals 44,544 cores. Each 12-core socket is coupled with a Cray Gemini routing and communications chip and shares 16 GB of main memory, giving the system a total of 59.4 Tb. The theoretical peak performance of the system is over 360 Tflops. Apart of the compute nodes there are 16 service blades, each with two dual-core processor sockets. They act as login nodes, controllers for the I/O and for the network.

HECToR use a 3D-torus topological Network, thus allowing all the three dimensions to be cyclic. For every two XE nodes there is one Gemini router chip. Gemini chips have 10 network links which are used to implement the 3D-torus of processors resulting to 1-1.5 microseconds latency between two nodes.
5.2.2 Operating System and File system

HECToR runs the Cray Linux Environment[33], a proprietary operating system from Cray. CLE comes with two different distributions on HECToR. A full-feature Linux distribution runs on the service nodes which are used for user login, I/O management etc. For the compute nodes a reduced version named Compute Node Linux is used. CNL is designed to minimise the effects of the operating system, improving the performance of the compute node by eliminating the unwanted features and services.

Lustre [26] is the parallel file system used by HECToR to access the 596 TB high-performance RAID disks in the shared, external filesystem. Lustre offers high performance and scalability to I/O operations.

5.2.3 Gemini Interconnect

The most important part of HECToR communications system is the Gemini interconnect [32]. It is especially designed for use with HPC systems. Thus it is capable of transferring millions of messages per second. Each node, which has 24 processors, is directly connected to the Gemini interconnect through a technology named Hyper-Transport 3.0. The use of this architecture limits bottlenecks that affect commodity network’s performance. Gemini provides to each node a peak of over 20 GB/s of injection bandwidth and can scale from hundreds to hundreds of thousands of cores without the increase in buffer memory required in the point-to point connection method of commodity interconnects.

5.3 SHMEM

SHMEM refers to the shared memory access library[27] available on Cray, SGI as well as on machines from other vendors. The library allows user to write parallel applications using a shared-memory programming model, where all the processes can operate on a globally accessible address space. The SHMEM data-passing library routines are similar to the message passing interface (MPI) library routines. They pass data between cooperating parallel processes. SHMEM contrasts with message passing via MPI in which a message is sent in a two-step process. With SHMEM, the sending of data involves only one CPU in that the source processor simply puts that data into the memory of the destination processor. Likewise, a processor can read data from another processor’s memory without interrupting the remote CPU. The remote processor is not made aware that its memory has been read or written, unless the programmer implements a special mechanism for this. The procedure calls have lower latency and higher bandwidth than MPI calls and can be used to optimise communication intensive sections of an MPI code to produce better scalability.
SHMEM functions work on symmetric accessible data objects that can be arrays or scalars. Remote data objects are identified by the address of the corresponding data object on the local CPU. The local existence of a corresponding data object implies that a data object is symmetric. In practice a symmetric data object is one where the local and remote addresses have a known relationship.

SHMEM routines can be used in conjunction with Message Passing Interface (MPI) routines in the same application. Programs that use both MPI and SHMEM should call MPI_Init followed by shmem_init. At the end of the program, shmem_finalize should be called followed by MPI_Finalize. SHMEM processing element numbers are equal to the MPI rank within the MPI_COMM_WORLD communicator if the MPI job consists of a single application.
Chapter 6

Results

6.1 Correctness Testing

For the DLBLib to function correctly it is important that the final outcome of the application being load balanced is not affected by the library. That means that during the load balancing process no load unit is lost or duplicated. That scenario can lead to unpredictable behavior for the application making it useless for its initial propose. In order to test the correctness of the DLBLib during the load balancing process we present a graphical representation of the load that is assigned to each processor. With this representation we can monitor the migration of the load units between processors and conclude if the DLBLib works according to the proposed model.

![Load Balancing over Iteration](image)

**Figure 6.1: Processor Load over Iterations in 8 Processor System**
To verify that the load units migrate correctly between the processors, a special serial number was included in every unit. The movements of the load units were monitored using these serial numbers.

The Figure 6.1 shows the amount of load that each processor has for a number of iterations. In the first iteration, we can see the initial distribution of the load among the processors. Processors 1, 5, and 7 (red, cyan, and gold respectively) are highly loaded since their load exceeds the average load of the system. These highly loaded processors split their excess load and pass it to their nearest left and right neighbors for the second iteration. This migration leads to the formation of a different set with highly loaded processors. In this step, processors 2, 5, and 8 (green, cyan, and blue respectively) are considered to be overloaded. Thus, they pack their excess load units and send half of them to each of their left and right neighbors. In this case, since the excess load unit is just one, it is sent to the right neighbor as instructed by the algorithm. The processors are arranged by the DLBLib into a virtual ring topology, so any excess load units from processor 8 migrating to the right neighbor will be received by processor 1. All the underloaded processors that receive load units from their neighbors add those units to their load so that the average load of the system is not an integral value. Thus, a number of processors will have the smallest integral value that is more than the average, and a number of them the largest integral value that is less than the average. This state of the system is also considered to be an optimal load distribution. We observe that our system reaches an optimal distribution state in iteration 9 and stays at this state until the end of the simulation. An important information obtained from the figure is that the total number of load units remains the same throughout the whole execution. By monitoring the serial numbers of the load units, we make sure that they are migrating to the correct processor. These two observations can guarantee that no load units are lost, duplicated, or migrate to a wrong processor, thus the final outcome of the application will not be affected by the load balancing process.

### 6.2 Effectiveness Testing

For a load balancing library to be effective, it must be able to balance the available load of the system as evenly as possible among the processors. In practice, this means that all the processors will finish executing their load almost at the same time, limiting the idle time and minimizing the total execution time. If the proposed library fails to minimize the execution time of the applications that is applied on then it fails its requirements. To test the effectiveness of our library, we measure the execution time of each iteration of the benchmarking application after using the DLBLib. The load balancing function of the library was called at the end of the execution of each iteration of the benchmark application. The results of this experiment are plotted in the Figure 6.2.
Figure 6.2 shows the effectiveness of the DLBLib in load balancing an application. As shown by the graph the library is extremely effective in reducing the execution time per iteration in the first stages of the simulation. In just five load balancing steps the library managed to reduce the execution time of a single iteration of the benchmark from above 40 seconds to well bellow 20. We call this behavior "fast start" and is one of the features of the proposed model. Just after the 10th iteration the balancing becomes less aggressive. It continues with small improvement until just after the 80th iteration where the system reaches to an optimal load balanced state. It is important to observe that only a small fraction of the iterations take significantly large amount of time to complete. If DLBLib was not used in this experiment the execution time for all the iterations would be more than 40 seconds each.

### 6.3 Optimal Load Balancing Convergence

The convergence to an optimal load balancing state is one of the most important features of a load balancing algorithm. It shows the number of steps that the balancing process has to be applied for the system to become load balanced. To test the convergence of the proposed algorithm and dynamic load balancing library we contacted a number of tests. First we tested the convergence of the model for a constant total load amount. That is when the total load of the system is the same across the whole execution of the application. Then we tested the convergence in a dynamic total load environment. In this case the total load of the application was dynamically varying across the execution of the application.
The results for the convergence of the model for the static total load scenario are discussed in Section 6.3.1. In Section 6.3.2 we discuss the results for the dynamically varying total load.

### 6.3.1 Static Total Load

For testing the convergence of the proposed dynamic load balancing library the DLBLib was used for load balancing the benchmarking application in a number of setups. The experiments measure the convergence in the number of the steps performed until the system reached a balanced state. The system setups used and their convergence iteration are shown in Table 6.1. In Figure 6.1 we give a graphical representation of the load and how it migrates between neighboring processors over the iterations. Additionally in Figures 6.3-6.5 we present some of the experiments in detail. In the figures we represent the load balancing process as it evolves over the iterations of the algorithm. We notice how the higher and lower loads of the system vary and finally converge to the average load of the system. It is important to note that in all the scenarios tested the system remain in a balanced state after reaching the optimal distribution.

<table>
<thead>
<tr>
<th>Test Number</th>
<th>System Size in CPU count</th>
<th>Convergence Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>46</td>
</tr>
<tr>
<td>5</td>
<td>128</td>
<td>63</td>
</tr>
<tr>
<td>6</td>
<td>256</td>
<td>130</td>
</tr>
<tr>
<td>7</td>
<td>512</td>
<td>200</td>
</tr>
</tbody>
</table>

Table 6.1: Overview of the tests conducted for the Static Total Load scenario.
Figure 6.3: Load Balancing Process in 8 Processor System

Figure 6.3 demonstrates the load balancing process in a system with 8 processors. The initial load of each processor was randomly determined and at the initial distribution load was varying between ten and zero. It is clear for the graph that the DLBLib is effective in such a system. In the first two iterations of the model the gap between the higher and the lower loads of the system closed from ten load units to just three. The smoothing of the load differences continued until iteration 9 where the system reached an optimal load balanced state. From that point forward the system remained load balanced due to the fact that the total load of the system was static throughout the whole execution.
If Figure 6.4 we see the balancing of a 128 processor system. Here we notice that the gap between the higher and lower load is much larger than for the previous scenario (8 processor system). Again the balancing model of the DLBLib smoothed out the extreme cases reducing the gap by more than half in just five iterations. We can see that after the 40th iteration the higher load value had almost reached the optimal value of the system, but the lower value remains relatively far from the system’s average. This is caused by the fact that a lot of the processors have in this case just one load unit more that the average value. In this situation, the load distribution is slowly changing, as every iteration only one load unit is migrating from each processor to its neighbor. We call this situation slow balancing. Although is takes a number of iterations for the library to smooth the slow balancing the execution time of these iterations is only one load unit above the optimal.
In the 512 processor system of the Figure 6.5 the DLBLib managed to load balance all the processors after about two hundred iterations. Here the optimal distribution is achieved although not all the processors have the same load allocated to them. This behavior is caused by the fact that the load is represented by discrete load units. The average load of the system, as shown from the figure, is not an integral value. As a sequence half of the processors of the system, after they reach their balanced state, have load assigned to them equal with the next higher integral value of the average. The other half of the processor have load equal with the previous lower integral value of the average load.

6.3.2 Dynamic Total Load

The convergence of the DLBLib library was also tested for the scenario of the dynamically varying total load. For this experiment the system was setup to start in a balanced state. At random points of the execution a processor was chosen and its load was increased or decreased by a random amount of load units. This process creates load imbalances to the processors of the system. According to the specifications set for the DLBLib it must be able to solve this imbalances after a number of iterations. To verify that, the state of the system was monitored at every iteration and the reaction of the library, to these dynamic load changes, was recorded. The results of this experiment are displayed in Figure 6.6.
Figure 6.6: Load Balancing Process in a dynamic load environment.

For this experiment we simulated an application, which load changes dynamically, that runs on an eight processor system. Figure 6.6 shows a graphical representation of the load of each one of the eight processors of the system throughout the whole execution. Each processors load is shown by one individual color bar. The eight bars are stacked so they can represent the total load of the system. The iterations at which the load of the system increases or decreases are shown by the points where the height of the stacked bars changes from a balanced state. At the point that the height of the bars rises so does the load, similarly when the height of the bars decreases the load is also reduced. At these points the system becomes unbalanced and the DLBLib, that detects the imbalance on the same iteration, begins the load balancing process. The iterations needed by the DLBLib to smooth the imbalance are shown by the parts of the graph where the load of the processors is changing every iteration.

The experiment shows us that the DLBLib after several changes in the total amount of the system’s load manages to balance again the system. At this point we note that the size of the system was chosen to be eight to simplify the presentation of the results. The results would follow the same pattern for every system size.

### 6.3.3 Worst Case Scenarios

In this section we present the two worst case scenarios that the DLBLib has to solve. For the first scenario all the load of the system is assigned to a single processor and the rest of the processors are not assigned with any load. For the second scenario all the processors of the system are in a balanced state except two. Those two processors,
one being overloaded and one underloaded, are not directly connected and the distance between them is relevantly large. To test the behavior of the DLBLib for this two scenarios we have simulate an eight processor system implementing the characteristics of the two scenarios. The results of the simulations are represented in Figures 6.7 and 6.8.

From Figure 6.7 we can see the reaction of the library to the first worst case scenario. At the beginning of the simulation all the load was assigned to one processor (CPU 0). The library operated in two steps. At the first step (iteration 0 - 10) DLBLib was moving load units in a effort to assign load to all the processors of the system. From that point on (iteration 10 - 23) the library was load balancing the processors to reach them to a balanced state.

This scenario highlights one very important feature of the wave propagation model implemented by DLBLib. The fact that each overloaded processor splits its excess load to its two neighbors also limits the execution time per iteration to more than 50% after the first iteration. The library continues to reduce the execution time per iteration until it reaches the optimum level of a load balanced system.
At the beginning of the simulation for the worst case scenario 2, shown by Figure 6.8, all the processors of the system are equally loaded except two. The processor represented by CPU 0 has double the load than the others and processor CPU 4 has no load assigned. The two processors have a large (according to the size of the system) distance between them. DLBLib, as shown by the representation of the balancing process, successfully solve this imbalance. In fact it took less iterations (almost half of them) for the library to bring the system to balance than for scenario 1, as all the processors had already an initial load.

6.4 DLBLib Scalability

Scalability is one of the most important attributes of a load balancing library. A library with good scalability will be able to be used with large systems, having thousands of processors, without adding to them many overheads in execution time. To test the scalability of the DLBLib we measure the overheads it introduces to an application. We tested two kind of applications, one with a small data set and one with large data set. For the small data set application the DLBLib had to move small amount of data between processors, on the other hand a larger amount of data had to be moved for the larger dataset application. As an overhead we measure the total cost of the balancer for the application of one iteration of the load balancing process. This include the cost of the communication for data migration. The cost is measure in seconds across a number of iterations and an average value was calculated. The scalability of the library was tested on a number of system setups. The different setups that we used differ only on their size.
From the experiments conducted, on both small and larger data set applications, we can conclude that the cost introduced by the balancing library is negligible when compared with the execution time of the application. The total cost for using the DLBLib is less than 0.02% of the total execution time. That cost, has in fact less significance, due to the fact that the total execution time of a load balanced application is far less that the execution time of an unbalanced application.

### 6.4.1 Small dataset application

As a small data set application benchmark we created a setup, where each load unit had a size of 4096 bytes. The data set contained in each load unit is equivalent with an array containing 1024 integer values and is exchanged between neighboring processors during the load balancing process. The setups used in this experiment as well as the cost of the balancer to the total execution time are shown in the Table 6.2. The cost of the balancer is also calculated as a percentage of the total execution time of the benchmark application. The setups listed in the table differ only in their system size.

<table>
<thead>
<tr>
<th>Test Number</th>
<th>System Size in CPU count</th>
<th>Overheads introduced in seconds per iteration</th>
<th>Overheads introduced in percentage of total execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>0.0000505</td>
<td>0.0009%</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>0.0000650</td>
<td>0.0011%</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>0.0000975</td>
<td>0.0017%</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>0.0001040</td>
<td>0.0018%</td>
</tr>
<tr>
<td>5</td>
<td>128</td>
<td>0.0001293</td>
<td>0.0022%</td>
</tr>
<tr>
<td>6</td>
<td>256</td>
<td>0.0001737</td>
<td>0.0029%</td>
</tr>
<tr>
<td>7</td>
<td>512</td>
<td>0.0001957</td>
<td>0.0033%</td>
</tr>
<tr>
<td>8</td>
<td>1024</td>
<td>0.0002327</td>
<td>0.0039%</td>
</tr>
</tbody>
</table>

Table 6.2: Overview of the tests conducted for evaluating the Scalability of DLBLib for 4096 byte load unit.

From the Figure 6.9 we can see that the size of the system is not affecting directly the cost of the DLBLib to the total execution time. This is because the number of load unit migrations is not directly connected to the system size as the processors only exchange load with their two neighbors. This is a very important feature of the proposed load balancer model that allows the use of the DLBLib library for balancing large systems with thousands of processors. The small increment in the cost of the library as the number of processors increases can be explained with the additional cost of the synchronization. As the size of the system increases the underlying communication library needs more time to synchronize all the processors. This additional cost can not be controlled by the DLBLib and depends only from the latency of the connecting network of the machine.
6.4.2 Large dataset application

For benchmarking a large dataset application, a setup with load unit size equal to 409600 bytes was created. The size of the load unit is equivalent with an array of 102400 integer values and is one hundred times larger than the data set of the previous experiment. The size is bigger than any scientific application would normally need, as this amount of data is contain in each load unit and each processor can execute any number of load units. The setups used in this experiment as well as the cost of the balancer to the total execution time are shown in the Table 6.3

<table>
<thead>
<tr>
<th>Test Number</th>
<th>System Size in CPU count</th>
<th>Overheads introduced in seconds per iteration</th>
<th>Overheads introduced in percentage of total execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>0.0012174</td>
<td>0.0200%</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>0.0012870</td>
<td>0.0151%</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>0.0013540</td>
<td>0.0212%</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>0.0014082</td>
<td>0.0178%</td>
</tr>
<tr>
<td>5</td>
<td>128</td>
<td>0.0014367</td>
<td>0.0182%</td>
</tr>
<tr>
<td>6</td>
<td>256</td>
<td>0.0014558</td>
<td>0.0180%</td>
</tr>
<tr>
<td>7</td>
<td>512</td>
<td>0.0015680</td>
<td>0.0170%</td>
</tr>
<tr>
<td>8</td>
<td>1024</td>
<td>0.0015567</td>
<td>0.0171%</td>
</tr>
</tbody>
</table>

Table 6.3: Overview of the tests conducted for evaluating the Scalability of DLBLib for 409600 byte load unit.

Figure 6.9: Load Balancing Cost over System Size for a small dataset.
By comparing Table 6.3 with the Table 6.2 from the small data set application benchmark we observe an increment in the cost of the load balancing process for the larger data set application. This increment was expected as the amount of the data that was exchanged packed in each load unit was increased by a factor of 100.

![Load Balancing Cost over System Size for a large dataset.](image)

The cost for using the DLBLib for load balancing applications with extremely large datasets is significantly larger than for small dataset applications. This is shown by comparing Figure 6.10 with Figure 6.9. The increment in the cost is not cause by the increment in the number of processors between the two experiments but in the increment of the transferred data. From this observation we can conclude that model implemented by the DLBLib can scale to larger processor numbers while maintaining its efficiency.
Chapter 7

Conclusions

This project focused on dynamic load balancing for distributed memory environments. After a careful review of the former load balancing models this work suggested a new model named Wave Propagation Model. The suggested model uses a diffuse method where each overloaded processor discards its additional load to its two nearest neighbors as described in Chapter 3. Also during this project a load balancing library named DLBLib was implemented as described in Chapter 4. The library, which implements the wave propagation model, can be used by a programmer to easily add dynamic load balancing to his applications. Finally a benchmarking application was implemented for testing the DLBLib. The result are discussed in Chapter 6.

The Wave Propagation Model is based on former diffusion models, reviewed in Chapter 2, but with some customization and added features that might be beneficial for its performance. The model is capable of more drastic diffusion of the load for larger differences in the load of individual processors. This results in significant gains towards the optimal load distribution with just a small number of iterations of the balancing model. As we can conclude from the interpretation of the results the model succeed in solving the dynamic load balancing problem for all the scenarios tested.

The suggested model was implemented in DLBLib, a C language dynamic load balancing library. The library was implemented on top of SHMEM communication library. SHMEM refers to the shared memory access library available on HPC machines from Cray and other vendors which can minimize the overhead associated with data passing requests, maximize bandwidth, and minimize data latency. The use of DLBLib not only reduced the load imbalance of the applications tested, but also balanced the systems to an optimum state. This resulted in the reduction of the execution time per iteration which in sequence reduce the total execution time.

Finally to test the correctness and efficiency on the DLBLib a benchmarking application was implemented. Using this benchmark, we tested the library for a number of different scenarios that simulate a wide range for real life scientific applications. All
the tests were performed on HECToR, UK’s national HPC service. The library showed its ability to load balance all the tested applications. As a result their execution time was significantly reduced. The results verified that the library can be used efficiently for load balancing applications for the HPC community.
Chapter 8

Future Work

A significant percentage of scientific applications use programming languages other than C. Commonly used languages include C++, Fortran and Java. A suggestion for future work is to implement the wave propagation model in libraries for those programming languages also. These implementations will enable the use of the DLBLib from programmers who prefer the use of languages other than C. Also it will make it possible for existing scientific applications that are written in those languages to be dynamically load balanced with minimum modifications.

Furthermore a suggestion is to implement a version of the DLBLib that runs on top of a message passing library other than SHMEM. Other libraries that can be used include MPI or even Parallel Global Address Space languages like Co-Array Fortran, UPC and Titanium.

Finally, time limitations did not allow an in-depth comparison of the DLBLib with other dynamic load balancing libraries. So, the last suggestion for future work is to compare in detail the differences in performance of DLBLib with existing dynamic load balancing libraries on real scientific applications.
Appendix A

DLBLib Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <mpp/shmem.h>
#include <omp.h>
#include <mpi.h>

long *_DLB_pSync;
float *_DLB_pWrk;
int *_DLB_right_target;
int *_DLB_left_target;
int *_DLB_my_pe;
int *_DLB_num_pes;
float *_DLB_load;
float *_DLB_global_load;
float *_DLB_start;
float *_DLB_end;
int *_DLB_num_units;

void DLB_Init(int *my_pe_, int *num_pes_, int num_units);
void DLB_Calculate_neighbors(int my_pe, int num_pes);
void DLB_Start_load_execution();
void DLB_End_load_execution();
float DLB_Get_load_average();
float DLB_Timer();
float DLB_Balance(void (*pack) (void *, void **, int, int*), void (*unpack) (void *, void *), void *data, float tolerance);
void DLB_Finalize();
float DLB_Get_load();
int DLB_Calculate_excess_units(float total_load, float my_load);
void *DLB_Malloc(int size_in_bytes);
void DLB_Free(void * ptr);
```

/****************************
 NAME : void DLB_Init(int *my_pe_, int *numPes_, int numUnits)
 DESCRIPTION : Initialises the Dynamic Load Balancing Library and allocates all
 the required memory for the library.
 INPUTS :
 int numUnits The number of the load units available
 in the application at start-up.

50
**OUTPUTS :**
- int *my_pe_ The id of calling the process as created by the shmem library.
- int *num_pes_ The sum of all the processes running the application.

**RETURNS :**

**NOTES :** It is the users responsibility to call the function before using the library.

********************************************************************************
*************************************

```c
#include <shmem.h>
#include <stdlib.h>

void DLB_Init(int *my_pe_, int *num_pes_, int num_units)
{
    int i;

    //initialise shmem library
    shmem_init();

    //dynamically allocate memory in the symmetric heap
    _DLB_pSync = shmalloc(_SHMEM_REDUCE_SYNC_SIZE * sizeof(long)); //used by shmem for synchronisation
    _DLB_pWrk = shmalloc(_SHMEM_REDUCE_MIN_WRKDATA_SIZE * sizeof(float)); //used by shmem for synchronisation
    _DLB_my_pe = shmalloc(sizeof(int)); //id of the calling process
    _DLB_num_pes = shmalloc(sizeof(int)); //sum of all the processes
    _DLB_num_units = shmalloc(sizeof(int)); //number of load units in the application
    _DLB_load = shmalloc(sizeof(float)); //the process load (in execution time)
    _DLB_global_load = shmalloc(sizeof(float)); //the total load (in execution time)
    _DLB_start = shmalloc(sizeof(float)); //timer for the load of a process
    _DLB_end = shmalloc(sizeof(float)); //timer for the load of a process
    _DLB_left_target = shmalloc(sizeof(int)); //left neighboring process
    _DLB_right_target = shmalloc(sizeof(int)); //right neighboring process

    //assign values to library variables
    *my_pe_ = shmem_my_pe();
    *num_pes_ = shmem_n_pes();
    *num_units = num_units;

    //initialisation required by shmem library
    for (i=0; i<_SHMEM_REDUCE_SYNC_SIZE; i++)
    {
        _DLB_pSync[i] = _SHMEM_SYNC_VALUE;
    }

    //calculate each process left and right neighbors
    DLB_Calculate_neighbors(*_DLB_my_pe, *_DLB_num_pes);
}
```

`**NAME : DLB_Calculate_neighbors(int my_pe, int num_pes)**`

```c
#include <shmem.h>
#include <stdlib.h>

void DLB_Calculate_neighbors(int my_pe, int num_pes)
{
    //calculate each process left and right neighbors
    //...
}
```
DESCRIPTION: Calculates the left and right neighboring processes of the calling process.

INPUTS:
- int my_pe The id of the calling process as created by the shmem library.
- int num_pes The sum of all the processes running the application.

OUTPUTS:
- int _DLB_left_target The left neighboring process of the calling process.
- int _DLB_right_target The right neighboring process of the calling process.

RETURNS:

NOTES: The function is used internally by the library.

*******************************************************************************
*******************************************************************************

void DLB_Calculate_neighbors(int my_pe, int num_pes){
    //calculate the id of the left and right neighbors based on the id of the calling process
    //it is assumed that the processes are in a ring
    if(my_pe == (num_pes-1)){
        _DLB_right_target = 0;
        _DLB_left_target = my_pe - 1;
    }
    else if(my_pe == 0){
        _DLB_right_target = my_pe + 1;
        _DLB_left_target = num_pes - 1;
    }
    else{
        _DLB_right_target = my_pe + 1;
        _DLB_left_target = my_pe - 1;
    }
}

*******************************************************************************
*******************************************************************************

NAME: void DLB_Start_load_execution()

DESCRIPTION: Starts the load measuring timer of the library.

INPUTS:

OUTPUTS:
- double _DLB_start The time-stamp of the starting point of the load execution.

RETURNS:

NOTES: It is the users responsibility to call the function at the beginning of the load execution.

*******************************************************************************
*******************************************************************************

void DLB_Start_load_execution(){
    //set the start timestamp of the execution
    _DLB_start = omp_get_wtime();
}
void DLB_End_load_execution()
{
    // set the end timestamp of the execution
    _DLB_end = omp_get_wtime();
    // calculate the process load (in execution time)
    _DLB_load = ((*_DLB_end) - (*_DLB_start));
    // update the global load
    shmem_float_num_to_all(_DLB_global_load, _DLB_load, 1, 0, 0, _DLB_num_pes, &DLB_pWrk[0], &DLB_pSync[0]);
}

float DLB_Get_load_average()
{
    // return the average load of all the processes
    return (*_DLB_global_load) / (float)*_DLB_num_pes;
}

****************************************************************************/
******************************************************************************
NAME: DLB_Calculate_excess_units(float total_load, float my_load)

DESCRIPTION: Calculates and returns the number of excess load units allocated to the calling process.

INPUTS:
- float total_load The sum of the load of all the processes.
- float my_load The load of the calling process

OUTPUTS:
- int excess_units The number of excess units of the process.

NOTES: The function is used internally by the library.

int DLB_Calculate_excess_units(float total_load, float my_load){
    int excess_units = 0;
    //number of excess units
    float load_per_unit = 0;
    //load per unit in execution time
    float excess_load = 0;
    //excess execution time of the process
    //calculate the excess load based on the average load
    load_per_unit = (float)total_load / DLB_num_units;
    excess_load = my_load - DLB_Get_load_average();
    //convert the excess execution time to load units
    excess_units = rint(excess_load / load_per_unit);
    return excess_units;
}

NAME: void DLB_Balance(void (*pack) (void *, void **, int, int*), void (*unpack) (void *, void *, int, int), void *data, float tolerance)

DESCRIPTION: Load balances all the calling processes based on the average load of the system.

INPUTS:
- void (*pack) The user specified function that packs the data that need to be moved to an other process.
- void (*unpack) The user specified function that unpacks the data received from an other process.
- void *data The data structure that holds the data of the calling process.
- float tolerance The tolerance of the balancer.

OUTPUTS:

RETURNS:

NOTES: It is the users responsibility to call the function at the point that the balancing is required.
- This function must be used after timing the load of the process.
using DLB_Start_load_execution() and DLB_End_load_execution()

********************************************************************************
*************************************

void DLB_Balance(void (*pack) (void *, void **, int, int*), void (*unpack) (void *
*, void *), void *data, float tolerance)

    void *load_from_left;
    //receive buffer for the data from left process
    void *load_from_right;
    //receive buffer for the data from right process
    void *load_to_left;  //send
    buffer for the data to the left process
    void *load_to_right;  //send
    buffer for the data to the right process
    int *buffer_size_to_left;  //size
    of the send to the left buffer
    int *buffer_size_to_right;  //size
    of the send to the right buffer
    int *buffer_size_from_left;  //size
    of the receive from the left buffer
    int *buffer_size_from_right;  //size
    of the receive from the right buffer
    float load_average = 0;
    //average load of the system
    int excess_units = 0;  //excess
    load units of the calling process
    int num_units_left = 0;  //number
    of excess units to be send to left process
    int num_units_right = 0;  //number
    of excess units to be send to right process

    //allocate memory in the symmetric heap
    buffer_size_to_left = shmalloc(sizeof(int));
    buffer_size_to_right = shmalloc(sizeof(int));
    buffer_size_from_left = shmalloc(sizeof(int));
    buffer_size_from_right = shmalloc(sizeof(int));

    *buffer_size_to_left = 0;
    *buffer_size_to_right = 0;
    *buffer_size_from_left = 0;
    *buffer_size_from_right = 0;

    //calculate if there are some excess load units in the calling process based
    on the tolerance specified by the user
    load_average = DLB_Get_load_average();
    if(_DLB_load > (load_average - (load_average * tolerance))){
        excess_units = DLB_Calculate_excess_units(_DLB_global_load, _DLB_load);
        num_units_left = excess_units / 2;
        num_units_right = excess_units - num_units_left;
    }

    //call the user specified function to pack the excess load units into buffers
    pack(data, &load_to_left, num_units_left, buffer_size_to_left);
    pack(data, &load_to_right, num_units_right, buffer_size_to_right);

    //notify the receiving processes of the size of the buffers that are going to
    be send to them
    shmmem_int_put(buffer_size_from_right, buffer_size_to_left, 1,
                   *DLB_left_target);
    shmmem_int_put(buffer_size_from_left, buffer_size_to_right, 1,
                   *DLB_right_target);

    shmmem_barrier_all();

    //allocate the specified size of buffer in the symmetric heap
    load_from_right = shmalloc(*buffer_size_from_right);
load_from_left = shmalloc(*buffer_size_from_left);

//send the packed data to the receiver processes
shmem_putmem(load_from_right, load_to_left, *buffer_size_to_left, _DLB_left_target);
shmem_putmem(load_from_left, load_to_right, *buffer_size_to_right, _DLB_right_target);

shmem_barrier_all();

//call the user specified function to unpack the data received
unpack(data, load_from_right);
unpack(data, load_from_left);

}
void DLB_Free(void *ptr){
    //deallocate the memory pointed by ptr
    shfree(ptr);
}

*******************************************************************************
*************************************
* NAME : void DLB_Finalize() *
* DESCRIPTION : Deallocates resources like symmetric buffers used by the library and then kills the processing elements created for the SHMEM Library. *
* INPUTS : *
* OUTPUTS : *
* RETURNS : *
* NOTES : The function should be called once at the end of the main program. *
*******************************************************************************
*************************************

void DLB_Finalize(){
    //make sure all pending operations are done.
    shmem_barrier_all();

    //deallocate symmetric buffers
    shfree(_DLB_pSync);
    shfree(_DLB_pWrk);
    shfree(_DLB_right_target);
    shfree(_DLB_left_target);
    shfree(_DLB_my_pe);
    shfree(_DLB_num_pes);
    shfree(_DLB_load);
    shfree(_DLB_global_load);
    shfree(_DLB_start);
    shfree(_DLB_end);

    //finalize the SHMEM library
    shmem_finalize();
}

Bibliography


[26] HECToR - Hardware. Available at:


[31] HECToR - Learn About HECToR. Available at:

[32] Cray XE6 Brochure. Available at:

[33] HECToR - Software on HECToR. Available at: