A MODEL FOR RESOURCE-AWARE LOAD BALANCING ON HETEROGENEOUS AND NON-DEDICATED CLUSTERS

By

Jamal Faik

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

Joseph E. Flaherty, Thesis Adviser

James D. Teresco, Thesis Adviser

Karen D. Devine, Member

Franklin T. Luk, Member

Mark S. Shephard, Member

Carlos A. Varela, Member

Rensselaer Polytechnic Institute
Troy, New York

July 2005
(For Graduation August 2005)

A MODEL FOR RESOURCE-AWARE LOAD BALANCING ON HETEROGENEOUS AND NON-DEDICATED CLUSTERS

By

Jamal Faik

An Abstract of a Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Examining Committee:

Joseph E. Flaherty, Thesis Adviser

James D. Teresco, Thesis Adviser

Karen D. Devine, Member

Franklin T. Luk, Member

Mark S. Shephard, Member

Carlos A. Varela, Member

Rensselaer Polytechnic Institute
Troy, New York

July 2005
(For Graduation August 2005)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Acknowledgments

I would like to gratefully thank my advisors Dr. Joe Flaherty and Dr. Jim Teresco. Their enthusiastic, compassionate and effective supervision and support have made this work possible.

Deep gratitude to my doctoral committee members for their valuable suggestionsand comments. In particular, I would like to thank Dr. Karen Devine and Dr. Mark Shephard for supporting this work from the beginning. Their proactive support, advice and ideas have been critically essential to this research.

Some parts of this work wouldn't have been possible without the active contribution of my colleague and friend Luis Gervasio and my former assistant Jin Chang and for that, I am very grateful.

I will never be able to thank Pam Paslow enough for her uninterrupted assistance. Not only did she make sure that I got my paychecks on time, but she simply solved all my non-technical problems. If only I had tried her for the technical part as well.

Special thanks to Dr. Toshi Ohsumi for his valuable insight. Many thanks to RPI SCOREC staff, particularly Christophe Dupre, for his constant help with the system related issues. I am also grateful to the RPI Computer Science labstaff for their valuable assistance.

Last but not least, I am forever indebted to my parents and my wife, Houda for their patience, inspiration, comprehension and encouragement when it was most needed. This thesis is dedicated to them.

Abstract

As clusters and grids become increasingly popular alternatives to custom-built parallel computers, they expose a growing heterogeneity in processing and communication capabilities. Performing an effective load balancing on such environments can be best achieved when the heterogeneity factor is quantified and appropriately fed into the load balancing routines. In this work, we discuss an approach based on constructing a tree model that encapsulates the topology and the capabilities of the cluster. The different components of the execution environment are dynamically monitored and their processing and communication capabilities are collected and aggregated in a simple form easily usable when load balancing is invoked. We used the model, called DRUM, to guide load balancing in the adaptive solution of three numerical problems on various cluster configurations. The results showed a clear benefit from using DRUM, in clusters with computational or network heterogeneity as well as in non-dedicated clusters. The results also showed that DRUM generates a very low overhead.

CHAPTER 1

Introduction

Clusters' popularity stems from their ability to offer cost-effective environments for running computationally-intensive parallel applications. Their wide acceptance as a viable alternative to tightly-coupled parallel computers has been further empowered by the development of open-source software such as Linux and standards such PVM and MPI.

One of the attractive features of clusters is the possibility to increase their overall computational power by incorporating additional nodes. This incremental expansion, motivated by the need to address increasingly growing computation workloads, eventually results in heterogeneous environments as the newly-added nodes often have superior computational capabilities. Today's grid environments expose even more heterogeneity. Several efforts to develop technologies to leverage the power of Internet-connected systems are being pursued. Grid technologies such as MPICH-G2 [30] and the Globus toolkit[1] have enabled computation on even more heterogeneous and widely-distributed systems.

In this work, we address the problem of dynamic Load Balancing (LB) and mesh partitioning on heterogeneous and non-dedicated clusters. We aim to increase the effectiveness of load balancing by achieving a good match between the partitions

[1] http://www-unix.globus.org/toolkit/

produced by a load balancer and the computational and communication capabilities of the host nodes on which these partitions are being mapped.

Speedup to be gained from a parallelization is bounded by the distribution of the application load over the nodes composing the execution environment and the capabilities of each individual node as well as the communication volume and behavior of the application. Traditional LB algorithms try to achieve a good balance by yielding partitions of similar sizes, while attempting to minimize the cost of inter-node communication. The presence of heterogeneity, either resulting from differences in hardware capabilities or from non-dedicated usage of the cluster, adds further constraints to the optimization problem at the core of any LB algorithm. Inclusion of these new constraints into the optimization equation mandates development of approaches to quantify the heterogeneity factor, which by itself requires techniques to measure the capabilities and utilization of the machine resources. Moreover, the collected information should be presented in a form such that its effective usage requires minimal or no changes to existing LB algorithms. A suitable approach to address those issues should also consider other constraints such as minimal intrusiveness of the data collection system, scalability and application-independent operation.

Our work involves an integrated approach that addresses these concerns. We are developing a middleware that allows dynamic load balancing routines to produce partitions with sizes compatible with capabilities of the computational resources. Specifically, we propose a model that encompasses the following features:

- A set of tools to discover and describe the execution environment. In particular, these tools include a mechanism to describe the interconnection topology of the resources and their static capabilities. Knowledge of the topology would, in particular, permit us to design topology-driven load balancing schemes.

- A metric to measure the degree of heterogeneity of the execution environment. This metric is useful in predicting and qualifying the eventual performance improvement to be gained from the use of the model.

- A system of application-independent, overhead-controlled, distributed agents for dynamic evaluation of the machine resources.

- An ensemble of heuristics to reduce the collected performance data to a simple scalar form, directly usable by existing load balancing algorithms with minimal or even no modification.

- A simple interface for integration with existing load balancing algorithms, particularly with the Zoltan toolkit for parallel data management and load balancing.

This document is organized as follows. The next chapter highlights the need for dynamic load balancing procedures, particularly for applications using adaptivity. The core of this section is a review of some recent efforts addressing load balancing for heterogeneous clusters. Chapter 3 describes our proposed model. The chapter starts by explaining our approach to describe the execution environment and then presents the proposed techniques for capabilities assessment, involving, in particular, a description of the agents used for monitoring. Heuristics to reduce performance data collected at each resource to a scalar number called power are subsequently described. The section concludes by showing a first experiment that uses the model to compute nodal powers. Chapter 4 briefly presents the public interface of DRUM and explains the interaction scenario between DRUM and the Zoltan library. Chapter 5 lists our experimental tests of the model. The chapter starts with a discussion of a metric to measure the degree of heterogeneity of a cluster. Then, in Section 5.1, we present an early experiment in which we use DRUM to drive load balancing in the solution of a two-dimensional Rayleigh-Taylor instability problem [43]. Section 5.2 gathers multiple experiments showing the use of DRUM in the solution of a Laplace equation on the unit square using the PHAML software [40]. Section 5.3 presents experiments using DRUM to guide load balancing in the solution of the perforated shock tube problem [25]. In Chapter 6, we explain the main contribution of our research, we survey the latest efforts in DRUM development and we list some ideas for future work. We describe DRUM important data types and public interface in the Appendix.

CHAPTER 2

Background and related investigations

In this work, we are concerned with improving the effectiveness of dynamic load balancing procedures used in methods based on discretization methods such as the Finite Element Methods (FEM) and Adaptive Mesh Refinement (AMR) techniques. Before we present relevant research with similar objectives as ours, we start by reviewing the basics of adaptive mesh refinement and why and how it requires load balancing.

2.1 Adaptive applications

Applications involving the solution of partial differential equations are among the most demanding computational problems, arising in fields such as fluid dynamics [47], materials science [11] and biomechanics [1]. Most of these applications use the FEM and related methods as standard analysis tools. Better solution approximations, increased robustness and higher space and time efficiency are attained when adaptivity is included in FEM.

In Adaptive FEM [6], the problem domain is discretized with a mesh. The adaptivity is achieved through any combination of the following processes:

- An *h-refinement* process in which portions of the mesh are refined or coarsened.

- A *p-refinement* process in which the method order is modified.

- An *r-refinement* process during which the mesh is moved to follow solution features.

The adaptive applications of interest in our study are computationally demanding and, as such, they require parallelism for satisfactory execution. The usual approach to parallelizing these problems is to distribute a mesh of the domain across cooperating processors, and then compute a solution, appraising its accuracy using error estimates at each time step. If the solution is accepted, the computation proceeds to the next step. Otherwise, the solution is adaptively enriched, and work is redistributed, if necessary, to correct for any load imbalance introduced by the adaptive step. Thus, dynamic partitioning and load balancing procedures become necessary.

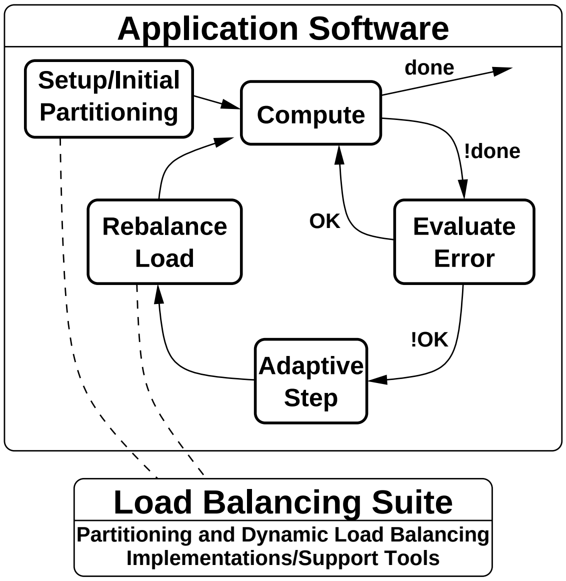


Figure 2.1: Program flow of a typical parallel adaptive computation using a load balancing suite such as Zoltan.

Figure 2.1 shows the typical interaction between parallel adaptive application software and dynamic load balancing procedures.

2.2 Mesh partitioning

Mesh partitioning, which consists of distributing regions of the mesh over a set of nodes, provides a natural route to parallelism. Formally, the mesh partitioning problem is described as follows. Given a graph $G = (V, E)$, where each vertex v in V corresponds to a mesh element and where each edge e in E models a connection between two mesh elements, a k-way partition is obtained by finding k subsets $V_1, V_2, ..., V_k$ of V such that:

$$V_1 \cup V_2 \cup ... \cup V_k = V$$

$$V_i \cap V_j = \emptyset, i, j \in \{1, ..., k\}, i \neq j$$

In Figure 2.2, we show a sample two-dimensional mesh (a) and a target parallel environment consisting of two 2-way SMP workstations connected by a network (b). Figure 2.2(c) shows a partitioning of this mesh and the assignment of those partitions to the processes and machines of the target environment. Six partitions are created and assigned to four processes, since four processors are available. Two processes are assigned two partitions, while the other two are assigned only a single partition. The four processes are further assigned to the available machines, two to each. Load balancing adds two extra constraints. The first one expresses the fact that the subsets $V_i, i = 1, ..., k$ need to have similar sizes within a certain tolerance. $|V_i| = \frac{|V|}{k} \pm$ tolerance, $i = 1, ..., k$

The second one emphasizes the necessity of a small number of edges between the

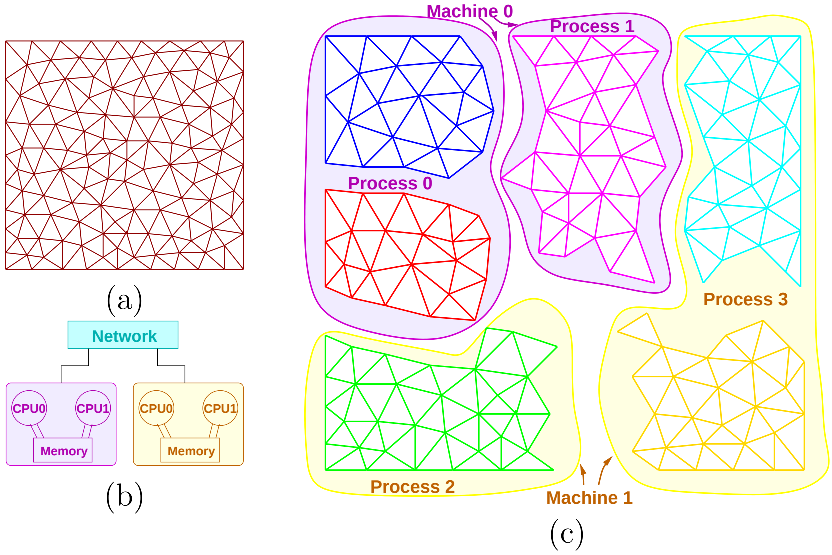


Figure 2.2: A sample two-dimensional mesh (a), target parallel environment in which the mesh is to be partitioned (b), and partitioning of the mesh and assignment to processes and machines for the parallel environment (c). ***Courtesy of Jim Teresco, Williams College.***

partitions (edge cut).

Minimize $|E_c|$ where $E_c = \{(u, v), u \in V_i, v \in V_j, i \neq j\}$

The load balancing problem is known to be an NP-hard problem. Given this, heuristics-based strategies for dynamic load balancing and partitioning have been developed. One class of strategies is formed by diffusive approaches [15, 61]. The general idea in diffusive load balancing is based on moving load from highly loaded processors to neighboring processors with less load. The process is repeated iteratively until a certain balance is reached. The dual approach, known as work stealing [7], is achieved when under-loaded processors pull the work from over-loaded processors.

Another class of LB techniques is composed of graph-based methods [8, 29, 32, 31, 58, 59, 57]. Many of the techniques is this class are based on either spectral methods [8, 29, 4, 3] or multilevel methods [32, 31, 58, 59, 57]. In multilevel methods, the original graph is first iteratively coarsened until a prescribed size is reached. Then an initial partitioning of the coarse graph is performed and the resulting partitions are mapped to the target processors. The partitions are then iteratively refined until the original graph is obtained. In each step of refinement, optimization techniques, which might generate data movement between partitions, are executed. Geometric methods such as those described in [60, 39, 42, 26] form a third class of partitioning techniques. One popular approach in this class is known as recursive coordinate bisection (RCB)[5]. In RCB, the mesh, often surrounded by a bounding box, is cut following a given dimension. The process is then recursively repeated until the number of partitions equals the number of target processors on which these partitions are (to be) mapped. A nice review of LB procedures and software is reported in [50].

The assumption of environment homogeneity in traditional LB techniques is increasingly unrealistic in today's systems. The quest for greater performance is answered by successive additions of new hardware to preexisting clusters. The new additions, consisting of more capable computational nodes and faster communication media, result in heterogeneous environments. Consequently, LB techniques need to account for heterogeneity in order to achieve reasonable effectiveness.

2.3 Load balancing for heterogeneous environments

The presence of hardware and software heterogeneity adds extra constraints to the optimization problem that load balancing procedures address. In the context of hardware heterogeneity, one of the questions is what sizes should partitions be such that they match the capabilities of the processors or collection of processors on which they are to be mapped. This question involves other questions such as how to quantify the capabilities of the target processors, how are these capabilities affected by non-dedicated usage of the resources, how to evaluate the success of a routine that takes heterogeneity into account when it does load balancing. Before we present our answers to these and related questions, we first discuss recent efforts that address load balancing and partitioning on heterogeneous clusters.

Walshaw and Cross [56] conduct multilevel mesh partitioning for heterogeneous communication networks. They base their technique on modifying a multilevel algorithm seeking to minimize a cost function based on a model of the heterogeneous communication network. The network model, which gives only a static quantification of the network heterogeneity, is supplied by the user at run-time as a Network Cost Matrix (NCM). The NCM implements a complete graph that models the processor links. Each edge is weighted as a function of the length of the path between processors. Empirical tests led the authors to consider a Quadratic Path Length (QPL), in which the path lengths are squared, as the NCM graph edge weight. We start our review of this work by reproducing some notations and definitions used by the authors. Let $G = (V, E)$ be an undirected graph. The vertices in V can either represent mesh nodes, mesh elements, a combination of both or some other special purpose representation. Edges in E represent the data dependencies in the mesh. $\mathcal{P}$ is a set of P processors on which the mesh is to be distributed. A partition $\pi : V \rightarrow \mathcal{P}$ is defined as a mapping of V into P disjoint subdomains S_p such that $\bigcup_P S_p = V$. The modified algorithm tries to achieve an even distribution of the load on the target computer system while also seeking to minimize an objective function Γ involving application edge weights and NCM weights

$$\Gamma = \sum_{(v,w) \in E_c} |(v,w)| \cdot |(\pi(v), \pi(w))|.$$

Here, E_c is the set of cut-edges (*i.e.*, edges between partition subsets), $|(v,w)|$ is an application-dependent edge weight, $\pi(v), \pi(w)$ are processors where vertices v, w are mapped on the target computer and $|(\pi(v), \pi(w))|$ is the NCM edge weight.

The multilevel algorithm starts by applying successive graph contraction to the application graph $G = (V, E)$ until a coarse graph where the number of vertices equals the number of processors is obtained. The contraction process consists in finding, at each step, a maximal independent set of edges[2] and contracting them. There might be multiple choices on which edges to select for contraction. The authors follow an idea developed by Karypis and Kumar [32] which consists of collapsing the most heavily weighted edges. This approach was shown to be beneficial to the optimization. Once the contraction process is complete, the algorithm partitions

[2] An independent set of edges is a set of edges that share no vertices.

the graph by assigning each vertex to a processor. A local optimization process is then administered to the partition before it is interpolated to its parent partition. The local optimization process is a variant of a Kernighan-Lin (KL) [35] bisection algorithm, which includes a hill-climbing mechanism to enable it to escape local minima. At the heart of the two nested-loop optimization algorithm, a gain function $g(v, q)$ of a vertex v that measures the gain in cost of a given partition if vertex v in subdomain S_p were migrated to sub-domain $S_q, q \neq p$ is computed as

$$g(v, q) = \Psi(\pi) - \Psi(\pi')$$

where π is the current partition, π' is the partition resulting from moving v to S_q, and Ψ is a partition cost function.

Given a vertex v in sub-domain S_p, the original gain function, computed with the assumption that network links have the same weight, is computed as

$$g(v, q) = |e_q(v)| - |e_p(v)|, \qquad e_q(v) = \{(v, w) \in E : w \in S_q\}.$$

The gain function is modified to include the weights of edges between sub-domains, which depend on the cost of communication between the processors owning these sub-domains. So, if $|(p, r)|$ denotes the weight of an edge between processors p and r (as specified in the NCM) and $\mathcal{P}$ is the set of processors of the target computer, the new gain function is computed as

$$g(v,q) = \sum_{r \in \mathcal{P}} |e_r(v)|(|(p,r)| - |(q,r)|).$$

Computation of the preference of a vertex v is a fundamental step in the optimization process. It consists in finding a processor q such that the migration of v to sub-domain S_q would yield a (local) maximum gain. To obtain this maximum for a given vertex v, one would need to iterate on each sub-domain S_q. The computation of the gain itself in not difficult and has complexity $O(P)$ where P is the number of processors in the cluster. Since the gain must be computed for each sub-domain and at each iteration of the optimization, the overall cost could be prohibitive if P is large. The authors proposed a simplification of the preference function that retains the spirit of the original while lowering its computational cost. Hence, for vertex $v \in S_p$, the maximum gain search space is reduced to a union of sub-domains adjacent to v together with processors adjacent to p in the processor graph.

An important part of Walshaw and Cross's work [56] consists in proposing metrics to evaluate the quality of a partition. Besides the classical metric for assessing partition quality

$$\Phi = |E_c| = \sum_{(v,w) \in E_c} |(v,w)|$$

which approximates the total communication volume for homogeneous networks, the authors use the following metrics:

- Network cost function (Γ):

$$\Gamma = \sum_{(v,w) \in E_c} |(v,w)| \cdot |(\pi(v), \pi(w))|$$

- Average dilation (Δ):

$$\Delta = \frac{\sum_{(v,w) \in E_c} |(\pi(v), \pi(w))|}{O(E_c)},$$

  with $O(E_c)$ denoting the number of edges in E_c (*i.e.*, the number of cut edges).

- Average path length/weighted dilation (δ):

$$\delta = \frac{\sum_{(v,w) \in E_c} l(\pi(v), \pi(w))}{O(E_c)},$$

  with $l(\pi(v), \pi(w))$ denoting the length of the path between $\pi(v)$ and $\pi(w)$.

The authors explain that a low average dilation, which indicates that most of the communication occurs across fastest links, is a signature of a good mapping. The proposed metrics, along with the partitioning run-time τ are used to evaluate the results of experimental comparisons of mappings of several 2-D and 3-D meshes on target clusters with topologies including 1-D array, 2-D mesh, SMP cluster, and meta-computer. Experiments confirmed that a quadratic path length effectively measures network cost between processors and the adjacent sub-domain/processor preference function in the computation of gain within the local optimization procedure. Walshaw and Cross [56] also showed that the resulting multilevel algorithm yielded low average dilation and better running time, without incurring a large increase in the cut-weight when compared with a standard multilevel algorithm coupled with a processor assignment algorithm.

Walshaw and Cross's approach [56] relies on the user supplying a NCM to be used at runtime. However, even with good data, the NCM cannot capture the dynamics of today's networks, especially in non-dedicated clusters. The authors are aware of this but argue that attempting to account for the network dynamics would cause the optimization to become intractable. Moreover, the mapping algorithm, again, noted by the authors, needs to be parallelized.

Minyard and Kallinderis [38] discuss the use of octree structures to perform parallel load balancing for dynamic execution environments. To account for the dynamic nature of the execution environment, they collect run-time measurements based on the "wait" times of the processors involved in the execution. These wait times measure how long each CPU remains idle while all other processors finish the same task. The cells are assigned a load factor that is proportional to the wait time of their processor. The octant load is then computed as the sum of the load factors of the cells contained within that octant. The octree then balances the load based on the weight factors of the octant, rather than the number of cells contained in each octant. Because the approach incurs a relatively small amount of data migration, it results in highly efficient load balancing. The proposed technique was shown to generate good quality partitions that remained almost the same, even after multiple adaptations and load balancings. The method was also shown to be effective in highly dynamic environments in which some processors might become overloaded [38].

Chen and Taylor [14] developed a tool for mesh partitioning on distributed systems. The tool, called PART, considers heterogeneity in the application as well as heterogeneity in processor and network performance. Execution environments consist of interconnected, possibly geographically-distributed, groups of processors. A group can be a cluster of homogeneous workstations, an SMP or a tightly-coupled parallel computer. Processors within a group are assumed to have the same performance. Communication within a group is referred to as local communication (C_L) while inter-group communication is labeled remote communication (C_R). In general, $C_R >> C_L$; thus, the basic idea of their approach is that processors having to do both local and remote communication should be given less work, in order to compensate for the overhead generated from remote communication. To balance execution time within a group, a "retrofit" step is used within PART. Chen and Taylor [14] first consider stripe partitioning for which communication occurs at most with two neighboring processors. In this special case, they measure the ideal reduction in execution time $T_{reduction}$ that can happen within a group G as

$$T_{reduction} = \frac{|G| - \nu}{|G|} \Delta(\nu, |G|)$$

where $|G|$ is the number of processors within group G, ν is the maximum number of processors among the groups that have remote communication and $\Delta(\nu, G)$ is the difference in execution time for the processors with local and remote communication versus processors with only local communication within G.

Chen and Taylor [14] argue that delegating remote communication to only one processor within a group can maximize the reduction in execution time. They study the case of a $2n \times n$ mesh partitioned in either $2P \ \ (n/\sqrt{P} \times n/\sqrt{P})$ blocks or $2P \ \ (n \times n/P)$ stripes. They evaluate the difference in total execution time between block and stripe partition as:

$$\Delta T_{Total}^{blocks-stripes} = \frac{2P - \sqrt{P} + 1}{P}\alpha_L + \frac{4 - 2\sqrt{P}}{\sqrt{P}} n\beta_L + \frac{\sqrt{P} - 1}{P}\alpha_R$$

where α_L and β_L are, respectively, the per-message and per-byte costs of local messages and α_R the per-message cost of remote messages. The values of α_L and α_R are two orders of magnitude larger than β_L. The first and third fractional terms above are always positive as $P > 1$. The middle term is also positive when $P \leq 4$, which means that the stripe partitioning yields a smaller execution time. Chen and Taylor [14] explain that even when $P > 4$, stripe partitioning is still advantageous unless n is so large that the absolute value of the middle fractional term is larger than the sum of the first and last fractional terms. In experiments conducted by Chen and Taylor [14], this happens when $n > 127KB$.

Partitioning in PART is a three-phase process. In the first phase, the mesh is partitioned into S divisions corresponding to the S groups composing the distributed environment. This step includes processor and element heterogeneity. In the second phase (retrofit), each of the S divisions is partitioned across the processors of its corresponding group. Heterogeneity in network performance and element type are taken into consideration. A third phase (global retrofit) may be performed if there is a big variance in the execution time of the different groups. This happens when there is a significant performance difference in interconnection networks of the groups. Both the local and the global retrofit phases use a parallelized simulated annealing with multiple Markov chains algorithm [12].

Chen and Taylor [14] conducted experiments to evaluate the efficiency of their partitioner. PART was applied to WHAMS2D [13], an explicit nonlinear finite-element code used to analyze elastic plastic materials. Globus was used to run the code on two geographically-distributed IBM SP computers. Nodes in the first SP were 1.6 times faster than those of the second. Code used both regular and irregular meshes. Results for the regular meshes showed a 33 to 46 percent improvement in efficiency when only heterogeneity in processors' performance is considered. An additional gain ranging from 5 to 15 percent was noticed when network heterogeneity was considered. The theoretical maximum for the network heterogeneity experiment was 15 percent. Significant improvement was also noticed for problems with irregular meshes. Results indicated a 36 percent increase in efficiency when both processor and network heterogeneity were taken into consideration.

Gross *et al.* [28] proposed a resource monitoring system called *Remos*, which allows applications to collect information about network and host conditions across different network architectures. Remos was used to run adaptive distributed applications on heterogeneous networks. Other efforts for load balancing on heterogeneous networks include the profile-based load by Banikazemi *et al.* [2] and the diffusive load balancing schemes of Elsasser *et al.* [23]

The research of Sinha and Parashar [46] in which they present a framework for adaptive system-sensitive partitioning and load balancing on heterogeneous and dynamic clusters is closer to ours. Their approach relies on the Network Weather Service (NWS) [63] tool to gather information about the state and capabilities of available resources. To each node in the execution environment, they associate a capacity number computed as a weighted sum of processing, memory, and communications capabilities.

For each node k, NWS tools are used to compute the percentage of available CPU $\mathcal{P}_k$, available memory $\mathcal{M}_k$ and available link bandwidth $\mathcal{B}_k$ for all K processors. These values are first normalized, *i.e.*,

$$P_k = \frac{\mathcal{P}_k}{\sum_{i=1}^{K} \mathcal{P}_i}, M_k = \frac{\mathcal{M}_k}{\sum_{i=1}^{K} \mathcal{M}_i}, B_k = \frac{\mathcal{B}_k}{\sum_{i=1}^{K} \mathcal{B}_i}$$

Then, the capacity of each node k is computed as:

$$C_k = w_p P_k + w_m M_k + w_b B_k$$

where w_p, w_m, w_b are weights measuring the contribution of each of the three factors in the overall capacity computation, with $w_p + w_m + w_b = 1$. These weights are application dependent and reflect its computational, memory and communication requirements.

The capacity calculator was integrated within the framework of GrACE (Grid Adaptive Computational Engine),“an object-oriented toolkit for the development of parallel and distributed applications based on a family of adaptive mesh-refinement and multigrid techniques” [46]. GrACE is a multi-layer infrastructure. The lowest layer implements a hierarchical distributed dynamic array (HDDA) data structure. HDDA objects hide from the user the complexity of managing data distributed across multiple address spaces. This includes a user-transparent management of communication, synchronization, consistency and dynamic load balancing. The HDDA data structure has standard array semantics which, according to the authors, has appreciable affinity with semantics of operations on grids and benefit from extensive compiler optimization. On top of the HDDA layer is implemented a layer that “provides an object-oriented programming interface for directly expressing multi-scale, multi-resolution AMR computation” [46]. The upper-layers of GrACE provide facilities for method-specific computations.

Sinha and Parashar [46] conducted experiments to evaluate the system-sensitive partitioning framework. Heterogeneity was simulated by adding multiple synthetic loads on selected processors. Experiments were run on a 32-node Linux cluster connected with a fast Ethernet (100MBs) network. Results showed that system-sensitive partitioning resulted in a decrease of application execution time ranging from 6% on 8 processors to 18% on 32 processors. The authors noticed even better improvement when dynamic sensing was used. They acknowledge that the weights used in the computation of the capacity are assigned arbitrarily. Our work uses a similar idea for computing a node's power as a weighted sum of contributing powers. Yet, the processing and communication powers used in our model handle more general cases that involve symmetric multiprocessors (SMPs), hierarchical clusters and also highly dynamic execution enviroments.

Wong, Jin and Becker [64] investigated the feasibility of running parallel applications on heterogeneous clusters. They run multi-zone versions of the NAS parallel benchmarks on a cluster made of two SGI origin 2000 servers and an Intel SMP Xeon server. The NAS parallel benchmarks used are BT-MZ, LU-MZ and SP-MZ. These consist of simulated applications that use different numerical schemes to compute the solution of the unsteady compressible Navier-Stokes equations on a 3D grid. The authors use a hybrid OpenMP [10] / MPI parallelization. OpenMP is used for intra-zone parallelism while MPI is used for inter-zone message passing based concurrency. The multi-zone load balancing is achieved via a modified bin-packing algorithm. The modification is introduced to handle OpenMP threads. The authors also propose an extension to handle heterogeneous clusters. This extension encompasses the following modifications:

- A notion of processor power factor f_p is introduced. f_p is a relative number that measure how fast processor p is when compared with a baseline processor. In the modified version, the bin-packing, which relies on the computational workload of each zone to assign zones to processors, takes into account the power factor when estimating a zone size.

- A customization of the number of threads acting on a zone group so that additional load could be adjusted for a specific processor type.

- A more dynamic approach to computing processor power factors based on a timing of an iteration of the actual benchmark.

Wong, Jin and Becker [64] conducted a variety of experiments using their multi-zone load balancing algorithm. The results showed the effectiveness of LB with a static approach based on OpenMP threads and a dynamic approach based on iteration profiling. They also showed that tightly-coupled hybrid applications can achieve fairly good performance on heterogeneous clusters.

Kumar, Das and Biswas [36] propose a graph partitioning algorithm for Parallel applications in Heterogeneous Grid Environments. They present a novel multilevel partitioning approach for irregular graphs and meshes that takes into account the inherent hardware heterogeneity present in grid environments. The algorithm, called MiniMax, produces and maps partitions onto a heterogeneous system with the objective of minimizing the maximal execution time of the parallel application. They start by proposing a workload and system model. The workload is modeled as weighted graph $G = (V, E)$. Each vertex v in V has a computational weight W^v, that indicates the amount of computation at v. A communication weight $C^{u,v}$ reflects the level of data dependency between vertices u and v. The heterogeneous system is also modeled as weighted graph $S = (P, L)$ where $P = \{p_1, p_2, ..., p_n\}$

denotes the set of processors in the system and $L = \{(p_i, p_j) | p_i, p_j \in P\}$ represents the communication links between the processors. A processing weight s_p is associated with each vertex p in P. This weight indicates the processing cost per unit of computation of processor p. Similarly, a link weight $c_{p,q}$ is attached to model the communication cost between processors p and q. A communication cost matrix gathers the communication costs between any pair of processors in P. When there is no direct edge between two processors, their link weight is the sum of weights of links forming a shortest path between them. The authors assume full duplex communication which yields a symmetric communication cost matrix.

For a given mapping of the workload graph on the system graph, the processing cost of a vertex v assigned to processor p is measured as:

$$W_p^v = W^v \times s_p$$

This cost reflects how many time units are needed by processor p to process vertex v. Similarly, the communication cost generated by data communication between any given vertices u and v is

$$C_{p,q}^{u,v} = C^{u,v} \times c_{p,q}.$$

The system is then evaluated as a whole. The potential of the system Φ_{sys} is quantified as the sum of individual processors potential:

$$\Phi_{sys} = \sum_{p \in P} \phi_p,$$

where the potential ϕ_p of an individual processor p is defined as:

$$\phi_p = \frac{1}{e_p + \delta c_p}$$

where e_p indicates the mean time required by processor p to compute a vertex, averaged over all vertices of the workload

$$e_p = \frac{\sum_{v \in V} W^v}{|V|} \times s_p.$$

Similarly, c_p indicates the mean time needed by p for an inter-vertex communication, averaged over all edges of the workload, i.e.,

$$c_p = \frac{\sum_{(u,v) \in E} C^{u,v}}{|E|} \times \overline{c_p}$$

where $\overline{c_p} = \sum_{q \in N, q \neq p} c_{p,q}/\delta_p$, indicates the mean link weight at p, averaged over all links incident on p, δ_p the degree of processor p and $\delta = 2 \times |E|/|V|$ the average degree of the workload graph. The system potential is, hence, a function of the number of processors, the processor and link weights and the network connectivity.

Before proposing their new partitioning algorithm, Kumar, Das and Biswas

[36] discuss some of the problems existing in the standard partitioning techniques. They note first that the standard approaches to partitioning do not satisfactorily address the mapping question, which controls the assignment of vertices to processors. Along with this remark, they explain how a separate mapping algorithm does not offer the flexibility of a posterior change of vertex assignment that might result in a reduced execution time. The authors then argue that most traditional partitioning schemes don't necessarily minimize the right metric. In fact, they operate by attempting to create an even distribution of the load across partitions while also minimizing the edge cut across partitions. They then state that minimizing the edge cut does not necessarily yield a minimization of the parallel execution of the application. The authors explain that this latter is rather dependent on the slowest processor in the execution environment of the parallel application. Therefore, they note, one should minimize the maximum execution time ET among processors:

$$ET = max\{ET_p\}, p \in P$$

where

$$ET_p = \sum_{v\mathcal{A}p} W^v \times s_p + \sum_{v\mathcal{A}p} \sum_{u\mathcal{A}q, p \neq q} C^{u,v} \times c_{p,q}.$$

Here, $v\mathcal{A}p$ represents an assignment of vertex v to processor p.

The authors propose MiniMax, a multilevel partitioning scheme that targets distributed heterogeneous systems such as the Grid. The novelty of the approach, the authors argue, is that it takes into account heterogeneity in both the system and workload graphs. They also state that MiniMax generates high quality partitions for all kinds of parallel Grid applications.

The algorithm operates in three steps. The first step consists of the workload graph coarsening. The coarsening is iteratively applied until a given threshold is reached. In the second step, the coarsest graph is partitioned among processors with the aim of achieving a minimal execution time of the parallel application. The authors correctly note that an optimal partition of the coarsest graph is not necessarily the best partition for the original graph. Therefore, in the third step of the algorithm, where an iterative refinement process is operated on the coarsest graph to obtain the original graph, an optimization technique is applied at each level in order to shorten the application execution time. This optimization might require a possible migration of vertices between processors as this could minimize the maximum execution time among all processors. The authors cite several approaches to obtain a maximal matching before concluding that a variant of the "heavy clique" approach works best for their partitioner.

The initial partitioning is performed when the number of vertices in the coarsest graph is below a predetermined threshold. The authors use a $|P|$-way graph growing algorithm to assign each vertex in V to one of the vertices in P. The authors propose a greedy strategy that constrains the growth of a region corresponding to a processor. This growth should in fact happen along a neighboring vertex w that yields a minimum increase in processor execution time. When vertex v is mapped to processor p, the gain in execution time is expressed as:

$$Gain_p^v = W^v \times s_p + \sum_{u\mathcal{A}q, \forall q \in P, q \neq p} C^{u,v} \times c_{p,q}.$$

The resulting execution time is therefore:

$$ET_p^v = ET_p' + Gain_p^v$$

where ET_p' is the execution time before the assignment of v to p.

Several levels of refinement are used to produce the original graph from its coarsened version. At each refinement step, the scheme investigates possible vertex migration that could optimize the application execution time. Kumar, Das and Biswas [36] propose three vertex migration strategies to reduce the maximum execution time of the application. They also note than no other schemes besides the three proposed could reduce the maximum execution time.

The authors conducted a variety of experiments to study and illustrate the performance of MiniMax. The quality of the partitions produced by MiniMax is quantified by three metrics:

1. Application execution time

2. Standard Deviation among processor execution time:

$$\delta = \sqrt{\sum_{p \in P} (ET_p - ET_{av})^2 / |P|}$$

where $ET_{av} = \sum_{p \in P} ET_p/|P|$ is the average execution time among processors.

3. Imbalance factor $Imb = MaxET/ET_{av}$ where $MaxET$ refers to the maximum execution time among processors.

The experimental results [36] showed that MiniMax produced high quality partitions. For all experiments, MiniMax produced values for δ orders of magnitude lower than ET, which indicated that all processors have close execution times and hence an imbalance factor close to 1.

The authors did not supply any comparisons with other approaches to partitioning for heterogeneous environments. They attribute the absence of such comparisons to the innovative aspect and the general environments their approach targets. They note, however, that a customized version of MiniMax which targets the usual (limited) problem domains covered by traditional approaches will open the possibility to conduct a comparative study.

CHAPTER 3

The Dynamic Resource Utilization Model (DRUM)

In this chapter, we describe our proposed model for load balancing on heterogeneous and non-dedicated environments. We present DRUM, a model that incorporates aggregated information about the capabilities of the network and computing resources composing an execution environment. DRUM can be viewed as an abstract object that

- encapsulates the details of the execution environment,
- provides facilities for dynamic, modular and minimally-intrusive monitoring of the execution environment and,
- exposes a simple interface for direct use by existing load balancers.

The Rensselaer Partition Model (RPM) [49], a precursor of this work, used a directed-graph hardware model. DRUM, in difference to RPM, relies on a tree structure. DRUM also incorporates a framework that addresses hierarchical clusters (*e.g.*, clusters of clusters, or clusters of multiprocessors) by capturing the underlying interconnection network topology. The tree structure in DRUM leads naturally to a topology-driven execution of hierarchical partitioning [52]. The root of the tree represents the total execution environment. The children of the root node are high-level divisions of different networks connected to form the total execution environment.

Sub-environments are recursively divided, according to the network hierarchy, with the tree leaves being individual single-processor (SP) nodes or shared-memory multiprocessing (SMP) nodes. *Computation nodes* at the leaves of the tree have data representing their relative computing and communication power. *Network nodes*, representing routers or switches, have an aggregate power calculated as a function of the powers of their children and the network characteristics.

We quantify the heterogeneity of the different components of the execution environment by assessing computational, memory and communication capabilities of each node. The collected data in each node is combined in a single quantity called the node "power." For load balancing purposes, we interpret a node's power as the percentage of overall load it should be assigned based on its capabilities.

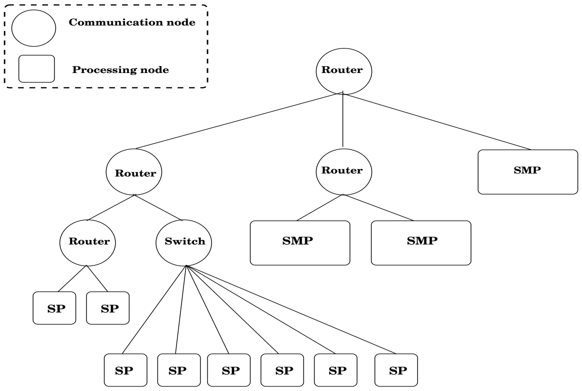


Figure 3.1: Tree constructed by DRUM to represent a heterogeneous network.

Figure 3.1 shows an example of a tree constructed by DRUM to represent a heterogeneous cluster. Eight SP nodes and three SMP nodes are connected in a hierarchical network structure consisting of four routers and a network switch.

Figure 3.2 shows the Bullpen heterogeneous cluster at Williams College. The cluster is composed of two 4-node SMPs, six 2-node SMPs and four SPs, all connected by a fast Ethernet switch. A more detailed description of the Bullpen cluster components is given in http://bullpen.cs.williams.edu and is reproduced here for convenience.

- One Enterprise 220R server with one 450MHz SPARC UltraII processor, 512 MB memory, and a 190GB RAID5 array, acting as file server and interactive login node. (bullpen)

- Two Enterprise 420R servers, each with four 450MHz SPARC UltraII processors, 4 GB memory, 72GB local disk. These are compute nodes. (rivera, wetteland)

- Six Enterprise 220R servers, each with two 450MHz SPARC UltraII processors, 512 MB or 1 GB memory, 36GB local disk. These are compute nodes. (mcdaniel, farr, lyle, gossage, righetti, arroyo)

- Four Ultra 10's, each with one 300 or 333 MHz SPARC UltraII processor, 128 MB memory, and 6 GB local disk. These are "slow" compute nodes, useful mainly in studies of heterogeneous systems. (mendoza, stanton, nelson, lloyd)

- Both 420R's (rivera, wetteland) and two of the 220R's (farr, mcdaniel) have gigabyte interconnect using WulfKits from Dolphin, Inc.

Figure 3.3 shows the model constructed by DRUM for this cluster. Figure 3.4 shows a different model of a reconfigured Bullpen cluster in which some computation nodes have been attached to a 10Mbs switch before being connected to the rest of the cluster through the 100MBs switch.

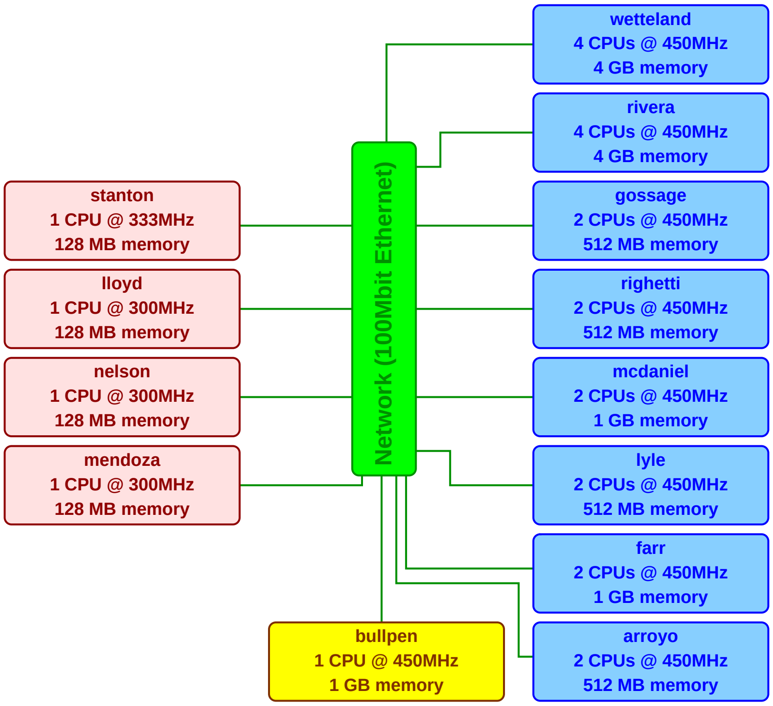


Figure 3.2: The "Bullpen Cluster" at Williams college.

3.1 Model creation

DRUM requires a tree model of the execution environment's underlying interconnection. An XML file (Figure 3.5), in which the list of nodes and description of their interconnection topology is used by a configuration tool to generate the initial data structures of DRUM. The configuration tool provides capabilities including (*i*) XML file generation using a graphical interface, (*ii*) initial assessment of node capabilities by running distributed benchmarks, and (*iii*) facilities to check availability of network management capabilities such as SNMP (Simple Network Management

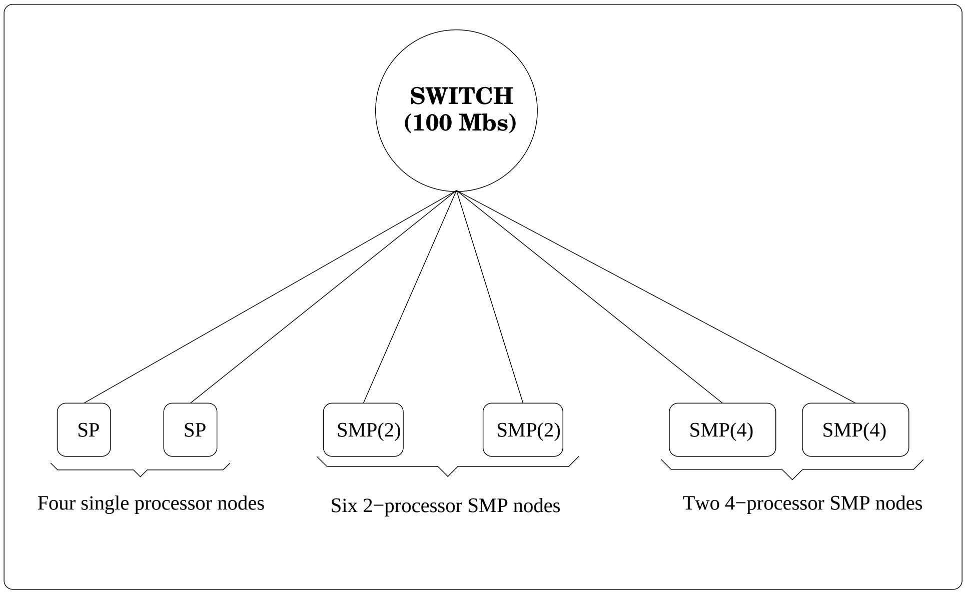

SWITCH
(100 Mbs)
SP
SP
SMP(2)
SMP(2)
SMP(4)
SMP(4)
Four single processor nodes
Six 2−processor SMP nodes
Two 4−processor SMP nodes

Figure 3.3: The "Bullpen Cluster" model built by DRUM.

Protocol) and threading capabilities. The configuration tool needs to be re-run only when hardware characteristics of the system have changed.

3.1.1 Capabilities assessment

Resource capabilities are assessed initially using benchmarks and are updated dynamically by *agents*: threads that run concurrently with the application to monitor each node. Currently, LINPACK [22] is used as a benchmark to compute a MFLOPS rating for the computation nodes of the cluster. The benchmark may be run from the configuration tool (Figure 3.6) or at the command line.

An application may use only the static information gathered from the original benchmarks (*e.g.*, if the system does not support threads), or may also use dynamic monitoring. The `startMonitoring()` function spawns the agent threads. Some

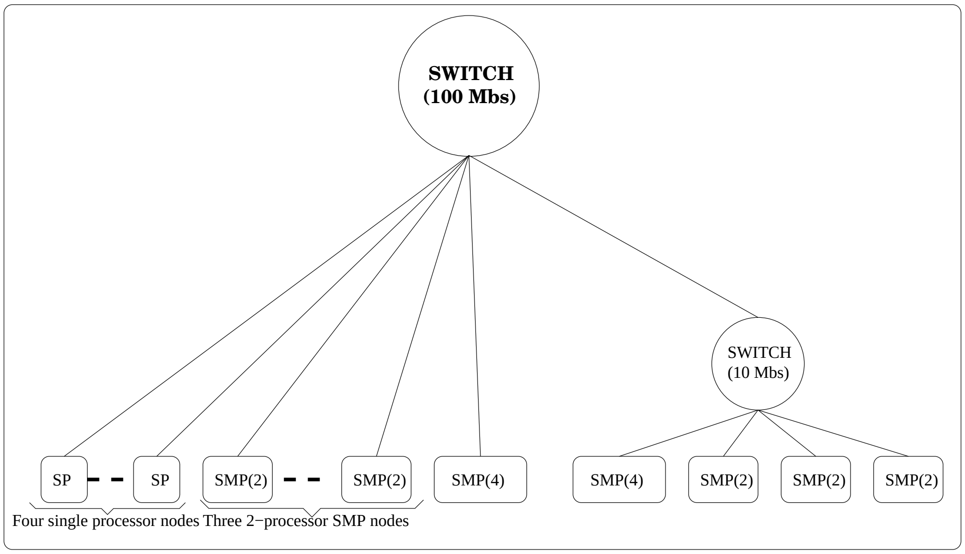


Figure 3.4: Reconfigured "Bullpen Cluster" model built by DRUM.

computation nodes, called *representatives*, are responsible for monitoring one or more network nodes. A call to `stopMonitoring()` ends the dynamic monitoring and updates the power of the nodes in the model. The monitoring agents contain (*i*) `commInterface` objects that monitor communication traffic, and (*ii*) `cpuMemory` objects that monitor CPU load and memory usage.

A `commInterface` object can be attached to either a computation or a network node. `commInterface` objects have been implemented using the `net-snmp` library[3] to collect network traffic information at each node. The network interfaces are probed for incoming and outgoing traffic. From this, we estimate the average rate of incoming packets λ and outgoing packets μ on each relevant communication interface. The *Communication Activity Factor* $CAF = \lambda + \mu$ is computed for

[3] `http://www.net-snmp.org`

```
<machinemodel numnode="7">
<node type="NETWORK_NODE" name="0" description="192.168.1.1" childrenNum="2"
 children="1;2;" IsMonitorable="1"></node>
<node type="NETWORK_NODE" name="1" description="192.168.2.2" childrenNum="2"
 children="3;4;" IsMonitorable="1"></node>
<node type="COMPUTING_NODE" name="3" description="n11.ns.cs.rpi.edu"
 childrenNum="0" children=";" IsMonitorable="1"></node>
<node type="COMPUTING_NODE" name="4" description="n12.ns.cs.rpi.edu"
 childrenNum="0" children=";" IsMonitorable="1"></node>
<node type="NETWORK_NODE" name="2" description="192.168.3.2" childrenNum="2"
 children="5;6;" IsMonitorable="1"></node>
<node type="COMPUTING_NODE" name="5" description="n13.ns.cs.rpi.edu"
 childrenNum="0" children=";" IsMonitorable="1"></node>
<node type="COMPUTING_NODE" name="6" description="n14.ns.cs.rpi.edu"
 childrenNum="0" children=";" IsMonitorable="1"></node>
</machinemodel>
```

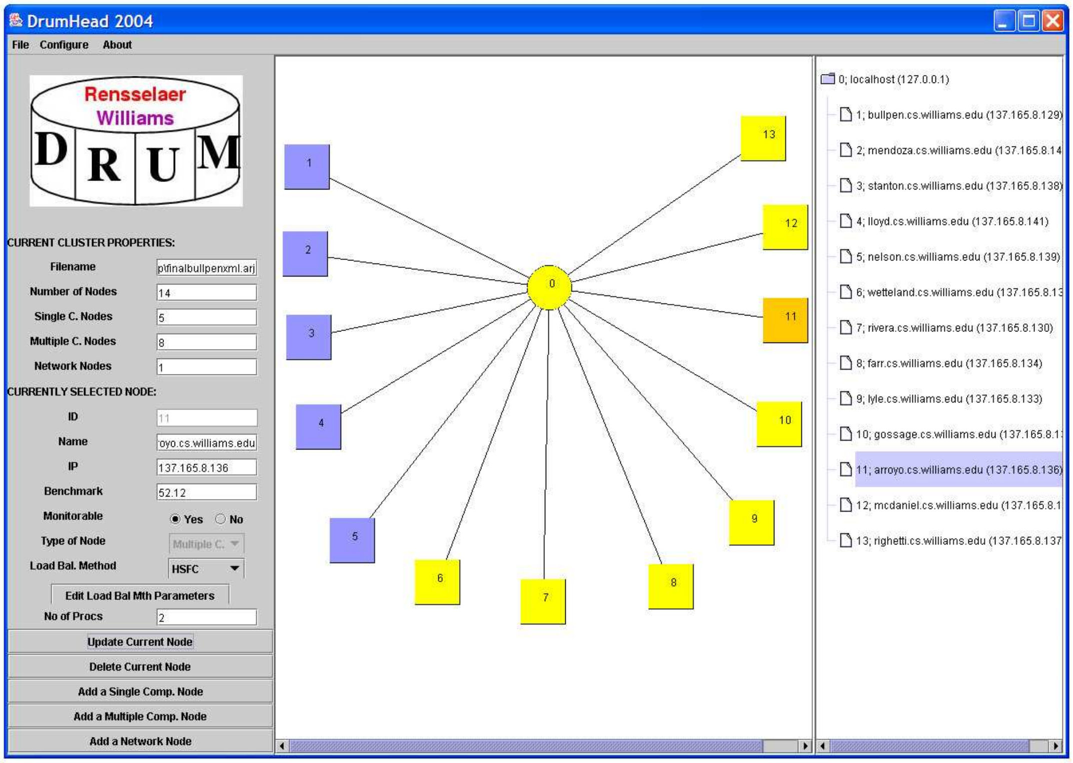
Figure 3.5: XML file description of the IBM Netfinity cluster used in the computation in Section 3.2. The cluster includes a hierarchical network.

all communication interfaces of each node. This also permits us to determine the average available bandwidth on each communication interface.

A `cpuMemory` object gathers information about the CPU load and the memory capacity of a computation node using kernel statistics. The statistics are combined with the static benchmark data to obtain a dynamic estimate of the processing power. A detailed description of the monitors, important data types and public interface of DRUM is given in the Appendix.

3.1.2 Node power

DRUM distills the information in the model to a power value for each node, a single number indicating the percentage of the total load that should be assigned to that node. This is similar Sinha and Parashar's approach [46]. Given power values for each node, any partitioning procedure capable of producing variable-sized



CURRENT CLUSTER PROPERTIES:

Filename	p\finalbullpenxml.arj
Number of Nodes	14
Single C. Nodes	5
Multiple C. Nodes	8
Network Nodes	1

CURRENTLY SELECTED NODE:

ID	11
Name	oyo.cs.williams.edu
IP	137.165.8.136
Benchmark	52.12
Monitorable	● Yes ○ No
Type of Node	Multiple C.
Load Bal. Method	HSFC
No of Procs	2

0; localhost (127.0.0.1)
- 1; bullpen.cs.williams.edu (137.165.8.129)
- 2; mendoza.cs.williams.edu (137.165.8.14
- 3; stanton.cs.williams.edu (137.165.8.138)
- 4; lloyd.cs.williams.edu (137.165.8.141)
- 5; nelson.cs.williams.edu (137.165.8.139)
- 6; wetteland.cs.williams.edu (137.165.8.13
- 7; rivera.cs.williams.edu (137.165.8.130)
- 8; farr.cs.williams.edu (137.165.8.134)
- 9; lyle.cs.williams.edu (137.165.8.133)
- 10; gossage.cs.williams.edu (137.165.8.1
- 11; arroyo.cs.williams.edu (137.165.8.136)
- 12; mcdaniel.cs.williams.edu (137.165.8.1
- 13; righetti.cs.williams.edu (137.165.8.137

Figure 3.6: Screen shot of the graphical program, drumhead, used to aid in the creation of the XML machine topology description. Here, a description of the cluster used for the computation in Section 5.2 is being edited.

partitions may be used to achieve an appropriate decomposition.

The power at each node depends on processing power, memory capacity, and communication power. At present, we compute the power of node n as the weighted sum of only the processing power p_n and communication power c_n

$$power_n = w_n^{comm} c_n + w_n^{cpu} p_n, \quad w_n^{comm} + w_n^{cpu} = 1.$$

The values of c_n and p_n are normalized across each level of the tree. The levels of the tree are identified in a top-down fashion where the root is at level 0, its immediate children at level 1 and so on. Figure 3.7 shows a DRUM tree composed of 4 levels. Let l denote the number of levels of a given DRUM tree and let $L_i, 0 \leq i \leq l$ denote the set of tree nodes at level i. The power of node n in L_i is given as

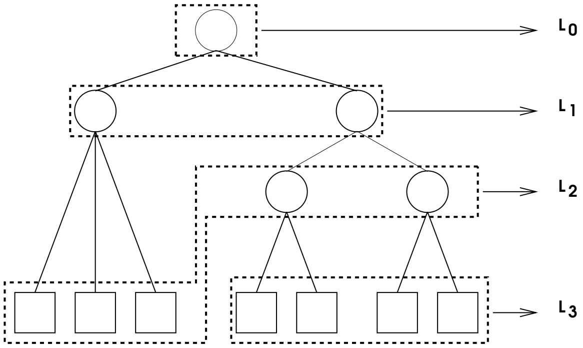


Figure 3.7: Levels of a tree constructed by DRUM.

$$power_n = pp_n \left(w_n^{comm} \frac{c_n}{\sum_{j=1}^{|L_i|} c_j} + w_n^{cpu} \frac{p_n}{\sum_{j=1}^{|L_i|} p_j} \right)$$

where pp_n denotes the power of the parent of node n (the power of the root node is 1).

3.1.2.1 Processing power

We evaluate a computation node's processing power based on: (*i*) CPU utilization u_n by the local process of the parallel job, (*ii*) Percent of CPU idle time i_n and (*iii*) the node's MFLOPS rating obtained from benchmarking b_n. In an early version of DRUM, we estimate the processing power node n as follows:

$$p_n = w_{dynamic}\ (u_n + i_n) + w_{static}\ b_n.$$

The weights $w_{dynamic}$ and w_{static} add to unity and their values depend on the length of the inter-balancing period. At the start of the computation, the value of $w_{dynamic}$ should be 0 since no dynamic monitoring has yet been performed. The rate at which $w_{dynamic}$ is increased depends on the length of the monitoring and the variability of the collected statistics. The maximum value of $w_{dynamic}$ should be bounded in order to ensure a non-negligible contribution of the static benchmarks in the overall expression.

In a more recent version of DRUM, we modified the expression of processing power to account for symmetric multiprocessors (SMPs). We also reformulated the expression in a way that did not require static and dynamic weights.

For a computation node n with m CPUs on which k_n application processes are running, we evaluate the processing power $p_{n,j}$ for each process j, $j = 1, 2, \ldots, k_n$, on node n based on:

- CPU utilization $u_{n,j}$ by process j. This indicates how much attention process j is getting from the node. This information is particularly important in the case of non-dedicated usage of the cluster.

- The fraction i_t of time that CPU t is idle. $t = 1, ..., m$.

- The node's static benchmark rating (in MFLOPS) b_n. Recall that the bench-

> mark number is obtained prior to the execution of the user application using DRUM; hence, it does not generate a computation overhead when the user application is running.

The overall idle time in node n is $\sum_{t=1}^{m} i_t$. However, when $k_n < m$, the k_n processes can make use of only k_n processors at any time, so the maximum exploitable total idle time is $k_n - \sum_{j=1}^{k_n} u_{n,j}$. Therefore, the total idle time that the k_n processes could exploit is $\min(k_n - \sum_{j=1}^{k_n} u_{n,j}, \sum_{t=1}^{m} i_t)$.

We work under the assumption that the operating system's CPU scheduler can be expected to give each of the k_n processes the same portion of time on a node's CPUs. In some systems, like SGI Irix, the scheduler might lock a given process to a specific CPU. In other systems, the scheduler might give higher priority to a specific process. However, the assumption of equal portion of time on CPUs is valid for most modern CPU schedulers. Hence, under this assumption, we assign all processes on node n equal processing power. We compute average CPU usage and idle times per process:

$$\overline{u}_n = \frac{1}{k_n}\sum_{j=1}^{k_n} u_{n,j}, \quad \overline{i}_n = \frac{1}{k_n}\min(k_n - \sum_{j=1}^{k_n} u_{n,j}, \sum_{t=1}^{m} i_t) \ .$$

Processing power $p_{n,j}$ is estimated as $p_{n,j} = b_n(\overline{u}_n + \overline{i}_n),\ j = 1, 2, \ldots, k_n$. Since $p_{n,j}$ is the same for all processes j on node n, $p_n = \sum_{j=1}^{k_n} p_{n,j} = k_n p_{n,1}$.

The processing power of internal nodes is computed as the sum of the powers of the node's immediate children. The propagation of powers to internal nodes is useful for hierarchical LB balancing strategies [52].

3.1.2.2 Communication power

The formulation of communication has evolved along the development of DRUM. In the earliest formulation, we view a node's communication power as inversely proportional to the value of the CAF at that node. The CAF inherently encapsulates dynamic information about the traffic passing through a node, the communication traffic at neighboring nodes, and, to some extent, the traffic in the overall system. The interface speed is also a factor; thus, given two nodes with equal CAF values, the one with the faster interface should be assigned a greater communication power. The communication power of nodes having more than one communication interface (*e.g.*, routers) is computed as the average of the powers on each interface. Thus, if we let $s_{n,i}$ denote the speed (expressed in MB/s), k the number of interfaces, and $CAF_{n,i}$ the CAF of interface i of node n, we compute the communication power as

$$c_n = \frac{1}{k}\sum_{i=1}^{k} \frac{log(s_{n,i})}{1 + CAF_{n,i}}.$$

The *log* function is applied to harmonize the range of values of $s_{n,i}$ and $CAF_{n,i}$. Similarly, let CAF_n represent the average CAF across all relevant interfaces of node n, then we model the communication weight factor as

$$w_n^{comm} = 1 - \frac{1}{1 + CAF_n}.$$

Nodes having communication interfaces with large CAF values do more communication than those with small CAF values. This weight is meaningful only if considered as a relative value.

In a second formulation of the communication power, we express the fact that a realistic weighting depends on the ratio of time spent in communication and time spent in processing. A correct estimation of this ratio without placing probes in the user application is not obvious. Even with such probes, a good approximation of these weights may be difficult if communication and processing are overlapped. Given this, we decide to assign the value of w_n^{comm} manually. In this version of DRUM, we still use the CAF number as the basis for communication power computation. Hence, if we let $CAF_{n,i}$ denote the CAF of interface i of a node n with s interfaces. We estimate the communication power as

$$c_n = \frac{1}{\frac{1}{s}\sum_{i=1}^{s} CAF_{n,i}}.$$

In practice, software loop-back interfaces and interfaces with $CAF = 0$ are ignored.

The third and most recent formulation of the communication power uses the available bandwidth at each node as the basis for measuring its communication power. For interface i of node n, we compute the available bandwidth $A_i(T)$ as:

$$A_i(T) = C_i - (\lambda_i(T) + \mu_i(T)).$$

where T denotes the monitoring period, C_i the maximum bandwidth at interface i and $\lambda_i(T)$ and $\mu_i(T)$ the rate of incoming and outgoing packets at interface i. Hence, for node n with s communication interfaces, we estimate its communication power as:

$$c_n = \sum_{i=1}^{s} A_i(T).$$

To compute per-process communication powers for processes $j, j = 1, 2, \ldots, k_n$, on node n on which k_n application processes are running, we compute c_n and associate $\frac{1}{k_n} c_n$ with each process. For consistency, if at least one non-root network node cannot be probed for communication traffic, all internal nodes are assigned CAF values computed as the sum of their immediate children's values.

3.2 Example of computed powers

We performed preliminary tests to experiment with the formulas involved in the power computation. The test bed is a sub-cluster of a previous configuration of the RPI netfinity cluster shown in Figure 3.8. We used the left sub-cluster rooted at Router $R1$. The four nodes involved have similar computational capabilities according to the LINPACK benchmark runs. We simulated heterogeneity by adding synthetic computation and communication load on selected processors. Processing and communication powers are computed dynamically. We run code solving a Rayleigh-Taylor instability problem in a rectangular parallelepiped containing an ideal gas. A complete description of the test problem can be found in [27].

Figure 3.10 shows the computed powers which represent percentages of the total load that should be assigned to each node. When no extra load is added to the homegeneous system, the computed powers, and subsequently the assigned load, are similar. In the second case, we added an extra communication load between nodes

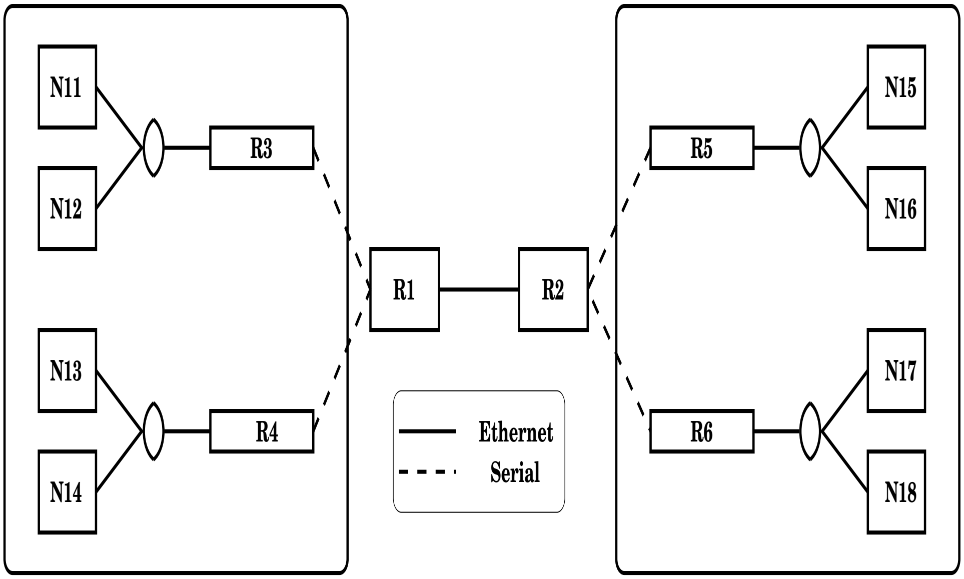


Figure 3.8: An early configuration of the RPI Computer Science Netfinity cluster.

$N11$ and $N12$. DRUM has reacted to the added load resulting in smaller relative powers assigned to $N11$ and $N12$. In the third case, we added extra computatonal load on node node $N14$. This resulted in a small power assigned to $N14$. In each case, $N11$, the master process, performs additional work coordinating the other processes and managing the standard output from all processes, so it is given a smaller percentage of work by DRUM. The ability to account for these otherwise hidden costs is a significant advantage of the dynamic monitoring approach.

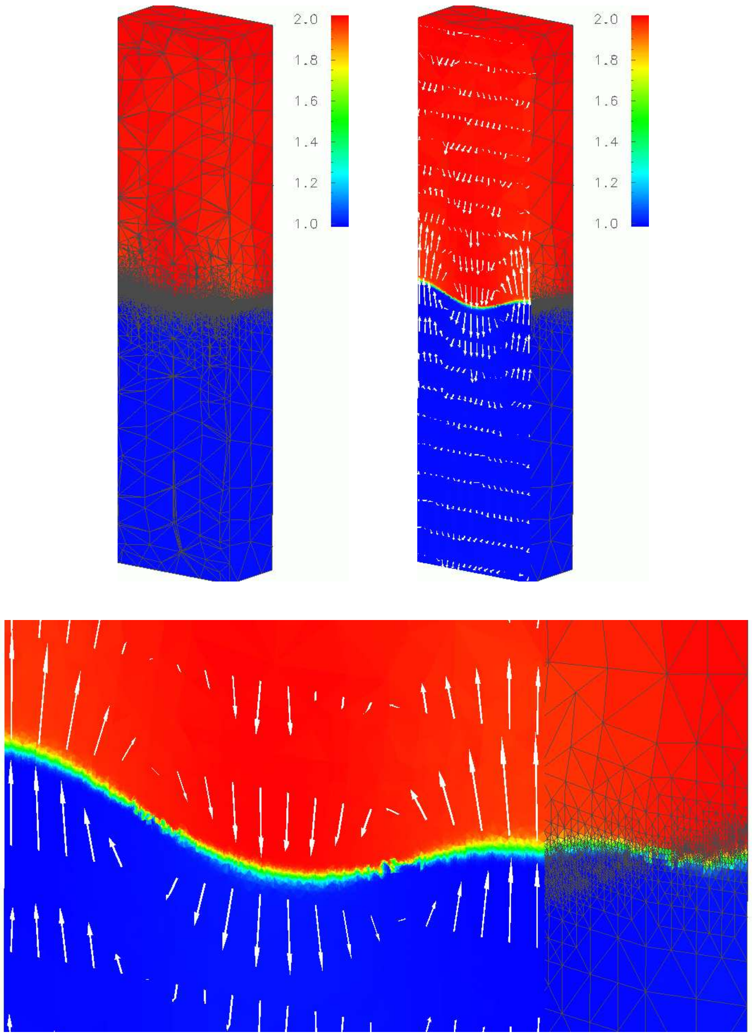


Figure 3.9: **Densities for a Rayleigh-Taylor flow projected on a plane through the center of the domain. Densities range from 1 (blue) to 2 (red). The projection on the upper left includes the mesh. Arrows shown on the cut plane at the upper right indicate velocity. The projection at the bottom is a closer view of the interface zone.** ***Courtesy of Ray Loy, Rensselaer Polytechnic Institute.***

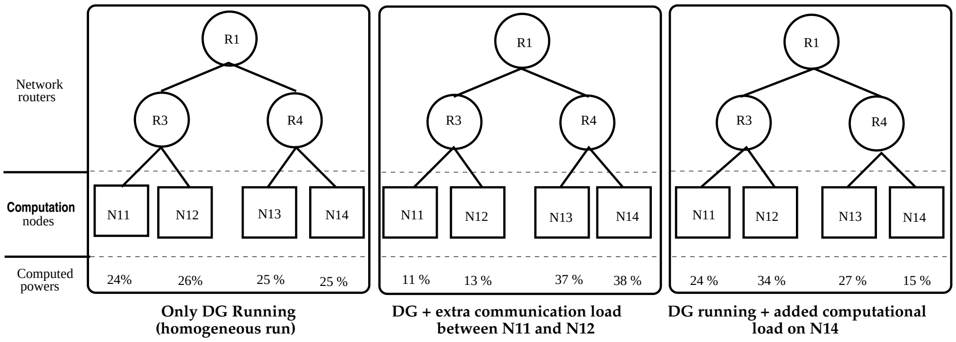

Network routers
Computation nodes
Computed powers
R1
R3
R4
N11
N12
N13
N14
24%
26%
25 %
25 %
11 %
13 %
37 %
38 %
24 %
34 %
27 %
15 %
Only DG Running (homogeneous run)
DG + extra communication load between N11 and N12
DG running + added computational load on N14

Figure 3.10: Power computation after four minutes of monitoring on a small cluster. The circles represent network routers, the squares represent identical computation nodes. The percentages below the squares are the powers computed at each node. (a) No extra load is added to the system. Given that the system is homogeneous, similar powers are computed on all nodes, (b) $N11$ and $N12$ communicate heavily, so those nodes are given less work to compensate for that communication, and (c) $N14$ is heavily loaded, so it gets a smaller percentage of the work.

CHAPTER 4

Interactions with load balancing procedures

We describe how DRUM can be used by existing load balancing algorithms. In designing DRUM, we were concerned in presenting the model as an abstract object. As we described earlier, the operation of DRUM is performed with the following main functions:

- **DRUM_createModel:** This function reads the XML file generated by the graphic configuration tool and creates a tree structure populated with the benchmark results. If the application opts not to use dynamic monitoring, then only this function needs to be called.

- **DRUM_startMonitoring:** This function starts the dynamic monitoring. Monitoring agents are started on each computing node. Some computing nodes, called representatives, take charge of monitoring one or more network nodes in addition to themselves. Starting the monitoring on each node consists of starting $commInterface$ monitor and a $cpuMemory$ monitor on that computing node. If the computing node is a "representative," $commInterface$ monitors to corresponding network node are also started. If the compute node is an SMP, only one $commInterface$ monitor is started for the whole node as processors in the SMP are assumed to share the same communication interface(s). The process elected to do the communication monitoring on an SMP is labeled *elected process*.

- **DRUM_stopMonitoring:** This function stops the dynamic monitoring and then computes the power of each node in the machine model. This operation involves a local data exchange on each SMP followed by global inter-node exchange.

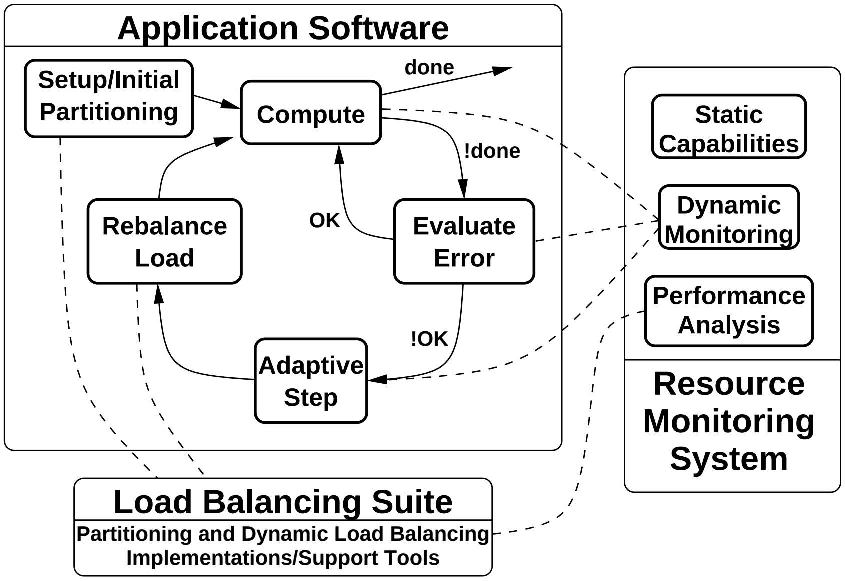


Figure 4.1: A typical interaction between an adaptive application code and a dynamic load balancing suite, when using a resource monitoring system (e.g., DRUM).

Figure 4.1 shows a typical interaction between an adaptive application code and a dynamic load balancing suite augmented with a resource monitoring system like DRUM.

DRUM is a stand-alone library, capable of interacting with any LB routine capable of producing partitions of different sizes. Yet, we have spent a special effort in integrating DRUM with the Sandia National Laboratories Zoltan data management toolkit. In this context, we give details of the overall design capturing the interactions between applications, DRUM and load balancing algorithms in Zoltan. However, we begin with a short overview of Zoltan.

"Sandia National Laboratories' *Zoltan* toolkit [20] provides a common interface to several state-of-the-art partitioners and dynamic load balancers [5, 9, 16, 33, 55] to distribute computations across cooperating processors and redistribute following adaptivity. Zoltan's design is data-structure neutral [18], relying on information gathered from application-specified callback functions and migration arrays to interact with application software. Most of these procedures seek to achieve an even distribution of computational work, minimize interprocess boundary interfaces that correspond to interprocess communication during the solution phase, and minimize data movement necessary to achieve the new decomposition." *courtesy of karen Devine, Sandia National labs.*

We borrow the following description of Zoltan from [19]. "Zoltan is unique in providing dynamic load balancing and related capabilities to a wide range of dynamic, unstructured and/or adaptive applications. Zoltan delivers this support in several ways. First, by including a suite of partitioning algorithms, Zoltan addresses the load-balancing needs of many different types of applications. Geometric algorithms like recursive bisection [5, 45, 48] and space-filling curve partitioning [60, 42] provide high-speed, medium-quality decompositions that depend only on geometric information and are implicitly incremental. These algorithms are highly effective in crash simulations, particle methods, and adaptive finite element methods. Graph-based algorithms, provided through interfaces to the graph partitioning libraries ParMETIS [34] and PJostle [54], provide higher quality decompositions based on connectivity between application data, but at a higher computational price. Graph algorithms can also be effective in adaptive finite element methods, as well as multiphase simulations and linear algebra solvers. Using Zoltan, application developers can switch partitioners simply by changing a Zoltan run-time parameter, allowing comparisons of the partitioner effect on the applications.

Second, Zoltan supports many applications through its data-structure neutral design. While similar toolkits focus on specific applications (e.g., the DRAMA toolkit [37] supports only mesh-based applications), Zoltan does not require applications to have specific data structures. Instead, data to be partitioned are considered to be generic objects with weights representing their computational cost. Zoltan also does not require applications to build specific data structures (e.g., graphs) for Zoltan. Instead, applications provide only simple functions to answer queries from Zoltan. These functions return the object weights, object coordinates, and relationships between objects. Zoltan then calls these functions to build data structures needed for partitioning. While some overhead is incurred through these callbacks, the cost is small compared to the actual partitioning time. More importantly, development time is saved as application developers write only simple functions instead of building (and debugging) complicated distributed data structures for partitioning. Third, Zoltan provides additional functionality commonly used by applications using dynamic load balancing. For example, Zoltan's data migration tools perform all communication to move data from old decompositions to new ones; application developers provide only callback functions to pack data into and unpack data from communication buffers. (In this respect, application-specific toolkits like DRAMA [37] can provide greater migration capabilities, as they have knowledge of application data structures.) While these tools operate well together, separation between tools allows application developers to use only the tools they want; for example, they can use Zoltan to compute decompositions but perform data migration themselves, or they can build Zoltan distributed data directories that are completely independent of load balancing. Zoltan's design is effective for both applications and research. It allows both existing and new applications to easily use Zoltan. New algorithms can be added to the toolkit easily and compared to existing algorithms in real applications using Zoltan." However, the procedures in Zoltan do not account for heterogeneity in the execution environment.

Our work provides support for heterogeneity in Zoltan. In this new version, Zoltan interacts with DRUM through DRUM's public interface. A boolean configuration parameter **USE_DRUM** is set to 1 to indicate to Zoltan that DRUM services will be used; otherwise, the parameter is set to 0. Figure 4.2 illustrates the use of DRUM within Zoltan. At the initiation of the Zoltan library, a call to **DRUM_createModel** followed by a call to **DRUM_startMonitoring** is performed. **DRUM_stopMonitoring** is invoked when Zoltan's main load balancing function is called. This call stops the monitors and computes the relative powers of each node in the model. These powers are used by the load balancer to produce partitions of appropriate sizes. Monitoring resumes when load balancing finishes. We would like to emphasize that it is by no means necessary to use Zoltan in order to be able to benefit from DRUM services. DRUM is a effectively a stand-alone library. The integration with Zoltan is more an addition to Zoltan than a requirement for DRUM use.

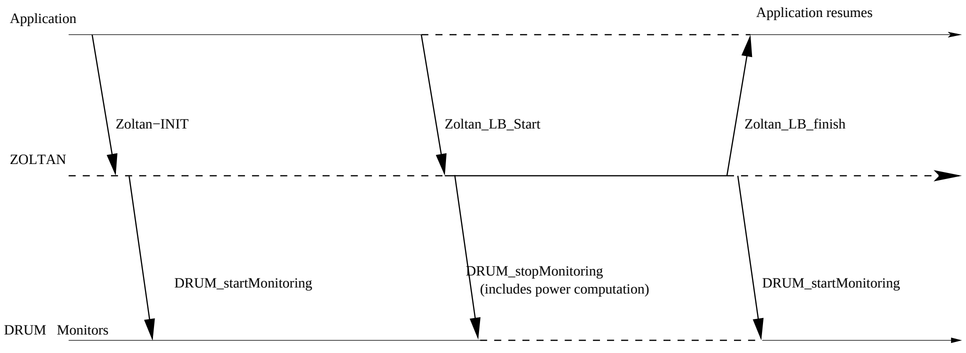


Figure 4.2: Interaction between the user application, Zoltan and the DRUM monitors. DRUM monitors collect data while the user application is running. They are stopped during the load balancing phase and restarted when the user application resumes.

CHAPTER 5

Experimental study

The potential improvement from resource-aware load balancing depends to a large extent on the degree of heterogeneity in the system. If the execution environment is nearly homogeneous, very little can be gained by accounting for heterogeneity. In such a situation, the overhead introduced by the dynamic monitoring may even slow the computation slightly. Hence, any measure of speedup should be tied to the degree of heterogeneity of the system.

Xiao *et al.* [65] propose metrics for CPU and memory heterogeneity defined as the variance of computing powers and memory capacities among the computation nodes. In particular, they define system CPU heterogeneity for a system with P processors as

$$H_{cpu} = \sqrt{\frac{\sum_{j=1}^{P}(\overline{W}_{cpu} - W_{cpu}(j))^2}{P}}$$

where $W_{cpu}(j)$ is a measure of the CPU speed of processor j relative to the fastest CPU in the system, computed as

$$W_{cpu}(j) = \frac{V_{cpu}(j)}{max_{i=1}^{p} V_{cpu}(i)}$$

and $\overline{W}_{cpu}$ is the average of relative CPU speeds

$$\overline{W}_{cpu} = \frac{\sum_{j=1}^{P} W_{cpu}(j)}{P}.$$

$V_{cpu}(i)$ is the MIPS (millions of instructions per second) rating for CPU i. We use the same formula to measure CPU heterogeneity, substituting the MFLOPS obtained from the benchmark for the MIPS values. We trust that MFLOPS provides a more reliable measure of CPU performance than raw MIPS. While MIPS provide a good indication of peak performance of a given CPU, it is often not a convenient performance metric for actual CPU performance. For example, consider the case where a floating point (FP) operation takes 20 single-cycle machine instructions on hardware without a FP co-processor. Now, let's assume that with the presence of a FP co-processor, the same FP operation takes only a single two-cycle instruction. Now the MIPS number is worse than before the FP co-processor addition, as it went down from 20 to 2. The actual performance, as could be measured by the execution time of the FP operation, has on the contrary dramatically increased. The execution time has been in fact reduced from to 20 cycles to 2 cycles.

In Table 5.1, we show an example of computed degrees of heterogeneity (DH) for various combination of fast 450Mhz and slow 333/330Mhz nodes of the Bullpen cluster. The fast nodes are indicated as **fast(number of CPUs)** and the slow nodes as **slow** as these latter have only 1 cpu per node.

Nodes	**DH**	Average Relative MFLOPS ($\overline{W}_{cpu}$)
8 (2fast(4))	0.00	0.9988
10 (2fast(4) + 2slow)	0.15	0.84
12 (2fast(4) + 4slow)	0.19	0.77
16 (2fast(4) + 4fast(2))	0.00	0.99
18 (2fast(4) + 4fast(2) + 2slow)	0.09	0.90
20 (2fast(4) + 4fast(2) + 4slow)	0.12	0.86

Table 5.1: Degree of heterogeneity for various combination of slow and fast nodes of the Bullpen cluster.

5.1 Raleigh-Taylor instability problem on the Bullpen cluster

We solved a two-dimensional Rayleigh-Taylor instability problem [43] on a rectangular domain on the Bullpen cluster at Williams. A description of the cluster is given in Chapter 3. The experiments involved "fast" 450MHz SPARC UltraII nodes and "slow" 300/333MHz SPARC UltraII processors, connected by fast (100 Mbs) Ethernet. We ran the experiments on four, six and eight nodes while keeping an even number of "fast" and "slow" nodes. The CPU degree of heterogeneity for this experiment was 0.18. The first formulations of the processing and communication powers were used in these experiements. Benchmark runs indicated that the fast nodes have a computation rate of approximately 1.5 times faster than the slow nodes. Given an equal distribution of work, the fast nodes would be idle one third of the time. By giving 50% more work to each of the fast nodes, a theoretical maximum speedup of 20% is would be possible. This maximum would be reached if we assume a perfect match between partition sizes and processor capabilities and

Processors	Octree	Octree + DRUM	Improvement	Max
4	16440s	13434s	18%	20%
6	12045s	10195s	15%	20%
8	9722s	7987s	18%	20%

Table 5.2: Execution times for running a Rayleigh-Taylor instability problem with standard Zoltan (Octree) and then with Zoltan augmented with DRUM.

if we ignore communication overhead and the serial sections of the code. For our computations, we used a partition-weighted version of the Octree/SFC procedure [9]. Table 5.1 summarizes the results obtained in terms of application execution time.

5.2 PHAML runs on the Bullpen cluster

We present experimental results using DRUM to guide resource-aware load balancing in the adaptive solution of a Laplace equation on the unit square, using Mitchell's Parallel Hierarchical Adaptive MultiLevel software (PHAML) [40]. After 17 adaptive refinement steps, the mesh has 524,500 nodes. The Bullpen cluster at Williams College is the execution environment used for these experiments. These experiments use the most recent power formulations.

We use Zoltan's Hilbert Space Filling Curve (HSFC) procedure for partitioning, though we re-emphasize that the powers computed by DRUM may be used by any Zoltan procedure and results are similar for other methods. A space-filling curve (SFC) maps n-dimensional space to one dimension [44]. In SFC partitioning [42, 60], an object's coordinates are converted to a SFC key representing the object's position

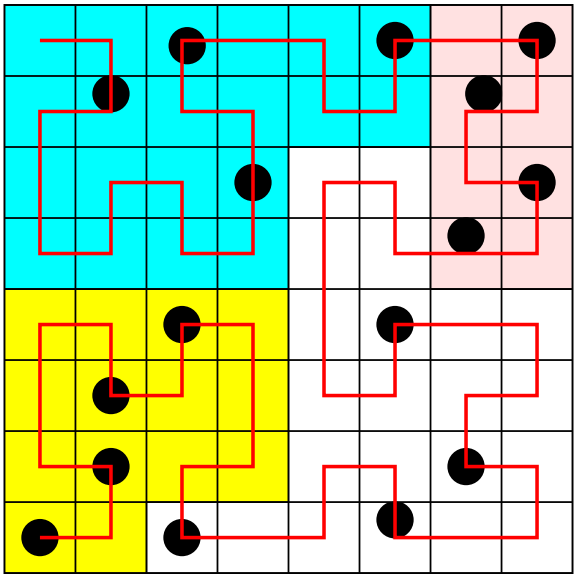


Figure 5.1: SFC partitioning example. Objects (dots) are ordered along the SFC. Partitions are indicated by shading. ***Courtesy of Karen Devine, Sandia National Labs.***

along a SFC through the physical domain. Sorting the keys gives a linear ordering of the objects (Figure 5.1). This ordering is cut into appropriately weighted pieces that are assigned to processors. Zoltan's HSFC [20] replaces the sort with adaptive binning. Based upon their Hilbert SFC keys, objects are assigned to bins associated with partitions. Bin sizes are adjusted adaptively to obtain sufficient granularity for balancing.

5.2.1 Experiment 1: Heterogeneous cluster with one process per node

The first set of experiments, which appear in [24], use combinations of fast and slow processors with one application process being run on each node. Figure 5.2 shows the run time of the PHAML application on various combinations of fast and slow processors and for communication weights (w^{comm} values) of 0, 0.1 and

0.25. Runs on only the homogeneous (fast) nodes show very low overhead incurred by the use of DRUM. On heterogeneous configurations, experiments using DRUM's resource-aware partitions show a clear improvement in execution time compared to those with uniformly sized partitions. In Figure 5.3, we compare the execution time

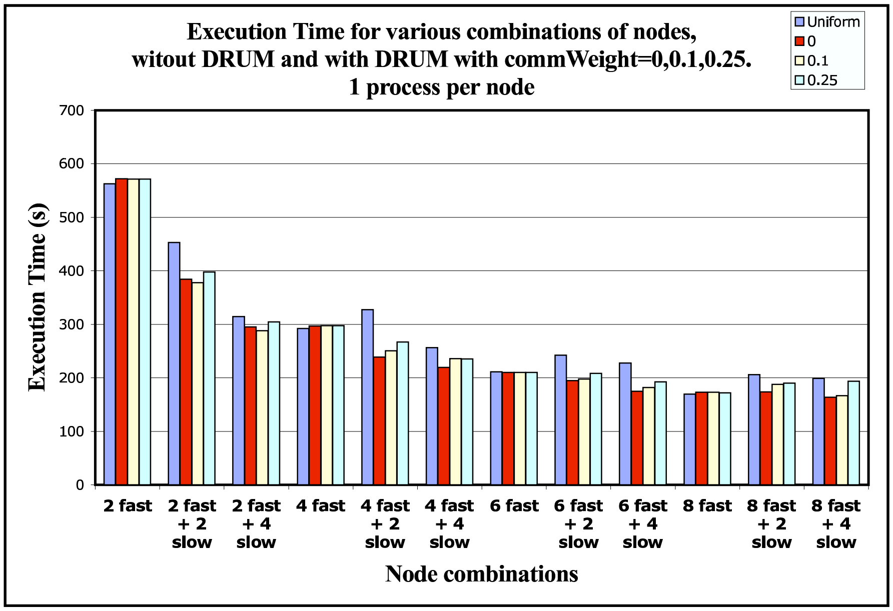


Figure 5.2: Execution times for PHAML runs when DRUM is used on different combinations of fast and slow processors, with uniform partition sizes and resource-aware partition sizes (with various values for w^{comm}). Here, only one application process is run on each node.

Relative Change (RC) achieved to a "Ideal" Relative Change (RC_{ideal}) for the same experimental data. RC is the variation in execution time relative to the original execution time:

$$RC = \frac{t_{uniform} - t_{DRUM}}{t_{uniform}}$$

where $t_{uniform}$ is the execution time of the application without using DRUM and t_{DRUM} is the execution time when DRUM is used. For this example, RC_{ideal} is the relative change that would be achieved if the fast processors were assigned exactly 50% more load than the slow ones and if communication overhead were ignored. In general,

$$RC_{ideal} = 1 - \frac{n}{\sum_{i=1}^{n} h_i}$$

where n is the total number of nodes running the application processes, and h_i is the ratio of ith processor's speed to that of the slowest processor. In our case, since the fast nodes are assumed to be 1.5 times faster than the slow ones, the h_i for each of the fast nodes is equal to 1.5. The assumption of no communication overhead is not realistic in most adaptive applications and, therefore, RC_{ideal} cannot practically be reached in most cases.

Results in Figure 5.3 show significant performance gains when DRUM is used in heterogeneous configurations. The results also show a small slowdown in some cases where only homogeneous configurations of nodes are used. In rare cases, DRUM shows a slight improvement in homogeneous configurations. We attribute such gains to the fact that DRUM compensate for the extra work the master process (process rank 0) has to do to manage standard output and coordinate communication between the other processes.

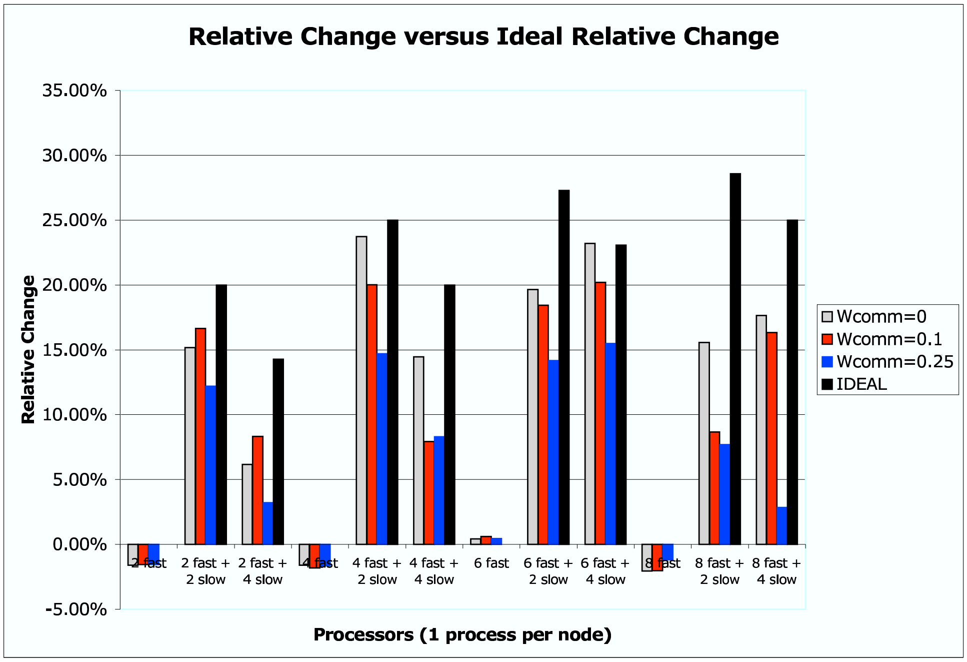


Figure 5.3: Relative change in CPU times for PHAML runs when DRUM is used on different combinations of fast and slow processors, contrasted with the ideal relative change. Only one application process is running on each node.

5.2.2 Experiment 2: Communication weight study

In order to study the effect of the communication weight w^{comm} in the overall execution time, we repeated the experiment for a wider range of communication weights and processor/process combinations. Here, multiple application processes are running on the SMP nodes. On both the slow and fast nodes, only one (application) process is running per processor. The results are reported in Figure 5.4. The combination of processes, processors and nodes are indicated as:

```
#total processes [#fast nodes(#processes per node) + #slow nodes(1)]
```

Again, these results indicate a low overhead of DRUM monitors in the cases of homogeneous node configurations, where only fast nodes are used. They also show significant benefits in the case of heterogeneous processor/process combinations. These computations also suggest that a communication weight of more than 0.5 is not appropriate for our test application. This is expected, since this application overlaps computation and communication. Figure 5.5 shows the best relative change values for each combination of processors and contrasts them with the ideal relative change.

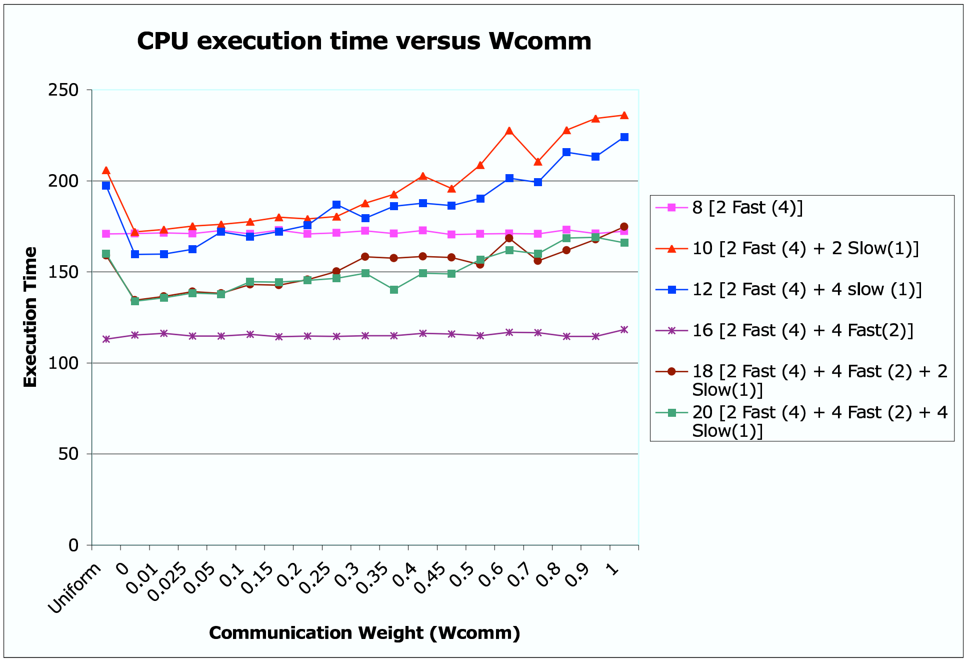


Figure 5.4: Execution times for PHAML runs when DRUM is used on different combinations of fast and slow processors and with different values for w^{comm}.

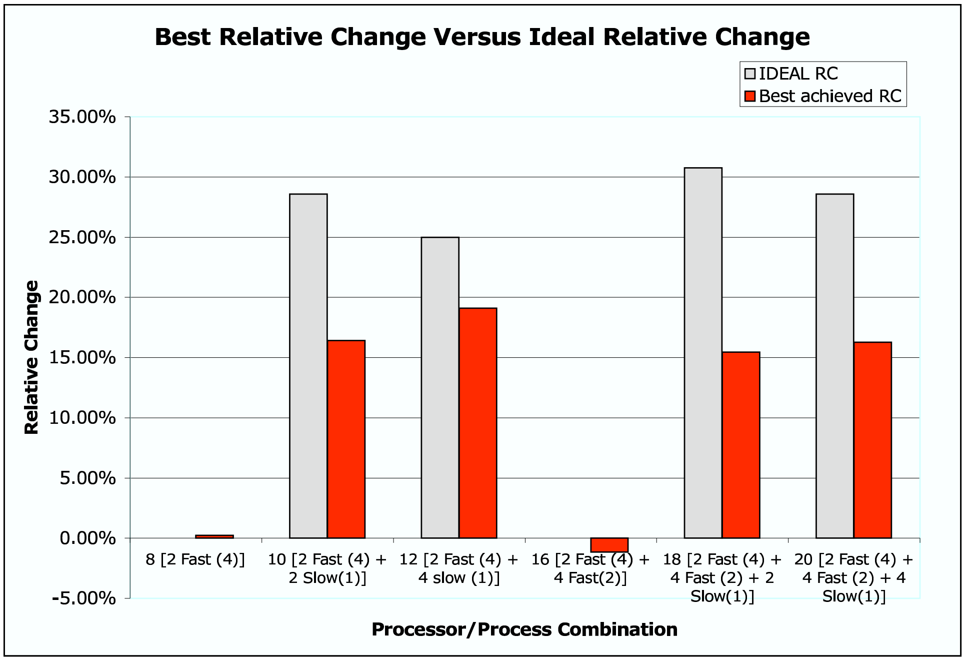


Figure 5.5: ideal and best observed relative changes across all values of w^{comm} for the timings shown in Figure 5.4.

5.2.3 Experiment 3: Correlation with Degree of Heterogeneity

In this experiment, we study the correlation of the speedup to be gained from the use of DRUM and the degree of heterogeneity of the execution environment. We use the formulas described in the beginning of this chapter to quantify the degree of heterogeneity of the cluster execution testbed.

Figure 5.6 shows the evolution of RC as a function of the degree of heterogeneity. As expected, DRUM has a greater impact on the execution time when the heterogeneity of the execution environment is greater.

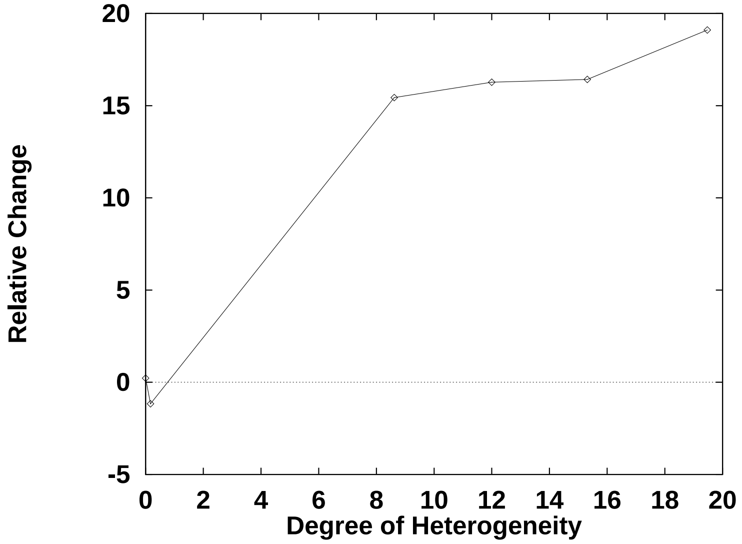


Figure 5.6: Relative Change in execution time as a function of the degree of heterogeneity.

5.2.4 Experiment 4: Non-Dedicated usage of the cluster

The most significant benefits of DRUM come from the fact that it accounts for both static information through the benchmark data and dynamic performance using monitoring agents. This provides a benefit both on dedicated systems with some heterogeneity that can be captured by the benchmarks, and highly dynamic systems where the execution environment may be shared with other processes. We have tested our procedures in these environments; preliminary results first appeared in [53] and are reproduced in Figure 5.7. In that case, we ran the same PHAML example but with two additional compute-bound processes (which are not part of the PHAML computation and are not explicitly being monitored by DRUM) running on the nodes on which the highest and the second highest rank PHAML processes are running. When using uniform partitions (with DRUM disabled) the computations are slowed significantly by the fact that some processors are overloaded. In particular in cases where the “slow nodes” are used, this results in a significant imbalance and most processors spend time waiting for the overloaded node to complete its part of the computation. When DRUM is used, we see significant improvement in running times, even in cases where the processors are all the same and the only source of heterogeneity is the external load that can be detected only at run time.

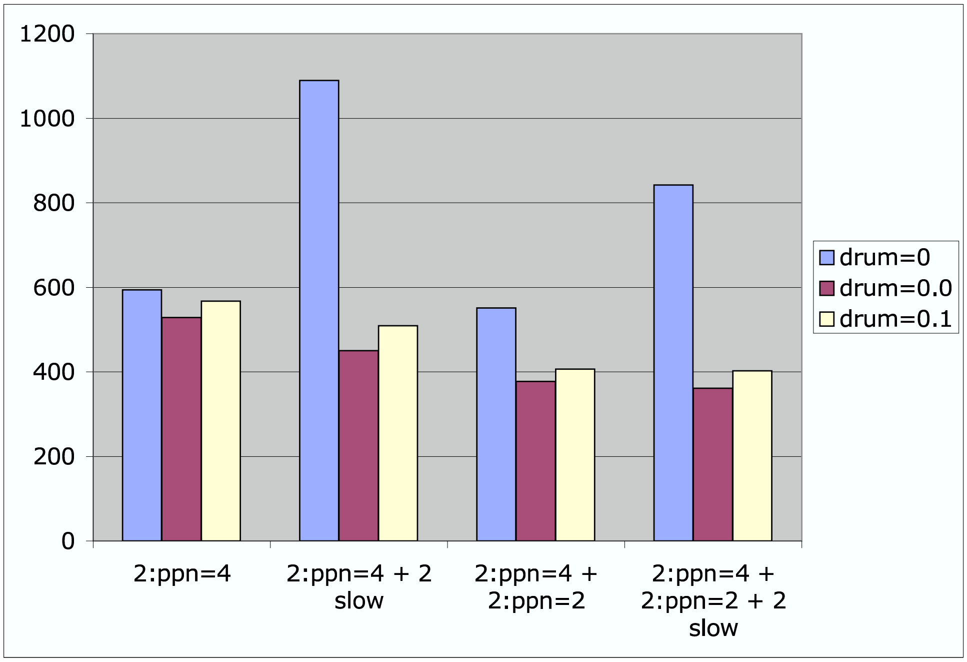


Figure 5.7: Execution times in seconds for PHAML runs when DRUM is used on different combinations of processors, but with two compute-bound external processes also running on the on the nodes on which the highest and the second highest rank PHAML processes are running. Times are shown for uniform partitions (drum=0) and DRUM-guided resource-aware partitions (drum=0.0 for processing power only, drum=0.1 with a communication weight of 0.1 applied). In this example, we run one application process for each CPU in each node.

Nodes	HSFC (NO DRUM)	HSFC + DRUM	Overhead
2 nodes	637.99	638.82	0.13 %
4 nodes	363.66	364.47	0.22 %
6 nodes	260.05	260.46	0.15 %

Table 5.3: PHAML on homogeneous nodes: evaluation of DRUM overhead.

5.2.5 Experiment 5: Evaluation of DRUM overhead for PHAML on Bullpen

In this experiment, we have run PHAML on a collection of 2, 4 and 6 homogeneous nodes of the Bullpen cluster. We launched two application processes per node. The objective of these homogeneous runs was to evaluate the overhead generated by DRUM. While DRUM monitoring was used, we did not use the computed powers to guide the load balancing. We report the results of these experiments in table 5.2.5.

This experiment shows that there is a negligible overhead incurred from the use of DRUM.

5.3 Network heterogeneity experiment: Running a perforated shock tube problem on the reconfigured Bullpen cluster

The objective of this experiment is to show the effect of DRUM when used in an environment with network heterogeneity. We modified the Bullpen cluster by introducing a slow 10 Mbs Ethernet hub and attaching some of the compute nodes to it. The resulting sub-cluster is then attached to the rest of the cluster via the 100 Mbs switch. Figure 5.8 first shown in Chapter 3 and reproduced here for convenience gives a description of the reconfigured cluster.

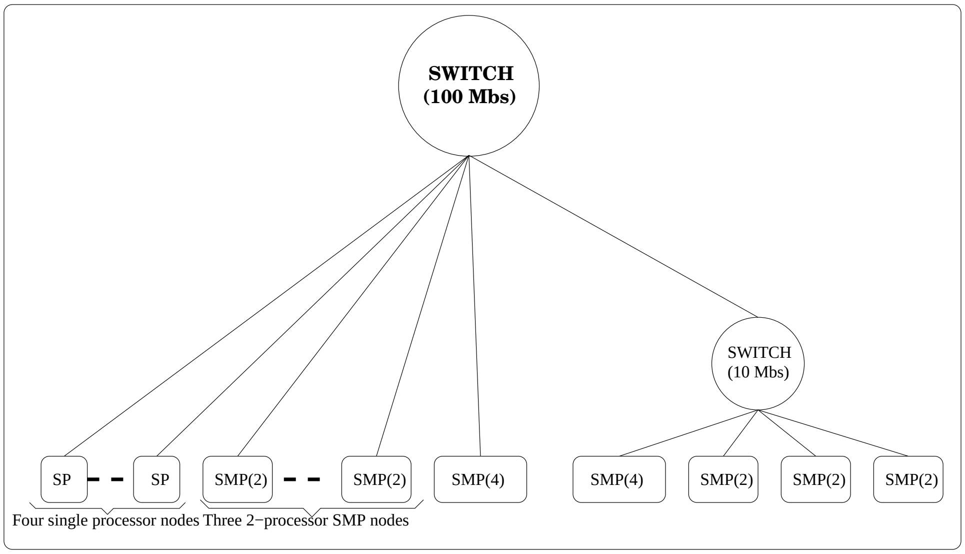


Figure 5.8: Reconfigured “Bullpen cluster” model built by DRUM

The perforated shock tube problem is described in [25] as follows. “Consider the three-dimensional unsteady compressible flow in a cylinder containing a cylindrical vent. This problem was motivated by flow studies in perforated muzzle brakes for large caliber guns [21]. We match flow conditions to those of shock tube studies of Dillon [21] and Nagamatsu *et al.* [41]. Our focus is on the quasi-steady flow that exists behind the contact surface for a short time. Using symmetry, the flow may be solved in one half of the domain bounded by a plane through the vent. The initial mesh (Figure 5.9) contains 45,093 tetrahedral elements. The mesh contains 80,659 elements after a prerefinement stage which forces refinement near the interface between the shock tube and vent. The larger cylinder (the shock tube) initially contains helium gas moving at Mach 1.23 while the smaller cylinder (the vent) is quiet. A Mach 1.23 flow is prescribed at the tube's inlet and outlet. The walls of the cylinders are given reflected boundary conditions, and a far field condition is applied at the vent exit. Flow begins as if a diaphragm between the two cylinders were ruptured."

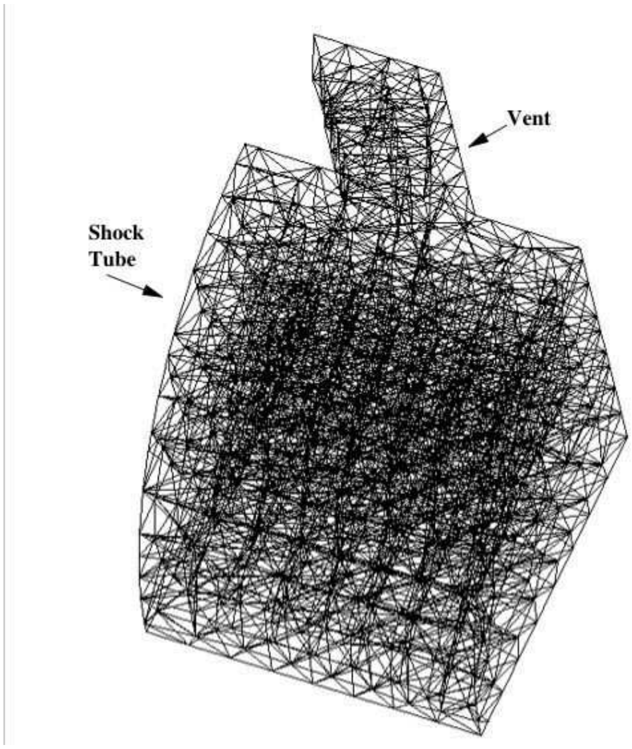


Figure 5.9: The initial "muzzle brake" mesh used for the perforated shock tube example. ***Courtesy of Jim Teresco, Williams College***
.

Figure 5.10 shows the execution times of running the perforated shock tube problem on eight homogeneous nodes of the modified cluster. The only source of heterogeneity is the slow switch connecting four of these 8 nodes. We varied the value of the communication weight w_{comm} from 0 to 1. The lowest execution time was obtained for $w^{comm} = 0.3$. Figure 5.11 shows with a high resolution what happens around the point $w^{comm} = 0.3$.

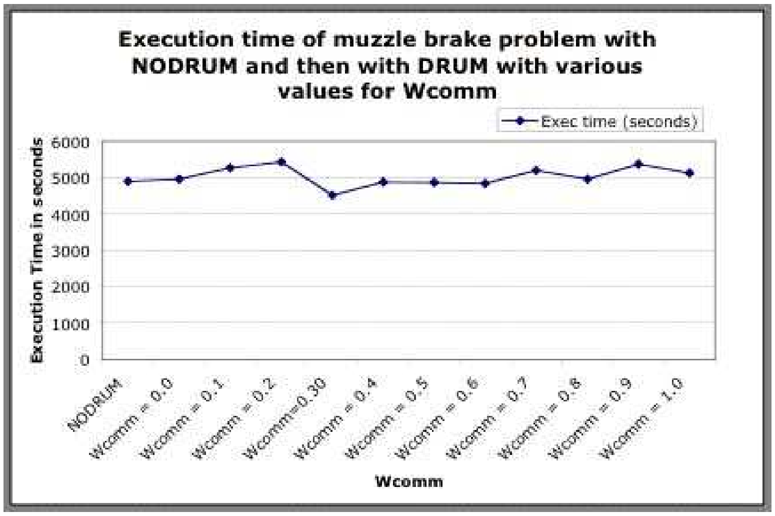


Figure 5.10: Perforated shock tube runs without DRUM and with DRUM using different values for w^{comm}.

In Table 5.4, we present the values of node powers computed by DRUM for different values for w^{comm} for the perforated shock tube experiments described earlier in this section. Nodes *rivera, righetti, arroyo* and *gossage* are connected to the slow 10Mbs switch as explained earlier.

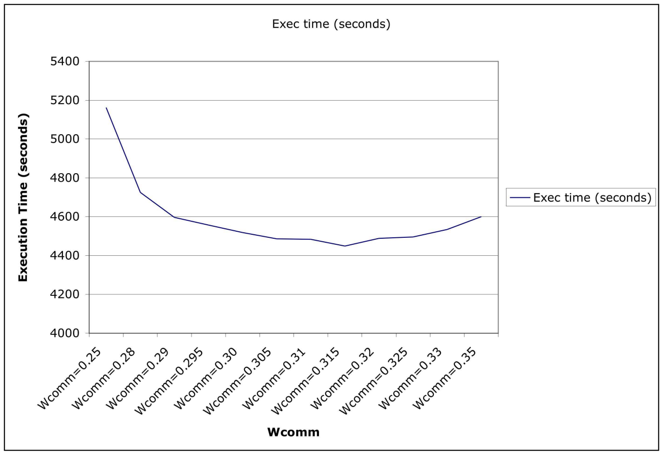


Figure 5.11: Perforated shock tube runs with DRUM using values for W^{comm} around 0.3.

w^{comm}	rivera	righetti	arroyo	gossage	wetteland	lyle	farr	mcdaniel
0.0	0.125	0.124	0.124	0.125	0.125	0.124	0.124	0.124
0.1	0.115	0.114	0.114	0.115	0.135	0.135	0.135	0.135
0.2	0.104	0.104	0.104	0.145	0.145	0.145	0.145	0.145
0.3	0.094	0.094	0.094	0.094	0.155	0.156	0.155	0.155
0.4	0.083	0.084	0.084	0.084	0.166	0.165	0.165	0.165
0.5	0.074	0.073	0.073	0.074	0.176	0.176	0.176	0.176
0.6	0.063	0.063	0.063	0.063	0.186	0.186	0.186	0.186
0.7	0.053	0.053	0.053	0.053	0.196	0.196	0.196	0.196
0.8	0.043	0.043	0.043	0.043	0.206	0.206	0.206	0.206
0.9	0.033	0.032	0.032	0.033	0.217	0.217	0.217	0.217
1.0	0.022	0.022	0.022	0.022	0.227	0.227	0.227	0.227

Table 5.4: Node powers computed by DRUM during the perforated shock tube experiments, for values of w^{comm} ranging from 0.0 to 1.0.

CHAPTER 6

Conclusion and future work

6.1 Contribution of this research

Our results show a clear benefit to resource-aware load balancing. DRUM accounts for both static information by using benchmark data, and dynamic performance data using monitoring agents. This provides a benefit both on dedicated systems with some heterogeneity captured by the benchmarks, and highly dynamic systems where the execution environment may be shared with other processes. We also have shown, through experimentation, that the more heterogeneity in the execution environment, the greater the performance improvement gained from using DRUM. In case of non-dedicated usage where some nodes might be busy running other application concurrently, DRUM has been shown to yield very significant performance gains exceeding 50% in execution time reductions. A study of DRUM's overhead has shown that DRUM has a negligible overhead than doesn't exceed 0.2% when the nodes are probed with a frequency of 1 second.

One of the good aspects of the software resulting from this research is that it is a self-sufficient tool that can be used by existing load balancing algorithms with almost no changes. While an effort has been successfully spent to integrate DRUM with Zoltan, it is by no means necessary to use Zoltan in order to benefit from DRUM monitoring and evaluation tools. DRUM has a simple interface that any load balancer capable of producing partitions of different sizes can directly call. In this context, it is clear that our model has some advantage when compared to similar same-goal efforts that generally propose tools that are tightly-coupled to specific load balancers (e.g. [46]).

One of the strengths of DRUM lies in its ease of use. In particular, it gives the user the option to use a graphical configuration tool to describe the execution environment. The tool has the ability to automatically generate, in an XML format file, a topology description of the execution environment augmented by initial statistics about the capabilities of its composing resources.

DRUM includes a power expression that uses heuristics to combine the collected information about resource utilization and capabilities into a scalar form, easily interpretable by load balancers. Just like [46], our expression involves a weighted sum. However, our power computation handles more general cases that involve symmetric multiprocessors, hierarchical clusters and also dynamic execution environments. DRUM has its own integrated monitors for performance data collection. With very little modification to the software, users could implement new power computation expressions and still rely on DRUM to supply the raw performance information collected by the monitors.

6.2 Latest DRUM development

Our research groups at RPI and Williams have been involved in many efforts aiming at providing more functionality for DRUM, extending its use to other platforms, and integrating it with other complimentary software.

We are currently perfecting a recent porting of DRUM to Linux. This would soon permit us to test DRUM on bigger clusters and with larger, three-dimensional adaptive applications written for the Linux platform.

An effort was spent on integrating DRUM with the Network Weather Service (NWS) [62] monitoring library [51]. The use of NWS is now a configuration option in DRUM. Our first experiments with DRUM/NWS did not show any added benefit when compared with using the internal monitors of DRUM. This results from the combination of these three facts: (*i*) the execution environment is characterized by a high latency network, (*ii*) the communication cost in PHAML is bound by latency as the communication pattern is characterized by short messages and (*iii*) the communication power expression does not include the latency factor.

We are currently working on integrating DRUM and the HIER software [52] which consists in implementing hierarchical techniques for load balancing, with the possibility of running both in the framework of Zoltan. In this research, we implemented hierarchical balancing procedures that interact with DRUM to tailor partitions to a given network topology [52]. In addition to the ability to produce weighted partitions, this allows different load balancing algorithms to be used, as appropriate, in different parts of the network hierarchy [52]. The DRUM configuration tool, DRUMHead, permits the user to specify the LB procedures to be executed at each level of the hierarchy [51].

6.3 Future work

A certain improvement in performance would be achieved if a more dynamic weighting scheme is used in DRUM power computation. These weights are application dependent and should reflect a ratio of computation to communication work in the application. We think it might be possible to estimate such weights without inserting probes into the user application. This could be achieved by exploiting data such as the number of incoming and the outgoing packets, the link bandwidth(s), the average CPU usage by the local process and the inter-probe interval. Estimation of this ratio is more complex in some cases such as when computation and communication are overlapped or when the environment is non-dedicated. In those situations, more intrusive and possibly less efficient techniques might be required. One idea that seems to combine the advantages of dynamicity without the drawbacks of significant intrusiveness consists in using tunable fixed weights. While, for example, the communication weight will be set globally in the application, it will be tuned at each node as a function of the amount of communication traffic the node is experiencing on its communication interfaces. The sensitivity of tuning as well as the correlation between amount of traffic and the extent of tuning is yet to be investigated.

It is necessary and we intend to include the memory factor into the power expression. Currently, DRUM agents monitor the available memory and the total memory on each computation node. Given this limited information, memory utilization should be a factor in the computation of a node's power only when the ratio of available memory to total memory becomes smaller than a specified threshold. More refined memory statistics (*e.g.*, number of cache levels, cache hit ratio, cache and main memory access times) are needed to capture memory effects more accurately in our model.

We also intend to port the DRUM library to other systems such as FreeBSD and MacOS X/Darwin. This would allow more interoperability in DRUM and, in conjunction with appropriate versions of MPI, it will allow an execution of DRUM across multiple operating systems.

Another effort we are currently investigating aims to extend the functionality of DRUM to widely distributed environment such as those covered by the grid model. We plan to build on the middleware-level load balancing ideas presented in [17]

Results obtained from the use of DRUM are very promising. As such, the software is currently undergoing a code review as a preparation for public release both as a stand-alone library and, possibly, in conjunction with the Zoltan toolkit. More details about the latest efforts in DRUM and related publications can be found at: http://www.cs.williams.edu/drum.

LITERATURE CITED

[1] The finite element method in biomedical engineering, biomechanics and related fields. In *Proceedings of FEM workshop 2003*, University of Ulm, Germany, 2002.

[2] M. Banikazemi, S. Prabhu, J. Sampathkumar, K. Panda, T. Page, and P. Sadayapan. Profile-based load balancing for heterogeneous clusters. Technical Report OSU-CISRC-03/98-TR16, Ohio State University, Computer and Information Science Department, 1998.

[3] S. T. Barnard. PMRSB: parallel multilevel recursive spectral bisection. In F. Baker and J. Wehmer, editors, *Proc. Supercomputing '95*, San Diego, December 1995.

[4] S. T. Barnard and H. D. Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):101–117, 1994.

[5] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Computers*, 36:570–580, 1987.

[6] R. Biswas, K. D. Devine, and J. E. Flaherty. Parallel, adaptive finite element methods for conservation laws. *Appl. Numer. Math.*, 14:255–283, 1994.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46:720–748, 1999.

[8] T. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorization". In *Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 445–452. SIAM, 1993.

[9] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco. Dynamic octree load balancing using space-filling curves. Technical Report CS-03-01, Williams College Department of Computer Science, 2003.

[10] R. Chandra, R. Menon, L. Dagum, D. Koh, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP.* Morgan Kaufmann, 2000.

[11] T. Chandrupatla and A. Belegundu. *Introduction to Finite Element in Engineering.* Prentice Hall Professional Technical Reference, August 1996.

[12] J. Chandy, S. Kim, B. Ramkumar, S. Pakes, and P. Benerjee. An evaluation of parallel simulated annealing strategies with application to standard cell placement. *IEEE Trans. Computers*, 16(7):398–410, April 1997.

[13] H. Chen, H. Gao, and S. Sarma. Whams3d project progress report (pr-2). Tech. Report 1112, CSRD, University of Illinois, 1991.

[14] J. Chen and V. E. Taylor. Mesh partitioning for efficient use of distributed systems. *IEEE Trans. Parallel and Distrib. Syst.*, 13(1):67–79, 2002.

[15] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7:279–301, 1989.

[16] H. L. de Cougny, K. D. Devine, J. E. Flaherty, R. M. Loy, C. Özturan, and M. S. Shephard. Load balancing for the parallel adaptive solution of partial differential equations. *Appl. Numer. Math.*, 16:157–182, 1994.

[17] T. Desell, K. E. Maghraoui, and C. Varela. Load balancing of autonomous actors over dynamic networks. In *Proc. Hawaii International Conference on System Sciences*, volume 9, page 90268a, 2004.

[18] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[19] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. New challenges in dynamic load balancing. *Appl. Numer. Math.*, 52(2–3):133–152, 2005.

[20] K. D. Devine, B. A. Hendrickson, E. Boman, M. St. John, and C. Vaughan. *Zoltan: A Dynamic Load Balancing Library for Parallel Applications; User's Guide.* Sandia National Laboratories, Albuquerque, NM, 1999. Tech. Report SAND99-1377. Open-source software distributed at `http://www.cs.sandia.gov/Zoltan`.

[21] R. E. Dillon Jr. A parametric study of perforated muzzle brakes. ARDC Tech. Report ARLCB-TR-84015, Benét Weapons Laboratory, Watervliet, 1984.

[22] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK User's Guide.* SIAM, Philadelphia, 1979.

[23] R. Elsässer, B. Monien, and R. Preis. Diffusion schemes for load balancing on heterogeneous networks. *Theory Compt. Syst.*, 35:305–320, 2002.

[24] J. Faik, J. D. Teresco, K. D. Devine, J. E. Flaherty, and L. G. Gervasio. A model for resource-aware load balancing on heterogeneous clusters. Technical Report CS-05-01, Williams College Department of Computer Science, 2005.

[25] J. E. Flaherty, M. Dindar, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. An adaptive and parallel framework for partial differential equations. In D. F. Griffiths, D. J. Higham, and G. A. Watson, editors, *Numerical Analysis 1997 (Proc. 17th Dundee Biennial Conf.)*, number 380 in Pitman Research Notes in Mathematics Series, pages 74–90. Addison Wesley Longman, 1998.

[26] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws. *J. Parallel Distrib. Comput.*, 47:139–152, 1997.

[27] J. E. Flaherty, R. M. Loy, M. S. Shephard, and J. D. Teresco. Software for the parallel adaptive solution of conservation laws by discontinuous Galerkin methods. In B. Cockburn, G. Karniadakis, and S.-W. Shu, editors, *Discontinous Galerkin Methods Theory, Computation and Applications*, volume 11 of *Lecture Notes in Computational Science and Engineering*, pages 113–124, Berlin, 2000. Springer.

[28] P. Gross and J. Subholk. A resource monitoring system for network aware applications. Tech. Report CMU-CS-97-194, Carnegie Mellon University, School of Computer Science, December 1997.

[29] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Scien. Comput.*, 16(2):452–469, 1995.

[30] N. T. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the Message Passing Interface. *J. Parallel Distrib. Comput.*, 63(5):551–563, May 2003.

[31] G. Karypis and V. Kumar. Parallel multilevel graph partitioning. Tech. Report 95-036, University of Minnesota, Department of Computer Science, Minneapolis, MN, 1995.

[32] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scien. Comput.*, 20(1), 1999.

[33] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. *SIAM Review*, 41(2):278–300, 1999.

[34] G. Karypis, K. Schloegel, and V. Kumar. *Parallel Graph Partitioning and Sparse Matrix Ordering Library.* University of Minnesota, Department of Computer Science and Army HPC Center, Minneapolis, MN, 1998. Version 2.0.

[35] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 29:291–307, 1970.

[36] S. Kumar, S. K. Das, and R. Biswas. Graph partitioning for parallel applications in heterogeneous Grid environments. In *Proc. IPDPS 2002*, Fort Lauderdale, 2002. IEEE.

[37] B. Maerten, D. Roose, A. Basermann, J. Fingberg, and G. Lonsdale. DRAMA: A library for parallel dynamic load balancing of finite element applications. In *Proc. Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, 1999. Library distributed under license agreement from `http://www.ccrl-nece.de/~drama/drama.html`.

[38] T. Minyard and Y. Kallinderis. Parallel load balancing for dynamic execution environments. *Comput. Methods Appl. Mech. Engrg.*, 189(4):1295–1309, 2000.

[39] W. F. Mitchell. Refinement tree based partitioning for adaptive grids. In *Proc. Seventh SIAM Conf. on Parallel Processing for Scientific Computing*, pages 587–592. SIAM, 1995.

[40] W. F. Mitchell. The design of a parallel adaptive multi-level code in Fortran 90. In *International Conference on Computational Science (3)*, volume 2331 of *Lecture Notes in Computer Science*, pages 672–680. Springer, 2002.

[41] H. T. Nagamatsu, K. Y. Choi, R. E. Duffy, and G. C. Carofano. An experimental and numerical study of the flow through a vent hole in a perforated muzzle brake. ARDEC Tech. Report ARCCB-TR-87016, Benet Weapons Laboratory, Watervliet, 1987.

[42] J. R. Pilkington and S. B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Trans. on Parallel and Distributed Systems*, 7(3):288–300, 1996.

[43] J.-F. Remacle, J. Flaherty, and M. Shephard. An adaptive discontinuous Galerkin technique with an orthogonal basis applied to compressible flow problems. *SIAM Review*, 45(1):53–72, 2003.

[44] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[45] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Comp. Sys. Engng.*, 2:135–148, 1991.

[46] S. Sinha and M. Parashar. Adaptive system partitioning of AMR applications on heterogeneous clusters. *Cluster Computing*, 5(4):343–352, October 2002.

[47] G. Sod. *Numerical Methods in Fluid Dynamic.* Cambridge University Press, Cambridge, 1985.

[48] V. E. Taylor and B. Nour-Omid. A study of the factorization fill-in for a parallel implementation of the finite element method. *Int. J. Numer. Meth. Engng.*, 37:3809–3823, 1994.

[49] J. D. Teresco, M. W. Beall, J. E. Flaherty, and M. S. Shephard. A hierarchical partition model for adaptive finite element computation. *Comput. Methods Appl. Mech. Engrg.*, 184:269–285, 2000.

[50] J. D. Teresco, K. D. Devine, and J. E. Flaherty. *Numerical Solution of Partial Differential Equations on Parallel Computers*, chapter Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations. Springer-Verlag, 2005.

[51] J. D. Teresco, L. Effinger-Dean, and A. Sharma. Resource-aware parallel adaptive computation for clusters. In V. S. Sunderam, G. D. van Albada, and P. M. A. Sloot, editors, *Proc. Computational Science – ICCS 2005: 5th International Conference*, volume 3515 of *Lecture Notes in Computer Science*, pages 107–115, Atlanta, 2005. Springer.

[52] J. D. Teresco, J. Faik, and J. E. Flaherty. Hierarchical partitioning and dynamic load balancing for scientific computation. Technical Report CS-04-04, Williams College Department of Computer Science, 2004. To appear, Proc. PARA '04.

[53] J. D. Teresco, J. Faik, and J. E. Flaherty. Resource-aware scientific computation on a heterogeneous cluster. *Computing in Science & Engineering*, 7(2):40–50, 2005.

[54] C. Walshaw. *The Parallel JOSTLE Library User's Guide, Version 3.0.* University of Greenwich, London, UK, 2002. Library distributed under free research and academic license at `http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/`.

[55] C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. *Parallel Comput.*, 26(12):1635–1660, 2000.

[56] C. Walshaw and M. Cross. Multilevel Mesh Partitioning for Heterogeneous Communication Networks. *Future Generation Comput. Syst.*, 17(5):601–623, 2001. (originally published as Univ. Greenwich Tech. Rep. 00/IM/57).

[57] C. Walshaw, M. Cross, and M. Everett. Dynaming load-balancing for parallel adaptive unstructured meshes. In *et al.*. M. Heath, editor, *Proc. Par. Proc. for Sci. Comp.* SIAM, 1997.

[58] C. H. Walshaw and M. Berzins. Dynamic load balancing for PDE solvers on adaptive unstructured meshes. *Concurrency: Practice and Experience*, 7(1):17–28, 1995.

[59] C. H. Walshaw, M. Cross, and M. Everett. Mesh partitioning and load-balancing for distributed memory parallel systems. In *Proc. Par. Dist. Comput. for Comput. Mech.*, Lochinver, Scotland, 1997.

[60] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proc. Supercomputing '93*, pages 12–21. IEEE Computer Society, 1993.

[61] M. Willebeek-LeMair and A. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Parallel and Distrib. Sys.*, 4(9):979–993, 1993.

[62] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1(1):119–132, 1998.

[63] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Comput. Syst.*, 15(5-6):757–768, October 1999.

[64] P. Wong, H. Jin, and J. Becker. Load balancing multi-zone applications on a heterogeneous cluster with multi-level parallelism. In *Proc. Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar'04)*, pages 388–393, Cork, 2004.

[65] L. Xiao, Z. Zhang, and Y. Qu. Effective load sharing on heterogeneous networks of workstations. In *Proc. IPDPS'2000*, Cancun, 2000.

APPENDIX A

DRUM Developer's Manual

We describe in this appendix the important data structures implemented in the DRUM software. We also describe in more detail the public functions in DRUM and how they operate on DRUM data structures.

A.1 DRUM machine model

The DRUM machine model (dmm) structure is the main data structure in DRUM. The dmm structure is used in all public functions of DRUM.

```
typedef struct DRUM_machineModel_struct {
  MPI_Comm comm;
  DRUM_list processList;
  struct DRUM_machineModelNode_struct **leaves;
  DRUM_machineModelNode *root;

  int rank;
  int numprocs;
  int debug;
  int initialized;
  int numberOfNodes;
  int monitoringFrequency;
  int allNodesMonitorable;
  int numberOfLeaves;
  int use_snmp;
  int use_kstat;
  int monitor_memory;
  int use_dynamicMonitoring;
  int monitors_running;
  int use_networkPowers;
  int use_flatModel;
  float fixed_networkWeight;
  char *xml_filename;
} DRUM_machineModel;
```

comm MPI Communicator for the entire "DRUM" universe.

processList Linked list of pointers to nodes in the tree for which process rank *"rank"* is responsible for monitoring.

leaves Linked list of pointers to the leaves of the tree. Permits direct access for the root to the leaves without having to traverse the whole tree.

root Root of the tree.

rank Rank of the local process within *"comm"*.

numprocs Number of processes in *"comm"*.

debug Debug level.

- DRUM_SILENT (0): No DRUM debug information is output.
- DRUM_INFO_MASTER (1): Prints some information, greeting messages, computed powers only from master process.
- DRUM_INFO_ALL (2): Prints some information, greeting messages, computed powers from all processes.
- DRUM_FORMULAS (3): Prints above plus information about computing formula.
- DRUM_MONITORS (4): Prints above plus information from monitors.
- DRUM_ALL (5): Prints all available debugging information.

initialized set to 1 if DRUM_init_Model has been called on the current machine model.

numberOfNodes Number of nodes in the tree.

monitoringFrequency How often to probe system for monitoring information (unit = 1 second).

allNodesMonitorable Set to 0 if at least one network node cannot be probed for network traffic (very common with simple network switches).

numberOfLeaves Number of leaves in the tree.

use_snmp Set to 1 if the Simple Network Management Protocol (SNMP) library is to be for network monitoring.

use_kstat Set to 1 if Solaris Kernel statistics are to be used for network and CPU monitoring (only on Solaris).

monitor_memory Set to 1 if memory monitoring is desired.

use_dynamic_monitoring Set to 1 if dynamic monitoring is desired (requires thread execution).

monitors_running Set to 1 if the monitoring threads have been started.

use_network_powers Set to:

- (0): Ignore network powers
- (1): Use same fixed weight for network powers in all the nodes.
- (2): Use dynamic weights for network powers (depend on traffic volume at each node).

use_flat_model Set to 1 if the execution environment is flat, i.e. all compute nodes are attached to a single switch or router.

xml_filename File name of the XML description of the system.

The recursive (tree) structure of the Machine Model is implemented through the Machine Model Node structure, defined as follows:

```
typedef struct DRUM_machineModelNode_struct {
  int    nodeCanBeMonitored;
  char  *networkIdentifier;
  double power;

  double commWeightSum;
  double commWeight;
  double commPowerSum;

  DRUM_typeOfNode nodeType;
   union nodeTypeStruct {
     DRUM_computingNode *compNode;
     DRUM_networkNode   *netNode;
  } type;

  struct DRUM_nic_struct *networkInterfaceCard;
  struct DRUM_cpuMem_struct *cm;

  int currentNodeNumber;
  int lowerBoundNodeNumber;
  int upperBoundNodeNumber;
} DRUM_machineModelNode;
```

nodeCanBeMonitored Set to 0 if a node cannot be monitored. A network Node cannot be monitored if no traffic info can be collected from that node (happens for simple switches for example).

networkIdentifier Identifier in the execution environment of the node. For now, we use the IP address as an identifier.

power Computed power of a node (between 0 and 1).

commWeightsum Sum of the communication weights of the node's immediate children (if any).

commPowerSum Sum of the communication powers of the node's immediate children (if any).

commWeight Communication power weight.

nodeType Type of node, could be either a computation node or a communication node.

type Actual computation or network node structure corresponding to this node.

networkInterfaceCard Pointer to the network interface card object that implements network monitoring for the current node.

cm Pointer to the objects that implements CPU and memory monitoring for the current node (if it is a computation node).

currentNodeNumber Index of the node in the tree. Numbering follows a postfix ordering.

lowerBoundNodeNumber Node number of the leftmost leaf of the subtree rooted at the current node.

upperBoundNodeNumber Node number of the rightmost leaf of the subtree rooted at the current node.

The DRUM_computingNode structure implements the computing node type, whereas the DRUM_networkNode structure implements the network node type. Descriptions of these two structures are presented in what follows:

```
typedef struct DRUM_computingNode_struct {
  double power;
  int   *sameNodeProcessList;
  int    processCount;
  int    electedProcess;
  int    representative;
} DRUM_computingNode;
```

power Computing power of the node.

sameNodeProcessList Array of ranks of processes running on the node.

processCount Number of processes on the same computing node. Default is 1.

electedProcess When multiple processes are running on the same computing node, only one process, the elected process, performs the network monitoring (NIC probing).

representative Rank of process monitoring an instance of a computation node (in case of multiple processes per node).

```
typedef struct DRUM_networkNode_struct {
  int representative;
  unsigned short numberOfChildren;
  struct DRUM_machineModelNode_struct **childrenList;
} DRUM_networkNode;
```

representative Rank of the process elected to request traffic information from the network node.

numberOfChildren Number of immediate children of the node.

childrenList Array of pointers to children of the node.

The monitoring is performed though two types of objects: (i) an networkInterfaceCard object for network Monitoring and (ii) a cpuMem object for CPU and memory monitoring. We describe these two objects in what follows:

```
typedef struct DRUM_nic_struct {
  int numberOfInterfaces;
  int monitoringThreadID;
  char *LoopBackOid;
  int LoopBackInterface;
  struct DRUM_interface *interfaces;
#ifdef HAVE_SNMP
  struct snmp_session *ss;
#endif
#ifdef HAVE_PTHREADS
  pthread_t threadID;
  int STOP_THREAD;
  pthread_mutex_t stopThreadMutex;
#endif
  DRUM_Timer *nicTimer;
  double power;
  double caf; /*average CAF across all interfaces of the same node*/
  DRUM_machineModel *dmm;  /* for availability in thread function */
} DRUM_nic;
```

numberOfInterfaces Number of communication interfaces of a given node.

monitoringThreadID ID of the thread performing the system probing for communication traffic.

LoopBackOid String describing the Object ID (SNMP OID) of the software loopback interface.

LoopBackInterface Index of the software loop back interface.

interfaces Array of communication interfaces (see DRUM_interface below).

ss Pointer to the SNMP session open.

threadID ID of the thread performing the communication traffic monitoring.

STOP_THREAD Set to 1 if the monitoring thread has been stopped.

stopThreadMutex Mutex variable for sequential access the the STOP_THREAD variable.

nicTimer Used to time the monitoring duration (high resolution timing when possible).

power Computed network power of the node to which the current network interface card object is attached.

caf Average CAF across all interfaces of the same node.

dmm Pointer to the whole machine model, for availability in thread function.

```
struct DRUM_interface {
  unsigned int  speed;
  char *description;
  unsigned long in_octets;
  unsigned long out_octets;
  double  caf;
};
```

speed Maximum bandwidth of the interface.

description Character Description of the interface (e.g. Interface0).

in_octets Number of octets/packets that have arrived through the interface.

out_octets Number of octets/packets that have departed through the interface.

```
struct DRUM_cpuMonitor {
  double mflops;
  double averageCpuUsage;
  double* CpuLoadAverage;
  double averageCpuIdleTime;
  double power;
};
```

mflops The MFLOPS number of the computation node as determined by the (LINPACK) benchmark run.

averageCpuUsage Average CPU usage by the target application process during a monitoring period, i.e. how much attention is the target application process is getting from the CPU.

cpuLoadAverage 3-elements array of CPU load averages during the last 1, 5 and 15 minutes (similar to Unix uptime).

averageCpuIdleTime Average CPU idle time during a monitoring period.

power Computed processing power.

```
struct DRUM_memoryMonitor {
  long  totalMemory;
  int   availableMemory;
};
```

totalMemory Total memory on node on which the CpuMemObject is attached.

availableMemory Available memory on node.

```
typedef struct DRUM_cpuMem_struct {
  struct DRUM_cpuMonitor cpu;
  struct DRUM_memoryMonitor memory;
  DRUM_Timer *cpuMemTimer;
  /* int  pid; not used? */
  int  MonitoringFrequency;
  int  numberOfCPUs;
#ifdef HAVE_PTHREADS
  pthread_t threadID;
  int TYPE2_THREAD_STOP;
  pthread_mutex_t type2_stopThreadMutex;
#endif
  DRUM_machineModel *dmm;  /* added to allow access to this from
  threads */
  int  TestsDone;
  int  LoadTestDone;
} DRUM_cpuMem;
```

cpu DRUM_cpuMonitor Object.

memory DRUM_memoryMonitor Object.

cpuMemTimer Used to time the monitoring period.

monitoringFrequency Probing frequency of the node for CPU and memory statistics (unit: seconds).

numberOfCPUs Number of CPUs in the computing node to which the CpuMem object is attached.

threadID Thread ID of the thread monitoring the computation node to which the cpuMem object is attached.

TYPE2_THREAD_STOP Boolean to control when the monitoring thread should be stopped.

type2_stopThreadMutex Mutex for sequential access the TYPE2_THREAD_STOP boolean.

dmm Pointer to the overall DRUM machine model, added to allow access to "this" from threads.

A.2 DRUM public interface

We present below the public functions of DRUM. We also attempt to describe in some detail what resources are involved in each call and what known errors each of the functions might generate. We start by giving a list of error conditions returned by DRUM functions.

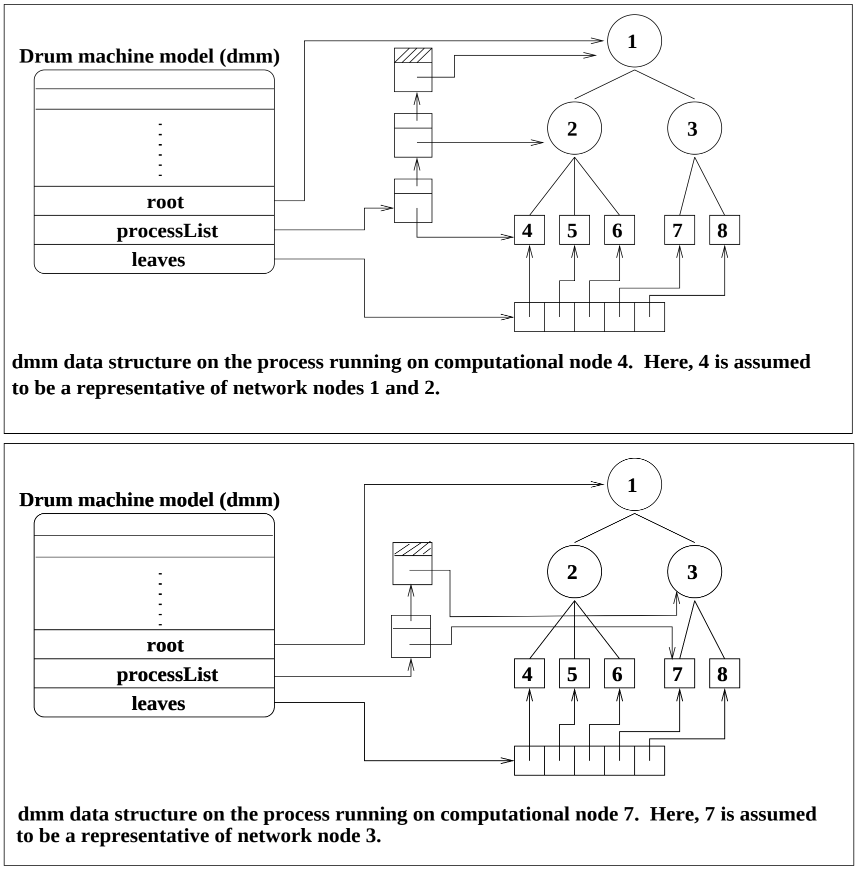
DRUM_OK(0) No errors.

DRUM_WARN(1) Some warning occurred in DRUM library. Application should be able to continue running.

DRUM_FATAL(-1) A fatal error occured.

DRUM_MEMERR(-2) Memory allocation failed. With this error, it could be possible to try a different, more memory-friendly approach.

```
**********************************************************************
DRUM_machineModel *DRUM_createMachineModel(MPI_Comm comm,
   int debug_level);
```



dmm data structure on the process running on computational node 4. Here, 4 is assumed to be a representative of network nodes 1 and 2.

dmm data structure on the process running on computational node 7. Here, 7 is assumed to be a representative of network node 3.

Figure A.1: View of DRUM machine model data structure on two processes running on two different computation nodes.

The call to DRUM_createMachineModel reads the XML description of the execution environment and builds a new machine model using processes specified by *comm.* The amount of output to be produced by DRUM is specified in debug_level. This function must be called by all processes in *comm.* It return NULL on error and a pointer to the newly-created machine model otherwise.

```
***********************************************************************
int DRUM_initMachineModel(DRUM_machineModel *dmm);
```

Numbers the tree nodes, set representatives, attach monitoring objects to the nodes and create “processList” to identify nodes to be monitored by processes.

```
***********************************************************************
int DRUM_deleteMachineModel(DRUM_machineModel *mm);
```

Destroys and deallocates a machine model. This function must be called by all processes in the *comm* that created it. Returns error code.

```
**********************************************************************
void DRUM_printMachineModel(DRUM_machineModel *mm, FILE *powerFile);
```

Prints a representation of the machine model and computed powers (when available) to the given open file pointer. Pass stdout or NULL to print to screen or log files.

```
**********************************************************************
int DRUM_setMonitoringFrequency(DRUM_machineModel *mm, int monFreq);
```

Sets monitoring frequency, measured in tenths of a second. Returns error code.

```
**********************************************************************
int DRUM_startMonitoring(DRUM_machineModel *mm);
```

Starts dynamic monitoring. This function will create pthreads to monitor the node and application performance. Returns error code.

```
**********************************************************************
int DRUM_stopMonitoring(DRUM_machineModel *mm);
```

Stops dynamic monitoring. This function will destroy pthreads and populate tree with performance data to be accessed by query functions below. Returns error code

```
**********************************************************************
int DRUM_computePowers(DRUM_machineModel *mm);
```

Computes node powers. Returns error code.

```
**********************************************************************
float DRUM_getLocalPartSize(DRUM_machineModel *mm, int *ierr);
```

Gets local partition size based on computed powers. Returns local partition size in function return, error code in ierr.

```
**********************************************************************
int DRUM_setParam(DRUM_machineModel *dmm, char *param, char *value);
```

Sets individual DRUM parameters. Returns error code. Parameters and values are specified by strings, even when param is a float or int value. For int values, can also specify "TRUE" for 1 and "FALSE" for 0. Strings are not case sensitive. Inspired by (and very simplified version of) Zoltan's Zoltan_Set_Param utility. Valid parameters, types, and defaults are described in Table A.2.

```
**********************************************************************
int DRUM_readParamFile(DRUM_machineModel *dmm, char *file);
```

Sets DRUM parameters from a file. Returns error code. Valid parameters are anything that can be passed to DRUM_setParam. File format is any number of lines with pairs of strings "PARAM" "VALUE". Each line results in a call DRUM_setParam(dmm, PARAM, VALUE).

Parameter	**Type**	**Default**
MONITORING_FREQUENCY	INT	1
USE_DYNAMIC_MONITORING	INT	1
USE_SNMP	INT	system-dependent
USE_KSTAT	INT	system-dependent
MONITOR_MEMORY	INT	0
USE_FLAT_MODEL	INT	1
USE_NETWORK_POWERS	INT	1
DEBUG_LEVEL	STRING/INT	Passed to createMachineModel
XML_TOPOLOGY_FILE	STRING	drum.xml

Table A.1: DRUM parameters, their types and default values.