A model for resource-aware load balancing on heterogeneous clusters

J. Faik*†, L. G. Gervasio*†, J. D. Teresco†‡, J. E. Flaherty*,
E. G. Boman§, J. Chang*, K. D. Devine§

Abstract

We address the problem of partitioning and dynamic load balancing on clusters with heterogeneous hardware resources. We propose DRUM, a model that encapsulates hardware resources and their interconnection topology. DRUM provides monitoring facilities for dynamic evaluation of communication, memory, and processing capabilities. Heterogeneity is quantified by aggregating the information from the monitors into a scalar form, easily usable by existing load balancing algorithms. We used DRUM to guide load balancing in the solution of a Rayleigh-Taylor instability problem on a heterogeneous cluster. A decrease of 22.6% in execution time out of a theoretical maximum of 24% is observed.

1 Introduction

Clusters have gained wide acceptance as a viable alternative to tightly-coupled parallel computers. They provide cost-effective environments for running computationally-intensive parallel and distributed applications. An attractive feature of clusters is the ability to expand their computational power incrementally by incorporating additional nodes. This expansion often results in heterogeneous environments, as the newly-added nodes often have superior capabilities. Grid technologies such as MPICH-G2 [7] have enabled computation on even more heterogeneous and widely-distributed systems. Internet-connected systems include more heterogeneity and extreme network hierarchy.

Applications involving the solution of partial differential equations are among the most demanding computational problems, arising in fields such as fluid dynamics, materials science, biomechanics, and ecology. Adaptive strategies that automatically refine, coarsen, and/or relocate meshes and change the method to obtain a solution with a prescribed level of accuracy as quickly as possible are essential tools to solve modern multi-dimensional transient problems [3]. The usual approach to parallelizing these problems is to distribute a

*Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180
†Computer Science Research Institute, Sandia National Laboratories, Albuquerque, NM 87185-1111
‡Department of Computer Science, Williams College, Williamstown, MA 01267
§Sandia National Laboratories, Albuquerque, NM 87185-1111

discretization (mesh) of the domain across cooperating processors, and then compute a solution, appraising its accuracy using error estimates at each step. If the solution is accepted, the computation proceeds to the next step. Otherwise, the discretization is refined adaptively, 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. Sandia National Laboratories' *Zoltan* library [5] provides a common interface to several state-of-the-art partitioners and dynamic load balancers [1, 2, 4, 8, 14]. Most of these procedures seek to achieve an even distribution of computational work, while minimizing interprocess communication and data movement necessary to achieve the new decomposition. However, these procedures do not account for heterogeneity in the execution environment.

We propose DRUM, a model that captures the structure and dynamics of heterogeneous clusters in order to increase the effectiveness of load balancing. DRUM (Dynamic Resource Utilization Model) encapsulates hardware resources, their capabilities and their interconnection topology in a tree structure, and provides a mechanism for dynamic monitoring and evaluation of their utilization. Monitors in DRUM run concurrently with the user application to collect memory, network, and CPU utilization and availability statistics. Information from the monitors is distilled to a scalar "power" value, readily used by load balancing algorithms capable of producing non-uniform partition sizes. Our model has been designed to work with Zoltan, but may also be used as a stand-alone library.

We describe related work in §2 and give details of DRUM in §3. §4 presents computations that use DRUM to reduce application run time on a heterogeneous cluster, and §5 discusses the results and describes plans for future development and experimentation.

2 Related Work

The popularity of clusters has motivated several recent efforts to study dynamic load balancing for heterogeneous systems.

Minyard and Kallinderis [9] monitor process "wait times" and use these to assign element weights. The main drawback of this approach is that it requires manual insertion of measurement probes in the user application. Walshaw and Cross [15]. modify a multilevel algorithm seeking to minimize a cost function based on a model of the heterogeneous communication network. Sinha and Parashar [11] use the Network Weather Service (NWS) [16] to gather information about the state and capabilities of available resources; then they compute the load capacity of each node as a weighted sum of processing, memory, and communications capabilities. Reported experimental results show that system-sensitive partitioning resulted in a decrease of application execution time ranging from 6% on 8 processors to 18% on a 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.

3 DRUM: Dynamic Resource Utilization Model

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 (*i*) encapsulates the details of the execution environment, and (*ii*) provides a facility for dynamic, modular and minimally-intrusive monitoring of the execution environment.

Unlike the directed-graph hardware model used in the Rensselaer Partition Model [13], DRUM uses 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 inherent structure of DRUM leads naturally to a topology-driven, yet transparent, execution of hierarchical partitioning [12]. 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.

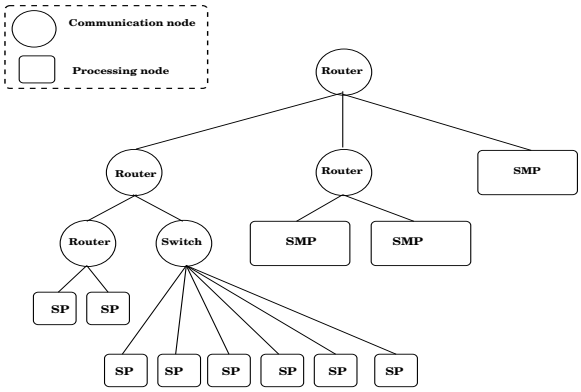

Figure 1: Tree constructed by DRUM to represent a heterogenous network.

Figure 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.

3.1 Model creation

DRUM requires a tree model of the execution environment's underlying interconnection. An XML file (Figure 2), 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 Protocol) and threading capabilities. The configuration tool needs to be re-run only when hardware characteristics of the system have changed.

```
<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 2: XML file description of the IBM Netfinity cluster used in the computation in Section 3.3.3. The cluster includes a hierarchical network.

3.2 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 [6] 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) 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 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[1] 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*

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

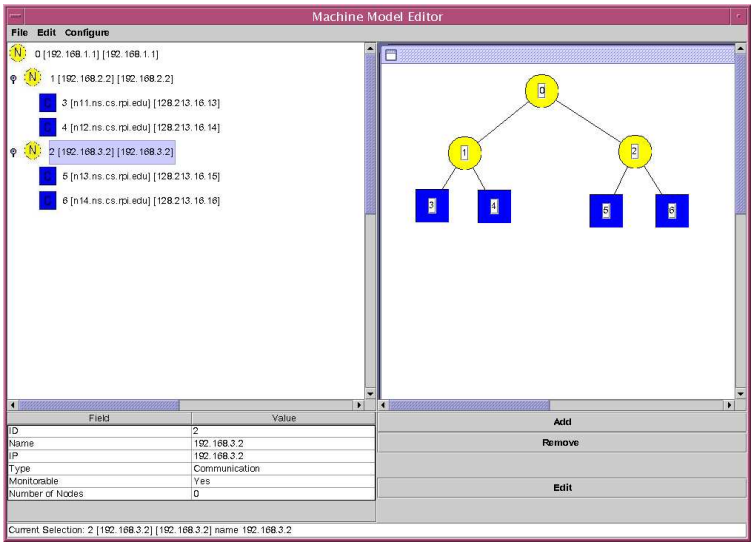

Figure 3: Screen shot of the graphical Java program, used to aid in the creation of the XML machine topology description. Here, a description of the cluster used for the computation in Section 3.3.3 is being edited.

factor, $CAF = \lambda + \mu$, is computed for all communication interfaces of each node.

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.

3.3 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 to the Sinha and Parashar approach [11]. Given power values for each node, any partitioning procedure capable of producing variable-sized 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.

3.3.1 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. Thus, for node n, we estimate the

processing power, p_n, as:

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

The weights $w_{dynamic}$ and w_{static} add to 1 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 as 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 $w_{dyanamic}$ should be bounded in order to ensure a non-negligible contribution of the static benchmarks in the overall expression. Currently, these weights are assigned manually. The processing power of internal nodes is computed as the sum of the powers of the node's immediate children.

3.3.2 Communication power

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

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

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. This weighting scheme is only a first approximation. We discuss possible improvements in §5.

3.3.3 Preliminary experiment

We performed preliminary tests to experiment with the formulas involved in the power computation. We use four identical processors connected by a hierarchical network, with heterogeneity simulated by adding synthetic computation and communication load on selected processors. Processing and communication powers are computed dynamically. Figure 4 shows the computed powers which represent percentages of the total load that should be assigned to each node. In each case, T1, the master process, must do some extra work coordinating the other processes and managing the standard output from all processes, so it is

given a smaller percentage of work by the model. The ability to account for these otherwise hidden costs is a significant advantage of the dynamic monitoring approach.

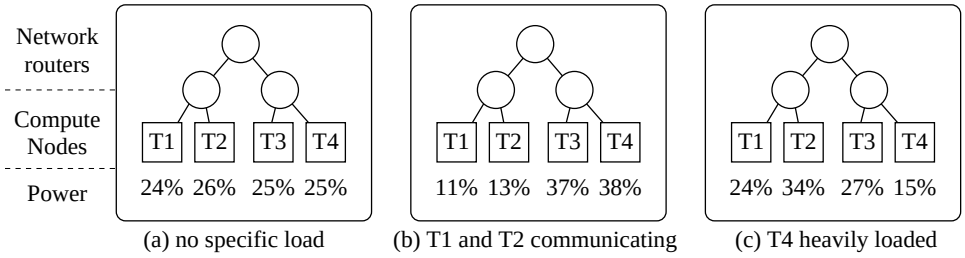

Figure 4: Power computation after four minutes of monitoring on a small cluster. The circles represent network routers, the squares represent identical computation nodes. The percentages below the squares are the powers assigned to each node. (a) The nodes and network are unloaded, so each node gets the same percentage of work, (b) T1 and T2 communicate heavily, so those nodes are given less work to reduce that communication, and (c) T4 is heavily loaded, so it gets a smaller percentage of the work.

4 Computational results

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.*, propose metrics for CPU and memory heterogeneity defined as the variance of computing powers and memory capacities among the computation nodes [17]. 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 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 formulas to measure CPU heterogeneity, substituting the MFLOPS numbers obtained from

the benchmark for the MIPS values. MFLOPS provides a more reliable measure of CPU performance than raw MIPS.

We consider a cluster with a homogeneous communication network, but where the speed of the computation nodes varies. We solve a two-dimensional Rayleigh-Taylor instability problem [10] on a rectangular domain on eight processors of the Sun cluster at Williams College: five 450MHz Sparc UltraII processors, and three 300MHz Sparc UltraII processors, connected by fast (100 Mbit) Ethernet. The CPU heterogeneity for our testbed cluster is 0.19. During 200 adaptive mesh refinement and rebalancing steps there was a maximum of 6320 triangular elements. Single-processor runs indicate that the fast nodes have a computation rate of approximately 1.5 times that of the slow nodes. Given an equal distribution of work, the fast nodes will be idle one third of the time. By giving 50% more work to each of the five fast nodes, an overall speedup of 24% is theoretically possible (assuming no communication). For our computations, we used a partition-weighted version of the Octree/SFC procedure. We observed a decrease in total wall clock time (including setup and all computation and communication) from 9507 seconds to 7351 seconds, an improvement of 22.6%, which is 94.1% of the theoretical maximum of 24%. A similar computation on a four-processor cluster showed an improvement of 10% out of a theoretical maximum of 11%.

5 Discussion

Our preliminary 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 that can be captured by the benchmarks, and highly dynamic systems where the execution environment may be shared with other processes.

We are currently testing the procedures on a wider variety of heterogeneous systems and for different applications. We are implementing hierarchical balancing procedures that interact with DRUM to tailor partitions to a given network topology [12]. In addition to the ability to produce weighted partitions, this will allow different load balancing algorithms to be used, as appropriate, in different parts of the network hierarchy [13].

The assignment of the weights in the power expression in DRUM should be improved. The weights should depend on a ratio of computation to communication. This ratio may be estimated without inserting probes into the application by using 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 when overlapping of computation and communication is considered.

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.

Acknowledgments

Faik, Gervasio, Teresco and Flaherty were supported by contract 15162 with Sandia National Laboratories, a multi-program laboratory operation by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000. Computer systems used include the "Bullpen Cluster" of Sun Microsystems servers at Williams College and the IBM Netfinity cluster at Rensselaer Polytechnic Institute.

References

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

[2] 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.

[3] K. Clark, J. E. Flaherty, and M. S. Shephard. *Appl. Numer. Math., special ed. on Adaptive Methods for Partial Differential Equations*, 14, 1994.

[4] 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.

[5] 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.

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

[7] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. Journal of Parallel and Distributed Computing (JPDC), to appear, 2003.

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

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

[10] 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.

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

[12] J. D. Teresco. Hierarchical partitioning and dynamic load balancing for scientific computation. In preparation, 2003.

[13] 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.

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

[15] 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).

[16] 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.

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