A model for resource-aware load balancing on heterogeneous clusters

J. Faik, J. E. Flaherty, L. G. Gervasio
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180

J. D. Teresco
Department of Computer Science
Williams College
Williamstown, MA 01267

K. D. Devine
Sandia National Laboratories
Albuquerque, NM 87185-1111

January 15, 2005

Keywords: I.3.1.d Parallel processing, C.1.2.e Load balancing and task assignment, C.0.d Modeling of computer architecture

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 merging the information from the monitors to produce a scalar number called "power." This power allows DRUM to be used easily by existing load-balancing procedures such as those in the Zoltan Toolkit while placing minimal burden on application programmers. We demonstrate the use of DRUM to guide load balancing in the adaptive solution of a Laplace equation on a heterogeneous cluster. We observed a significant reduction in execution time compared to traditional methods.

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 [8] have enabled computation on even more heterogeneous and widely-distributed systems. Internet-connected systems include more heterogeneity and extreme network hierarchy.

Our goal is to improve the efficiency of scientific computations in these heterogeneous environments, while putting as little burden as possible on application programmers. We propose the Dynamic Resource Utilization Model (DRUM)[1] [5, 23], a persistent and dynamic model of the execution environment that captures the structure and dynamics of heterogeneous clusters in order to increase the effectiveness of load balancing. The 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. Since our initial focus is on guiding a resource-aware dynamic load balancing, information from the monitors is distilled to a scalar "power" value, readily used by load-balancing algorithms capable of producing non-uniform partition sizes.

We apply DRUM to applications involving the parallel adaptive solution of partial differential equations. These are among the most demanding computational problems, arising in fields such as fluid dynamics [14], materials science [1], biomechanics [12], and ecology [2]. 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 discretization (mesh) of the domain across cooperating processors, and then compute a solution for a global time step and appraise its accuracy at each step. If the solution is accepted, the computation proceeds to the next time step. Otherwise, the mesh is adjusted adaptively, and work is redistributed, to correct for any load imbalance introduced by the adaptive step. If necessary, the numerical method may be changed. Thus, dynamic partitioning and load-balancing procedures become necessary

[1] `http://www.cs.williams.edu/drum`

since the locations where meshes must be refined or simpler methods replaced by more complex ones are not known *a priori* and are determined as part of the solution process. Sandia National Laboratories' Zoltan Toolkit [6] provides a common interface to several state-of-the-art partitioners and dynamic load balancers 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; [20] includes a survey of such procedures. Figure 1 shows the interaction between parallel adaptive application software

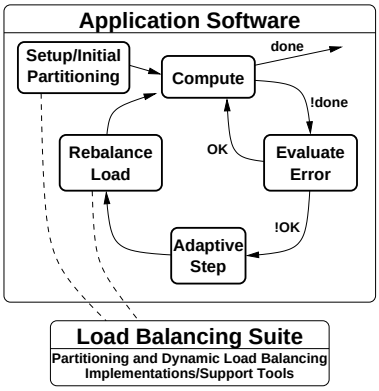

Figure 1: Program flow of a typical parallel adaptive computation using a load balancing suite such as Zoltan.

and a dynamic load balancing suite such as that in Zoltan. After an initial partitioning, the application performs computation steps, periodically evaluating error estimates and checking against specified error tolerances. If the error is within the tolerance, the computation continues. Otherwise, an adaptive refinement takes place, followed by dynamic load balancing before the computation resumes. However, these dynamic load balancing procedures do not directly account for heterogeneity in the execution environment. Results herein use power values computed by DRUM to guide Zoltan procedures to produce resource-aware partitions. Figure 2 shows the interaction among an application code, a load balancing suite such as Zoltan, and a resource monitoring system such as DRUM for a typical adaptive computation. When load balancing is requested, the load balancer queries the monitoring system's performance analysis component to determine appropriate parameters and partition sizes for the rebalancing step. While our examples will use DRUM with Zoltan, DRUM may also be used as a stand-alone library.

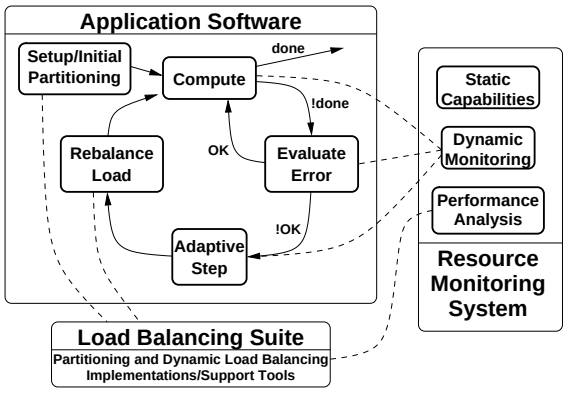

Figure 2: A typical interaction between an adaptive application code and a dynamic load balancing suite, when using a resource monitoring system (e.g., DRUM).

The next section describes some related work. We present the details of DRUM in Section 3. Section 4 contains results using DRUM to reduce application run time on a heterogeneous cluster. A discussion of the results and a presentation of plans for future development and experimentation are the subject of Section 5.

2 Related Work

The popularity of clusters has motivated several recent efforts to study dynamic load balancing for heterogeneous systems. Minyard and Kallinderis [10] use octree structures to conduct partitioning in 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 computation. These "wait" times measure how long each CPU remains idle while all other processors finish the same task. The cells are assigned load factors that are proportional to the "wait" times of their respective owning processes. Each octant load is subsequently computed as the sum of load factors of the cells contained within the octant. The octree algorithm then balances the load factors based on the weight factors of the octants, rather than the number of cells contained within each octant. Walshaw and Cross [24] conduct multilevel mesh partitioning for heterogeneous communication networks. They modify a multilevel algorithm, seeking to minimize a cost function based on a model of the heterogeneous communication network. The 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 representing processor interconnections. Each graph edge is weighted as a function of the length of the path between its corresponding processors. Lowekamp, et al. [9] present a resource monitoring system called Remos. Remos allows applications to collect information about network and host conditions across different network architectures. Sinha and Parashar [17] present a framework for adaptive system-sensitive partitioning and load balancing on heterogeneous and dynamic clusters. They use the Network Weather Service (NWS) [26] 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 significantly decreases application execution time. Most of these approaches are either related to a specific load-balancing algorithm or rely on information supplied externally at run-time or through instrumentation probes added to the user application. Our proposed system addresses some of these issues. DRUM does not require a specific load-balancing algorithm and relies on both static and dynamic information to evaluate resource usage.

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 [19], 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 Zoltan's hierarchical partitioning capabilities, where different load balancing procedures are used at different levels in the net-

work hierarchy [22]. 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.

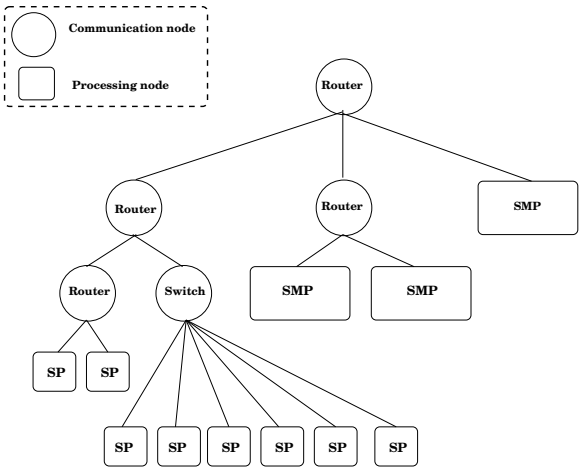

Figure 3: Tree constructed by DRUM to represent a heterogeneous network.

Figure 3 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's tree model of the execution environment is created upon initialization using an XML-format configuration file (Figure 4) that contains a list of nodes and description of their interconnection topology. This configuration file is created manually or with a graphical configuration tool called DrumHead [21]. DrumHead (Figure 5) provides capabilities including specification of cluster nodes and their characteristics, network topology and capabilities, specification of load balancing procedures and parameters for hierarchical balancing, initial assessment of node capabilities by running distributed benchmarks, and facilities to check availability of network management capabilities such as SNMP (Simple Network Management Protocol) and threading. The configuration tool needs to be re-run only when hardware characteristics of the system have changed.

```
<machinemodel><node>
type="NETWORK" nodenum="0" name="" IP="" isMonitorable="false"
parent="-1" imgx="361.0" imgy="52.0"
<lbmethod lbm="HSFC" KEEP_CUTS="1"></lbmethod>
</node><node>
type="SINGLE_COMPUTING" nodenum="2" name="mendoza.cs.williams.edu"
IP="137.165.8.140"  isMonitorable="true" parent="0"
imgx="50.0" imgy="138.0"
</node><node>
type="MULTIPLE_COMPUTING" nodenum="3" name="rivera.cs.williams.edu"
IP="137.165.8.130" isMonitorable="false" parent="0"
imgx="74.64 "imgy="263.0" numprocs="4"
<lbmethod lbm="HSFC" KEEP_CUTS="1">
</lbmethod></node>
...
</machinemodel>
```

Figure 4: An excerpt from the XML configuration file generated by DrumHead for the Bullpen Cluster at Williams College (see also Figure 6).

3.2 Capabilities Assessment

Resource capabilities are assessed initially using benchmarks. Currently, LINPACK [7] is used as a benchmark to compute a MFLOPS rating for the computation nodes of the cluster.

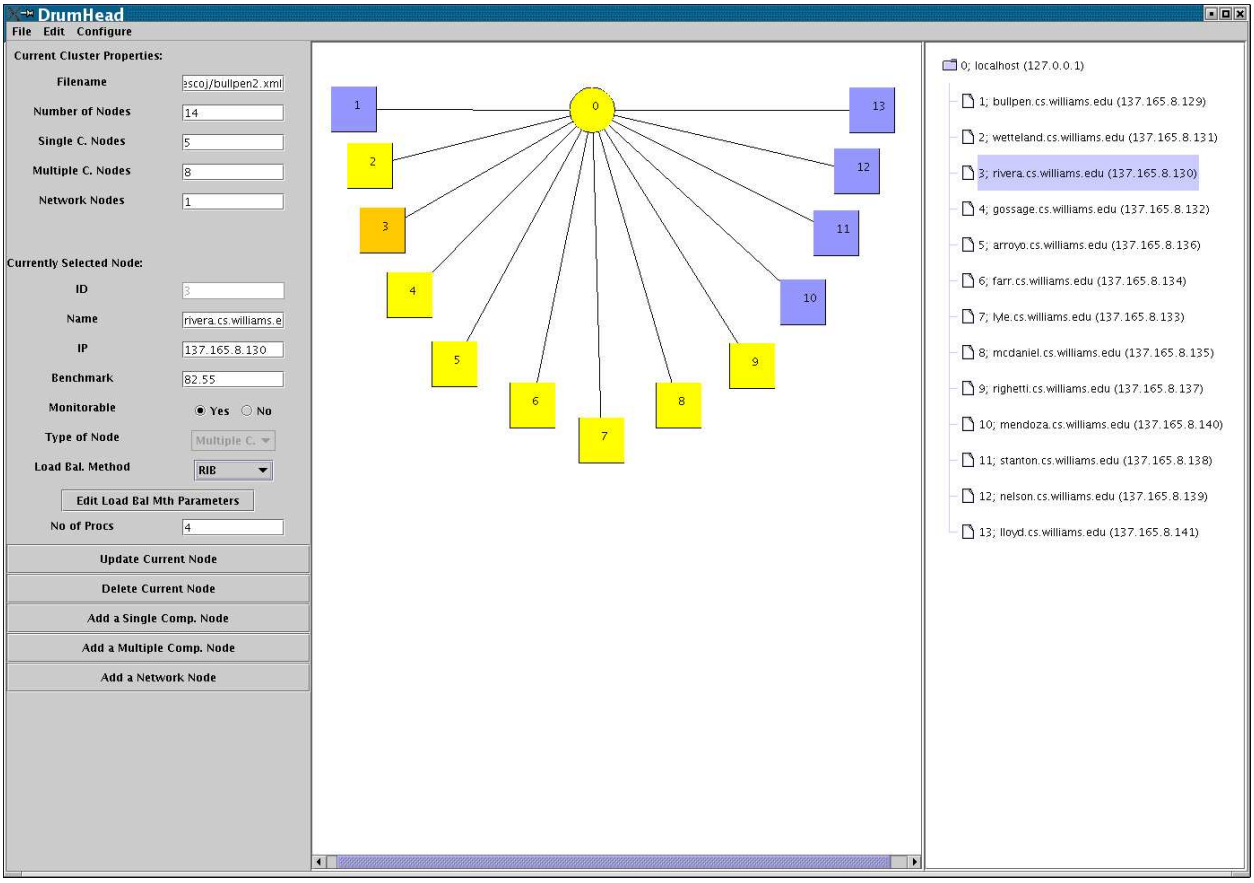

Figure 5: Screen shot of DrumHead, the graphical Java program used to aid in the creation of the XML machine topology description. Here, a description of the Bullpen cluster at Williams College is being edited.

The benchmark may be run from the configuration tool (Figure 5) or at the command line. The benchmarks may not accurately reflect the characteristics of a particular computation. We intend to allow user-specified callbacks that could be used to calibrate the machine model, likely using the same software that will be used for the actual computation.

During the course of a computation, the availability of resources can change dramatically, particularly in shared environments. In these dynamic environments, DRUM's capability assessments can be updated by *agents*: threads that run concurrently with the application to monitor each node. An application may choose to 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 `DRUM_startMonitoring()` function spawns the agent threads.

A call to `DRUM_stopMonitoring()` ends the dynamic monitoring. The monitoring agents contain `DRUM_nic` objects that monitor communication traffic and `DRUM_cpuMem` objects that monitor CPU load and memory usage. Some computation nodes, called *representatives*, are responsible for monitoring one or more network nodes.

A `DRUM_nic` object can be attached to either a computation or a network node. Versions of `DRUM_nic` objects have been implemented using the `net-snmp` library[2] and kernel statistics to collect network traffic information at each node.

A `DRUM_cpuMem` object gathers information about the CPU usage 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.

After monitoring has been stopped, `DRUM_computePowers()` updates the power of the nodes in the model. These powers, which are appropriate to use as resource-aware partition sizes in a dynamic load-balancing procedure, are queried with `DRUM_getLocalPartSize()`.

3.3 Node power

DRUM distills the information in the model to a power value for each node, a single number indicating the portion of the total load that should be assigned to that node. This is similar to the Sinha and Parashar approach [17]. Given power values for each node, any partitioning procedure capable of producing variable-sized partitions, including all Zoltan procedures, may be used to achieve an appropriate decomposition. Thus, any applications using a load-balancing procedure capable of producing non-uniform partitions can take advantage of DRUM with little modification. Applications that already use Zoltan can make use of DRUM simply by setting a Zoltan parameter, with no further changes needed.

The power at each node depends on processing power and communication power. We compute the power of node n as the weighted sum of the processing power p_n and communication power c_n:

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

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

3.3.1 Processing power

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 (i) CPU utilization $u_{n,j}$ by process j, (ii) the fraction i_t of time that CPU t is idle, and (iii) the node's static benchmark rating (in MFLOPS) b_n. 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 $\min(k_n - \sum_{j=1}^{k_n} u_{n,j}, \sum_{t=1}^{m} i_t)$. Since the operating system's CPU scheduler can be expected to give each of the k_n processes the same portion of the time on a node's CPUs, we assign all processes on node n equal 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, $p_n = \sum_{j=1}^{k_n} p_{n,j} = k_n p_{n,1}$. On internal nodes, p_n is the sum of the processing powers of the nodes' children.

3.3.2 Communication power

We estimate a node's communication power based on the communication the node. At each computation and (when possible) network node, DRUM's monitoring agents compute the rate of incoming packets λ and outgoing packets μ on each relevant communication interface. Just like in [18], for each interface i, we compute the available bandwidth (ABW) using measurements over a time interval T as

$$A_i(t, T) = \frac{1}{T} \int_t^{T+t} (C_i - (\lambda_i(t) + \mu_i(t)))dt,$$

where $A_i(t, T)$ is the available bandwidth at interface i through the time interval from t to $t + T$, and C_i is the link's maximum bandwidth. We view a node's communication power

as proportional to the available bandwidth. For a node n with s interfaces, we estimate the communication power as

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

In practice, software loop-back interfaces are ignored. To compute per-process communication powers for processes $j, j = 1, 2, \ldots, k_n$, on node n, 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 ABW values computed as the sum of their immediate children's values.

4 Computational results

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) [11]. After 17 adaptive refinement steps, the mesh has 524,500 nodes. We use a Sun cluster at Williams College consisting of nodes with "fast" (450MHz Sparc UltraII) and "slow" (300/333MHz Sparc UltraII) processors (Figure 6). Fast processors are either in four-way or two-way SMP nodes. Slow processors are in

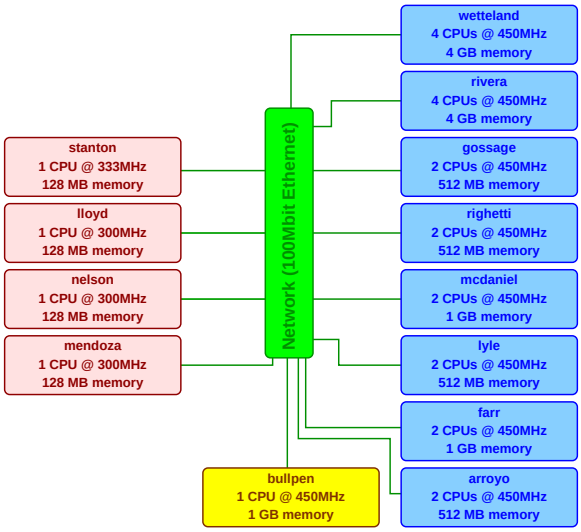

Figure 6: The "Bullpen Cluster" at Williams College.

uniprocessor nodes. All nodes are connected by fast (100 Mbit) Ethernet. Benchmark runs

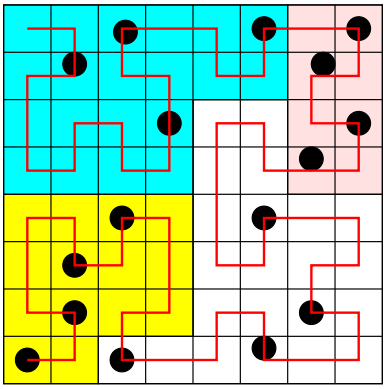

Figure 7: SFC partitioning example. Dots (objects) are ordered along the SFC. Partitions are indicated by shading.

indicated that the fast processors have a computation rate of approximately 1.5 times faster than the slow processors. Given an equal distribution of work, the fast processors would be idle one third of the time.

We use the 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 [16]. In SFC partitioning [13, 25], an object's coordinates are converted to a SFC key representing the object's position along a SFC through the physical domain. Sorting the keys gives a linear ordering of the objects (Figure 7). This ordering is cut into appropriately weighted pieces that are assigned to processors. Zoltan method HSFC [6] 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.

4.1 Experiment 1: Heterogeneous cluster with one process per node

The first set of experiments, which appear, in part, in [23], uses combinations of fast and slow processors with one application process being run on each node. Figure 8 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 9, we compare

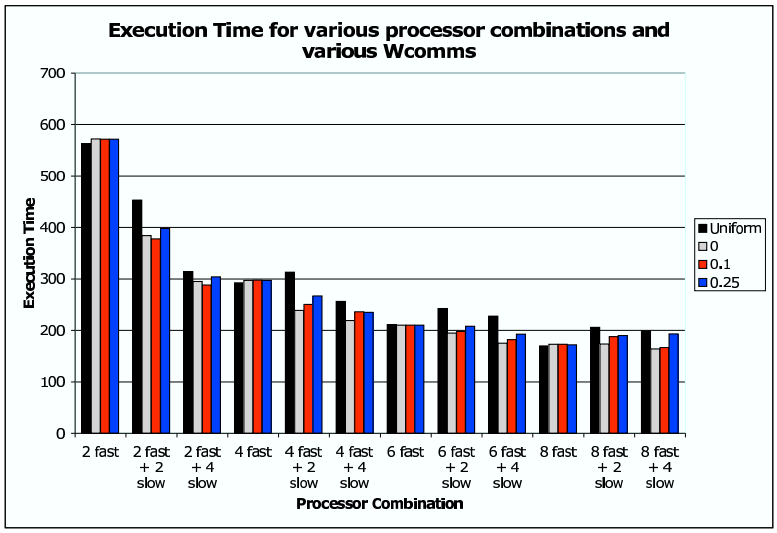

Figure 8: Execution times for PHAML runs when DRUM is used on different combinations of fast and slow processors, with uniform partition sizes and resource-aware partition sizes (with various values for w_{comm}). Here, only one application process is run on each node.

the execution time Relative Change (RC) achieved to an "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,

$$RC_{ideal} = 1 - \frac{n}{\sum_{i=0}^{n} h_i}$$

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.

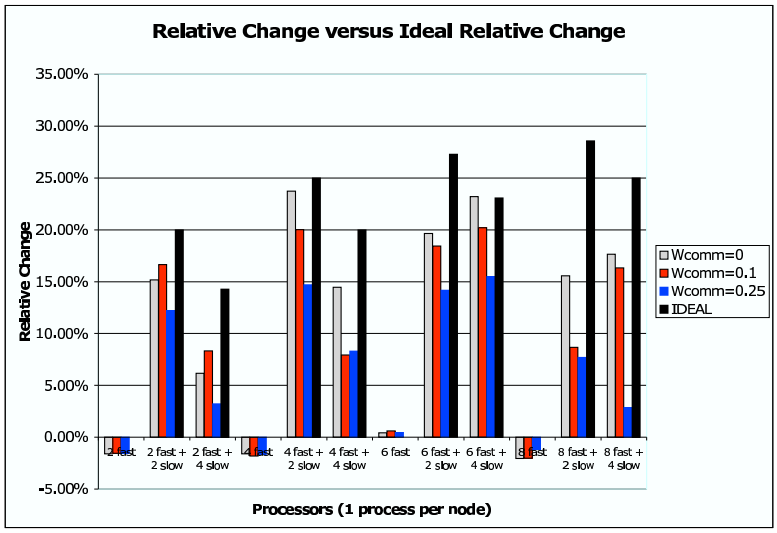

Figure 9: Relative change in CPU times for PHAML runs when DRUM is used on different combinations of fast and slow processors, contrasted with the ideal relative change. Only one application process is running on each node.

4.2 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 10. The combination of processes, processors and nodes are indicated as:

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

In particular, these results confirm the low overhead of DRUM monitors in the cases of 8 and 16 processor runs, 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 11 shows the best relative change values for each combination of processors and contrasts them with the ideal relative change.

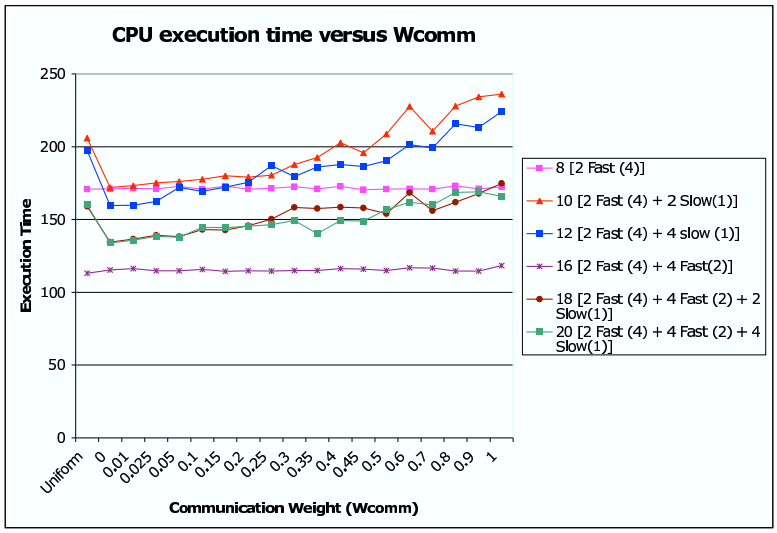

Figure 10: Execution times for PHAML runs when DRUM is used on different combinations of fast and slow processors and with different values for w_n^{comm}.

4.3 Experiment 3: Correlation with Degree of Heterogeneity

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 improvement should be tied to the degree of heterogeneity of the system.

Xiao, *et al.*, propose metrics for CPU and memory heterogeneity defined as the standard deviation of computing powers and memory capacities among the computation nodes [27]. In particular, they define system CPU heterogeneity for a system with P processors as

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

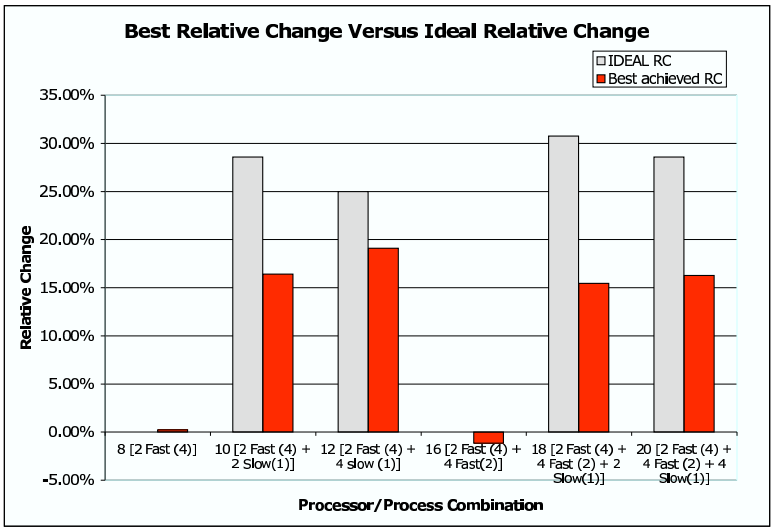

Figure 11: Ideal and best observed relative changes across all values of w_n^{comm} for the timings shown in Figure 10.

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.

Figure 12 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.

4.4 Experiment 4: Non-Dedicated usage of the cluster

The most significant benefits of DRUM come from the fact thatr it accounts for both static information through the benchmark data and dynamic performance using monitoring agents.

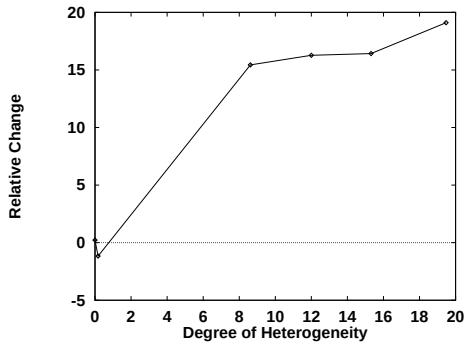

Figure 12: Relative Change in execution time as a function of the degree of heterogeneity.

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 [23]. 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 highest-rank node in the computation. 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 only be detected at run time.

5 Discussion

Our preliminary results show a clear benefit to resource-aware load balancing. We are currently testing the procedures on a wider variety of heterogeneous systems and for different applications. Additional examples demonstrating DRUM's dynamic capabilities are being devised. The real advantages of the communication power can only be seen when network resources are non-uniform or part of the network is heavily loaded. We also plan to allow DRUM to trigger an application's load balancing phase when its monitors detect an imbal-

ance, even in cases where the application has not encountered an adaptive step that would normally trigger load balancing.

We have implemented hierarchical balancing procedures that interact with DRUM to tailor partitions to a given network topology [22]. In addition to the ability to produce weighted partitions, this strategy allows different load-balancing algorithms to be used, as appropriate, in different parts of the network hierarchy [19, 22]. Future plans include a tighter integration of DRUM with hierarchical balancing to allow partitions that can take advantage of hierarchical partitions to account for network hierarchy and DRUM-guided partition sizes to account for heterogeneity and non-dedicated network and processor usage.

We are currently integrating tools for data noise filtering and forecasting. This would allow DRUM to produce better estimates for resource utilization and would also permit an implementation of an adaptive probing procedure in the monitors.

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.

Currently, DRUM requires a description of the computing environment, which may be generated by DrumHead, and uses its own monitoring tools to gather benchmarks and run-time performance statistics. In grid environments and other situations this information may be available from other tools (*e.g.*, Globus Monitoring and Discovery Service (MDS) [4], the Network Weather Service (NWS) [26], Ganglia [15]). We plan to interface DRUM with these tools. We have implemented an interface to NWS [21] and intend to test this interface further and compare its performance with DRUM's native monitoring system. While we will not require NWS or any other external packages to make use of DRUM, we would like to be able to make use of information that these packages can provide when they are available.

Acknowledgments

Faik, Flaherty, Gervasio, and Teresco were supported in part by Contract 15162 with Sandia National Laboratories, a multi-program laboratory operated 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. The authors would like to thank William Mitchell for his help with the PHAML software used for many of the computational results. Williams College undergraduates Laura Effinger-Dean and Arjun Sharma worked on DRUM as part of the Williams College Summer Science Research program. Effinger-Dean implemented the DRUM interface to NWS. Sharma redesigned and implemented an improved version of DrumHead. Erik Boman and Bruce Hendrickson at Sandia National Laboratories also contributed to the design of DRUM.

References

[1] S. Adjerid, J. Flaherty, J. Hudson, and M. Shephard. Modeling and the adaptive solution of CVD fiber-coating processes. *Comput. Methods Appl. Mech. Engrg.*, 172:293–308, 1999.

[2] T. Caraco, S. Glavanakov, G. Chen, J. Flaherty, T. Ohsumi, and B. Szymanski. A diffusion model for vector-borne infection: Lyme disease. *American Naturalist*, 160:348–359, 2002.

[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] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, 2001.

[5] 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. Technical

Report Technical Report CS-04-02, Williams College Department of Computer Science, 2004. To appear, *Appl. Numer. Math.*

[6] 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`.

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

[8] 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.

[9] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *Proc. 7th IEEE Symp. on High-Performance Distributed Computing*, 1998.

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

[11] 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.

[12] T. Ohsumi, J. Flaherty, V. Barocas, S. Adjerid, and M. Aiffa. Adaptive finite element analysis of the anisotropic biphasic theory of tissue-equivalent mechanics. *Computer Methods in Biomechanics and Biomechanical Engineering*, 3:215 – 229, 2000.

[13] 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.

[14] 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.

[15] F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler. Wide area cluster monitoring with Ganglia. In *Proc. IEEE Cluster 2003*, Hong Kong, 2003.

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

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

[18] J. Strauss, D. Batabi, and F. Kaashoek. A measurement study of available bandwidth estimation tools. In *Proc. 3rd ACM SIGCOMM conference on Internet measurement*, pages 39–44. ACM Press, 2003.

[19] 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.

[20] 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.

[21] J. D. Teresco, L. Effinger-Dean, and A. Sharma. Resource-aware parallel adaptive computation for clusters. Technical Report CS-04-13, Williams College Department of Computer Science, 2005. Submitted to *ICCS '05*, Workshop on High Performance Computing in Academia: Systems and Applications.

[22] 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. Submitted to Proc. PARA '04.

[23] J. D. Teresco, J. Faik, and J. E. Flaherty. Resource-aware scientific computation on a heterogeneous cluster. Technical Report CS-04-10, Williams College Department of Computer Science, 2005. To appear, *Computing in Science & Engineering*.

[24] 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).

[25] 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.

[26] 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.

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