An IP-level network monitor and scheduling system for clusters

Daniel Andresen, Sadanand Kota, Madhu Tera and Timothy Bower
Department of Computing and Information Sciences
234 Nichols Hall, Kansas State University
Manhattan, KS 66506
{dan, sada, mst9696, tim}@cis.ksu.edu
Office: (785)532-6350, Fax: (785)532-7353

Abstract

Current systems for managing workload on clusters of workstations, particularly those available for Linux-based (Beowulf) clusters, are typically based on traditional process-based, coarse-grained parallel and distributed programming. The DESPOT project is building a sophisticated thread-level resource-monitoring system for computational, storage and network resources based on SGI's Performance Co-Pilot (PCP). In this paper we present our architecture for low-overhead, fine-grained resource-monitoring tools for network communication, and present our open-source graphical interfaces to this data. We also present our scheduling system utilizing this data, with experimental results indicating the overhead of our system is minimal while providing significantly better performance than several popular scheduling techniques.

1 Introduction

Current systems for managing workload on clusters of workstations, particularly those available for Linux-based (Beowulf) clusters, are typically based on traditional process-based, coarse-grained parallel and distributed programming [11, 3]. In addition, most systems do not address the need for dynamic process migration based on differing phases of computation. The DESPOT project is building a sophisticated, thread-level resource-monitoring system for computational, storage, and network resources. We plan to use this information in an intelligent scheduling system to perform adaptive process/thread migration within the cluster. At the National Computational Grid level, we will integrate our system with meta-computing systems such as GLOBUS via a link to the Network Weather Service (NWS) in order to explore new scheduling heuristics based on the more detailed information provided by DESPOT.

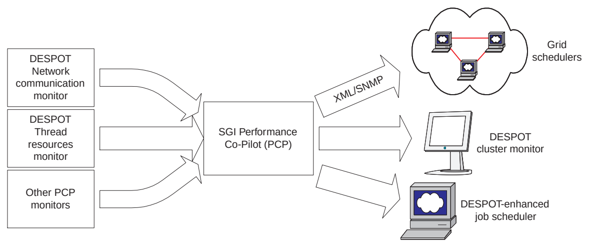
DESPOT Network communication monitor
DESPOT Thread resources monitor
Other PCP monitors
SGI Performance Co-Pilot (PCP)
XML/SNMP
Grid schedulers
DESPOT cluster monitor
DESPOT-enhanced job scheduler

Figure 1. DESPOT architecture.

Current systems for monitoring resource usage within a Beowulf cluster usually either are message-passing-system dependent, such as LAM for MPI, or, like the SMILE system, cannot provide process-to-process communication details [13, 7]. In this paper, we introduce our Beowulf-dependent architecture for monitoring distributed process-to-process communication in a language- and messaging system-independent manner, rather than just those developed to work in a particular programming framework. We indicate how we can monitor communication at the thread/LWP level, including over bonded links [1]. We introduce an open-source graphical interface to this information (see Figure 6), and our initial scheduling algorithm with experimental results.

Section 2 discusses related work. Section 3 covers the design of our system. Section 4 describes our first scheduling algorithm. Section 5 offers our initial experimental results, and finally, Section 6 offers our conclusions and final thoughts.

2 Background

We first briefly discuss monitoring and scheduling tools for Beowulf clusters such as PCP, and go on to the MOSIX

```
hinv.ncpu PMID: 60.0.32 [number of CPUs in the system]
    value 2
kernel.percpu.cpu.user PMID: 60.0.0 [percpu user CPU time metric from
/proc/stat]
    inst [0 or "cpu0"] value 18178010
mem.util.used PMID: 60.1.1 [used memory metric from /proc/meminfo]
    value 237211648
mem.util.free PMID: 60.1.2 [free memory metric from /proc/meminfo]
    value 26615808
network.interface.total.packets PMID: 60.3.17 [network total (in+out) packets
from /proc/net/dev per network interface]
    inst [0 or "lo"] value 22988
    inst [1 or "eth0"] value 1974784
    inst [2 or "eth1"] value 0
proc.psinfo.ppid PMID: 60.8.3 [parent process identifier]
    inst [1 or "000001 init [5]"] value 0
...
    inst [404 or "000404 (lockd)"] value 1
...
```

Figure 2. PCP sample output.

environment [3, 11, 8].

Performance monitoring tools Several commercial and open-source tools have been developed to monitor the performance of a large number of computers such as a typical computing cluster. In contrast with existing systems, which usually display information only graphically, the DESPOT project integrates performance monitoring with scheduling systems. In the following sections, we discuss open-source cluster-monitoring tools.

Several tools have been developed to monitor a large number of machines as stand-alone hosts as well as hosts in a cluster. These tools can be useful because they monitor the availability of services on a host and detect if a host is overloaded, but they do not generally provide performance-monitoring information at the level of detail needed to tune the performance of a Beowulf cluster. Examples of these systems are PaRe Procps [12], BWatch [10], Mon [15], Nocol [9], and Netsaint [6].

Detailed tracing of message-passing events in a cluster is afforded with the *Conch Visualization Package* from Georgia Tech [14]. This tool can be used to perform detailed traces of the execution of a distributed program, but the data reported is more relevant to tracing and debugging than measuring performance.

The *SMILE Cluster Management System* (SCMS) is an extensible management tool for Beowulf clusters [7]. SCMS provides a set of tools that help users monitor, submit commands, query system status, maintain system configuration, and more. System monitoring is limited to heartbeat-type measurements.

The Network Weather Service, although not targeted at Beowulf clusters, is a distributed system that periodically monitors and dynamically forecasts the performance various network and computational resources can deliver over a given time interval [16, 17]. The service operates a distributed set of performance sensors (network monitors, CPU monitors, etc.) from which it gathers system condition information. It then uses numerical models to generate forecasts of what the conditions will be for a given time frame. NWS is used for various meta-computing systems such as Globus and APPLeS [5, 4].

MOSIX is a popular platform for supporting distributed computing. It enhances the Linux kernel with cluster computing capabilities. In a MOSIX cluster, there is no need to modify applications to run in the cluster, or to link applications with any library, or even to assign processes to different nodes. MOSIX does it automatically and transparently. The resource sharing algorithms of MOSIX attempt to equalize the processing load of the machines in the cluster. However, the scheduling algorithms only consider the total CPU load and memory usage of each machine. Per process load and network load measurements are not considered [8].

MOSIX was useful for our experiments for two reasons. First of all, it provides a framework and an API for migrating processes between machines. Thus it is a convenient platform for the development of prototype scheduling algorithms. Secondly, the built-in MOSIX scheduling algorithm offers a baseline measuring stick for comparing our own scheduling algorithms.

PCP The SGI Performance Co-Pilot (PCP) provides a systems-level suite of tools that cooperate to deliver distributed, integrated performance management services. PCP provides the ability to quickly isolate and understand performance behavior, resource utilization, activity levels and performance bottlenecks. Performance data may be collected and exported from multiple sources, most notably the hardware platform, the IRIX kernel, layered services, and end-user applications, and is returned in a structured text format (for example, Figure 2).

There are several ways to extend the PCP by programming certain of its components: by writing a Performance Metrics Domain Agent (PMDA) to collect performance metrics from an uncharted performance domain, or by creating new analysis or visualization tools using documented routines from the Performance Metrics Application Programming Interface (PMAPI).

Collection tools (called PMDAs) extract performance values from target systems but do not provide graphical user interfaces. Systems supporting PCP services are broadly classified into two categories: *Collector* hosts have the Performance Metrics Collection Daemon (PMCD) and one or more PMDAs running to collect and export performance metrics. *Monitor* hosts import performance metrics from one or more collector hosts to be consumed by tools to monitor, manage, or record the performance of the collector hosts. Each PCP-enabled host can operate as a collector, or a monitor, or both. The monitoring tools consume and process performance data using a public interface, the Performance Metrics Application Programming Interface (PMAPI). Below the PMAPI level is the pmcd process, which acts in a coordinating role, accepting requests from clients, routing requests to one or more PMDAs, aggregating responses from the PMDAs, and responding to the requesting client. Each performance metric domain (such as IRIX or some DBMS) has a well-defined name space for referring to the specific performance metrics it knows how to collect. Monitoring tools communicate only with pmcd. The PMDAs are controlled by pmcd and respond to requests from the monitoring tools that are forwarded by pmcd to the relevant PMDAs on the collection host.

Each PMDA provides a domain of metrics, whether they be for IRIX, a database manager, a layered service, or an application module. These metrics are referred to by name inside the user interface, and with a numeric Performance Metric Identifier (PMID) within the underlying PMAPI. The PMID consists of three fields: the domain, the cluster, and the item number of the metric. The domain is a unique number assigned to each PMDA. For example, two metrics with the same domain number must be from the same PMDA. The cluster and item numbers allow metrics to be easily organized into groups within the PMDA, and provide a hierarchical taxonomy to guarantee uniqueness within each PMDA.

3 System design

Our project's goal is to take standard Beowulf technology and create a more fine-grained, smarter Beowulf, which actively adapts to varying configurations and loads. A typical Beowulf cluster consists of a number of processing nodes connected via a network fabric, all monitored and scheduled by a dedicated management node acting as a bridge to the outside network. Many Beowulf configurations are heterogeneous, with nodes differing in number of processors, processor speeds, amount of memory, and I/O systems (disk, network). Many computing environments expect a symmetric cluster configuration, which can lead to a mismatch between performance and expectations. DESPOT makes no assumptions regarding the (non-)homogeneity of its systems.

We extend SGI's PCP (performance co-pilot) as an interface to allow clients to retrieve and process performance-related information easily. By use of dynamically linked modules, we can provide traditional Unix statistics (netstat, ipcs), Linux-related information (information provided by the /proc file system), and new, non-traditional clustering information through a single, well-defined, consistent interface.

Although we will mainly be using PCP for its plugin framework to provide an improved scheduling system, it has the added benefit of giving an abundance of performance information that may prove useful in the everyday debugging of a high-performance cluster (such as SCSI/raid, NFS, interrupt, and daemon statistics) with interesting features such as logging, event recording, and playback. Sample output from PCP is shown in Figure 2. PCP and the text-based user interface are available under the Gnu General Public Licenses (GPL). A graphical viewer is provided by SGI's commercial ACE product.

Monitoring threads The ability to do thread-level monitoring of computation and communication is an integral part of DESPOT, including both UNIX and Java threads. Under Linux, Java native threads are separate lightweight processes, allowing them to be monitored. DESPOT requires the use of Java native threads if per-thread communication/processor monitoring is to be used. The use of traditional, or "green," threads, causes all threads to be multiplexed onto a single process, so all behavior will be aggregated at that level. Use of Java native threads allows us to treat Java threads as standard Linux threads and provide this important capability.

Monitoring communication Through integrating PCP's *netstat* module across all nodes in a cluster, we can create a matrix outlining process-to-process connections. However, this does not indicate the volume of data being passed or shared between processes. Thus, we actually have three PCP modules to perform the communication monitoring duties: *Pnet*, *Pcap*, and *PPair*. *Pnet* uses PCP's netstat module to provide information on sockets from individual nodes. *Pcap* sits on top of the Linux networking code to monitor packet traffic, and *PPair*, running on a single node on the system, acts as the central clearinghouse to correlate the *Pnet* and *Pcap* data across the cluster. This allows us to retrieve information about the entire cluster from a single machine and reduces the load on individual nodes. *Pcap* and *PPair* are discussed in more detail below.

Since connections are monitored at the socket/packet level, some connections are unable to be monitored completely. This would include communication through shared memory segments; or sockets to machines outside the cluster, in which case statistics are not known for the remote process.

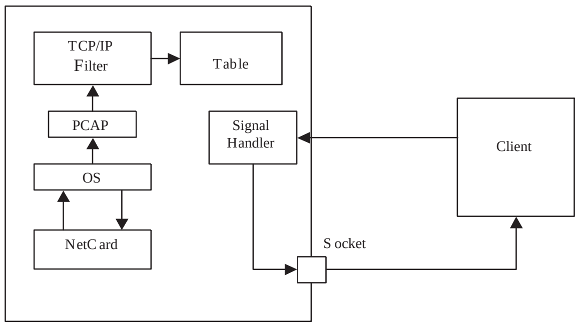

Figure 3. Traffic capture module structure.

It is not possible to monitor the amount of intra-node communication in generic Beowulf systems (i.e., those not running DSM or other distributed IPC systems) without creating several headaches. First, a large amount of kernel hacking would be required, including changes to sensitive areas of code – particularly the networking and shared-memory sections. Second, by adding monitoring code, we would end up significantly impacting performance by adding latency to transmission times. And finally, this performance-monitoring kernel code would have to be maintained as the kernel is revised.

We could solve this intra-node communication problem by implementing a layer above the standard system library calls that records relevant information. This has the same pitfalls as a kernel modification, but has the possibility of missing data through the use of static linking or nonstandard libraries in the observed applications.

The **Pcap** module is responsible for collecting information about the amount of traffic between each node in the cluster, and consists of two processes: a statistics process

```
129.130.10.139:664 260 30
128.169.93.174:8050 384 32
129.130.10.139:1044 6240 112
129.130.10.73:1023 602668 23618
129.130.10.140:22 500 29
129.130.10.139:22 7071221 33864
```

Figure 4. Pcap information buffer.

which gathers information from the network interface cards in promiscuous mode, and an agent process listening and honoring requests for these statistics The statistics process collects packet counts as well as total byte counts. Thus, we can also distinguish data transfer connections from interactive connections based on the ratio of bytes to packets (a low ratio probably indicates an interactive session). The module retains and reports only recent traffic. Longterm trends should be recorded and analyzed with another module if needed.

The implementation is largely based on the PCAP library. *Tcpdump* is a front-end pretty-printer of the information which libpcap gathers. The statistics process re-uses the tcpdump code, replacing pretty-printing with statistics-gathering code. Time stamps, byte counts, and packet counts are recorded in a hash table, which can then be queried through the second process. The PCAP library works through inserting an IP filter within the kernel to promiscuously sort through all packets received by a node.

To avoid modifying libpcap, we had to use a two-process system (Figure 3). Currently, libpcap has its own event loop with no hooks to register new events for which to trigger call-backs (such as IO on a file descriptor) except the traditional signal API. Thus, if queries from outside the machine are to be honored, a separate process is needed to listen for such queries, signal the statistics process, and relay the info to the network. This agent process can be implemented to honor INET socket-based requests, PCP requests, or requests of any other type.

PPair - information integration *PPair* uses a binary tree whose node contains the local address with port number and the foreign address with its port number. Using RPC, we acquire a snapshot of the netstat data on all the machines in the existing Beowulf cluster, and we dynamically create a binary tree using the information.

Pcap gathers information about bandwidth usage by various processes, which is expressed in terms of packets per seconds and bits per second. For example, from the first line in Figure 4, 129.130.10.139:664 indicates PID of the process, which is using 260 bits per second of bandwidth, and sending 30 is packets per second. So the complete snapshot gives information about six processes' recent network usage. This can then be combined with the information from Pnet to associate communicating processes across the cluster. PPair exports this data after aggregation with the other nodes in XML format (Figure 5) for use with scheduling systems and graphical monitoring tools.

```
<?xml version="1.0"?>
<address>
<set>
<address1>129.130.10.140:1787</address1>
<ad dress2>129.130.10.139:22  </address2>
<bps>2345</bps>
<pps>12</pps>
</set>
<set>
<address1>129.130.10.140:22  </address1>
<address2>129.130.10.139:625 </address2>
<bps>5643</bps>
<pps>53</pps>
</set>
<set>
<address1>129.130.10.139:22  </address1>
<addres   s2>129.130.10.140:1787</address2>
<bps>6453</bps>
<pps>18</pps>
</set>
<set>
<address1>129.130.10.139:625 </address1>
<address2>129.130.10.140:22  </address2>
<bps>5483</bps>
<pps>37</pps>
</set>
</address>
```

Figure 5. PPair XML output.

Graphical display The intended consumer for the detailed information being produced by our PCP modules is advanced scheduling algorithms. However, it is also useful for system administrators. We have developed a graphical display in Java to show this information in a per-node and aggregate form (Figure 6). The display shows the per-process communication, as well as aggregate resource usage for memory, disk, number of processes, and network traffic. It also shows the same information on a per-node basis. We plan to make this available under the GPL as an open-source alternative to SGI's ACE product, which is typically available only with the purchase of SGI systems. The application uses JNI to connect with the PCP system and acquire the data to be displayed. Data from PPair is in XML format, which is parsed, aggregated, and shown.

4 Process Scheduling

Our initial experiments used PCP to collect and archive system performance data and used the MOSIX environment to support process migration. We present a scheduling technique which balances the CPU load, network traffic and memory usage based on measurements collected for each process as well total system load measurements.

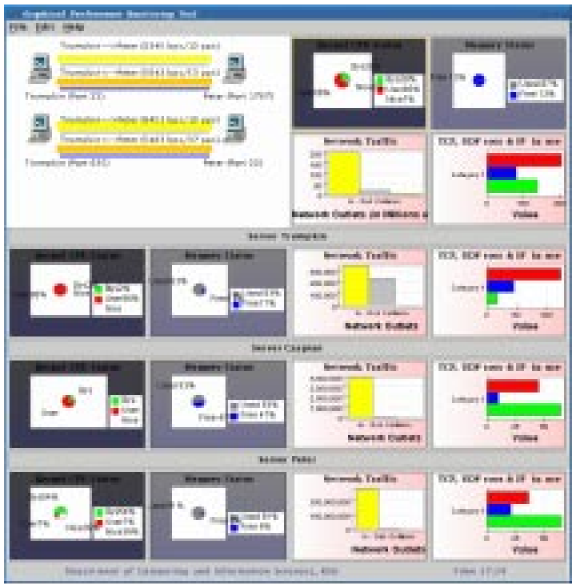

Figure 6. Process memory/network usage display.

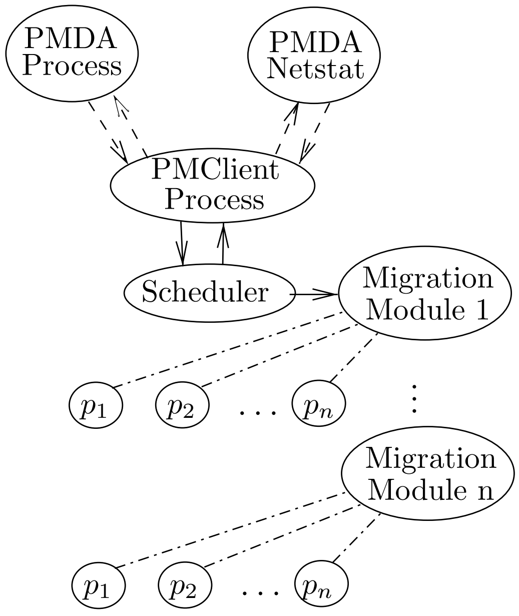

Figure 7. Centralized scheduler associated with PMDA and Migration Modules.

The scheduling technique, that we have implemented, assigns a 'happiness' value to each process and sums these happiness values to come up with a total happiness for the cluster. A process is considered to be happiest if it receives the total amount of CPU, memory and network bandwidth that it requires. We estimate the desired resource usage based on usage from the preceding time interval. The algorithm attempts to maximize the total happiness. The premise of this algorithm is similar to related algorithms used by our laboratory for clustered web servers [2].

Happiness (H_{ij}) for a particular process (i) on machine (j) is

$$H_{ij} = \alpha H_{CPU_{i,j}} + \beta H_{NET_{i,j}}$$

where α and β are typically both 0.5, meaning that the weighting between network and CPU is equal. $\alpha = 1.0$ and $\beta = 0$ is equivalent to a load-balancing algorithm such as that used by MOSIX. We define

$$H_{CPU_{i,j}} = \frac{CPU_{avail}}{CPUdesired_{i,j}}$$

If the process is currently using an amount of CPU proportional to the number of other processes on the same processor, we assume $CPU_{desired}$ is equal to an entire CPU. If the process is using *less* than its "fair share", then we assume its CPU needs are being adequately met. We apply a similar logic for $NET_{desired}$, additionally assuming that communicating processes on the same node have adequate bandwidth.

The system begins with a round robin assignment of processes to machines in the cluster and then dynamically shift processes between nodes. For each machines in the cluster, we fetch the process metrics for all cluster processes and find the available CPU and network bandwidth. Then we find total happiness (H) for the cluster as a sum of all the H_{ij} values calculated from the above equation.

Next we calculate predicted happiness for other process to node assignments. Predicted happiness is reduced by 10% if process migration is required for the cost of performing the migration. Once it is determined that a process migration will result in improved overall happiness, then the MOSIX process migration methods are called.

To reduce the algorithm's exponential complexity, we use the following heuristics. First, to minimize memory swapping, we ignore any combination which requires more memory for the processes on a machine than the machine's available RAM. Secondly, we do not investigate all the combinations before migrating a processes. We stop our search once a configuration is found which improves the overall happiness of the cluster by a tunable factor (currently 20%). These have served to keep scheduling times within a reasonable period, and we are investigating further optimizations.

5 Experimental results

The small cluster used to prototype the scheduling algorithm consisted of two dual processor 300MHz Pentium 2 systems with 256MB RAM and two 600MHz Intel Celeron systems with 256MB RAM. The machines were connected by 100Mbps switched Ethernet. The software used was Red Hat Linux 6.2 with kernel 2.2.19, MOSIX 0.98 and PCP 2.2.1.

Our algorithm performed better than MOSIX by an amount of 20-30% for computationally intensive process with bursts of network communication over two machines, and averaged approximately 15% improvement for four machines (six processors). Table 1 lists results for our four-node prototype system. Our scheduler is equivalent to the MOSIX and round robin scheduler for homogenous CPU intensive programs. We anticipate our scheduler will outperform round robin by a substantially higher margin with heterogenous processes.

6 Conclusions and future work

In this paper we have presented our system for monitoring communication within a Beowulf cluster at the process-to-process level, and shown how the system can also be used to monitor thread-level communication as well. Other monitoring systems for Beowulf clusters are either methodology-specific (such as LAM), or present only aggregated communication results. We have also given experimental results indicating that the system has the rapid response time and low overhead to be useful in our application domain. Finally, we have also developed an open-source graphical front-end to PCP that displays the enhanced level of detail.

We plan to extend the system and viewing application to a hierarchical organization to increase its scalability over the current, centralized system. We also plan to increase the flexibility of the graphical monitor to allow users to click on various computer pairs and get a detailed analysis of the communication patterns. Another planned enhancement is the inclusion of a history mechanism.

Test Program Type	DESPOT μ (sec)	DESPOT σ^2	MOSIX μ (sec)	MOSIX σ^2	MOSIX %Δ	Round Robin μ (sec)	Round Robin σ^2	Round Robin %Δ
All CPU bound 4 processes	266	2.0	259	0.316	-2.7	same as MOSIX		
All CPU bound 8 processes	511.1	4.0	502.9	1.6	-1.5	same as MOSIX		
Network bound 5 proc., 60 1kB mesg.	308	0.3	349	0.3	+11.7			
Network bound 5 proc., 75 1kB mesg.	389.8	0.2	460.6	0.7	+15.4			
Combination CPU–Net 70% CPU, 30% Net	708	5.0	855.5	1.3	+17.2	790.9	1.52	+10.5

Table 1. Scheduling algorithm results. Shown are mean, std. dev., and percent change relative to DESPOT.

References

[1] D. Andresen and Z. Baosong. Heterogeneous channel bonding on a Beowulf cluster. In *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, pages 2479–2484, Las Vegas, June 2000.

[2] D. Andresen, T. Yang, O. Ibarra, and O. Egecioglu. Adaptive partitioning and scheduling for enhancing www application performance. *Journal of Parallel and Distributed Computing (JPDC)*, 49(1):57–85, February 1998.

[3] D. Becker. *The Beowulf project*, July 2000. http://www.beowulf.org.

[4] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing'96*, Pittsburgh, PA, November 1996.

[5] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[6] E. Galstad. *Netsaint Network Monitor*. http://www.netsaint.org/.

[7] P. R. Group. *SCMS Web Page*. Kasetsart University. http://smile.cpe.ku.ac.th/.

[8] *The MOSIX Project Homepage*. http://www.mosix.cs.huji.ac.il/.

[9] Netplex Technologies Inc. *Nocol System Monitoring Tool*. http://www.netplex-tech.com/software/nocol.

[10] J. Radajewski. *bWatch - Beowulf Monitoring System*. University of Southern Queensland, Apr. 1999. http://www.sci.usq.edu.au/staff/jacek/bWatch/.

[11] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, Aug. 1995.

[12] F. Strauss and O. Wellnitz. *Procps Monitoring Tools*. Technical University Braunschweig, June 1998. http://www.sc.cs.tu-bs.de/pare/results/procps.html.

[13] L. Team. *LAM Overview*. University of Notre Dame. http://www.mpi.nd.edu/lam/overview.php3.

[14] B. Topol. *Conch Visualization Package*. Graphics, Visualization and Usability Center; Georgia Institute of Technology. http://www.cc.gatech.edu/gvu/people/ Undergrad/Brad.Topol/conchviz.html.

[15] J. Trocki. *Mon System Monitoring Tool*. Transmeta Corporation. http://www.kernel.org/software/mon/.

[16] University of California San Diego. *The Network Weather Service Homepage*. http://nws.npaci.edu/NWS.

[17] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, April 1998.