Performance Evaluation of Load Sharing Policies with PANTS on a Beowulf Cluster

James Nichols and Mark Claypool
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609 USA
{jnick | claypool}@cs.wpi.edu

Abstract.

Powerful, low-cost clusters of personal computers, such as Beowulf clusters, have fueled the potential for widespread distributed computation. While these Beowulf clusters typically have software that facilitates development of distributed applications, there is still a need for effective distributed computation that is transparent to the application programmer. The PANTS Application Node Transparency System (PANTS), enables transparent distributed computation. The system employs a fault-tolerant communication architecture based on multicast communication that minimizes load on busy cluster nodes. PANTS runs on any modern distribution of Linux without requiring any kernel modifications. The initial design and implementation of the load balancing algorithm used only a measurement of CPU usage to make distribution decisions. While CPU usage is the typical metric used in load distribution, other system resources such as disk and memory can become loaded and be a performance bottleneck. In this work, we examine PANTS in the context of load distribution algorithms, build new load indices, develop benchmarks, and evaluate performance. We find our load indices can reduce workload running times by 1/2 of the time of the original PANTS indices.

1. Introduction

Cluster computing offers several benefits to application programmers and to users. A large class of computations can be broken into smaller pieces and executed by the various nodes in a cluster. However, sometimes on a cluster it can be beneficial to run an application that was not designed to be cluster-aware. A main goal of our research is to support such applications.

We have developed a system called PANTS[1] for distributed applications running in a cluster environment. PANTS automatically detects which nodes of the cluster are overloaded and which are underloaded, and transfers work from overloaded nodes to underloaded nodes so that the overall application can complete more quickly. PANTS runs on a Beowulf cluster, which is a cluster of PC-class machines running a free/open-source operating system, in our case Linux, and connected by a standard local area network. Two fundamental design goals of PANTS are that its operation

[1] PANTS stands for *PANTS Application Node Transparency System*.

should be transparent to the programmer and to the user of the distributed application, and that PANTS should impose only minimal overhead on the system.

PANTS is designed to be transparent to the application as well as the programmer. This transparency allows an increased range of applications to benefit from process migration. Under PANTS, existing multi-process applications that are not built with support for distributed computation can run on multiple nodes by starting all the processes of the application on a single node and allowing PANTS to migrate the individual processes of the application to other nodes. As far as the application is concerned, it is running on a single computer, while PANTS controls what cluster computation resources it is using.

The PANTS design provides a method for minimal inter-node communication and for fault tolerance. In a Beowulf system, communication over the network can often be the performance bottleneck. With this in mind, PANTS keeps the number of messages sent between machines low and also uses a protocol that does not exchange messages with nodes that are busy with computations. Built-in fault tolerance allows the cluster to continue functioning even in the event that a node fails. In the same way, nodes can be added or removed from a cluster without having to restart PANTS.

Early research results showed PANTS provided a near linear speedup for computationally intensive applications [DHV00, CF01]. Unfortunately, programs that impart load on the CPU of a machine are not the only applications that are desirable to run on Beowulf clusters. An application could read or write many files to disk imparting load on the I/O subsystem, maintain large data structures loading memory, or cause system events such as interrupts and context switches. For example, compiling the Linux kernel is a typical example of an application that requires some CPU resources, but does not significantly load the CPU while imparting heavy load on the I/O resources of a system. Thus, load measurement can also be based on I/O in terms of number of blocks read and written to the disks, memory load in terms of total pages read and written per second, context switches per second, or interrupts per second. Load can also be measured as a mix of all of these criteria. When just looking at CPU usage as the only measurement of load, a system compiling the kernel would not “load" the system, while there would be a performance benefit if the load measurements included I/O use.

In this work we examine PANTS in the context of load distribution algorithms, to devise new methods of capturing load metrics and also to measure the performance of new metrics and policies we have devised. We design and implement new ways to measure load in PANTS including I/O usage, memory usage, context switches, and interrupts, as well as improving the way CPU usage is measured over past implementations. To test and verify correctness of the new measures of load we built a micro benchmark for each new load metric and used a real-world application as a benchmark: a distributed compilation of the Linux kernel.

2. PANTS

2.1 PANTS Load Distribution Algorithm

An important aspect of any process migration scheme is how to decide when to transfer a process from a heavily loaded node to a lightly loaded node. Many load-distribution algorithms for distributed systems have been proposed; there is a summary in [FW95]. We have implemented a variation of the multi-leader load-balancing algorithm proposed in [FW95].

In this algorithm, one of the nodes is required to be the leader. The leader can be any node in the cluster and is chosen randomly from among all of the nodes through a voting procedure. The leader has three basic responsibilities: accept information from lightly-loaded ("available") nodes in the cluster, use that information to maintain a list of available nodes, and return an available node to any client that requests it. The algorithm is fault tolerant; it utilizes the election procedure to select a new leader in case the leader node fails.

An available node is one that is lightly loaded, as measured by some load metric, such as CPU utilization. When any node in the cluster becomes available, it sends a message to the leader, indicating that it is free to accept new work. If a node becomes unavailable, for example, if it begins a new computation, it sends another message to the leader. Thus, the leader always knows which nodes are available at any time. If a node wants to off-load work onto another node, it need only ask the leader for an available node, then send the process to that node. If one of these non-leader nodes fails it will simply be removed from the list of free nodes that the leader maintains. This makes PANTS fault-tolerant of failure by any nodes in the cluster.

The actual implementation is a variation of the multi-leader policy described in [FW95] and implemented in [DHV00]. In many other load-balancing algorithms, either nodes that are available or nodes that need to off-load work broadcast their status to all the nodes in the cluster. These broadcast messages frequently need to be handled by machines that are heavily loaded ("busy machines"). The multi-leader algorithm avoids these "busy-machine messages" by sending messages only to the leader multicast address.

We modified the policy in [FW95] to simplify the implementation and improve fault tolerance, at the cost of a small increase in the amount of network traffic. In PANTS, there are two multicast addresses. One of the addresses is read only by the leader; both available nodes and nodes with work to off-load send to the leader on this address. Because the leader is contacted via multicast, the leader can be any node in the cluster, and leadership can change at any time, without needing to update the clients. Available nodes receive on the other multicast address when the leader needs to discover, for the first time, which nodes are available. Because multicast is used to communicate with available nodes, busy nodes are not even aware of the traffic.

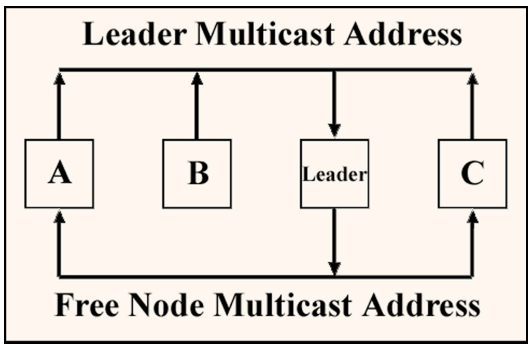

Fig. 1: PANTS Multicast Communications

Figure 1 depicts the multicast communication among PANTS nodes. There are four nodes in this Beowulf cluster, one of which is the leader. Nodes A and C are "free" nodes, being lightly loaded and Node B is a "busy" node. All nodes can communicate with the leader by sending to the leader multicast address. The leader communicates with all free nodes, A and C in this example, by sending to the free node multicast address. Node B is not "bothered" by messages to the free nodes since it is not subscribed to that multicast address.

Previously, PANTS monitored the load at a node by periodically checking the CPU usage through the `/proc` file system. The time that the CPU spent processing user, system, nice, and idle process types in the last 5 seconds are obtained. The user, system, and nice times are summed and divided by the total, obtaining a percentage which is then compared against a static threshold set at compile time.

2.2 PANTS Implementation

PANTS is composed of two major software components, the PANTS daemon and `prex`, which stands for PANTS remote execute. The PANTS daemon is responsible for coordinating the available resources in the cluster. It communicates among nodes to determine which nodes are available to receive processes. `prex` intercepts the execution of a process, queries the PANTS daemon for an available node and remotely executes the process to distribute load among the nodes in the cluster.

Using the multicast policy described in Section 2.3, the PANTS daemon can respond quickly to a request from `prex` for an available node. `prex` is made up of a library object called `libprex.o` and a remote execution program called `prex`. The library object is designed to intercept programs when they are initially executed and then send them to the `prex` program.

The way `libprex` works with the C library allows it to intercept processes transparently. To enable `libprex`, the Linux environment variable LD_PRELOAD is set to reference the `libprex` library object. Doing this causes the library functions in `libprex` to override the usual C library functions. When programs call the

`execve()` function to execute a binary (for example, when a user executes a binary from a command line shell), our version of `execve()` inside `libprex` is used instead of the original one. Inside of our `execve()` function, the real C library `execve()` is invoked to execute `prex`, whose arguments are the process name, followed by the original arguments for that process. `prex` can then use `rsh` to remotely execute the process.

Several user-defined bits within the flag area of the ELF binary's header signal whether the process should be migrated. These bits can be set using a simple command line utility. When `prex` is invoked for a process, the process is checked for migratability, which is determined by flags set within the binary. If the binary is migratable, `prex` queries the local PANTS daemon to determine if the local machine is busy with other computations. If the process is not migratable, or the local machine is not busy, the binary is executed on the local machine by calling the original version of `execve()`. If the local machine is busy, `prex` queries the local PANTS daemon for an available node. If a node is returned, `prex` calls `rsh` to execute the process on that node. If all nodes are busy, the process is executed locally.

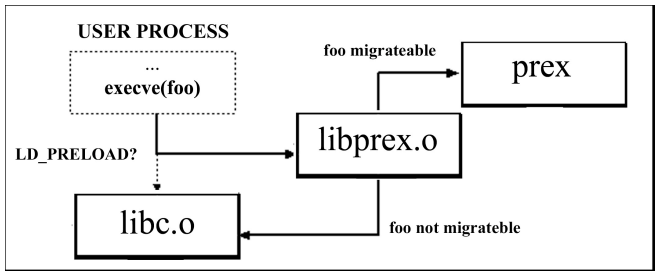

Fig. 2: `prex` Functionality

Figure 2 depicts the functionality of `prex`. When a process calls `execve()` and the environment variable LD_PRELOAD is set, `libprex.o` intercepts the `libc.o` version of `execve()`. If the executable file is migratable, `libprex.o` invokes `prex` to communicate with the PANTS daemon to find a free node and migrate the process. If the executable is not migratable, `libprex.o` invokes the normal `libc.o` version of `execve()`.

Research has also been completed [STW01] which demonstrated that PANTS is compatible with the distributed interprocess communications package DIPC [KK]. The use of DIPC requires the programmer to include DIPC-specific code in the program, and thus compromises our transparency goal of not requiring any special code for the distributed application. However, the use of DIPC does allow the programmer to use a wider range of interprocess communication primitives, such as such as shared memory, semaphores, and message queues, than would otherwise not be available.

2.3 Current Configuration

Our PANTS Beowulf cluster is composed of seven 600MHz Alpha machines. The physical memory ranges from 512MB on the NFS file server to 64MB on several of the node machines, and all machines have at least 128MB of additional disk swap space. Each machine is equipped with a PCI Ultra-Wide SCSI controller and hard drives, and a 100Base-T network card. The network is arranged in a star topology with a switch in the middle, and one machine providing a gateway to the outside world through an additional network card and IP-Masquerading software.

All of the machines in the cluster run RedHat Linux 7.1, the last release to support the Alpha architecture. The Linux kernel is version 2.4.18 and each kernel image is compiled with NFS support. The cluster shares a common `/home` directory which is shared via NFS from the file server. This common file system is necessary to facilitate `rsh` execution of child processes and provides a simple means of sharing data. The NFS server machine is equipped with extra RAM and SCSI hard disks and its kernel image has a larger maximum read/write NFS block size than the default. The read/write block size settings on remote machines are also increased for better performance. To see a listing of all the processes running on each system in the cluster see [JN02], however it is the minimal required for system functionality. For more details of the PANTS implementation and the configuration of our cluster, including the logs of configuration for each node, see [JN02].

3. Methodology and Approach

Since CPU usage is not the only applicable load metric when classifying different types of load we implemented services to measure the load on the system by looking not only at CPU usage, but I/O usage, context switches, interrupts, and memory usage. We then developed micro benchmarks to stress the system and evaluate each metric for verification purposes. Finally, we use a real world application to evaluate the performance of the algorithm.

Table 1: Load Metrics

CPU	(%) jiffies
I/O	blocks/sec
Context switches	switches/sec
Memory	page operations/sec
Interrupts	Interrupts/sec

3.1 Load Metrics

The load metrics include CPU usage, I/O measurement, number of context switches, memory pages read and written, and overall interrupts generated on the system. These numbers are read from `/proc/stat` into a data structure and are then time stamped. In Linux `/proc` is a special file system that provides an interface to read the values of certain kernel variables and data structures. The metrics are summarized in Table

1, for details of how the metrics are implemented see [JN02]. Each metric has a threshold value that is used to determine the availability of the node. If the measurement produced by the metric is over this threshold the node is considered loaded and will return an unavailable status to the leader node. The node will not be listed as available until the loads are under the thresholds, and the leader node is then informed of the node's availability.

3.2 Micro-Benchmark

To verify the correctness of our load metrics and to refine our understanding of the way that PANTS shares load, we use tests with fine control over the CPU usage, interrupts thrown, disk usage, memory usage, and context switching. We create micro benchmarks for this purpose to allow us to test specific actions of PANTS in controlled situations. For any given test, different amounts of different kinds of work can be assigned to the system, and the sharing of load can be studied for tuning and analysis purposes.

The micro benchmarks create a task that consists primarily of one of the types of load we have measured and reports how fast the assigned tasks were completed. The types of load created and measured by the benchmarks are CPU usage, I/O to the disk drive or drives, and memory usage. Because PANTS works by relocating processes before executing them, these benchmarks start a number of processes on one node, and then allow PANTS to remotely execute them. Each benchmark test begins with several minutes of idle time to allow the systems to reach their baseline measurements.

When the benchmarks begin, they write a time stamp to a logfile to mark the start of their execution. The controlling process will then begin spawning a number of identical child processes at fixed intervals. These child processes generate the actual load of the benchmark, and are flagged migratable so they can be passed to a remote node for execution. When all the child processes are done, the logfile is again time stamped to mark the end of execution. We analyze the these log files to collect performance statistics.

The CPU test consists of a parent process that spawns four child processes at five minute intervals. Each of these child processes performs a large number of floating point operations (FLOPs). FLOPs were chosen for this benchmark because they are frequently used for some distributed computing. This benchmark shows system load conditions under a system that is loaded primarily with CPU intensive processes, while ignoring other types of load.

The I/O test is designed to load the disks of a machine while leaving the CPU relatively unloaded. A copy of a large directory structure (384MB including files and directories) is created on the local hard disk of each machine. The parent process then spawns four child processes at five minute intervals. Each child process copies the directory structure to a new location on the local hard disk. Many small files are involved requiring more writes than one large sequential file of the same size; new files and directories must be written as well as the actual data contained in the files.

The memory usage test consists of a parent process which spawns four child processes at five minute intervals. Each of these child processes use the `malloc()` function to allocate 100MB of memory, and calls the `mmap()` function to map an existing section of memory into this allocated area. After the memory is allocated and mapped it is released using the `free()` function. This process is repeated ten times in each child process, creating a rise and fall of virtual memory. None of the systems involved have 100MB of free physical memory, so each machine must page memory in and out of its swap space during each cycle to provide enough virtual memory to fit the allocated structure.

3.3 Application Benchmark

To further evaluate the performance of PANTS we used a real-world application as a benchmark. The application is a distributed compilation of the Linux kernel, executed by the standard Linux program `make`. This distributed compilation exemplifies an application which imparts significant load on the I/O and memory resource with modest load on the CPU.

To make the compilation distributed via PANTS we had to make slight modifications to the compiling process. First, we marked the `gcc` compiler binary as migratable, which allows PREX to remotely execute `gcc` on remote nodes. Second, we modified the standard Makefile included with the Linux source tree to use a script program (`my_gcc`) as the compiler. This script program simply made any file references passed from make into absolute paths. Relative paths to files are not translated properly by `rsh` when sent to remote nodes, as the working directory is where the binary is located on the remote filesystem.

The `my_gcc` script then uses an `execve()` of the real `gcc` giving it the absolute file names it determined as arguments. PREX then intercepts this `execve()` call, checks and sees that `gcc` is migratable, and then hands the process to the PANTS system. We also modified `ld`, the link editor program used by `make`. We modified it to wait until all the files were present on the NFS mount before it attempted to link them. This added a small measure of robustness to the compilation, but in practical use we feel it is unnecessary as a properly equipped and configured NFS server can keep up with the demand placed on it during the compile.

The Linux kernel source tree was located on the NFS mount and all output files were sent to this same location. This made all of the files available to all the nodes. The build was started from a node in the cluster by simply typing `make vmlinux` from the NFS mounted Linux kernel directory. The Linux kernel version was 2.4.18. The kernel build was configured to compile a kernel image identical to those on which the machines in the cluster currently ran. Overall, 432 files were compiled, with the mean source file size being 19KB.

4. Results

Results are obtained from the log files produced by the PANTS daemon providing information for CPU, memory, disk, context switching, and interrupts at five second intervals. We measured the load on an idle machine and obtained baseline measurements for each metric, summarized in Table 2 along with the thresholds we used for the experiments in Section 4.2.

Table 2: Load Metrics

Metric	Baseline	Threshold
CPU (%)	0%	95%
I/O (blocks/sec)	250	1,000
Context switches (switches/sec)	950	6,000
Memory (pages/sec)	0	4,000
Interrupts (interrupts/sec)	103K	115K

4.1 Micro benchmark results

Once we implemented our new load measurements we devised a process to validate our measurement techniques. We wanted several "micro benchmarks" that were simple programs designed to stress a particular system resource. The micro benchmarks were run on unloaded systems, only running basic system services. These services included the `inetd` daemon, which is responsible for `rsh`, the system logging `deamon`, `crond`, `swapd`, and a few other services. To see a listing of all the processes running on each system in the cluster see [JN02].

The CPU benchmark displayed the typical behavior of the PANTSD system. As each node surpassed 95% CPU utilization it was removed from the list of free nodes and did not take any more processes. The total load of the benchmark was divided evenly between all four nodes. Figure 3 shows the average load cluster wide for each metric during the benchmark.

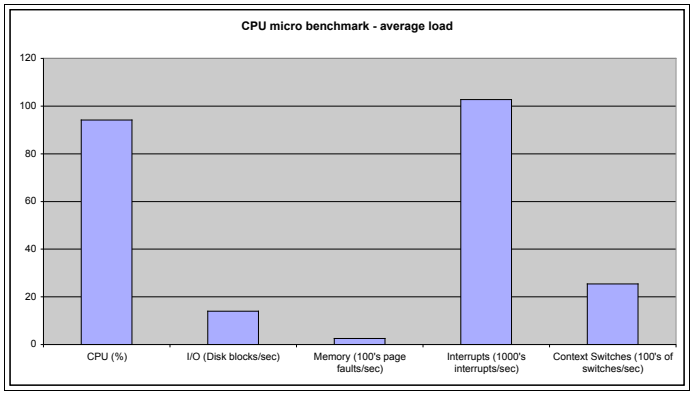

Fig. 3: CPU micro benchmark

The memory benchmark was first run with only the CPU metric enabled and the CPU threshold at 90%. The figures for this and the remaining benchmarks are not shown. The parent node never became unavailable, as CPU utilization stayed around 7%. However, memory usage on the parent node was extremely heavy, and the machine became unresponsive to user control for most of the test. While the memory load is high, high I/O load is incurred because the memory metric is page faults per second and page faults by their nature effect the load on the I/O subsystem. The standard deviation for the memory metric across all nodes in this benchmark is approximately 12,000 page faults/sec, which is expected as no load is being distributed so there is significant variation.

The memory metric was then enabled for the second run, with the threshold set at 10,000 pages/second. After the first process was spawned the parent node was loaded with 50,000 pages/second, switched to the unavailable state and was removed from the free node list. The second process was then passed to another node, which immediately became unavailable when it was loaded at 100,000 pages/second. The third process was passed to another node, and the fourth to the leader node. The memory load average is increased as the previously idle nodes are now participating, and the standard deviation is much lower at 575 page faults/sec. The load on the I/O system is dramatically lower for two reasons: the load on memory is not as high so not as many disk accesses are necessary to swap pages; furthermore some machines in the cluster have more memory than others and not as many swaps are required on these machines.

The disk benchmark was first run with only the CPU metric enabled and the CPU threshold. CPU utilization stayed between 4% and 8% for the duration of the run. The parent machine was fairly responsive to user input until the second process was spawned, at which time it stopped responding to other commands until the test completed. For this experiment, the average for I/O load is low, but the standard deviation is very large.

For the second run the disk threshold was set to 10,000 blocks/second in addition to the CPU threshold. Immediately after the first process was spawned the parent node was loaded at 20,000 blocks/second and became unavailable. The second process was passed to another node, which immediately displayed over 100,000 blocks/second and also immediately became unavailable. The third process was passed to another node, and the fourth process to the leader. The cluster wide load average is considerably increased, but the standard deviation, and indicator of variation in load amongst nodes in the cluster is greatly decreased, as the load is more evenly distributed. Using these simple micro benchmarks to generate certain types of load we verify that our metrics are measuring the load on the system correctly.

4.2 Application Benchmark Results

To evaluate the performance of the distributed compilation of the Linux, we obtained timing measurements for several variations of the compilation for comparison purposes. Our first variation was a compile where the files were stored on the local hard

disk and no PANTS load distribution. The second was a compile of the kernel tree which was stored on the NFS mounted disk. The next three were all compiles where the source was mounted over NFS with PANTS running. In the first of these three the `gcc` binary was flagged do not migrate, in the second PANTS used the default load metrics, and finally PANTS used our new load metrics and policy. Five compile times were averaged to obtain the results shown.

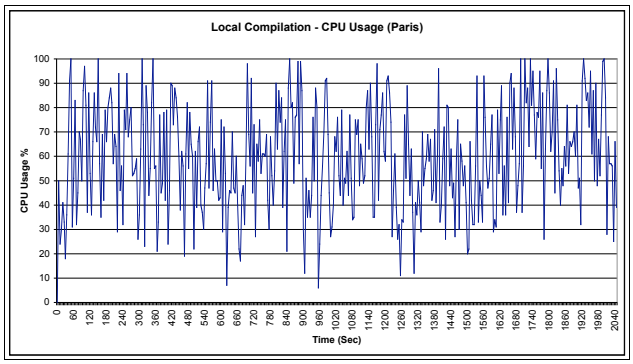

Fig. 4: CPU Usage: Local Compilation

Figure 4 depicts the CPU usage over time on one of the nodes for a local disk compilation, which shows some load being imparted on the CPU, but rarely is the utilization above 95%, the CPU load threshold. The compilation with PANTS running but migration disabled was achieved by flagging the `gcc` binary as not migratable. This evaluation was done to give baseline results and an idea of the overhead involved with running the PANTS daemon. We conclude that there was very little overhead incurred: the compile with PANTS running took 5 seconds longer out of a total of 1520 seconds than an NFS compile. This overhead is the slight delay induced by `prex`'s checking the `gcc` binary for migratability, which was done at least 432 times, once for each .c source code file, during the course of the compile. Additional checks were made of the make binary, and the utilities that make uses, such as `ld` and `ar`.

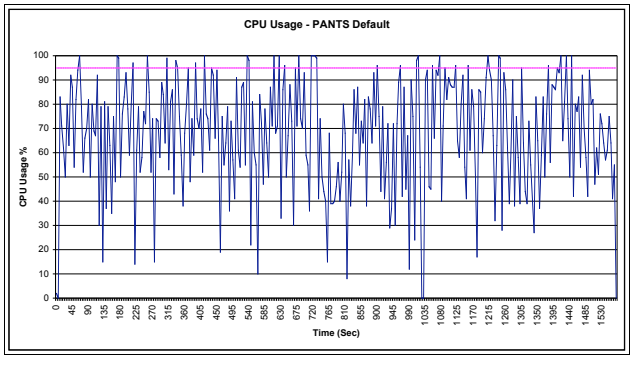

Figure 5: CPU Usage: PANTS default

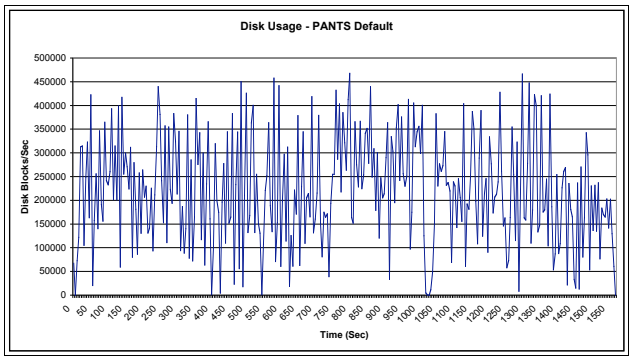

Figure 6: Disk Usage: PANTS default

The next evaluations were with PANTS running using the default load metrics. Specifically, as noted in section 2.1, the CPU metric was the only one being used and the threshold was set at 95%. As shown in Figure 5, the CPU usage during a compile rarely goes above 95%, and only at these times are process migrated. This lack of migration yields no throughput increase or load distribution throughout the cluster. In Figure 6, it is clear that the disk is being heavily loaded. In addition to the disk usage, the memory is being loaded and there are also a large number of context switches and interrupts generated during the compilation.

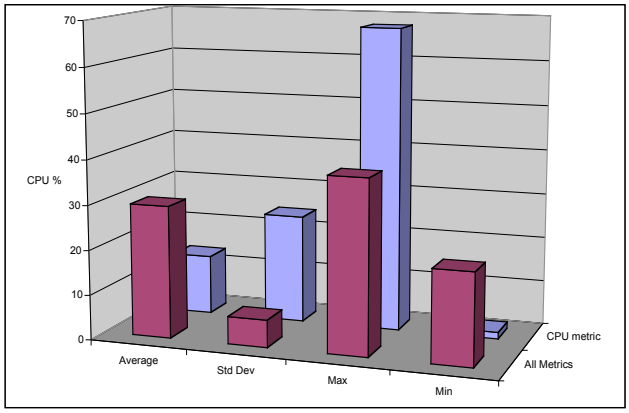

Figure 6: Cluster CPU Load Average

Using our new load metrics and policy the compile time is dramatically decreased because of load distribution. Figures 6-10 show a system level perspective of the results with our new metrics and policies. The figures compare the average CPU usage, I/O, memory, and context switch load using the PANTS default policy and the new policy and metrics we created. In these figures, the standard deviation is decreased considerably, and the maximum and minimum values are brought closer to the average, illustrating better load sharing.

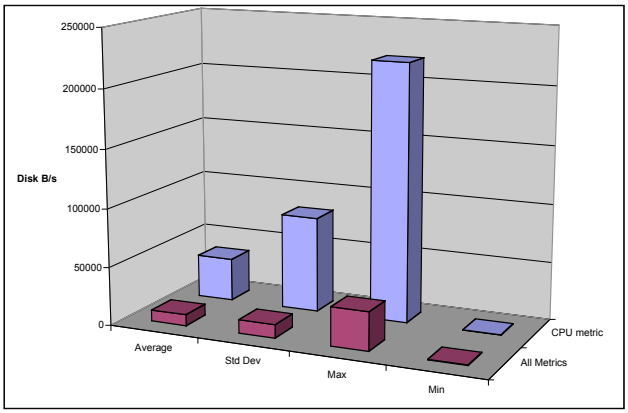

Figure 7: Cluster I/O Load Average

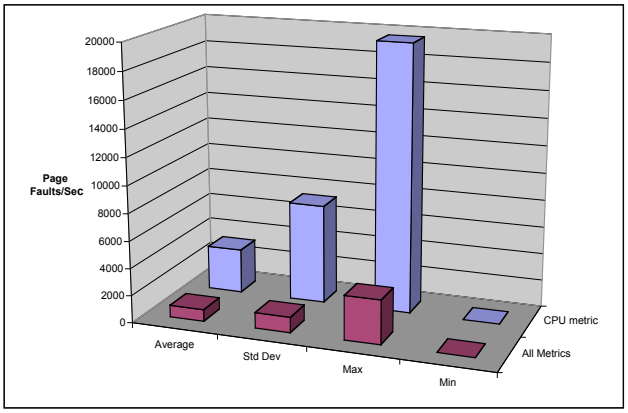

Figure 8: Cluster Memory Load Average

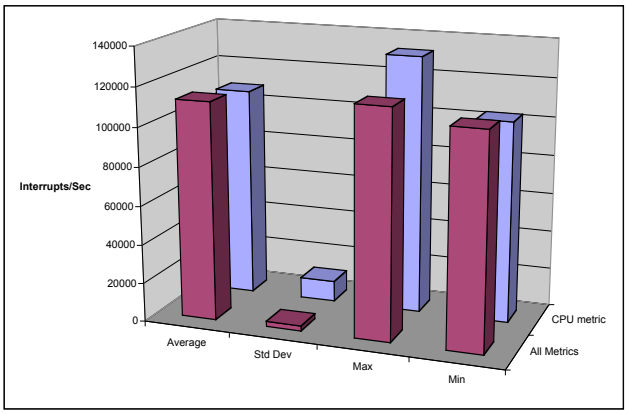

Figure 9: Cluster Interrupts Load Average

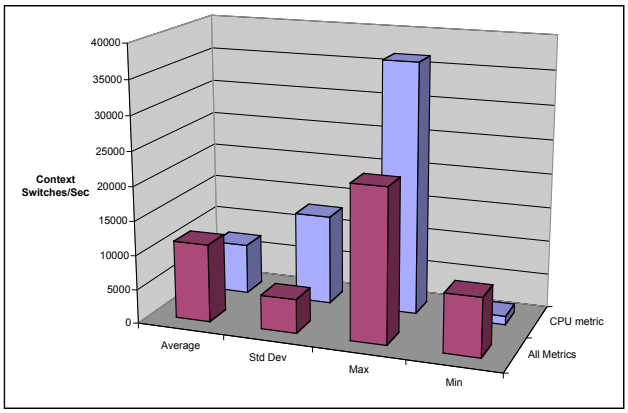

Figure 10: Cluster Context Switches Load Average

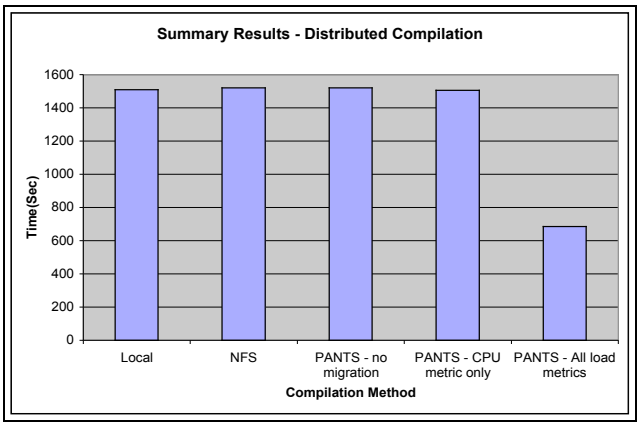

Figure 11: Summary of Linux Kernel Compilation Results

Several metrics show interesting behavior: the I/O and memory load average went down using our new policy. This may seem counter intuitive, but can easily be explained by the fact that some machines in the cluster have more RAM then others and so do not have to use swap space or have few page faults. There still remains some amount of variance in the load between different machines in the cluster using our new policy. This is due to the fact that there is some load that cannot be migrated from the originating node. This load includes that of running the `make` program and managing over 400 `rsh` remote executions.

In Figure 11, we compare the average time it took for each compilation method. From a user's perspective, our new metrics and polices result in a 54% reduction in the compilation time over the default policy, which just measured the load on the CPU. Furthermore, we decrease the time of the local disk compilation by 56%, and the NFS compilation by 56%.

5. Conclusions

Distributed computation has grown in popularity recently, earning attention from scientists and corporations. Accompanied with the dramatic growth of Linux, Beowulf clusters provide a cost effective solution to today's large computation needs. PANTS and its use of transparent load sharing takes another step in the direction of ever more powerful cluster technology. PANTS is designed to run on any system running a modern distribution of Linux regardless of the underlying hardware architecture. Transparency, reduced busy node communication, and fault tolerance make PANTS a viable solution for more effective use of a Beowulf cluster.

In this work we designed and implemented ways to measure additional types of load beyond that of typical CPU utilization and incorporated their use into the PANTS load sharing algorithm. We developed micro benchmarks to test and verify our new load metrics and to gain insight into how the system was behaving when loaded. Finally, we used a real world application as a benchmark, the performance of a distributed compilation of the Linux kernel with PANTS.

When using the default PANTS load metric and policy, only examining CPU load, we found that there was very little process migration. Subsequently, there was no increase in throughput and no sharing of load throughout the cluster. Using our new metrics and policy we achieve better throughput, decreasing the length of the compile by more than 50% from the default policy. From a system level perspective, we lowered the standard deviation from the average cluster-wide load for each of the metrics considerably sharing load very well amongst the nodes in the cluster. Including I/O, memory, context switches, and interrupt load metrics has many benefits when used in load distribution.

6. Future Work

When PANTS makes a decision to run a process on a particular node the job runs to completion. A job may end up running on a loaded node perhaps because no other nodes were free or a bad load distribution decision was made, preemptive migration allows this situation to be corrected. Preemptive migration is the term used to describe the act of stopping a process that has started execution, moving it to another machine, and resuming the execution. Some load distribution algorithms make use of preemptive migration to allow overloaded nodes to send processes currently running to other nodes. The primary advantage of this preemptive migration is that if a job is running on a busy node while another is less loaded the load can still be distributed. PANTS may benefit from using preemptive migration, but we have not yet implemented this feature because most migration techniques and implementations are architecture bound. Finally, further analysis is required to compare the relative cost of migration using preemption and non-preemption.

Including a network component of the load metrics may be beneficial to PANTS. The current metrics may overlap network usage somewhat, as context switches and interrupts are caused by network activity. This metric could be as simple as parsing in-

formation retrieved by the `ifconfig` command, or a slightly more elegant approach by making use of the `/proc/net` information. Perhaps the most benefit can be found in the use of adaptive thresholds. Currently, the thresholds used by PANTS are static, although they can be modified without restarting the daemon. If the thresholds were adaptive PANTS might be able to respond more appropriately when the load on the system fluctuates. The current work in progress is to investigate adaptive thresholds along with some heuristic based load distribution to send processes to nodes that perform better at processing a particular type of workload.

Acknowledgements

This research was supported by equipment grants from Alpha Processor, Inc. and from Compaq Computer Corporation. The authors would also like to thank the following WPI students and faculty who have contributed to the various phases of the PANTS project: Jeffrey Moyer, Kevin Dickson, Chuck Homic, Bryan Villamin, Michael Szelag, David Terry, Jennifer Waite, Seth Chandler, and David Finkel.

References

[CF01] Mark Claypool and David Finkel. Transparent Process Migration for Distributed Applications in a Beowulf Cluster, In Proceedings of the International Networking Conference (INC), Plymouth, UK, July 2002. Online at: `http://www.cs.wpi.edu/~claypool/papers/pants`

[DHV00] K. Dickson, C. Homic, and S. Bryan Villamin. Putting PANTS On Linux: Transparent Load Sharing In A Beowulf Cluster. *Major Qualifying Project CS-DXF-9918*, May 2000. Advisor Mark Claypool and David Finkel.

[EH] Hendriks, E., "BPROC: Beowulf Distributed Process Space", Online at: `http://www.beowulf.org/software/`

[FW95] David Finkel and Craig E. Wills. Scalable Approaches to Load Sharing in the Presence of Multicasting. *Computer Communications*, 8(9), September 1995.

[JN02] James Nichols. Performance Evaluation of Load Sharing Policies on a Beowulf Cluster. *Major Qualifying Project CS-MLC-BW01*, April 2002. Advisor Mark Claypool.

[KK] Karimi, K., "Welcome to DIPC", Online at: `http://www.gpg.com/DIPC/`

[PVM] PVM. Parallel Virtual Machine. Online at: `http://www.epm.ornl.gov/pvm/pvm home.html.`

[STW01] Michal Szelag, David Terry, and Jennifer Waite. Integrating Distributed Inter-Process Communication with PANTS on a Bewoulf cluster. *Major Qualifying Project CS-DXF-0021*, March 2001. Advisors Mark Claypool and David Finkel.