Putting PANTS on Linux: Transparent Load Sharing in a Beowulf Cluster [1]

Kevin Dickson, Chuck Homic, Bryan Villamin

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609 USA

April 30, 2000

[1] This research has been supported by equipment grants from Alpha Processor, Inc. and from Compaq Computer Corporation.

Abstract

PANTS is the PANTS Application Node Transparency System. It provides automatic and transparent load sharing on a Beowulf cluster of Linux computers. PANTS manages the resources of the cluster and executes processes remotely to share the computational load among the individual nodes. Even multiprocess applications not designed for a cluster can gain improved performance. This is achieved with a daemon running on each node and PREX, PANTS remote execute, intercepting and remotely executing initiated processes.

Contents

1 Introduction **4**

2 What is Beowulf? **5**
2.1 History 5
2.2 Advantages 5
2.3 Current implementations 6
2.3.1 PVM 6
2.3.2 MPI 6
2.3.3 DIPC 6
2.3.4 BPROC 6
2.4 Beowulf Cluster Configuration 7
2.4.1 Hardware 7
2.4.2 Software 8
2.4.3 Writing distributed applications 8

3 Historical PANTS **10**
3.1 Design 10
3.1.1 Multicast Load Sharing 11
3.1.2 Transparent Process Migration 11
3.2 Toward A New PANTS 12

4 Today's PANTS **13**
4.1 Goals 13
4.2 What's new? 14
4.3 Design 14
4.3.1 PANTSD 14
4.3.2 PREX 18

5 Results **23**

6 Conclusion **25**

7 Future Work **26**
7.1 Preemptive migration . . . 26
7.2 IPC . . . 27
7.3 Tune Load Variables . . . 27
7.4 Prevent Overloads . . . 28

A Using PANTS **29**
A.1 Hardware and Software Requirements . . . 29
A.2 Installing PANTS . . . 30
A.3 Testing the Installation . . . 30
A.4 When Things Go Wrong . . . 31
A.5 Running Applications with PANTS . . . 32

B A Sample Distributed Application **34**
B.1 sum.c . . . 34
B.2 multisum . . . 35
B.3 total.c . . . 35
B.4 bigsum . . . 36
Bibliography . . . 37

Chapter 1

Introduction

PANTS is the PANTS Application Node Transparency System. It provides automatic and transparent load sharing on a Beowulf cluster of Linux computers. PANTS manages the resources of the cluster for the user and executes processes remotely to share the computation load among the nodes in the cluster.

The benefits of cluster computing are well known [12]. A large class of computations can be broken into smaller pieces and executed in parallel by various nodes in a cluster. Sometimes, however, it can be beneficial to run an application which is not cluster-aware on a Beowulf cluster. This is one of the main goals of PANTS.

PANTS was 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, not built with cluster computing in mind, can run on multiple nodes by invisibly migrating the individual processes of the application. As far as the application is concerned, it is running on a single computer, while PANTS controls what resources it is using.

The PANTS design also contains a method for minimal inter-node communication and fault tolerance. In a Beowulf system, the network is most often the performance bottleneck [12]. With this in mind, PANTS keeps the number of messages that move between machines low and also uses a protocol which does not exchange messages with nodes busy with computation. 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 dramatic consequences.

Chapter 2

What is Beowulf?

A Beowulf is a collection or cluster of workstations. These workstations are usually low-cost personal computers on a local area network (LAN) connected to each other via Ethernet. The main idea behind the Beowulf project was to build a low-cost but highly scalable parallel system using relatively inexpensive personal computers.

2.1 History

The Beowulf was developed by Donald Becker and Thomas Sterling of the Center of Excellence in Space Data and Information Sciences (CESDIS) [8] in the summer of 1994. CESDIS is a division of the University Space Research Association (USRA) at Goddard Space Flight Center (GSFC) in Greenbelt, Maryland. CESDIS is a NASA contractor, supported in part by the Earth and Space Sciences (ESS) project. The ESS project is a research project within the High Performance Computing and Communications (HPCC) program.

The first Beowulf was built to address a computational requirement of the ESS community. The initial cluster consisted of sixteen 486 DX4 100MHz CPUs. It was constructed using 10Mbps Ethernet for the system area network (SAN). Because the Ethernet card's throughput proved to be inadequate for the speed of the CPUs, two Ethernet cards (per machine) were used. This required binding two Ethernet addresses to one IP, known as channel bonding. One Ethernet card dealt with the outgoing information, the other card with the incoming data.

2.2 Advantages

The biggest advantage of a Beowulf system over massively parallel processors (MPPs) or supercomputers is the cost. Since inexpensive personal computers are used as nodes, a powerful Beowulf system can be built without spending a fortune. This cost advantage often can be as much as an order of magnitude

over commercial systems of comparable capabilities [12]. The Beowulf currently provides the most bang for the buck. Another advantage of a Beowulf cluster is scalability. A wide range of system sizes is possible from a small number of nodes connected by a single low cost hub to a system incorporating complex topologies of many hundreds of processors. And these systems can be expanded over time as additional resources become available or extended requirements drive system size upward [12]. The Beowulf is affordable, powerful, scalable, and easily expandable.

2.3 Current implementations

There are many libraries designed to simplify the creation of parallel applications for the Beowulf. The most widely used libraries are Parallel Virtual Machine (PVM), Message Passing Interface (MPI), Distributed Interprocess Communication (DIPC), and Beowulf Distributed Process Space (BPROC).

2.3.1 PVM

Parallel Virtual Machine (PVM) was developed at Oak Ride National Labs. It provides a complete runtime system for parallel computation with a powerful message-passing interface [11]. The PVM runtime message-passing system is easy-to-use, portable, and widely popular on parallel systems. It has been designed so that users without system-administration privileges could install the software and run parallel jobs from their shell accounts.

2.3.2 MPI

Message Passing Interface (MPI) was designed to allow for efficient implementation on all existing distributed memory parallel systems [5]. The MPI provides a complete library specification for message passing primitives and has been widely accepted by vendors, programmers and users.

2.3.3 DIPC

Distributed Inter-Process Communication (DIPC) provides distributed program developers with semaphores, messages and transparent distributed shared memory. A patch to the Linux kernel and a set of user-space utilities provide this functionality [3].

2.3.4 BPROC

Beowulf Distributed Process Space (BPROC) provides a distributed process ID (PID) space [6]. This allows a node to run processes that appear in its process tree even though the processes are physically present on other nodes. The remote processes also appear to be part of the PID space of the front-end node and not the node on which they are running.

2.4 Beowulf Cluster Configuration

Beowulf is a set of nodes connected by a network. A node is usually a low-cost personal computer. There are at least two kinds of nodes in a cluster, the head node and the worker nodes. The head node is where users can login, and usually, the node from which most tasks are initiated. This machine is usually set up with at least two network adapters to separate local and remote traffic. The worker node is used for computation. The nodes are often rack-mounted to conserve space, as some of the bigger Beowulf systems consist of several hundred nodes.

2.4.1 Hardware

The nodes of a Beowulf cluster are generally Intel-based personal computers. This doesn't mean that it's limited to inexpensive personal computers. A node could be a higher-end machine too, like an Alpha or SPARC workstation. Beowulf can be run on almost all hardware architectures. It is, however, suggested to have a homogeneous cluster with nodes of the same CPU type and all the nodes running the same operating system [12] since a cluster with similar nodes is easier to maintain. The task of creating parallel applications for a homogeneous cluster is simpler than creating programs for a non-homogeneous cluster. There is always a chance of a programming language or hardware discrepancy between the nodes.

A node needs at least a motherboard, processor, RAM, hard drive and a network card. The head node requires a video card, monitor, keyboard, mouse, and two or more network cards, depending on its network setup. For easier maintenance, all nodes must have a floppy drive and video card. The floppy drive is needed in case a boot-up disk is needed for node failure recovery. Some of the newer motherboards come with on-board video (and audio) for extra, but minimal cost. Although not required for all the nodes. Having on-board video card will save the system administrator time; swapping cards between nodes can get tedious.

The Beowulf cluster used for this MQP is made up of seven 64-bit Alpha workstations donated by Compaq Corporation and Alpha Processor Inc. (API). Each machine runs at 600MHz and contains between 64 and 512 MB of RAM.

Networking

In order to have a Beowulf cluster, a network must be present to connect the nodes together. This is generally done by having at least one Ethernet card installed on each node, and these cards connected together using a hub. Both Fast Ethernet cards and a wide variety of hubs are generally inexpensive and readily available. Faster network connection can also be obtained through slightly more expensive equipment such as 100Mbps Ethernet cards and high speed switches.

10 Mbps hubs are being used to connect the nodes of our Beowulf cluster. Each node contains a 10/100Mbps Ethernet card.

2.4.2 Software

Beowulfs use an open source operating system where all source code is available at no extra cost. Linux is the most popular open source operating system for Beowulfs. Linux is a monolithic, multitasking, virtual memory, demand paged, POSIX compliant operating system [12]. Complete source code for Linux is readily available for download from the Internet ready for editing which makes it a top choice for Beowulf projects.

Linus Torvalds developed Linux in 1991 at the University of Helsinki in Finland. Torvalds' early version of Linux was primitive. A graphical interface was not built in, instead just a simple command line interface was provided. However, over the last decade, Torvalds and various developers from around the world have greatly improved Linux. Linux now supports the complete GNU programming environment including gcc and g++, emacs, as well as compiler tools such as lex and bison [4]. It provides the X Windows user interface and window managers, remote access, a complete file system and NFS for distributed files.

There are several Linux features that are especially useful for setting up a Beowulf system:

The /proc file system

Linux has a process file system, which serves as an interface to kernel data structures. The /proc file system doesn't actually exist on the local hard drive, but is made to appear that way for ease of use. By doing this, system information and devices can be manipulated by standard tools and remotely accessed through a network file system instead of using special tools specifically designed for that purpose.

Loadable kernel modules

Loadable kernel modules are dynamically loaded extensions to a monolithic kernel. In a Beowulf system, loadable modules make system management a much easier task. It avoids the need to recompile a kernel, install it on every node of the system, and reboot every node. Instead, only the particular feature being added needs to be compiled as a module, and then installed on every node without rebooting a single machine

Our cluster uses Red Hat, and Red Hat based distributions of Linux. All of the nodes are using Linux kernel version 2.2.12.

2.4.3 Writing distributed applications

One reason for putting together a Beowulf system is to run applications which require a lot of computational power to complete. Applications for analyzing large amounts of data would complete sooner if the computations could be broken up into smaller individual computations. These applications would need to be designed to run on distributed systems such as a Beowulf cluster. A

program has to be "parallelized" so that it will work effectively and efficiently on a Beowulf. Process level parallelism is often the easiest and most effective way to utilize a Beowulf system. Since Message Passing Interface (MPI) is the most common library used by distributed applications programmers, a short description of how it is used is presented.

MPI is primarily a communications library. As mentioned before, MPI provides message passing primitives that help in creating distributed programs. An MPI program usually consists of a several processes running in parallel with each other. Each of the processes in an MPI program is unique, with separate and unshared address space. MPI however, does not provide automatic load balancing. Load balancing is the act of running applications on multiple nodes in such a way as to keep any one node from doing too much of the work. The programmer must handle the load balancing decisions when using MPI.

The communication between the processes is achieved by explicitly calling MPI messaging procedures at both the sending and receiving end. An MPI process is just a typical program written in C or C++ and linked with the MPI libraries. MPI programs can be executed on Beowulf. No special kernel, operating system, or language is needed.

To create a MPI distributed application, the program must be first initialized to load the MPI specific messaging primitives. `MPI_Init` is used for initialization and `MPI_Finalize` for termination. Invoking MPI procedures before `MPI_Init` and after `MPI_Finalize` are illegal. Other primitives include `MPI_Comm_size`, `MPI_COMM_WORLD`, and `MPI_Get_processor_name` for determining the current node's relation to the other nodes it the cluster. For communication between processes, `MPI_Send` and `MPI_Recv` are used and are the two most important MPI primitives. These primitives cause the transmission of a message from a sender's memory to a receiver's memory. The location, size and type of the contents of a message are specified by the first three arguments to `MPI_Send` and `MPI_Recv`. These primitives are "blocking" procedures, that is, they do not return to their caller until the requested operation is complete. Using MPI to create distributed applications is fairly easy, but it does force the programmer to know certain aspects of the cluster in order to best design the application.

Chapter 3

Historical PANTS

This MQP is a continuation upon the PANTS project which was originally developed by Jeff Moyer [9]. PANTS sought to achieve transparent load sharing for Beowulf systems, that is, run programs on a Beowulf cluster that were not explicitly written for a cluster environment. In this chapter, Jeff Moyer's implementation of PANTS is referred to as "PANTS v1.0."

3.1 Design

Computations written explicitly for parallelism have the potential to outperform traditional serial or procedural programs. However, under many circumstances, it is desirable to run an application that was not specifically designed for cluster computation on a Beowulf system. PANTS was created by Jeff Moyer with this goal in mind [9]. However, this is not the only possible use. PANTS can also be used to simplify the development of new parallel applications, by handling all the details of process migration for the programmer.

PANTS consisted of two major parts: A load sharing policy which makes decisions about which node receives a particular process, and the facility to actually move a process to the selected node. The load sharing policy was a multicast-based policy developed by WPI Professors Craig Wills and David Finkel in [13]. This protocol was designed to minimize the number of "busy machine messages," that is, the number of messages that need to be processed by a node which is busy working on a computation. This PANTS v1.0 implementation of the multicast load sharing policy was called TPMD (Transparent Process Migration Daemon).

To actually achieve load sharing, PANTS v1.0 used EPCKPT [10], a patch to the Linux kernel which adds process checkpoint and restart capability. Using this system, PANTS v1.0 can halt and package a running program, transport it to an available node in the cluster, and restart it remotely.

3.1.1 Multicast Load Sharing

Multicasting provides the unique opportunity to send messages that will be received on a LAN only by those nodes that require them. The multicast leader policy proposed by Wills and Finkel uses two multicast addresses, one for free nodes, and the other for the leader.

The free node multicast address is used to eliminate network messages sent to busy nodes, as well as limit the number of total network messages. Each time a node becomes free, it sends a message to the free node multicast address. In this way, each free node is aware of all the free nodes that come after it. If the leader node becomes busy or unavailable, then the free node that is aware of the greatest number of free nodes becomes the new leader. Therefore, there is only a single message each time a node changes status, as well as only a single message to change leadership.

Because there is only one leader at any time, using multicast for leader messages means that the leader can always be reached at the same address. This is important because each client must be able to communicate with the leader at any time. If the leadership were to change, for example if the leader crashed and a new leader were chosen, each client would have to be notified of the change. Using multicast eliminates this problem, providing a consistent address for the leader.

3.1.2 Transparent Process Migration

The multicast load sharing system handled the coordination of the cluster, determining which nodes are free and deciding where processes should be delivered. PANTS v1.0 used EPCKPT, a process checkpoint/restart utility developed by Eduardo Pinheiro [10], to actually deliver these processes.

EPCKPT is a patch to the Linux kernel which can save the state of an entire process so that it can be removed from the system and restored at any time. Saving state in this manner is called checkpointing. Checkpointing is used for an array of purposes, including fault-tolerance, application debugging and analysis. PANTS v1.0 used checkpointing to stop a process on an overloaded node, move it to a free node, and restart it there. It is in this way that PANTS v1.0 distributed processor load in a cluster.

In EPCKPT, checkpointing is transparent, meaning that the procedure is done in such a way that the application is not even aware of the change. This is beneficial because using this facility, applications that were not explicitly designed to be run on a Beowulf can be manipulated by PANTS to improve performance on the cluster. It also means that a developer of new distributed applications can count on PANTS to handle resource management for the cluster, and concentrate on developing the application itself.

3.2 Toward A New PANTS

Jeff Moyer's implementation of PANTS created an effective means for process migration on a cluster. Programs to be run on PANTS did not need to be cluster aware, allowing programs to benefit from PANTS without needing built-in migration methods. Its use of multicasting efficiently used the network, while providing a simple means of coordinating the various nodes in the cluster. The ability to move processes in mid-execution also made PANTS a very powerful system by utilizing all the available processing power, even in situations where load demands changed unexpectedly.

Although the design of PANTS was a powerful load sharing system, its implementation was incomplete, providing no inter-process communication, and supporting only Intel systems. The goal of our MQP was to further develop PANTS, making it useful for real-world cluster computing, continuing the goal of transparent load sharing, and adding architecture independence so it would no longer be bound to just Intel-based clusters.

The next chapter discusses in more detail the goals of our new PANTS implementation, as well as how we changed the PANTS architecture to achieve them.

Chapter 4

Today's PANTS

4.1 Goals

The goals for PANTS can be viewed from two angles. First there are the goals of transparency and the ability to distribute processes across a cluster for effective load sharing. This defines PANTS and makes it unique among the different methods for cluster organization. Second there are the specific goals for this MQP. These are fault tolerance, architecture independence, and the minimizing of inter-node communication.

Load sharing is the purpose for cluster solutions such as PANTS. What makes PANTS unique is its ability to perform load sharing transparently — to execute a task on one node from another without the user or the task explicitly knowing that the execution has taken place remotely. Our use of transparency gives the user more freedom than is available in other cluster organization techniques. Users do not need to create programs specifically for PANTS using special libraries or develop them with a specific cluster topology in mind. Instead the user can be sure that any multi-process application can potentially benefit by being run with PANTS. PANTS itself will handle the details of load sharing. The result is performance benefits for applications that are not cluster aware. By using the PANTS design, a process will not know that it had been executed remotely, because it will appear that it is running on the node from which is was originally executed.

Minimizing inter-node communication was also a goal for this project. In most Beowulf systems, the network is often the bottleneck. Although the individual nodes of the cluster may be fast, communication between them uses Ethernet, which is comparatively slow. Keeping the amount of network overhead low leaves more bandwidth for inter-process communication. It is also best to communicate with busy nodes as little as possible.

Fault tolerance is another important issue when performing large computations. If a single node fails, the rest of the cluster should still continue functioning. In many of the current implementations for cluster organization, a static

head node is used. This provides centralized cluster organization, but if this head node should fail, the coordinator is gone and the cluster becomes useless. In PANTS, the leader is dynamic and can be chosen at any time. If the current leader node should for some reason fail, a new one is elected to take over the cluster leadership.

The final goal is architecture independence. We want to be able to run PANTS on any homogeneous cluster running Linux regardless of the underlying hardware architecture.

4.2 What's new?

Several elements make the new implementation of PANTS different than the version created by Jeff Moyer. In terms of what it offers, PANTS v2.0 has removed the architecture-dependent EPCKPT, temporarily foregoing the ability to preemptively migrate processes. Instead, load sharing occurs upon the instantiation of the process' execution. Additionally, the daemon has been re-written to be cleaner and to use a slightly different implementation of the multicast messaging policy in order to improve reliability. Also, PANTS no longer makes changes to the Linux kernel code to perform its task. Instead a new library object has been created to intercept process executions and trigger a program to remotely execute the process, which is also new to the PANTS design.

4.3 Design

The design of PANTS is divided into two separate pieces to perform its task. First is the PANTS daemon which acts as the administrative unit of PANTS. Its job is to coordinate the resources of the cluster. Second is PREX, the workhorse of our PANTS implementation. PREX does the actual remote execution of the processes when the process is started.

4.3.1 PANTSD

PANTSD is the PANTS daemon, and is loosely based upon TPMD (Transparent Process Migration Daemon), from the previous version of PANTS. The goal of the new daemon is to provide much of the same functionality as TPMD, while simplifying the design, and allowing room to expand the capabilities of the PANTS architecture.

Overview

The PANTS daemon is responsible for watching the cluster and keeping track of free nodes, so that when a new process is started, it can be directed to a node that is not busy working on a computation. To achieve this, a daemon runs on each node. The daemon is a single executable which can act as either a leader

or a client, depending upon which node is elected leader. The leader and client daemon each have their own particular responsibilities.

Like TPMD from the previous version of PANTS, the PANTS daemon uses multicast messages to communicate among nodes. Multicast messages are packets that can be received by any number of nodes on a LAN simultaneously. A node that wishes to receive multicast messages subscribes to a particular multicast address. This instructs the Ethernet interface to pick up multicast messages bound for that address. If a node is not subscribed to a particular multicast address, then messages to that address are ignored by the Ethernet interface.

PANTSD uses this multicast communication to send messages to the leader, and to free nodes. Figure 4.1 illustrates multicast communication in PANTS. There is only one leader in the cluster, and this can be any node. The leader must subscribe to the leader multicast address, and respond to requests from client nodes for the address of an available node when it is requested. It must also handle messages from clients about load information, so that it knows about every free node in the cluster at all times.

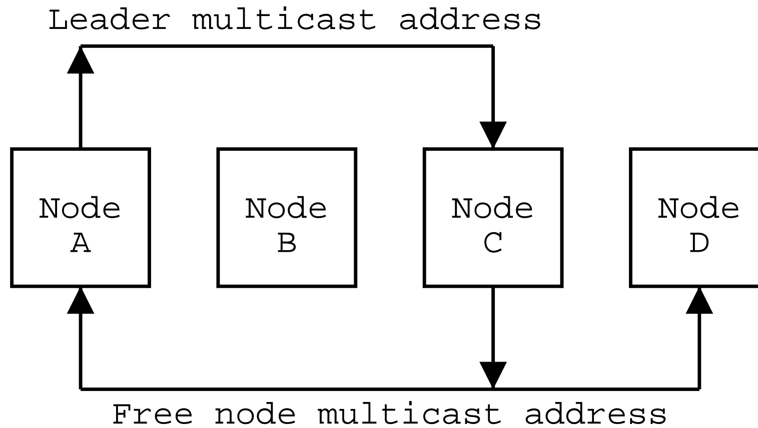

Figure 4.1: **PANTS Multicast Communication.** This figure depicts the multicast communication among PANTS daemons. There are four nodes in this Beowulf cluster, one of which is the leader. Nodes A and D are "free" nodes, having little computation load and Node B is a "busy" node. All Nodes can communicate with the leader (C) by sending to the leader multicast address. The leader communicates with all free nodes, A and D 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.

Every node in the cluster runs a client daemon. The client monitors load on each node to determine whether the node is available for computation, or busy. This measurement is made so that PANTS can make load balancing decisions. If a node is free, meaning the amount of load is low, then it is eligible to receive a process from another node. If it is busy, the load is too high to take on another process. Whenever the status changes, for example a free node becomes busy,

it must update this status with the leader. Since the leader is always aware of changes in availability, it can keep its free node list up-to-date.

Another responsibility of the client daemon is to respond to queries for a free node from PREX. When PREX asks the local daemon for a free node, the daemon must obtain the free node address from the leader, and return the result back to PREX.

Implementation of PANTSD

When PANTS is started on a cluster, the daemons must first agree on a leader. This process is called arbitration. Since it does not matter which node is the leader, the choice may be made randomly. It would be possible to allow the cluster user to choose a leader node; however this choice would be static, and would interfere with our goal of fault-tolerance. If this node were to fail, then there would be no leader. Thus, we chose to use nondeterministic leader arbitration.

When the PANTS daemon is started, it first subscribes to the leader multicast address, and sends a LEADER_CHALLENGE packet. This packet is used to discover if there is already a leader on the cluster. If a leader already exists, that leader will send a SMACKDOWN packet to the leader address, which tells everyone that they are not the leader, and should unsubscribe from the leader address and go away. If a daemon receives SMACKDOWN, it immediately begins acting as a client.

The LEADER_CHALLENGE packet contains a 32-bit random number N. This number is used in case there are multiple nodes in arbitration at the same time, a common occurrence during a cluster boot. If a node in arbitration receives a LEADER_CHALLENGE with an N larger than its own, it defers leadership to the remote node by leaving the leader multicast group, and becoming a client. On the other hand, if a node receives an N smaller than its own, it reasserts its own leadership by resending its LEADER_CHALLENGE packet, with its higher value of N. In the unlikely case that two nodes choose the same value of N, the decision is made by IP address, which must be unique. In this way, no matter how many nodes are in arbitration, only a single node will emerge the leader.

If a daemon sends a LEADER_CHALLENGE and receives no response with a greater N within a short period of time, it assumes the cluster is without leadership, and becomes the leader. Currently, this time period is 5 seconds, but much shorter values are possible. The daemon retains its membership in the leader multicast group, and creates a list of free nodes. To populate this list, a message packet called WHODAT is sent to the free node multicast group. Any node that is free will be listening to the free node multicast address, and respond by sending an ADD_AVAIL message to the leader address, indicating that the node should be added to the leader's free node list. Busy nodes are not subscribed to any multicast channel, and are unaware of any of this activity.

When a daemon becomes a client, it begins periodically monitoring CPU load to determine whether the node is "free" or "busy." The /proc/stat interface to the Linux kernel provides this information in a way that is easy to read from a

user space application. The first part of /proc/stat looks like this:

```
cpu 99576 0 14942 3796680
```

This output represents the number of kernel scheduling cycles that have been used by user space applications, "nice" applications, system processes, and idle time, respectively. These are cumulative totals, so it becomes possible to read /proc/stat again and learn more:

```
cpu 101146 0 15197 3827803
```

Now, to determine how the CPU was used during the interval, simply subtract the values. In this example, 1570 cycles were used by applications, 255 were used by the system, and 31,123 cycles were unused. Clearly, this node would be classified as "free" by the PANTS daemon.

PANTSD checks the CPU load in this manner every 5 seconds, although this time interval can be changed in the code. Whenever a node changes from free to busy during this interval, the daemon sends the leader a REMOVE_AVAIL packet, indicating that this node should be removed from the free node list. It also unsubscribes from the free node multicast group. Now that it is busy, it need not be bothered by any multicast traffic.

If load changes from busy to free, it once again sends ADD_AVAIL to the leader, indicating that it should be put back on the list. Thus the leader knows which nodes are free in the cluster at all times. This information comes in handy when PREX is looking for a free node.

When PREX determines that a program can possibly be executed remotely, it queries the local PANTS daemon via a local socket. The daemon first looks at the most recent load measurement. If the node is free, no load sharing needs to take place. PREX is instructed by PANTSD to simply execute the process locally. However, if the local node is busy, the daemon must quickly find a free node to which the process can be sent. To do this, it sends a GETNODE packet to the leader. The leader consults the free node list, chooses a node randomly, and returns its address in a FREENODE packet. The leader also removes that node from the list, anticipating that it will soon be busy. The local daemon returns the address to PREX, and PREX is free to execute its process on the free node. If there are no free nodes, the FREENODE packet contains no address, and PREX must run the process locally.

Changes from TPMD

The PANTS daemon differs from TPMD in several important ways. First, the original protocol outlined in [13] does not account for packets which are lost due to network congestion. This is important because free nodes are expected to update their own free node lists when additional nodes become available. If a packet is lost, the free nodes would fall out of synchronization, which may lead to a node not being used. Both TPMD and PANTSD solve this problem by having free nodes periodically send a message to the leader address. In PANTSD, this

is implemented by occasionally retransmitting the ADD_AVAIL message to the leader address.

This periodic update means that it is no longer necessary for free nodes to maintain their own incremental free node lists, as such a list can be regenerated by these periodic updates. Further, a new free node list can be regenerated with a WHODAT message in case the leader is somehow lost. The PANTS daemon has this ability for the leader to query the available address so that all free nodes can respond and re-populate the free node list at any time. This general query is only used at cluster boot time, to create the initial list, or when a new leader is elected. Since the loss of a leader should be very rare, this temporary increase in network traffic is negligible, and does not warrant a more complicated protocol to eliminate it.

TPMD used EPCKPT [10] to checkpoint running processes and restore them on remote nodes. Because EPCKPT is specific to the Intel architecture, we have chosen to eliminate its use in favor of architecture independence.

Because much of the benefit of process migration can be achieved at process initiation, the PANTS daemon provides free node information to PREX. Remote execution has the potential to be more efficient than preemptive migration, since initial network traffic is limited to the executable path and arguments. Preemptive migration requires that the entire process space be transported before it can be restarted on the remote node.

TPMD used the one-minute load average to classify a node as available, unavailable, or overloaded. While this is useful information, it means when an available node is assigned a process, it could be up to one minute before it is removed from the free node list. It is possible the node could be assigned another process during this time that results in an overloaded node. Obviously, if there is another node available in this situation, it should be used instead. TPMD solved this problem by choosing available nodes nondeterministically, greatly reducing the chance that a node is overburdened. PANTSD also chooses nodes randomly, but further improves the situation by making faster load measurements.

Availability of a node in PANTSD is determined from the CPU load information exported by the /proc/stat interface. This makes it possible to compute load over any arbitrary interval, such as 5 or 10 seconds. This reduces the window during which a node could become overloaded. Also, when a node is selected by the leader, it is removed from the free node list, preventing it from being selected again for a period of time, to ensure its load information has settled before it is assigned another process.

4.3.2 PREX

Overview

PREX, which stands for "PANTS remote execute," is our system for remotely executing a new process in the cluster. PREX has been broken into two separate parts, each performing a distinct task. In order to operate transparently, PREX must be able to intercept processes before they are loaded. The library object

libprex is used to intercept the initiated process. Actual process migration is performed by the application prex.

The prex application can be initiated from the command line to start up a process on a free node determined by the PANTS daemon. This is done by calling prex with the binary name of the process as the first argument with the arguments to that binary as the remaining arguments to prex. Internally, prex first determines if the binary is migratable. If the process is indeed able to be migrated, prex will query the local daemon about where to send the process. If the current node is not heavily loaded with processes, the local daemon returns a signal to execute the process locally. Otherwise, the daemon returns the address of a node determined by the leader to be free. When the address is received, prex will execute the process on that node by making a remote shell call. If a node address is not returned or the binary is not found to be migratable, then it is automatically executed on the local node.

Although the user can run prex from the command line, this method would not result in transparency for the user. Instead, PANTS needs to perform the load sharing transparently. To do this a library object has been created to intercept process executions and to hand them off to prex for possible remote execution. This is accomplished by using the environment variable "LD_PRELOAD" which, when set to libprex, loads the prex library and uses it to intercept the process execution by redefining execve() to execute prex instead. The operation of libprex and prex is illustrated in Figure 4.2.

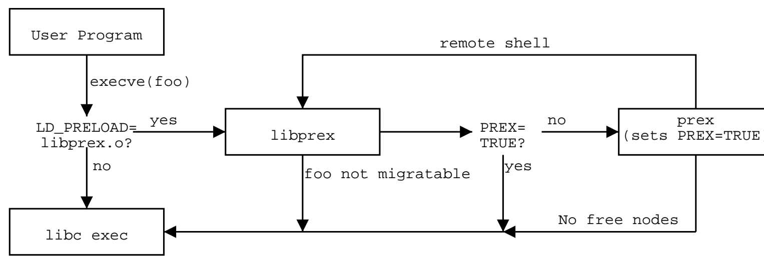

Figure 4.2: **PREX Functionality.** When a process calls execve() and the environment variable LD_PRELOAD is set, libprex.o intercepts the libc.o version of execve(). If the executable file is migratable, libprex.o invokes prex to communicate with the PANTS daemon to find a free node and execute the process via remote shell. If the executable is not migratable, or there are no free nodes, libprex.o invokes the normal libc.o version of execve(). After a remote shell, libprex.o is triggered on the remote node, and must break out of the loop by consulting the PREX environment variable.

Implementation of PREX

PREX is made up of libprex, the interception library object, prex, the application for migrating processes, and chmig application used for marking a binary as migratable.

In the PANTS system, ELF binaries can be marked with certain flags to indicate the binary's ability to be migrated. ELF stands for Executable Linking Format, and is the standard binary format for Linux systems. Flags within an ELF binary can be changed with our "change migratability" program called `chmig` which can set, remove, and check these PANTS-specific flags. The ELF binary format contains an easily accessible area specifically for one bit flags which `chmig` can set. There are three different flags currently settable by `chmig`. These flags allow for the PREX program to properly determine how to handle incoming binaries. These are the "migrate" flag, the "migrate immediate" flag and the "migrate once" flag. When the "migrate" flag is set, `prex` will know that the given binary is allowed to be remotely executed on another machine. If this flag is not present then the binary is executed directly on the node doing the execution. The "migrate immediate" and "migrate once" flags deal with preemptive migration, and are present for future expansion.

`prex` is the program which handles sending newly executed binaries to a free node for execution. It is called with its arguments being the binary name and the arguments for that binary. Additionally, the force option, "-f", causes `prex` to skip over the binary migratability check. When called, `prex` first checks the binary for migratability. Doing this checks that the binary is of the ELF format while binaries of other formats produce a prex error and are executed locally. Next, the header information is analyzed to determine if the binary was compiled for a 32-bit or 64-bit processor. This needs to be determined in order to know which data structure type to use for accessing the flags area of the binary's header data. After this, the flags in the header are checked for the migratability flag. After passing these checks, the binary is prepared for remote execution. A destination node is determined at this point by sending a query to the local PANTS daemon by creating a single use local socket for communication. If the local node is free, the daemon will instruct PREX to execute the process locally. Otherwise, it will return the address of a free node in the cluster. The node identification returned is used as the destination for the remote shell (`rsh`) call which executes the binary on that node with the other `rsh` arguments set. Since `rsh` is being used, the program to be executed must reside on all the nodes with the same absolute path. While the `rsh` command is issued, an environment variable is set in order to prevent a remotely executed process from being immediately migrated again. This "PREX" environment variable is set to "TRUE" to indicate that the executed process has been through the PREX program. After this set up, `rsh` is issued through an `execve` call with the updated environment.

The `prex` program works to find a free node, and complete the remote execution, but does not directly result in a transparent system of remote process execution. A broad view of the method for achieving migration transparency is to intercept each executed process at its start up and then send it through the `prex` program. Two methods for doing this were devised, and the second was eventually chosen over the first for reasons of simplicity, clean code and ease of use for the user.

The first method involved the use of a kernel patch which changed the way

Linux behaved as it handled incoming processes. Every executed process must make its way through certain sections of the kernel, and since the kernel code is available and editable, it is a convenient and effective means of catching and manipulating the process. Operating in kernel space is also fast and memory efficient. On the other hand, the Linux kernel is known to have frequent releases. If the kernel code undergoes big changes in areas that our patch modifies, a new patch would need to be made and distributed to work with the new kernel code. We'll call this process of adapting to new patches "version chasing." So the use of a kernel patch would have had higher than desired maintenance responsibilities.

From the user space, all processes are started with the libc `execve` call. Process execution eventually makes its way to the central function `do_execve` which is the last point where the argument list and environment variables look to be manipulatable as they are themselves arguments of the `do_execve` function. Unfortunately, these values cannot be changed without significant difficulty. In an attempt to overcome this, we move to a higher level of abstraction. The `do_execve` function is called by `sys_execve` which is located in its respective architecture specific file. This function and the functions calling it must be platform dependent, because it makes use of a data structure which represents the data stored in the CPU's registers. Registers are of course different on each of the separate architecture types. So in order to adapt the kernel at this level or higher requires editing the functions for each architecture type. One of our goals it to make PANTS architecture independent, so this is an unattractive solution. If we were to edit the kernel code for many different architectures and the next release of the Linux kernel were to include a new architecture, our kernel patch would not be able to handle it. To solve this, we moved out of the kernel level and into an earlier point during execution in user space.

The method currently used for PANTS involves intercepting the C Library `execve()` call. This reduces the version chasing scenario and also allows for a much neater and more modularized approach. The GNU C Library, `glibc` is freely available and is part a of most if not all distributions of Linux, so incompatibility problems should not be present. Also, by using `glibc`, the underlying architecture does not need to be considered as the complier will take care of that for us. Version chasing will be minimized as this method does not involve changes to the `glibc` code but rather adds external functionality to it.

This method intercepts the `execve` function call which initiates the execution of a program. When `execve` is called, its arguments are the binary name, the argument list and the environment variable list. Intercepting the calling of this function allows us to manipulate the arguments before sending it on its way to be executed. The first step was to create a library with our own definition of `execve` which first checks the environment variables for the PREX environment variable. Depending on this variable, the incoming process will be sent on its way unchanged, or the call will be changed to execute `prex` by changing the arguments of the `execve` call. The real libc `execve` function is called with the help of the dynamic library loader. Calls to `dlopen` and `dlsym` are used to retrieve an address to where the real `execve` function is located in the C library. This function pointer is then used to send the process on its way to be executed.

When executed, the program goes through the loader to load dynamic libraries by first looking at LD_PRELOAD. In PANTS, LD_PRELOAD is set to equal `libprex.o`, a library object containing our own definition of `execve`. By doing this, any calls at the application level to `execve()` will use our version of `execve` instead of the standard definition provided by libc. With the library in place to silently intercept calls to `execve`, PANTS will now be able to achieve load sharing transparently.

These two pieces, the PANTS daemon and PREX, together form PANTS. The PANTS daemon uses multicast messaging to keep track of which nodes are free, and PREX uses this information to execute processes on free nodes. The result is efficient and transparent load sharing.

Chapter 5

Results

In order to show that PANTS v2.0 works effectively, we ran and benchmarked a simple multiprocess application. This application is called "bigsum," because it computes the sum of large sequences of numbers. Bigsum consists of four major parts:

- `sum`. The basic computational unit of bigsum. `sum` accepts the start and end values of a range, and computes the sum of the range using brute force.
- `multisum`. This shell script is just a loop which runs multiple instances of the `sum` program.
- `total`. A simple adder. `total` accepts a list of numbers, and computes the sum. This is used to accumulate the results of `multisum`.
- `bigsum`. A small shell script with verifies command line parameters, and pipes the output of `multisum` to `total`.

Although it has many pieces, the application is not too complicated. In order to benefit from being run in a cluster however, the application needs to use multiple processes. Something similar is needed even to exploit multiple processors in a single computer. To look at this application more closely, refer to Appendix B.

Summation is a simple computation that easily exploits parallelism. It is part of a large class of applications which can be broken into several distinct parts which may be computed independently, and then accumulated to produce a result. In this case, the only inter-process communication necessary takes place at the beginning and end of each of the smaller computational components.

The results of our trials are given in Table 5.1. Bigsum is run on ranges varying from 1 million to 2 billion. On one node, it is clear that the time to complete a sequence is linear to the size of the sequence. Using PANTS to run the application on two nodes tells a different story. For small ranges, such as

1 million and 10 million, it actually takes longer to complete the computation. This is because there is significant overhead in executing a remote shell to move one of the processes to another node. From the table, it appears that it takes about 1 second of overhead to perform the remote execution. Increasing the size of the range however, eventually makes the overhead negligible, and the speed of computation approaches double the speed of a single node.

The results are similar for three nodes. Performance is poor for small ranges, but approaches triple speed for large ranges. Since PANTS was clearly not designed for applications which take less than one second to complete, it is definitely a reasonable method for executing this type of distributed application in a cluster environment.

Nodes	Range $\times$ 1,000,000	Time (seconds)
1	1	0.06
	10	0.28
	100	2.42
	200	4.81
	1000	23.89
	2000	47.74
2	1	0.99
	10	1.10
	100	1.34
	200	2.53
	1000	12.02
	2000	23.94
3	1	1.00
	10	1.07
	100	1.85
	200	2.60
	1000	9.04
	2000	17.19

Table 5.1: **Results using bigsum on PANTS.** We ran the bigsum program on sequences ranging from 1 million to 2 billion, on a varying number of nodes. Note that aside from a small startup penalty, performance improves linearly with the number of nodes.

Chapter 6

Conclusion

PANTS v2.0 is a success. It performs as it was intended, organizing the resources of the cluster and allowing for efficient load sharing, transparent to the user and the executed programs. The goals previously set forth were satisfied in our implementation of PANTS. Minimal inter-node communication was achieved by designing the daemon to use the multicast policy set forth in the previous version of PANTS, and modified for this project. The leadership policy used by the PANTS daemon uses dynamic leadership which makes sure that there is always one leader selected with an efficient election policy. PANTS was coded throughout to be architecture independent in order to run on a broad range of clusters.

Chapter 7

Future Work

Although PANTS is effective for certain distributed applications, there are still a few limitations which restrict the types of applications that can benefit from PANTS. In this chapter, we discuss a number of ways in which we feel PANTS can be improved to make it more useful for a wider range of computations.

7.1 Preemptive migration

Currently, PANTS only has the ability to remotely execute a new process. This in itself has its own advantages but other situations may occur where extended use of load balancing is needed. Take for example a 3 node cluster running with all of the nodes busy doing computations. Now a user starts another process at node A. The daemon can not return a free node address since there are no free nodes. Therefore the new process is started locally and node A is now overloaded. After some time passes, node B completes its computations and is added back to the free node list. It would be beneficial, at this point, if node A could lighten its load by giving node B a process to work on.

The new process could be killed, and remotely executed on node B, but this would cause all the computation done up this point to be lost as the new process would be restarted. Here is where preemptive migration comes into play.

Preemptive migration is when a currently running process is frozen in its current state, moved to another machine and continued from where it left off on the new node. This can aid load balancing greatly by allowing overloaded nodes to send some of its processes to free nodes in the cluster. To actually migrate the process, the state of the process must be frozen, moved and reinstated on the new node. This involves saving all process information, including current register values, and memory corresponding to the process. This information must be packaged and sent to the new node. As stated in Chapter 3, Jeff Moyer's implementation of PANTS performed migration by using EPCKPT, but EPCKPT is bound to the Intel platform.

BPROC, the Beowulf Distributed Process Space [6], should be investigated

for adding preemptive migration to PANTS. As mentioned in Chapter 2, BPROC has the ability to preemptively migrate processes and is implemented on many different architectures. However, BPROC requires that a single, static node be chosen to coordinate activity. This conflicts with PANTS's goal of fault-tolerance, because if that node were to fail, the capabilities of BPROC would no longer be available. Also, process migration is explicitly triggered by the application. The use of BPROC in PANTS would require that this migration be controlled by the PANTS daemon.

Overcoming these difficulties should not be impossible. The addition of BPROC to PANTS would dramatically improve the load sharing power of PANTS, as PANTS would then be able to respond dynamically to the condition of overloaded nodes, rather than simply using remote execution to attempt avoiding overloads.

7.2 IPC

Extending the interprocess communication methods in PANTS would also serve to make the system more powerful. Currently, only communications through stdin and stdout are available, and are provided through `rsh`. Being able to communicate with pipes, shared memory, message queues and semaphores would allow a broader range of application to be used on PANTS.

DIPC [7] provides shared IPC services to distributed applications. As with BPROC, DIPC requires a static leader, and is not completely transparent, but these problems should be easily overcome. In many applications, shared memory or message queues are a much more natural means of communicating among different parts of an application. With PANTS and DIPC, these applications would be able to benefit from the cluster environment.

7.3 Tune Load Variables

The main purpose of PANTS is load sharing. But to achieve load sharing, it it first necessary to understand what "load" is. PANTSD measures load by looking at the percentage of CPU utilization during each time interval. A threshold is chosen, and a node with CPU use above this threshold is considered "busy." Others are considered "free." The value of this threshold, or the use of this particular method of CPU load measurement, may have an impact on the determination of availability of a node. Further, this could affect the efficiency of the cluster as a whole. In the current version of PANTS, a CPU load measurement technique and load threshold were chosen that seem to perform well, but the issue was not fully investigated.

Also, the choice of measuring CPU load, although it seems obvious, is not the only option. Another measure of load on a node is the amount of memory used by applications. If too much memory is in use, and a node begins thrashing its swap space, the node should also be considered overloaded, regardless of

the CPU usage. The MOSIX operating system migrates processes based upon excessive memory use. This process is called “memory ushering.” [2]

7.4 Prevent Overloads

One current weakness of PANTS is its handling of new processes when there are no free nodes. In this situation, PANTS currently executes the new process on the node that initiates it. While this may be acceptable in some circumstances, more often than not it results in an overloaded node. An alternative may be to suspend the new process as well as the calling process in PREX, before it is executed, until a node becomes available. In this manner, the overloading of nodes is prevented. This technique would greatly improve performance of applications that create more processes than there are nodes in the cluster. Process creation and process completion would progress almost in lockstep, fully utilizing the cluster during the duration of the computation.

Appendix A

Using PANTS

A.1 Hardware and Software Requirements

In order to use PANTS, there must first be a physical cluster set up. This requires having two or more computers connected together on an Ethernet. The speed and quality of the network components are not critically important, but a faster network can be helpful. Each of the nodes in the cluster must be of the same hardware architecture, whether it be Intel, Alpha, SPARC or another architecture. Although it is helpful, it is not necessary for each node to be the same speed. Also, other hardware, such as the type of hard drive or the amount of RAM do not need to be identical on each machine.

The cluster must be running a Linux distribution using libc 6 or higher. Most if not all modern Linux distributions will have the appropriate C library included. It is also required that distributed applications that run under PANTS be presented with a consistent filesystem on every node. Thus, the working directory of distributed applications should be shared via NFS, or some other shared filesystem. A shared `/home` partition can serve this purpose. Read-only directories, such as `/usr` do not need to be NFS; it is possible to mirror these directories on each node. In fact, there is a significant performance benefit to providing each node with its own `/usr`. This will avoid a large amount of network traffic when executing most programs, installed in `/usr/bin`. However, this decision is in the hands of the cluster maintainer, and may be influenced by available hardware as well. Read [12] to find out more about Beowulf cluster setup.

Since PANTS uses multicast communication to coordinate the cluster, you must have IP multicasting enabled in the kernel. PANTS also uses local sockets (also called Unix domain sockets) for communication between PANTSD and PREX, and the `/proc` filesystem to make load measurements. These options must also be enabled. In the "Networking options" section of the Linux kernel configuration, enable "IP: multicasting" and "Unix domain sockets." Also, in the "Filesystems" section, enable the option "/proc filesystem support." Then

recompile the kernel, if necessary, and install it appropriately on each node.

PANTS uses `rsh` to execute processes remotely. As such, your system must be configured so that `rsh` can be used within the cluster without using a password. To do this, make sure the address of each node in the cluster is in `/etc/hosts.equiv` on every node. Consult the `rsh` man page for more information.

A.2 Installing PANTS

Once the cluster is setup and running, PANTS can now be built and run on the cluster.

In the PANTS distribution directory, simply run `make` to compile `pantsd`, `libprex`, `prex`, and `chmig`. Then run `make install` to copy the programs to the appropriate directories. If your cluster does not share a common `/usr` via NFS, you will need to run `make install` on each node in the cluster.

A shell script, `prexify`, which simplifies the startup of PREX is also installed. If you wish to remove the PANTS elements from your system, run `make uninstall` at the command line. A simple program that can be used for testing the installation is created by running `make bigsum`.

Now that PANTS is installed, it must be started up by running `pantsd` on each node. Immediately a leader will be chosen and a free node list created and populated. The daemon is now ready to respond to queries for free nodes. Executing `pantsd` from within a startup script will set up PANTS when the machine is booted up. Once the daemon is running on each node, you will need to activate PREX to achieve load sharing. You can use PREX in three ways. First you can call `prex` from the command line to directly execute a process on a remote node. But this requires you to think about PREX every time you want to run a migratable task and removes the benefits of transparency. Second, a single shell can be "prexified." This is done by simply setting the environment variable LD_PRELOAD to `<libpath>/libprex.o` where `<libpath>` is the absolute path to libprex.o. This is generally `/usr/lib`. After this, start a new shell. The `prexify` shell script will automate this process when executed at the command line. Executing processes from this shell will cause the processes to be loaded by `libprex.o` and handled by `prex`. The third way requires a restart. In this method, LD_PRELOAD is set in the very beginning by your system startup script and will affect all users and all executed processes. This results in the most transparency for the user.

A.3 Testing the Installation

We have provided a simple multiprocess application called "bigsum" to test your cluster configuration. This application uses brute force computation to calculate the sum of a large sequence of numbers. Although this is not a brilliant algorithm (this sum could obviously be computed in constant time), it is effective

in consuming large amounts of CPU time, and demonstrating a performance improvement once PANTS is enabled.

To build `bigsum`, simply type `make bigsum` from the PANTS distribution directory, and move the resulting files, `bigsum`, `multisum`, `sum` and `total`, to a location accessible by all nodes. The makefile will automatically use `chmig` to enable the migratability flag in the `sum` executable, the program which does the bulk of computation.

Now, to try out `bigsum`, add up a bunch of numbers:

```
shell$ ./bigsum <processes> <low> <high>
```

`<processes>` specifies the number of processes bigsum should run to complete the computation. When running under PANTS, you should make this the number of nodes in your cluster.

`<low>` and `<high>` specify the range of numbers to be added. Depending upon your hardware, you may need to use an extraordinarily large range of numbers for the computation to take a significant amount of time. Since `bigsum` uses 64-bit integers, you may safely add a range as large as from 1 to 2 billion. (6 billion should be possible, but the shell uses 32-bit numbers.)

When you run `bigsum` with one process, note the time it takes to complete. Then, make sure PANTSD is running on each node, and LD_PRELOAD is enabled to use PREX. Run `bigsum` again and see how long it takes. It should run much more quickly when it is run under PANTS. You may confirm that the load sharing is working by running `top` on some of the nodes. While `bigsum` is running, you should notice the `sum` program using significant CPU time on each node.

A.4 When Things Go Wrong

In order for PANTS to work properly, PANTSD must run on each node and communicate via multicast with other nodes. Also, PREX must be enabled with LD_PRELOAD, communicate with PANTSD via a local socket, and execute tasks with `rsh`. In addition, applications which are to be migratable, must have the appropriate flags in the binary set. If any of these are not working, then you will be unable use PANTS for load sharing. Here is a quick checklist to look over in case you are having difficulty:

- If the intended processes are not being remotely executed, check that the application binary has the appropriate flags set. The status of the flags can be viewed by calling:

  ```
  shell$ chmig -v <the application>
  ```

  Something similar to the following will appear:

  ```
  Not 64bit, class=1
  <the application>:  Migrate Enabled
  ```

```
<the application>:  Migrate Immediate Disabled
<the application>:  Migrate Once Disabled
```

If "Migrate" is not set to "enabled", then run:

```
shell$ chmig -m <the application>
```

- Check network hardware and software. Make sure that the nodes are properly connected to the network, and you can communicate among them. Also, make sure you are running the proper kernel, with all of the required features listed above enabled.

- Using `ps`, ensure that PANTSD is running on each node. If PANTSD refuses to start up, inspect the output from the program. If it is unable to create a multicast or local socket, or some other configuration error prevents it from working properly, it will display an error message and exit.

- Try using `rsh` to execute a process on a remote node. For example: `rsh <nodeaddr> ls`. This should remotely log into the node specified by `<nodeaddr>` and produce a directory listing. If this doesn't work, or prompts you for a password, then `rsh` is misconfigured.

- View the LD_PRELOAD environment variable. It should look similar to:

  ```
  LD_PRELOAD=/usr/lib/libprex.o
  ```

  If LD_PRELOAD is not present, PREX will not be invoked automatically. If LD_PRELOAD is present, check to make sure the specified file exists. Also, make sure the `prex` executable is installed in the proper location, usually `/usr/bin`.

If things seem interminable, you may also contact the authors. There may be a unique configuration problem with your cluster that was unforeseen.

A.5 Running Applications with PANTS

Although PANTS strives to be transparent, the current version has two major limitations that restrict the applications it can support. You must take these considerations into account when designing distributed applications to run with PANTS. First, PANTS cannot tell your application how many processes to create. This is because PANTS does not talk to busy nodes, and therefore doesn't even know exactly how many nodes exist. Second, PANTS currently only supports stdin/stdout via `rsh` for inter-process communication.

Process count will have a significant effect on the performance of your computation under PANTS. Running too few processes will obviously under-utilize the cluster, since there are not enough processes to go around. Running too many processes will cause the excess processes to be executed by a single node,

leading to an overload condition; when the other nodes are finished with their processes, the overloaded node will still be hard at work. The "Future Work" section of this document discusses a couple ways that this problem may be handled in future version of PANTS. For now, we must temporarily sacrifice a bit of transparency and fix the number of processes to the number of nodes available.

Inter-process communication (IPC) is another area where the transparency of PANTS is currently limited. Because `rsh` is used to remotely execute tasks, the only method of IPC is stdin/stdout. Luckily, this is the easiest way to communicate with a process, so as long as applications use this method, they will work properly under PANTS. "Future Work" also describes ways that IPC support can be improved in PANTS.

Appendix B

A Sample Distributed Application

To illustrate an application that can benefit from the transparent remote execution of PANTS, we created the "bigsum" program. Although it looks like a significant amount of code to add up a sequence of numbers, each component is very simple. Note too, the lack of any PANTS-specific code or functions. The application must create a number of processes, but it is otherwise unaware of the operation of PANTS.

B.1 sum.c

This is the main computational component of bigsum. The `sum` program simply adds up a sequence of numbers and returns the result.

```
#include <stdio.h>

int main(int argc,char *argv[]) {
   long c,low,high,sum;

   if (argc<3) {
      fprintf(stderr,"sum <low> <high>\n");
      return -1;
   }

   low=atoi(argv[1]);
   high=atoi(argv[2]);

   sum=0;
   for (c=low;c<=high;c++) {
      sum+=c;
```

```
   }

   printf("%ld\n",sum);
   return 0;
}
```

B.2 multisum

This simple shell scripts takes a range of numbers, and executes multiple instances of `sum`. Because it executes multiple processes in the background, it is able to benefit from the transparent remote execution of PANTS.

```
#!/bin/bash

PROCS=$1
START=$2
STEP=$3
END=$4

CURRENT=$START;
until test $PROCS = 0; do
   let RANGE=$CURRENT+$STEP-1;

   if test $PROCS = 1; then
      ./sum $CURRENT $END &
      else
      ./sum $CURRENT $RANGE &
      fi

   let CURRENT=$CURRENT+$STEP
   let PROCS=$PROCS-1;
      done
```

B.3 total.c

`total` just adds up all of the results, and displays a single sum.

```
#include <stdio.h>

int main(int argc,char*argv[]) {
   long sum,n,t;

   if (argc<2) {
      printf("total <n>\n");
      return -1;
```

```
   }

   n=atoi(argv[1]);

   sum=0;
   while (n--) {
      scanf("%ld",&t);
      sum+=t;
   }
   printf("%ld\n",sum);
}
```

B.4 bigsum

The `bigsum` script verifies the command line parameters, and pipes the output of `multisum` to `total`.

```
#!/bin/bash

PROCS=$1;
START=$2;
END=$3;

let STEP=$END-$START;
let STEP=$STEP/$PROCS;

if test $STEP -lt 2; then
   echo You don\'t need that many processes for this sequence\!
   exit
   fi

echo Using $PROCS processes to sum from $START to $END

./multisum $PROCS $START $STEP $END | ./total $PROCS
```

Bibliography

[1] Amnon Barak, Avner Braverman, Ilia Gilderman and Oren Laden. "Performance of PVM with the MOSIX Preemptive Process Migration Scheme." *Proc. 7th Israeli Conf. on Computer Systems and Software Engineering*, June 1996.

[2] Amnon Barak and Oren Laden. "The MOSIX Multicomputer Operating System for High Performance Cluster Computing." `http://www.cs.huji.ac.il/mosix`

[3] Tim Bynum. `http://wallybox.cei.net/dipc/`

[4] *GNU Project - Free Software Foundation (FSF).* `http://www.gnu.org/software/software.html`

[5] Bill Gropp, Ewing Lusk. *The Message Passing Interface (MPI) standard.* `http://www-unix.mcs.anl.gov/mpi/index.html`

[6] Erik Hendriks. *BPROC: Beowulf Distributed Process Space.* `http://www.beowulf.org/software/bproc.html`

[7] Kamran Karimi. "DIPC." `http://www.gpg.com/DIPC/`

[8] Phil Merkey. *Beowulf Project at CESDIS.* `http://cesdis.gsfc.nasa.gov/linux/beowulf/beowulf.html`

[9] Jeffrey Moyer. *PANTS Application Node Transparency System.* `http://segfault.dhs.org/ProcessMigration/`

[10] Eduardo Pinheiro. *EPCKPT - A Checkpoint Utility for Linux Kernel.* `http://www.cs.rochester.edu/u/edpin/epckpt/` *(Mirror site)*

[11] *PVM: Parallel Virtual Machine.* `http://www.epm.ornl.gov/pvm/pvm_home.html`

[12] Thomas L Sterling. *How to Build a Beowulf.* U.S.A.: The MIT Press, 1999.

[13] Craig E. Wills and David Finkel. "Scalable Approaches to Load Sharing in the Presence of Multicasting." *Computer Communications*, 18(9):620-630, September 1995.