FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

PERFORMANCE IMPROVEMENT IN A GRID COMPUTING ENVIRONMENT USING CONDOR AND BEOWULF CLUSTERS

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bangalore Rao Guruprakash

2001

To: Dean Arthur W. Herriott
College of Arts and Sciences

This thesis, written by Bangalore Rao Guruprakash, and titled Performance Improvement in a Grid Computing Environment using Condor and Beowulf Clusters, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

Xudong He

Shu-Ching Chen

Nagarajan Prabakar, Major Professor

Date of Defense: May 09, 2001

The thesis of Bangalore Rao Guruprakash is approved.

Dean Arthur W. Herriott
College of Arts and Sciences

Dean Douglas Wartzok
Division of Graduate Studies

Florida International University, 2001

DEDICATION

I dedicate this thesis to my parents for their love, encouragement and guidance without which the completion of this work would not have been possible.

ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. Nagarajan Prabakar, for having faith and confidence in my abilities. I would also like to thank him for the direction and motivation that he constantly provided.

I would like to thank the Director of the Center for Advanced Systems Engineering and committee member, Dr. Xudong He, for providing me with the facilities to implement and work on my thesis. I would also like to acknowledge the constant feedback that I received from my committee member, Dr. Shu-Ching Chen.

I would like to thank Dr. Laird Kramer from the Physics Department, for his time, insight and providing us with a real engineering application to test our systems. I would also like to thank Eric Johnson, System Administrator for the Computer Science department, and John Flynn from the Systems Support Group, for them valuable assistance in setting up the systems for testing.

I would like to thank my friends, Vinay, Anil, Nikhil, Mhatre, Suresh, Humberto and Tom who were always patient, understanding and helpful. And, a special thanks to Smitha for her love and encouragement.

ABSTRACT OF THE THESIS

PERFORMANCE IMPROVEMENT IN A GRID COMPUTING ENVIRONMENT

USING CONDOR AND BEOWULF CLUSTERS

by

Bangalore Rao Guruprakash

Florida International University, 2001

Miami, Florida

Professor Nagarajan Prabakar, Major Professor

Advances in the field of Measurement have made it possible for scientists to collect a wide variety and quantity of experimental data. Often these data sets are of the order of a few Mega/Giga bytes and processing these data sets could take an extremely long time and would warrant the use of special and/or additional resources. Often engineering problems, some of which that do not necessarily have such huge data sets, can still be computationally intensive and require the use of computing resources for an extended period of time. To meet such resource requirements, we make use of the Grid Computing Environment (or the Globus) where in the users (geographically dispersed) share their resources. Although our system is a node on the Globus, the efficiency of resource utilization needs to be enhanced at this node. To achieve this, we have come up with a three-layer model in which the Globus is the topmost layer. In the middle layer, we have a Condor system, which hunts for unused or idle resources on the network and utilizes them to process jobs. At the bottom layer, we have the resource pool, which consists of a variety of resources that can be utilized to run jobs. To further improve the efficiency, in the bottom layer, as a part of the Condor Network resource pool, we implement a Beowulf cluster. Our tests have shown positive results that prove that the three-layered framework can be used to achieve significant improvement in resource management.

TABLE OF CONTENTS

CHAPTER PAGE

1 Introduction 1

2 Background Study 3

2.1 The Grid Computing Environment 3

2.2 The Condor System 6

2.3 The Beowulf Clusters 11

2.4 The Message Passing Interface (MPI) 15

3 Design 19

3.1 The 3-Layer Model 19

3.2 The Globus Resource Management Service 20

3.3 The Condor local resource management system 21

3.4 The Scyld Beowulf Cluster 27

3.5 Summary 31

4 Tests and Results 33

4.1 Testing without Beowulf, Condor and Globus 34

4.2 Testing with Beowulf but without Condor and Globus 35

4.3 Testing with Beowulf and Condor but without Globus 38

4.4 Testing with Beowulf, Condor and Globus 41

4.5 Conclusion 45

5 Applications 47

5.1 Monte Carlo simulation 47

5.2 GeneWise 48

5.3 Wise2 49

5.4 BLAST 49

6 Conclusion.....51

References 52

Appendix 54

LIST OF TABLES

TABLE PAGE

Table 1. A view of the Condor Pool Status 24

Table 2. A view of the Condor job Queue 25

Table 3. The Condor command file 26

Table 4. A sample program to execute on a single machine 34

Table 5. A sample MPI program for the cluster 35

Table 6. Output of sample program running on cluster 36

Table 7. Time taken for the MPI program to run on the cluster 37

Table 8. Command file for the non-parallel program 39

Table 9. Command file for the parallel program 39

Table 10. Time taken for the MPI program to run on condor without the cluster 40

Table 11. A sample parallel program for submitting via Globus 43

Table 12. Makefile for the Globus parallel program 43

Table 13. RSL file for the Globus parallel program 44

LIST OF FIGURES

FIGURE PAGE

Figure 1. A Layered view of the Globus 4

Figure 2. A real-life Globus application [5] 5

Figure 3. Architecture of a Condor Pool [20] 7

Figure 4. The Condor System Overview 9

Figure 5. The Beowulf Cluster Layout 12

Figure 6. The Three-Layer Model 19

Figure 7. Globus Resource Management Service 20

Figure 8. An Active Condor Pool [20] 22

Figure 9. The Scyld Beowulf Cluster Layout 28

Figure 10. The Beowulf Cluster Status (non-graphical) 29

Figure 11. The Beowulf Cluster Status (graphical) 30

Figure 12. The Beowulf Cluster setup tool 30

Figure 13. The test bed 33

Figure 14. Testing with Beowulf Cluster 37

Figure 15. Testing with Condor but without Beowulf Cluster 40

Figure 16. Testing with Globus, Condor and Beowulf Cluster 45

Figure 17. The CLAS detector [18] 47

1 Introduction

There are many important scientific applications and problems that require large amounts of computational power over a long period of time. For many scientists, the quality of their research depends heavily on the computing throughput. Scientists involved in this type of research need a computing environment that delivers large amounts of computational power over a long period of time. Such an environment is called a High Throughput Computing (HTC) environment [1]. And the key to high-throughput is the efficient use of available resources.

In the past, the scientific community used to rely mostly on large mainframe computers to do their computational work. However, since mainframe computers were not easy to afford, a large number of individuals and groups were forced to share this resource. Scientists would have to wait for their turn on the mainframe, and would have only a certain amount of time allotted to them. This would force them to limit the size of their problems to fit in their allocated compute time. While this environment was inconvenient for the users, it was a very efficient utilization of resource, since the mainframe was busy nearly all the time.

In the last few years, computers have become smaller, faster and cheaper. Though these computers are not as powerful as the mainframes, it is quite affordable to have a number of such computers for individuals or a small group to work on, whenever they want to. This is an environment of distributed ownership, where individuals throughout the organization utilize their own resources. The total computational power of the organization as a whole might rise dramatically as a result of such a change, but the resources available to the individual users remained roughly the same. While this environment is more convenient for the users, it is also much less efficient since many machines sit idle for long periods of time while their users are busy doing other things [2]. By the use of a good

local resource management system, it is possible to convert these unutilized computational resources and turn them into a High Throughput Computing (HTC) environment.

In general, the types of users who benefit from HTC environments are:

- Users whose computational requirements exceed the capabilities of their own hardware.
- Users whose computational time requirements cannot be satisfied by a single Workstation.

For users whose computational requirements exceed the capabilities of their own hardware, a local resource management system, like the Condor System [4], can provide for a good resource sharing among the users. However, if the resources required are not available locally, then a Grid Computing Environment, like the Globus [5] can provide resources from outside the local domain.

For users whose computational time requirements are cannot be satisfied by a single Workstation, a cluster of workstations, like the Beowulf Cluster [6], can provide a high computational power.

By combining the Globus, the Condor and the Beowulf Cluster, it is possible to meet the requirements of the users identified above. Since performance is a criterion, we have come up with an architecture that will improve the efficiency in a Grid Computing Environment. We have also identified and established a criterion for job submission and remote resource request in such an environment.

2 Background Study

Local resource management systems and cluster of workstations are an interesting topic and lot of research and development has already been done. Of particular interest are the Grid Computing Environment, the Condor local resource management system and the Beowulf clusters. All of these can be used in a High Throughput Computing environment to better utilize the local and foreign resources. In the next few sections, these technologies are discussed briefly.

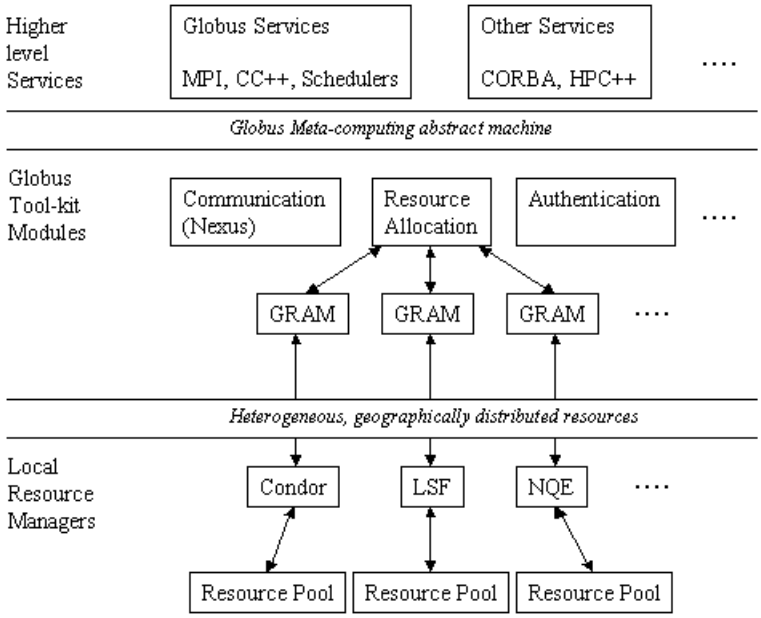
2.1 The Grid Computing Environment

Computational Grids open up access to globally available resources to people who require them. State of the art and emerging scientific applications require fast access to large quantities of data and fast computational resources. Both resources and data are often distributed in a wide-area network with components administered locally and independently. Computations may involve hundreds of processes that must be able to acquire resources dynamically and communicate efficiently.

New applications based on high-speed interaction among users, computers, databases, instruments, etc., are fast emerging. Online Instruments, use of remote software, data intensive computing, distributed super computing, very large scale simulation and high throughput computing are a few examples of these new kinds of applications. In order to provide a solution to such applications, the Grid computing environment makes it possible to seamlessly integrate various kinds of resources and computational powers. Globus provides a framework for such a computing infrastructure.

Globus provides an infrastructure that integrates geographically distributed computational and information resources for high performance applications. People who need access resources and people who have resources collaborate by sharing resources for particular periods of time. Each user

on the Grid has a public key and needs to authenticate once during the initiation of the computations that will acquire resources, use resources and then release resources.

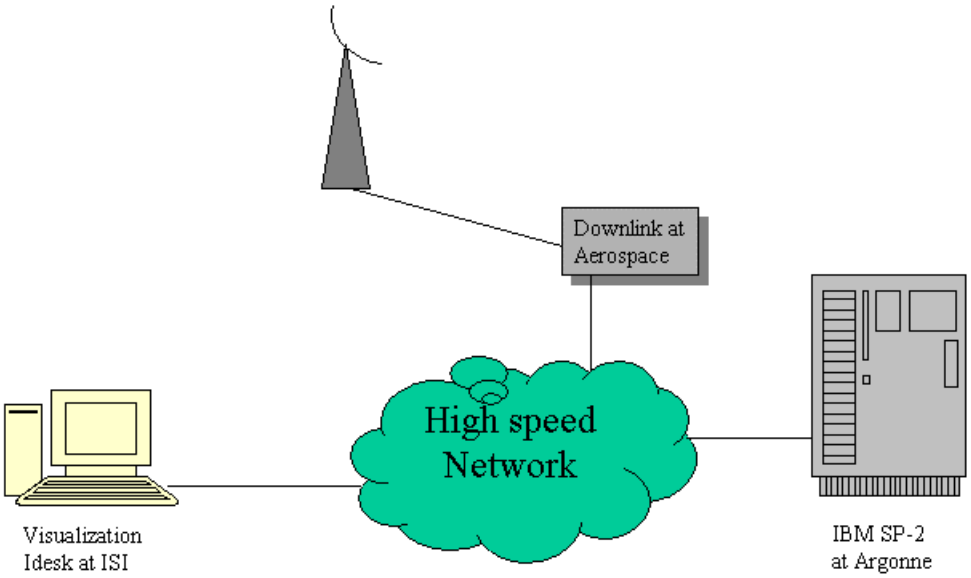


A layered view of the Globus

Figure 1. A Layered view of the Globus

Globus is a layered architecture in which high-level global services are built on top of an essential set of core local services. At the bottom of this layered architecture, the Globus Resource Allocation Manager (GRAM) provides the local component for resource management [5]. Each GRAM is responsible for a set of resources operating under the same site-specific allocation policy, often implemented by a local resource management system, such as Load Sharing Facility (LSF) or Condor. For example, a single manager could provide access to the nodes of a parallel computer, a cluster of

workstations, or a set of machines operating within a Condor pool [5]. Thus, a computational grid built with Globus typically contains many GRAMs, each responsible for a particular "local" set of resources.



This figure shows a networked supercomputing system used during the I-WAY experiment for real time analysis of data from a meteorological satellite. The satellite downlink is located in El Segundo, California, the supercomputer at Argonne, Illinois and the visualization engine and graphics device in Los Angeles.

Figure 2. A real-life Globus application [5]

A real-life application involving data from a meteorological satellite, super-computer and visualization software is shown in the figure above. It can be noted that the satellite is a specialty instrument, i.e., not available locally, from which data is collected. Hence, given sufficient access privileges, it is possible to connect to specialty equipment using Globus.

Applications such as application web portals and problem solving for computational chemistry require high throughput computing environments. Globus can be used for managing such jobs. It is possible to schedule many such independent tasks using Globus. Depending on the resource requirements for each task, the jobs are routed to the appropriate nodes for execution. Also, depending on the complexity and the requirements of the problem, multiple domains could be involved.

The advantages and flexibility that Globus provides can sometimes become issues for concern. The resource pool for the Globus is dynamic. This means that resources may not be always available for use. Also, since the resources are widely dispersed, resource discovery becomes an issue.

Globus involves remote job submission. This brings in the issue of security and accounting. These jobs will have to be monitored and closely controlled to eliminate the threat of any malicious interference by the currently executing task. Computational Grid Systems may require any or all of the standard security functions, including authentication, access control, integrity, privacy and nonrepudiation [23].

2.2 The Condor System

Condor is a software system that runs on a cluster of workstations with an intention to harness wasted CPU cycles. It achieves a High Throughput Computing environment by managing a very large collection of distributively owned workstations. Typically, a Condor pool consists of any number of machines, of possibly different architectures and operating systems that are connected by a network.

Condor achieves a high throughput, by providing two important functions. First, it makes the available resources more efficient by finding idle machines and putting them to work [3]. Second, it expands the resources available to a given user, by functioning well in an environment of distributed ownership.

Condor's development was motivated by the ever-increasing need of scientists and engineers to harness the capacity of the large collections of distributively owned workstations. Its environment is based on a novel layered architecture that enables it to provide a powerful and flexible suite of Resource Management services to sequential and parallel applications.

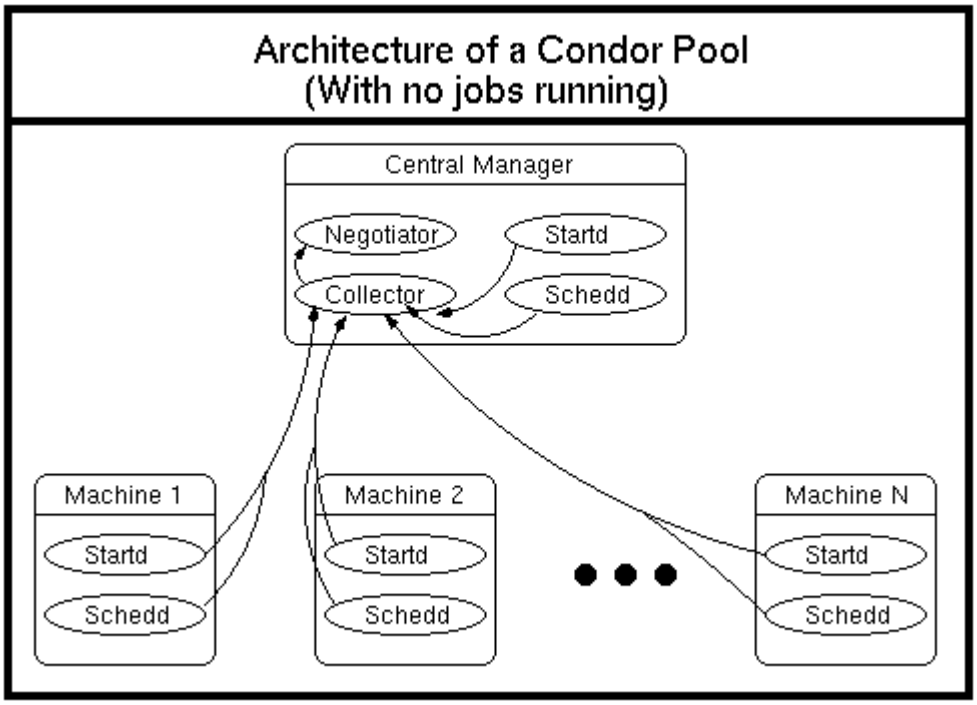


Figure 3. Architecture of a Condor Pool [20]

Figure 3 shows the architecture of an idle Condor pool. The Condor runs programs called the Condor *daemons*, which run on all the systems in its pool. Each of these daemons performs certain specific tasks that are needed for the Condor system to run effectively. These daemons are also used to monitor the status of the individual computers in a cluster.

On every system in the Condor pool, a daemon called the *master* makes sure that the rest of the daemons on that machine are running. If any of the other daemons dies, the master detects it and tries to restart it. But if the daemon continues to die, the master sends an e-mail to the Condor administrator and stops trying to start it.

Additionally, two other daemons run on every machine in the pool, the *startd* and the *schedd*. Since the Condor uses only the idle machines to run jobs, the *startd* daemon is used to determine whether a machine is idle and available to run a Condor job. The *startd* daemon monitors the information about the machine such as keyboard and mouse activity, and the load on the CPU. The *startd* daemon also notices when a user returns to the machine and removes the job from that machine. A daemon called *schedd* keeps track of all the jobs that have been submitted on a given machine.

A set of tools, besides the daemons, is also available on the Condor. These tools are programs that are used to help manage jobs and monitor their status, monitor the activity of the entire pool, and gather information about jobs that have been run in the past.

Some of the important mechanisms involved in the Condor system are Matchmaking, Check pointing, Migration, Remote system calls and Sandboxing. Matchmaking is used to match the resource requests with the available (or published) resources. It uses the idea of Class Ads in which the resource availability and resource requests are published globally. The resource that best matches

the job's resource requirements is selected and the job is submitted to that machine for execution. The *collector* and *negotiator* daemons keep track of the Class Ads by the resources and the jobs. Figure 4 shows these daemons and gives an overview of a Condor system. It also shows important mechanisms and illustrates the working of Condor.

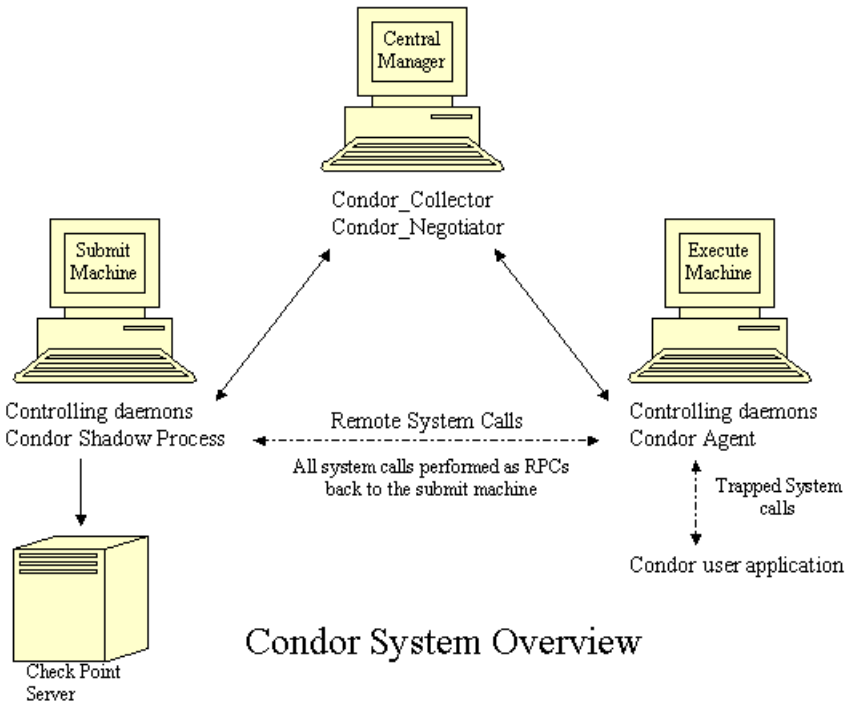


Figure 4. The Condor System Overview

Check pointing mechanism is used to save the state of an application (that is being executed), for every given interval of time. This ensures that if a job execution is interrupted, then the execution can be resumed from the last check point rather than having to start the execution all over again. This is an important mechanism since it ensures that work loss is minimal.

Process Migration is done sometimes when an application requests additional resource or when the application execution has been interrupted. It could also be because of certain user policies that would restrict the time for which the process is executed. The process is migrated to another machine that is idle and best matches the resource requirements for the application.

A submit machine in a Condor pool, is a machine which submits the job to the Condor's central manager. This job will be placed on the Condor queue before being scheduled for execution on any other machine. The machine on which the Condor job is executed is called the execute machine.

Remote system calls are issued among the Condor machines when an application requests additional resources. An agent is created on the execute machine that traps all system calls. If a system call cannot be satisfied by the local machine, then a remote system call is made to the submit machine. All remote system calls are performed as RPCs (Remote Procedure Calls).

Some of the important features of the Condor are –

- Better Resource utilization
- Provides access to non-local resources
- Well suited for parallel and distributed applications
- Makes use of wasted resources in the network
- Provides great flexibility in selecting resources to run a job
- Uses check-pointing mechanism that ensures that work is not lost.

However, it does have its shortcomings. The resource pool is dynamic and since Condor hunts for idle machines, there could be few or none available during certain periods of time every day. So, once

a job is submitted to the Condor, it would be difficult to estimate the job's completion time. Also, since the jobs could be submitted to virtually any machine in the pool, the issue about security arises. Hence jobs have to be monitored and accounted for before submitting it to the job queue.

2.3 The Beowulf Clusters

The Beowulf-class parallel machine evolved from early work in low cost computing. The first work in this area centered around clusters of workstations [10]. These clusters are often built using existing workstations that are used as interactive systems during the day, can be heterogeneous in composition, and rely on extra software to balance the load across the machines in the presence of interactive jobs.

As it became obvious that workstations could be used for parallel processing, several research teams began to build dedicated machines from inexpensive, non-proprietary hardware. These "Pile-of-PCs" consists of a cluster of machines dedicated as nodes in a parallel processor, built entirely from commodity off the shelf parts, and employing a private system area network for communication [11]. The Beowulf workstation's availability of most system software source encourages customization and performance improvements. Experiments have shown that Beowulf workstations are capable of providing high performance for applications in a number of problem domains.

A Beowulf cluster is a kind of high-performance massively parallel computer built primarily out of commodity hardware components, running a free-software operating system like Linux or FreeBSD, interconnected by a private high-speed network. It consists of a cluster of PCs or workstations dedicated to running high-performance computing tasks. The nodes in the cluster are dedicated to running cluster jobs. It is usually connected to the outside world through only a single node called the master.

Beowulf clusters used mainly to run jobs in parallel. The cluster provides dedicated machines that can handle computation intensive jobs and jobs that require co-ordination among multiple processes. The Figure 5 below shows a simple layout of a Beowulf cluster. The master node has two network interfaces. One of the interfaces is used to connect it to the external network. The other interface is used to connect to the slaves through a common switch.

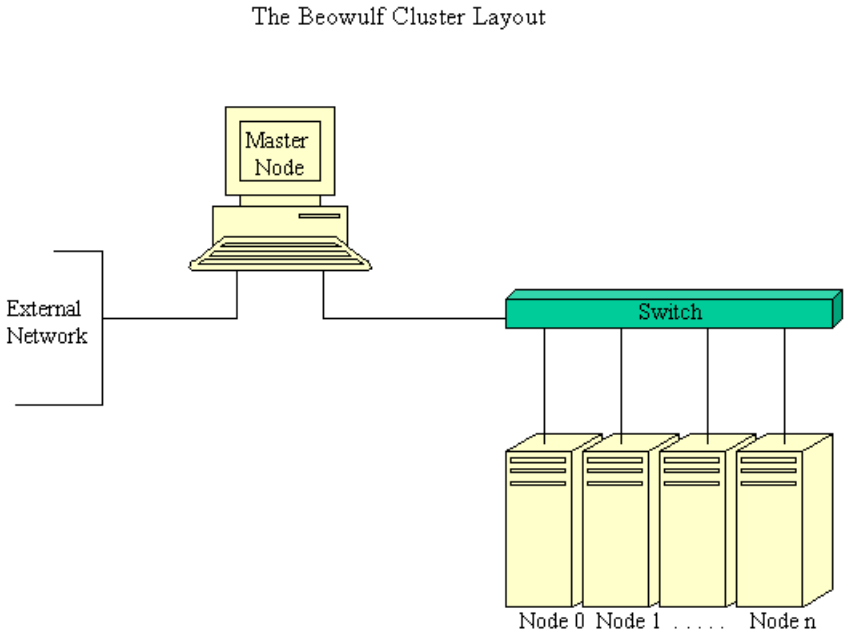


Figure 5. The Beowulf Cluster Layout

A Beowulf class cluster computer is distinguished from a Network of Workstations (NOWs) by several subtle but significant characteristics. First, the nodes in the cluster are dedicated to the cluster. This helps ease load balancing problems, because the performance of individual nodes are not

subject to external factors. Also, since the interconnection network is isolated from the external network, the network load is determined only by the application being run on the cluster. This eases the problems associated with unpredictable latency in NOWs. All the nodes in the cluster are within the administrative jurisdiction of the cluster. For examples, the interconnection network for the cluster is not visible from the outside world so the only authentication needed between processors is for system integrity. On a NOW, one must be concerned about network security. Another example is the Beowulf software that provides a global process ID. This enables a mechanism for a process on one node to send signals to a process on another node of the system, all within the user domain. This is not allowed on a NOW. Finally, operating system parameters can be tuned to improve performance. For example, a workstation should be tuned to provide the best interactive feel (instantaneous responses, short buffers, etc), but in cluster the nodes can be tuned to provide better throughput for coarser-grain jobs because they are not interacting directly with users [13].

There are a few technicalities that one has to be aware of while building a Beowulf cluster. The master node has to be booted up first where the Beowulf software has already been installed. Later the slave nodes have to be booted from a boot disk (or from any source that has the initial Beowulf boot image). The slave node boot up consists of three phases [14]. By manipulating the sequences of process or the processes themselves, that occur during the various boot phases, it is possible to change the behavior of the cluster. For example, it is possible to alter the place and permissions on the master's root file systems that are mounted.

Phase 1: - Initial (Floppy) boot

Phase 1 is the initial boot up of the slave machine from the initial (floppy) image. This image may be stored either on a floppy disk or in the BeoBoot partition of the node's hard drive. The processes that occur during this phase are:

1. The BIOS loads a sector from the slave node boot image.
2. The boot loader on the floppy (or hard disk) takes over and loads the rest of the data stored in the initial image.

The slave node initial boot image contains a minimal kernel image and an initial ramdisk image. These images probe the PCI for network hardware, configure network interfaces and download the final kernel image and ramdisk that the machine will run. The final image and ramdisk will be started via a *Two Kernel Monte* [14].

Phase 2: - Final Kernel - initrd

In phase 2, the slave node is running the final kernel image, which was downloaded in phase 1. The root file system is the ramdisk image downloaded during phase 1. This image contains all the kernel modules for this final kernel. The PCI probe will load all relevant drivers at this time. The ramdisk image contains a smaller image, which will be used as the permanent root file system. The boot program for this phase takes this smaller ramdisk image and copies it into one of the /dev/ramX devices.

Phase 3: - Final Kernel – ramdisk Root

In phase 3, the linuxrc would have exited and the new smaller root file system would have been mounted. The init program used is *boot*. In this capacity, it starts the BProc slave daemon and waits for it to exit. If the slave daemon dies for any reason, the init program will reboot the system.

Beowulf clusters provide a cost effective alternative to supercomputers. However, they have to be setup properly to utilize their full potential. They provide a high throughput computing environment

and hence they have to be used for critical applications like real-time applications, high priority applications or compute intensive applications.

2.4 The Message Passing Interface (MPI)

The Message Passing Interface (MPI) Forum has been meeting since 1992 and is comprised of high performance computing professionals from over 40 organizations. [12]. Their goal is to develop a message passing interface that meets the needs of the majority of users in order to foster the use of a common interface on the ever growing number of parallel machines. By separating the interface from the implementation, MPI provides a framework for Massively Parallel Processors (MPP) vendors to utilize in designing efficient commercial implementations.

MPI is a software system that allows users to write message-passing parallel programs that run on a cluster, in Fortran and C. MPI (Message Passing Interface) is a defacto standard for portable message-passing parallel programs standardized by the MPI Forum and is available on all massively parallel supercomputers.

Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. Although there are many variations, the basic concept of processes communicating through messages is well understood. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm. Each vendor has implemented its own variant. More recently, several systems have demonstrated that a message passing system can be efficiently and portably implemented. It is thus an appropriate time to try to define both the syntax and semantics of a core of library routines that will be useful to a wide range of users and efficiently implementable on a wide range of computers.

The main advantages of establishing a message-passing standard are portability and ease-of-use. In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message passing routines. The benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard, such as the mechnism proposed here, provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

This standard is intended for use to those who want to write portable message-passing programs in Fortran 77 and C. This includes individual application programmers, developers of software designed to run on parallel machines, and creators of environments and tools. In order to be attractive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user while not semantically precluding the high-performance message-passing operations available on advanced machines.

The attractiveness of the message-passing paradigm at least partially stems from its wide portability. Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all of these. In addition, shared-memory implementations are possible. The paradigm will not be made obsolete by architectures combining the shared- and distributed-memory views, or by increases in network speeds. It thus should be both possible and useful to implement this standard on a great variety of machines, including those *machines* consisting of collections of other machines, parallel or not, connected by a communication network.

The advantages of the Message Passing model are [7]:

- Flexibility

  The message-passing model can be easily implemented on processors that are connected by a communication network. Thus, it can be used on parallel supercomputers (that have multiple processors inside) and also by a network of workstations and dedicated PC clusters. Hence, they are quite flexible and the same code can be implemented on distributed or parallel processors.

- Expressive

  Message passing is well suited for expressing parallel algorithms. It provides a better control for data-parallel and compiler-based models in dealing with data locality. Also, this model can be used in programs that can be made tolerant of the imbalance in process speeds found on shared networks and for scheduling algorithms. In short, they are well suited for formulating parallel algorithms and load balancing.

- Ease of debugging

  Debugging of parallel programs can be a difficult task since they could be hard to analyze. Usually, multiple processes communicate between themselves and manipulate shared memory. This can be quite confusing as the number of processes increase. The most common cause of error in the shared memory model is the unexpected overwriting of memory. But in the message-passing model, controlling memory references more explicitly minimizes these kinds of memory read-write errors.

- Performance

  Message passing provides a way for the programmer to explicitly associate specific data with processes and thus allows the compiler and cache-management hardware to function fully.

Hence performance of the message-passing model increases since it uses the cache and memory on the CPUs to do run jobs or processes.

There are over a hundred functions available in the mpi library. But most programs use as few as six basic mpi functions. Some of the important mpi functions are MPI_Init, MPI_Finalize, MPI_Send, MPI_Recv, MPI_Comm_Rank, MPI_Comm_Size. The MPI forum maintains the documentation for the MPI functions and can be found at their website [21]. Many different versions of MPI are available. For all MPI programs, we have used the beoMPI version from the Scyld Computing Corporation, as it provides some additional support for the Beowulf clusters.

3 Design

3.1 The 3-Layer Model

With the goal of improving the efficiency of end nodes in a Grid Computing Environment, we have come up with a 3-layer model design. In this model, the top-most layer is the Globus, or the Grid Computing Environment. The middle layer consists of the local resource management system. The bottom layer consists of a resource pool for the Condor. In our case, the middle layer is the Condor system and the bottom layer consists of a combination of Workstations and Beowulf clusters.

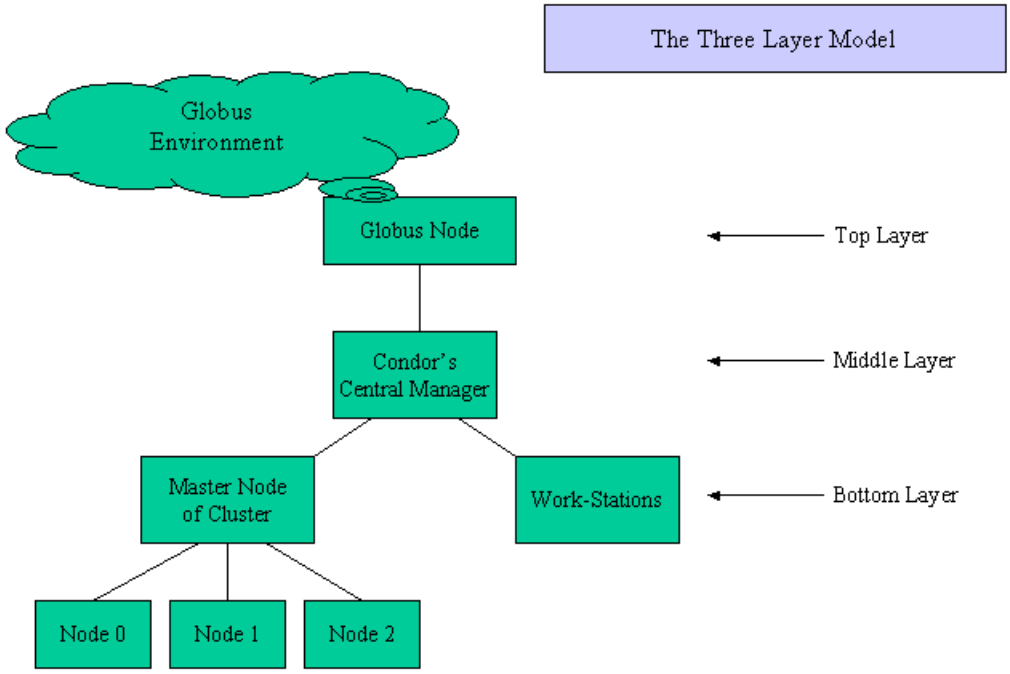


Figure 6. The Three-Layer Model

Figure 6 shows the layout for the 3-layer model. The Globus node provides interfaces for the Internet Protocol, the Nexus Communication [25] and the Globus Resource Allocation Manager

(GRAM) Protocol. Of particular interest is the GRAM, since it provides us with a way to effectively implement a local resource management system that could be tailored to improve the efficiency of our resource management. A brief discussion of the components in each of the layers that make up this 3-layer model is discussed in the following sections.

3.2 The Globus Resource Management Service

An important feature of the Globus is that it distinguishes between local services, which are kept simple to facilitate deployment, and global services, which are constructed on top of local services and may be more complex. This feature provides us with the flexibility to choose a convenient local resource management system.

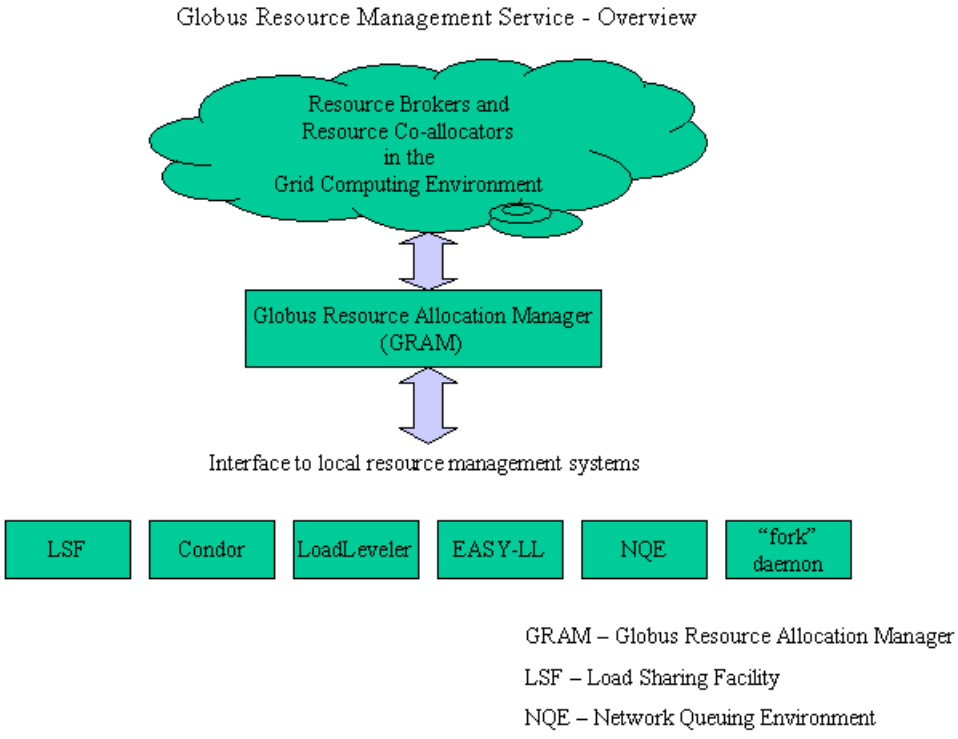


Figure 7. Globus Resource Management Service

The above figure illustrates how the high-level Globus global services are layered on top of the local resource management systems. As shown above, Globus Resource Allocation Manager provides an interface for the global and local services to interact with each other. Currently, the GRAM can operate in conjunction with six different local resource management tools: Network Queuing Environment (NQE), EASY-LL, LSF, LoadLeveler, Condor, and a simple "fork" daemon. Within the GRAM API, resource requests are expressed in terms of an extensible Resource Specification Language (RSL) [24].

RSL is used throughout the Globus architecture as a common notation for expressing resource requirements. Resource requirements are expressed by an application in terms of a high-level RSL expression. A variety of *Resource Brokers* (as shown in the figure above), implement domain-specific resource discovery and selection policies by transforming abstract RSL expressions into progressively more specific requirements until a specific set of resources are identified.

The *Resource Co-allocators* (as shown in the figure above), co-allocate resources, ensuring that a given set of resources is available for use simultaneously. The *Resource Co-allocators* break the RSL into pieces and distribute to various GRAMS. The GRAMS in turn contact the local resource management system that is responsible for managing the local resources.

We have used a Condor system as our local resource management system. The way Condor handles the resource requests and its advantages are discussed in the next section.

3.3 The Condor local resource management system

Condor has several advantages that led us to choose it as our local resource management system. One of its major advantages is that the Condor's resource pool can have a variety of systems, such as

a Network of Workstations (NOWs), and/or dedicated machines for computation, such as a Beowulf Cluster. In a computation intensive and job intensive environment, Beowulf clusters are most sought off since these are characteristically dedicated resources for computation. Hence, the load on these machines is usually high and would result in some of the machines being put in the job queue.

In such a scenario, a Condor system would hunt for alternate resources that are available for computation and thus reducing the waiting time for queued jobs.

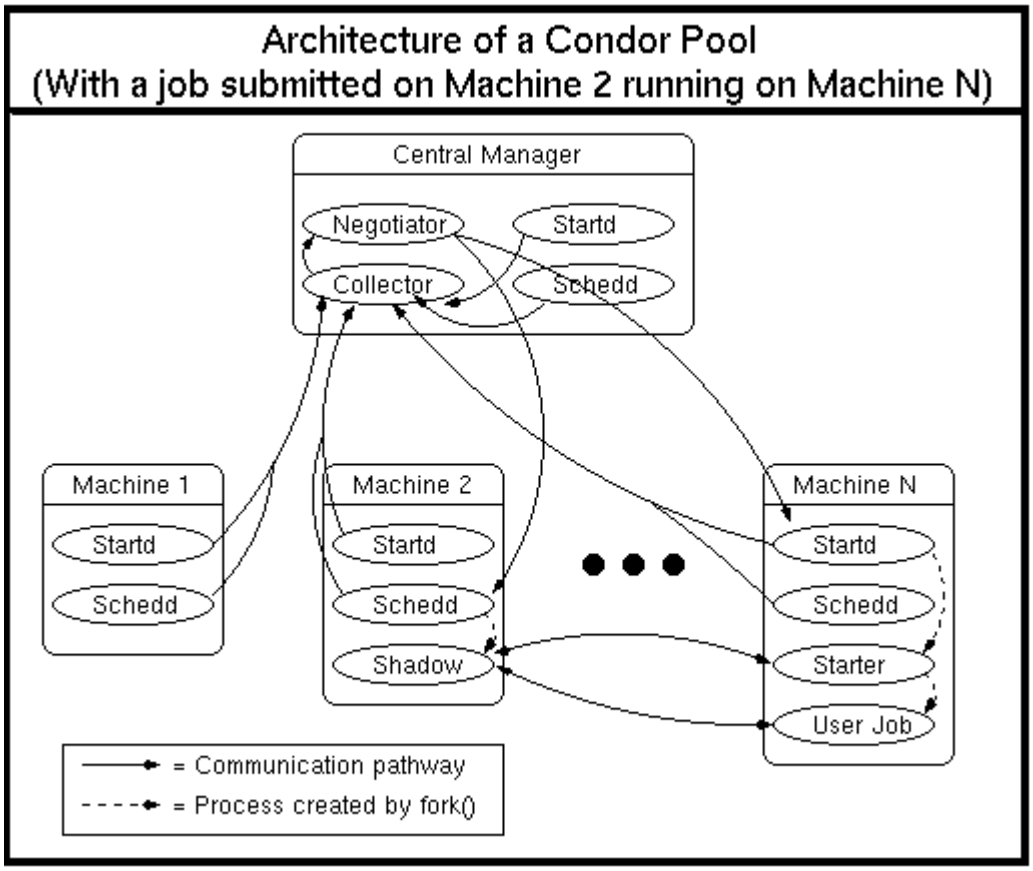


Figure 8. An Active Condor Pool [20]

From the figure above, it can be seen that every Condor job involves three machines. The machine on which the job is submitted is the submitting machine. The Central Manager is the second machine. The machine on which the job runs is the executing machine. The Central Manager finds an idle machine for executing the job and takes the job from the submitting machine and puts it on to the idle machine. The executing machine need not be just a single machine. It could be multiple machines depending on the job's life and requirements.

Condor operates on the fact that every job has a set of resource requirements and every machine in the resource pool has a set of resources that it can offer. The Condor system comes up with a match between the resources requested and the resources available. The match making could involve a lot of additional considerations to be taken into account such as the job owner's preferences for the executing machine's resources. The job owner might prefer to have a certain set of resources available while the job is executing, but these might not be absolutely essential for the execution of the job. If no machines that match the preferences are available, the job will be assigned to the best matched machines.

The properties of the computing resources are reported to the central manager by the *startd* on each machine in the pool. The *negotiator's* task is not only to find idle machines, but machines with properties that match the requirements of the jobs, and if possible, the job preferences.

When a match is made between a job and a machine, the Condor daemons on each machine are sent a message by the central manager. The *schedd* on the submitting machine starts up another daemon, called the *shadow*. This acts as the connection to the submitting machine for the remote job, the shadow of the remote job on the local submitting machine. The *startd* on the executing machine also creates another daemon, the *starter*. The starter actually starts the Condor job, which involves

transferring the binary from the submitting machine (as shown in the figure above). The starter is also responsible for monitoring the job, maintaining statistics about it, making sure there is space for the checkpoint file, and sending the checkpoint file back to the submitting machine (or the checkpoint server, if one exists). In the event that its owner reclaims a machine, it is the *starter* that vacates the job from that machine [3].

The status of the machines in the Condor pool can be determined by using the *condor_status* command. The table below shows the output for the c*ondor_status* command.

[root@matrix sbin]# condor_status

Name	OpSys	Arch	State	Activity	LoadAv	Mem	ActvtyTime
matrix.cs.fiu	LINUX	INTEL	Owner	Idle	0.590	61	0+00:35:12
cnode.cs.fiu.	LINUX	INTEL	Owner	Idle	0.000	61	0+00:30:04

	Machines	Owner	Claimed	Unclaimed	Matched	Preempting
INTEL/LINUX	2	2	0	0	0	0
Total	2	2	0	0	0	0

Table 1. A view of the Condor Pool Status

The above table shows the status of all the machines in the Pool. It can be seen that there are two machines in the pool. Both machine – matrix.cs.fiu.edu and cnode.cs.fiu.edu, were idle when the *condor_status* command was executed. *Condor_status* is a good way of monitoring the status of the condor pool. There are other utility commands in Condor that help monitor the status of jobs in the pool. To view the jobs that are placed in the Condor job queue, we could use the command *condor_q*.

The output for the *condor_q* command is shown in the table below.

guru@matrix.cs.fiu.edu:~/condor-6.2.0/examples 21% condor_q

-- Submitter: matrix.cs.fiu.edu : <131.94.126.147:1610> : matrix.cs.fiu.edu

ID	OWNER	SUBMITTED	RUN_TIME	ST	PRI	SIZE	CMD
1.0	guru	5/5 15:17	0+00:00:00	I	0	0.0	sh_loop 60
2.0	guru	5/5 15:29	0+00:00:00	I	0	2.6	io.remote 200
3.0	guru	5/5 15:29	0+00:00:00	I	0	2.6	env.remote foo bar
4.0	guru	5/5 15:29	0+00:00:00	I	0	2.8	fstream.remote
5.0	guru	5/5 15:29	0+00:00:00	I	0	2.6	loop.remote 200
5.1	guru	5/5 15:29	0+00:00:00	I	0	2.6	loop.remote 200
5.2	guru	5/5 15:29	0+00:00:00	I	0	2.6	loop.remote 300
5.3	guru	5/5 15:29	0+00:00:00	I	0	2.6	loop.remote 300
5.4	guru	5/5 15:29	0+00:00:00	I	0	2.6	loop.remote 500
6.0	guru	5/5 15:29	0+00:00:00	I	0	2.6	registers.remote
7.0	guru	5/5 15:29	0+00:00:00	I	0	2.7	reader.remote
8.0	guru	5/5 15:29	0+00:00:00	I	0	2.7	printer.remote
9.0	guru	5/5 15:29	0+00:00:00	I	0	2.7	fortIO.remote
10.0	guru	5/5 15:29	0+00:00:00	I	0	0.0	sh_loop 60

14 jobs; 14 idle, 0 running, 0 held

Table 2. A view of the Condor job Queue

In the above table, it can be seen that each job in the job queue has an unique ID. It is possible to submit a variety of jobs to the condor. For each job submitted, we have to specify a command file, which is a collection of the specifications and requirements for a given job. Here it is possible to specify whether a job has to be run in the default environment or in parallel environments, such as MPI, PVM, LAM, etc. A simple command file for a job is shown in the table below. This file can also be found in the examples sub-directory from the Condor installation directory.

```
guru@matrix.cs.fiu.edu:~/condor-6.2.0/examples 26% more loop.cmd

## Test Condor command file

## Specify the executable file and the input/output files here
executable      = loop.remote
output          = loop.$(Process).out
error           = loop.$(Process).err
log             = loop.log

## Specify the arguments for the executable and the number of instances
## of the executable that has to be run here
arguments       = 200
queue 2

## Specify a different set of arguments for the executable and the number
## of instance of the executable that has to be run here.
arguments       = 300
queue 2

## Possible to further customize the input/output files for each process.
arguments       = 500
output          = loop.last.out
error           = loop.last.err
queue
```

Table 3. The Condor command file

As it can be seen from the above table, Condor provides great flexibility for job submission. To submit a parallel job the following line has to be added to the command file.

environment = mpi

This will ensure that the job is run using the mpi environment. We could further make sure that the job is submitted to the Beowulf cluster by specifying it as the *requirements* variable value in the command file.

The resource specification language (RSL) of the Globus is similar to the condor's command file. Hence if a job is submitted through Globus then the Globus's RSL file can be automatically converted using scripts, into the Condor's command file.

Thus, in our 3-Layer model, it is possible to submit jobs that are either parallel or non-parallel. If the job is parallel or if it has a high priority, it can be sent to the Beowulf cluster for execution. If the job is non-parallel, then other machines in the pool are checked to see if they are idle, before submitting it to the cluster. Thus the cluster is kept free for executing parallel jobs and high priority jobs.

3.4 The Scyld Beowulf Cluster

Beowulf clusters used mainly to run jobs in parallel. The cluster provides dedicated machines that can handle computation intensive jobs and jobs that require co-ordination among multiple processes.

Many vendors provide the software for running and managing Beowulf clusters. We have used the Scyld Beowulf Release 27bz-7 (based on Red Hat Linux 6.2) from the Scyld Computing Corporation. This can be used to build a cluster of machines running on the Linux Operating system.

A brief overview of the capabilities and features of the Scyld Beowulf clusters are discussed and it is shown how the Condor's central manager can make use of these facilities to run and monitor jobs on the cluster.

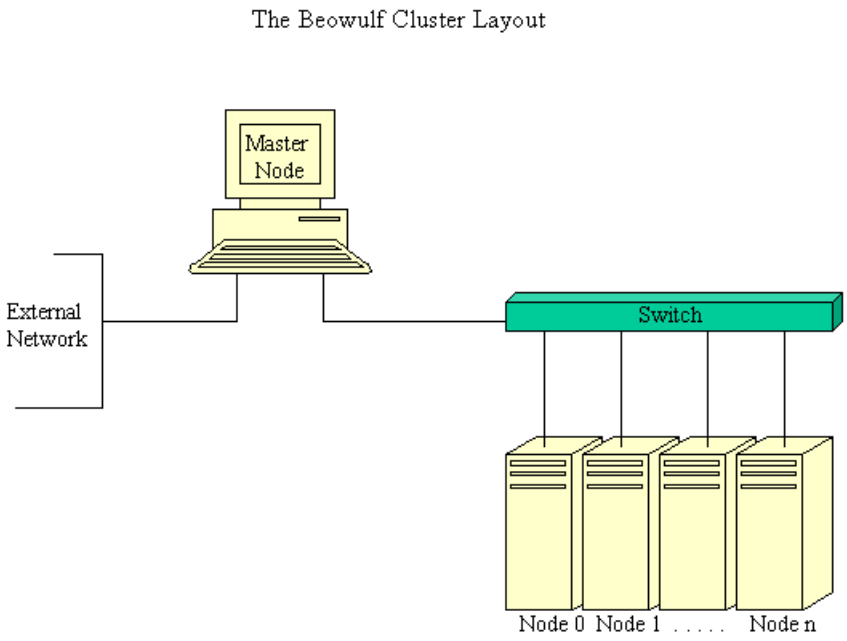


Figure 9. The Scyld Beowulf Cluster Layout

The general layout of the Scyld Beowulf Cluster is shown in the figure above. The master node has two network interfaces one of which connects to the external network and the other connects to the slaves through a common switch. The slave nodes are dedicated to the master node. This means that only the master node and not the external network can see the slave nodes. Similarly, the slave nodes can only see the master node and not the external network. This is useful since the slave provide the

high throughput computing ability and it becomes important not to put any network traffic on these slave nodes that could affect their performance.

The slave nodes mount the file system of the master node. Hence they will be able to use the files on the master to run a task. Depending on who submits a job (the root or the user), the permissions are transferred to the slave nodes. Hence for a job started by a user on the master node, the slave nodes will also have the same access permission to the master's filesytem as that of the user. This is an important security issue and is handled well in the Scyld's Beowulf implementation.

The status of the nodes in the Scyld Beowulf cluster can be determined by using the command *bpsh.* There is also a command, *beostatus,* which is a graphics utility that will display the status of the slave nodes in the cluster.

```
guru@matrix.cs.fiu.edu:~ 1% bpstat
Node    Address       Status  User    Group
0       192.168.1.100 down    any     any
1       192.168.1.101 up      any     any
2       192.168.1.102 up      any     any
3       192.168.1.103 down    any     any
```

Figure 10. The Beowulf Cluster Status (non-graphical)

The *beostatus* tool shows the important parameters such as the CPU usage and the network load. It reads data from the *bproc* process that manages the Beowulf cluster and displays it in a graphical form.

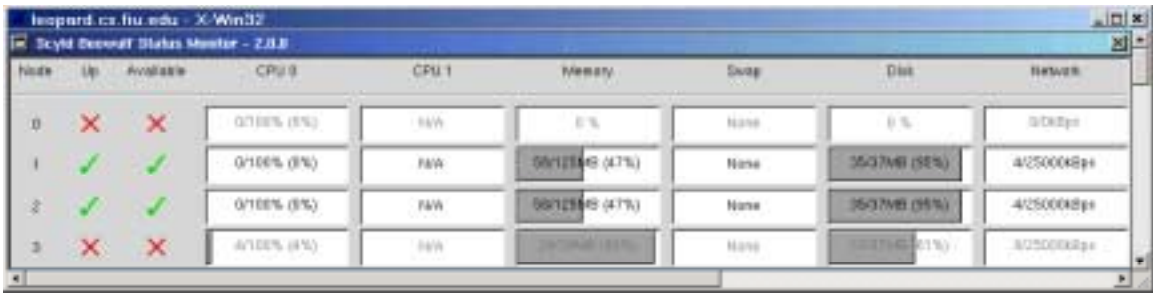


Figure 11. The Beowulf Cluster Status (graphical)

The slave nodes in the cluster can be configured using the *beosetup* tool. The *beosetup* tool reads from a config file that contains all the information about the slaves. It is possible to manually edit this file to alter the configurations for the slave nodes.

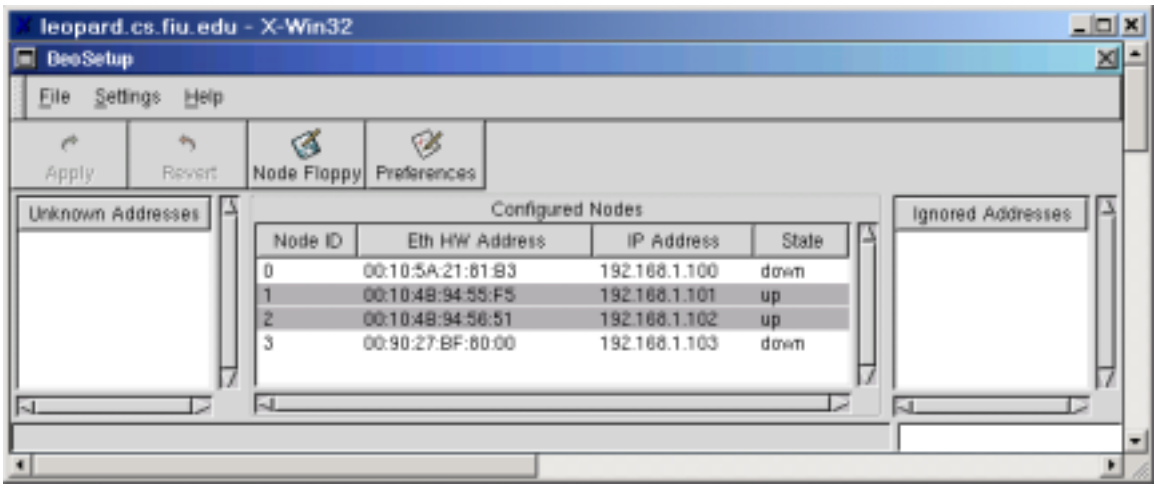


Figure 12. The Beowulf Cluster setup tool

The Scyld Beowulf Distributed Process Space (BProc) is a set of kernel modifications, utilities and libraries that allows a user to start processes on other machines in a Beowulf-style cluster. Remote processes started with this mechanism appear in the process table of the front-end machine in a cluster. This allows remote process management of slave nodes using the normal UNIX process

control facilities. Signals are transparently forwarded to remote processes and exit status is received using the usual wait() mechanisms.

BProc also provides process migration mechanisms for the creation of remote processes. These mechanisms remove the need for most binaries on the remote nodes.

In the 3-Layer Model, when the Condor central manager passes on a job to the Beowulf cluster, the job is run using the BProc. There are two types of jobs that could be submitted to the cluster by the Condor manager. One, a parallel job that can be run on the cluster using the mpirun command and the other, a job that is not parallel. For jobs that are not parallel, the Scyld Beowulf cluster runs them using the mpprun command. This command is used to run programs, which have not been parallelized internally.

Thus, the Condor's central manager can submit any kind of job to the cluster and get the results back. Ideally, real-time applications and applications that require high throughput should be placed on the cluster and batch jobs should be given to other machines in the condor pool.

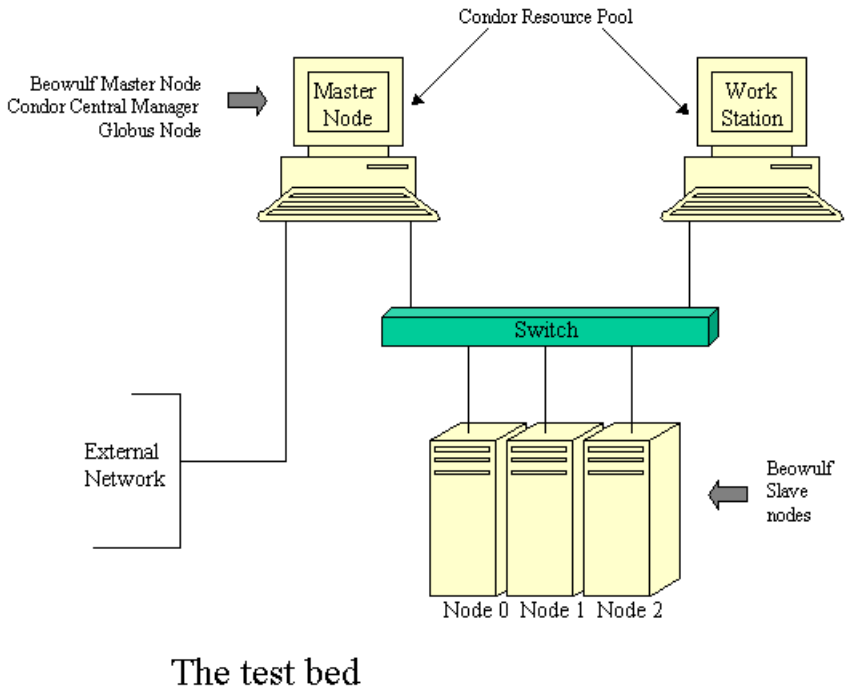
3.5 Summary

After the above discussions on the Grid, the Condor and the Beowulf clusters it can be seen that when these three technologies are put together, they will be able to provide the end user with a lot of flexibility and computing power. The 3-Layer model defines a strategy in which these technologies can be brought together to make use of the advantages of each of them and to eliminate their drawbacks.

We have successfully implemented the 3-Layer model using the facilities in the Center for Advanced Distributed Systems Engineering (CADSE) labs at the School of Computer Science in FIU. Engineering and Scientific programs from various departments and organizations were run on this setup and tested for their performance. The next chapter discusses the tests and results obtained during the performance-testing phase. An overview of the applications that were run on this setup is discussed in the subsequent chapter.

4 Tests and Results

The 3-Layer model was tested on a test bed of five machines. Test programs were written in the C programming language using the MPI libraries for parallelizing the code. The figure below shows the layout of the test bed.



The test bed

Figure 13. The test bed

The test bed consists of a three Beowulf slave nodes connected to a switch. The master node of the Beowulf cluster is also the Central Manager for the Condor. This master node is also Globus enabled. The master node has two network interfaces. One of its interfaces connects to the external network and the other connects to the switch forming an internal network. There is also a Work

Station on the internal network. This workstation has Condor installed on it. The Master Node (along with its slave nodes) and the Work Station forms the Condor Resource Pool. The testing is done on the internal network that is not exposed to the external network traffic.

The testing was divided into four test cases:

1. Without Beowulf, Condor and Globus.
2. With Beowulf but without Condor and Globus.
3. With Beowulf and Condor but without Globus.
4. With Beowulf, Condor and Globus.

4.1 Testing without Beowulf, Condor and Globus

A simple program to output the result "Hello World" was written and the time taken to execute this process measured. The intention here was to get a measure of the time required by the node to do the preprocessing before the actual output was written to the screen.

The following is a small program that will help achieve the result.

```
#include <stdio.h>
int main(){
        printf("Hello World from matrix.cs.fiu.edu");
        return 0;
}
```

Table 4. A sample program to execute on a single machine

A small script (using the time command in Unix) was also written, to measure the time taken to execute this program. The average time taken for running this program measured by the script was **0.11 seconds** on a single machine.

4.2 Testing with Beowulf but without Condor and Globus

The program to print “Hello World” is extended so as to generate a number of processes and each process prints out its rank number. This program is written in MPI, and executed on the Beowulf cluster.

The following is the program that will help achieve the result.

```
#include <stdio.h>
#include "mpi.h"

int main( argc, argv )
int  argc;
char **argv;
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    printf( "Hello world from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Table 5. A sample MPI program for the cluster

The program was executed with 8 threads for 500 times. To reduce the noise, the network was isolated from the cs.fiu.edu domain and also a sleep time of 10 seconds was given between each invocation of the program. A sample output for the program is shown on the following page.

```
[root@matrix hello]# mpirun --np 8 --log-thresh progress ./hello
argv[5] (./hello) is not an option
mpirun option processing stopped at ./hello
scheduling the job
calling bproc_numnodes
bproc says there are 3 nodes

permission match (superuser)
node 0 is 0

permission match (superuser)
node 1 is 1

node 2 is not up (status is 0)

the system has 2 nodes, 3 cpus
allocating a schedule with job size 8
placing rank 0 on the frontend
placing rank 0 on node -1
placing rank 1 on node 1
placing rank 2 on node 0
placing rank 3 on node 1
placing rank 4 on node 0
placing rank 5 on node 1
placing rank 6 on node 0
placing rank 7 on node 1
spawning the job: argc=1
Using p4 spawner

generating process group file
Writing line: 1 1 ./hello
Writing line: 0 1 ./hello
Writing line: 1 1 ./hello
Writing line: 0 1 ./hello
Writing line: 1 1 ./hello
Writing line: 0 1 ./hello
Writing line: 1 1 ./hello

Done writing scheduler file
running MPI program
environ is 0x8049870, __environ is 0x8049870
P4_PG=/proc/self/fd/3
environ is 0x8049950, __environ is 0x8049950
calling execv to run ./hello
Hello world from process 0 of 8
Hello world from process 6 of 8
Hello world from process 2 of 8
Hello world from process 4 of 8
Hello world from process 7 of 8
Hello world from process 3 of 8
Hello world from process 1 of 8
Hello world from process 5 of 8
```

Table 6. Output of sample program running on cluster

The following is a table of the results.

Time measured in seconds.	
Avg Preprocessing Time	7.483435
Avg Total Time	8.157763
Avg Runtime	0.674329

Table 7. Time taken for the MPI program to run on the cluster

A graph of the above results is plotted to give a better understanding of the results.

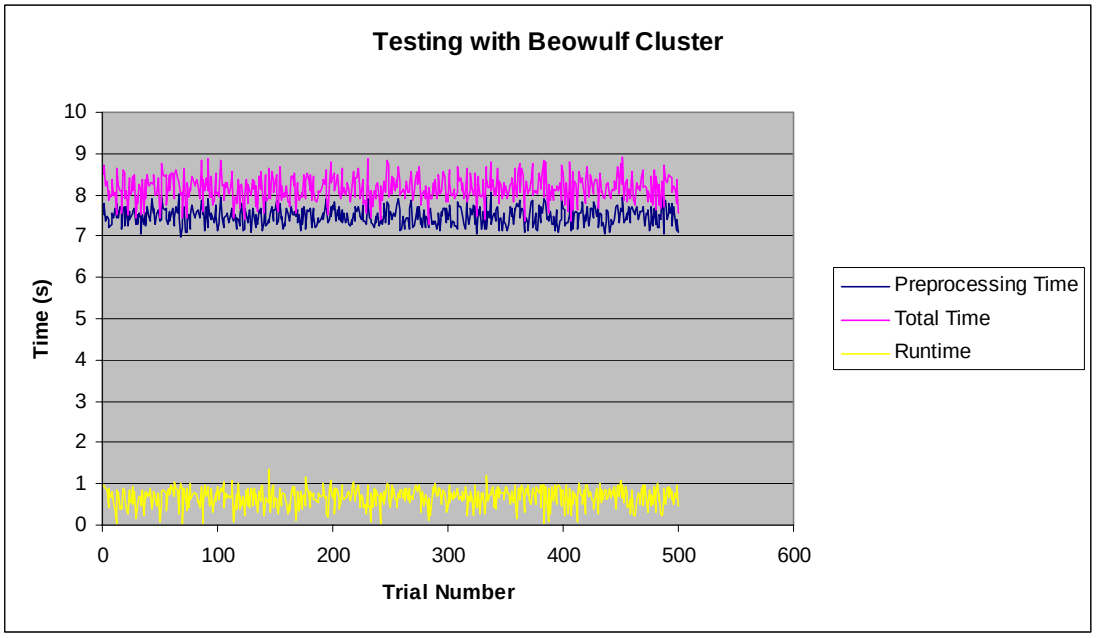


Figure 14. Testing with Beowulf Cluster

We can make an important observation from the above results. The average runtime for the above program is roughly 0.67 seconds whereas the pre-processing time is roughly 7.48 seconds. This means that the pre-processing takes an extremely long time compared to the actual runtime. However, the pre-processing time is the time taken to create processes and allocate it to the nodes on the cluster. This means that given the number of processes that have to be created, the pre-processing time will be fairly constant.

It can also be noted that the actual runtime for the program improved when run on the cluster. The program showed an actual runtime of 0.11 x 8 seconds (for running it 8 times) when run on a single machine, where as it showed a runtime of 0.67 seconds (for running 8 processes) on the cluster. This is an improvement of about 25%.

So, we can conclude that if the actual runtime for a job takes more than the pre-processing time, then we stand to benefit by submitting the job on to the cluster for execution. In our case, if the runtime for a job is greater than, say 8 seconds, then it makes sense to use the cluster for running the job.

4.3 Testing with Beowulf and Condor but without Globus

Both the parallel and the non-parallel versions of the simple program, shown in Tables 4 and 5 respectively were used for this test case. But this time, instead of submitting the job directly to the Beowulf cluster, it was submitted to the Condor's central manager. The Command files for the two programs had to be written separately.

The table below shows the command file for the non-parallel version of the program.

```
# name of the application to be run
executable      = hello

# specify the output, error and the log files
output          = hello.$(Process).out
error           = hello.$(Process).err
log             = hello.log

# specify the number of instances of the program to be run
queue 8
```

Table 8. Command file for the non-parallel program

The table below shows the command file for the parallel version of the program.

```
# specify the parallel environment that has to be used
universe        = mpi

# name of the executable to be run
executable      = /usr/bin/mpirun

# specify the output, error and the log files
output          = hello.$(Process).out
error           = hello.$(Process).err
log             = hello.log

# specify the arguments to be used to run the application in parallel
arguments       = -np 8 ./hello
priority        = 1
machine_count   = 3
```

Table 9. Command file for the parallel program

The program was executed with 8 threads for 500 times. To reduce the noise, the network was isolated from the cs.fiu.edu domain and also a sleep time of 10 seconds was given between each invocation of the program.

The output for the program was as follows:

Time measured in seconds	
Avg Total time	13.81223
Avg Preprocessing time	10.29474
Avg Runtime	3.517488

Table 10. Time taken for the MPI program to run on condor without the cluster

A graph of the above results is plotted to give a better understanding of the results.

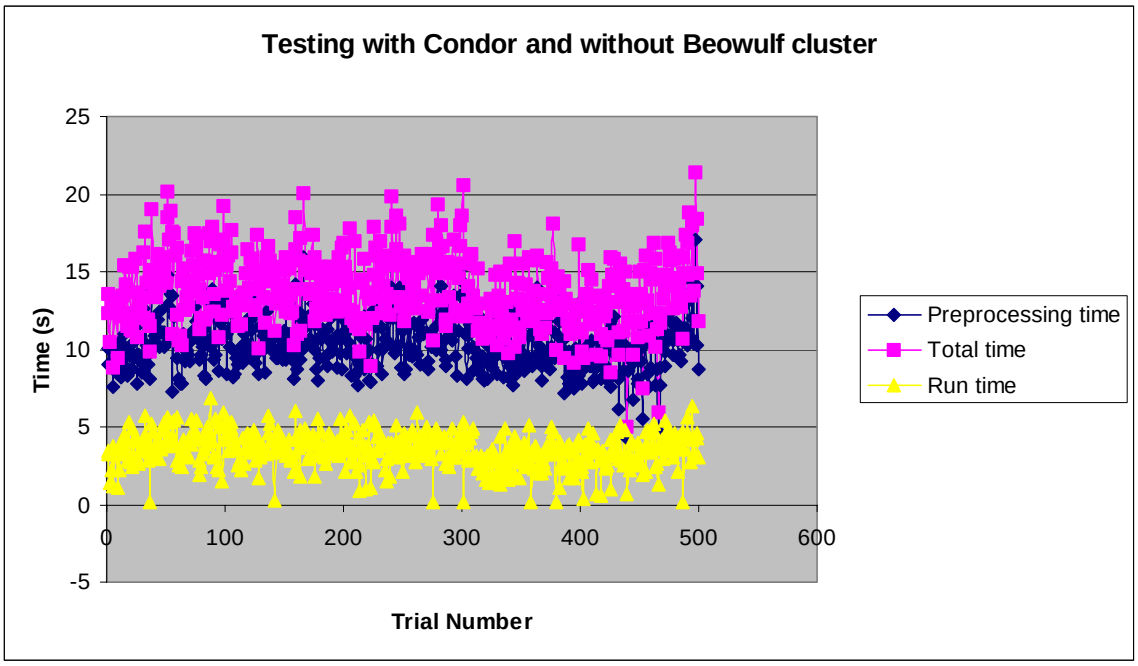


Figure 15. Testing with Condor but without Beowulf Cluster

The jobs were placed on the job queue before being submitted to the appropriate machine. When the cluster was available, the processing time for the application was found to be the same as in the previous section. But when the cluster was not available, the processing time increased.

When a job is submitted to the Condor, there are two cases. First – when the cluster is available, second – when the cluster is busy (not available). In the first case, the testing did not show any difference in values from the previous section. This was because as soon as a job was submitted to the Condor, it was passed on to the cluster. Thus it would look like the job had been directly submitted to the cluster. Hence, it showed no significant changes in the runtime values.

We carefully study the second case, as it is more interesting due to the lack of resources (cluster not available). The runtime for the job increased considerably in this case, but this was because the cluster was not available. If we were to rely only on the cluster, then the job would never have been completed. But our 3-Layer model gives us the flexibility to run the job even when the resources are not fully available. Of course, the runtime is higher in this case, but it is a lot better than having to wait infinitely until the cluster becomes available.

This can be seen as one of the advantages of the 3-Layer model.

4.4 Testing with Beowulf, Condor and Globus

A parallel job was submitted to the Condor's Central manager through Globus.

This job makes use of the globus_duroc_runtime library (Dynamically-Updated Request Online Coallocator v0.3 [22]). MPI_Init has a Globus-enforced DUROC barrier that waits for *all processes*, across *all machines*, to be loaded and start execution before proceeding. If we are using MPICH

version of MPI, then it distills all communication (including collective operations) into its constituent point-to-point components before passing them on to the lower-level device. The globus device is therefore presented with only point-to-point communication requests. The choice of protocol (TCP or vendor-supplied MPI) is based on the source/destination, that is, vendor-supplied MPI for intra-machine messaging (assuming MPICH-G2 was configured with an "mpi" flavor of Globus) and TCP for all other messaging.

This means that if a MPI program is submitted through the Globus to another Globus (submit) node, then the MPI version used will be the one that is present at the Globus submit node. This gives a great flexibility in that, the submit node can have any vendor specified MPI package installed and still be able to run the program submitted via Globus. A small program, which prints “Hello World”, is shown below.

```
#include "mpi.h"
#include "globus_duroc_runtime.h"

int main(int argc, char **argv)
{
   MPI_Init(&argc, &argv);

   globus_module_activate(GLOBUS_DUROC_RUNTIME_MODULE);
   globus_duroc_runtime_barrier();
   globus_module_deactivate(GLOBUS_DUROC_RUNTIME_MODULE);

   printf("hello, world\n");

   MPI_Finalize();
}
```

Table 11. A sample parallel program for submitting via Globus

```
# Modify this file by:
#     1. set GLOBUSDIR to your Globus installation
#     2. set FLAVOR to one of the Globus flavor directories
#     3. select one of the two DEFINES and comment out the other

GLOBUSDIR = /depot/globus-1.1
FLAVOR    = i686-pc-linux-gnu_nothreads_standard_debug

# use 'DEFINES = -DVMPI' if you have specified an *_mpi_* FLAVOR of above
# otherwise use 'DEFINES ='

DEFINES   =

# The rest of the file should _not_ change.
include $(GLOBUSDIR)/development/$(FLAVOR)/etc/makefile_header
hello:
      $(CC) $(DEFINES) $(CFLAGS) $(GLOBUS_DUROC_RUNTIME_CFLAGS) -c hello.c
      $(LD) -o hello hello.o \
      $(LDFLAGS) \
      $(GLOBUS_DUROC_RUNTIME_LDFLAGS) \
      $(GLOBUS_DUROC_RUNTIME_LIBS) \
      $(LIBS) \
      -lmpi
clean:
      $(RM) -rf *.o hello
```

Table 12. Makefile for the Globus parallel program

The corresponding Makefile for the program in shown above. Care has to be taken to set the environment variables (for e.g. GLOBUS_INSTALL_PATH) before running the command “make hello”. To run the program using Gloubus, a RSL (resource specification language) file has to be created. This is shown in the table below.

```
+
( &
  (resourceManagerContact="matrix.cs.fiu.edu")
  (count=8)
  (label="linuxjob 0")
  (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0))
  (directory=/home/guru/globus)
  (stdout=/home/guru/globus/hello.out)
  (stderr=/home/guru/globus/hello.err)
  (executable=/home/guru/globus/hello)
)
```

Table 13. RSL file for the Globus parallel program

The output for the program was as follows:

Time measured in seconds	
Avg Total time	24.48343
Avg Preprocessing time	23.87297
Avg Runtime	0.61046

It can be seen that the preprocessing time required by the Globus is a lot higher, it doesn’t really affect the execution much, since we expect jobs that use this 3-Layer model to be of the order of a few hours. A notable observation is that the average runtime is **0.61 seconds** (same as in the second case). This implies that performance of the job completely depends on the resource pool at the

submit node of the Globus. We already showed in the previous sections how the performance is improved at the submit node of the Globus by using Condor and Beowulf clusters.

A graph of the above results is plotted to give a better understanding of the results.

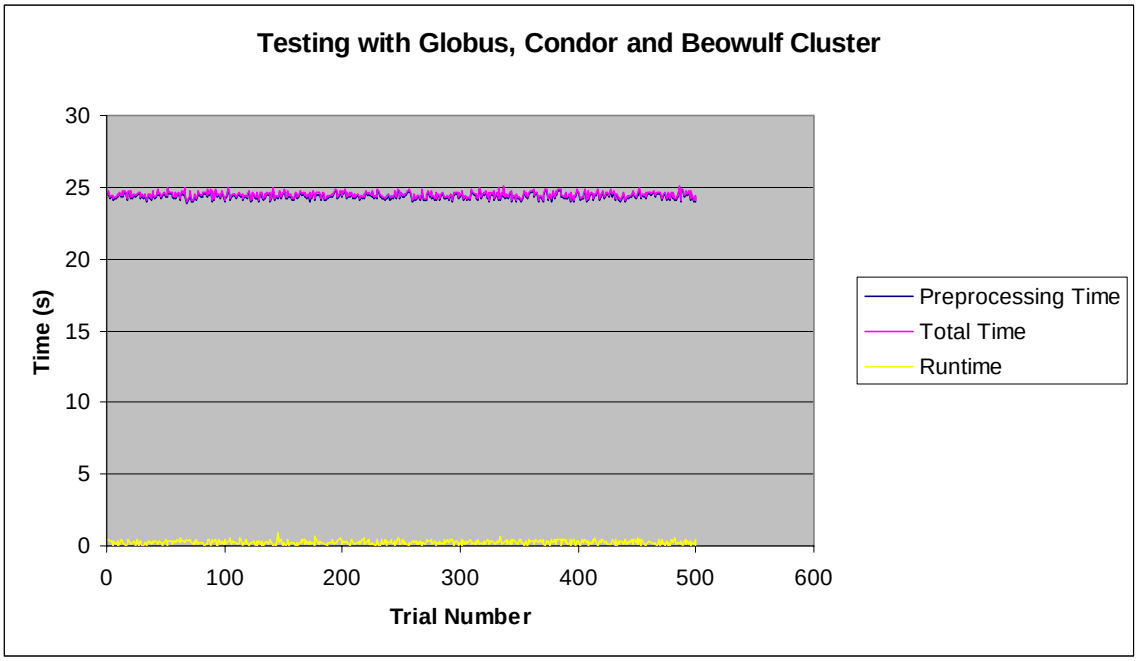


Figure 16. Testing with Globus, Condor and Beowulf Cluster

4.5 Conclusion

The 3-Layer model improves the performance and resource management at a Globus submit node. When a job is submitted via Globus, the Condor's central manager in the 3-Layer model will submit the job to the most powerful and ideal resource first (which is the cluster). If the cluster is not available then it submits the jobs to other available machines in the resource pool.

It can be seen from the test cases above, that the jobs running on the cluster take lesser time to execute. Also the cluster provides support for both parallel and non-parallel jobs. When the cluster is not available, the time taken to run the jobs on other available resources increases. But this is definitely better than having to wait till the cluster becomes available to process a job.

Hence there is a considerable improvement in the performance of the Globus node by using the 3-Layer model.

5 Applications

The 3-Layer model was developed with the intention to help scientists and engineers to reduce their wait time for their computational work. There are quite a few applications that we identified and tested on our 3-Layer Model. Some of the applications that were tested on the 3-Layer model setup are listed below.

5.1 Monte Carlo simulation

The Monte Carlo simulation of the CLAS detector [18] involves the prediction and calculation of the path taken by the nuclear elements inside a nuclear reactor. The figure below shows a reactor in which a high-speed electron is passed in and the path it takes.

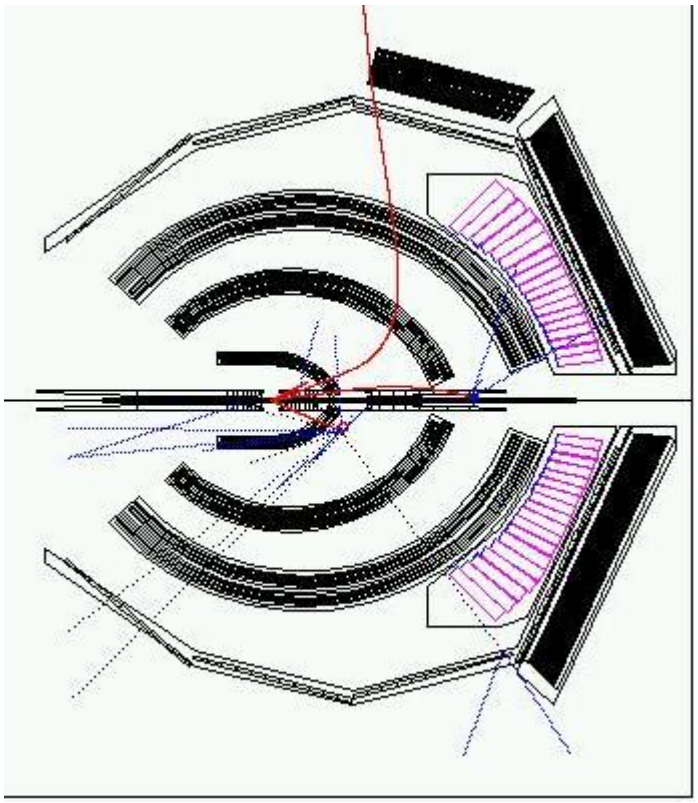


Figure 17. The CLAS detector [18]

The GSIM package uses the GEANT routines from the CERN libraries as a framework for a Monte Carlo simulation of the CLAS detector [18]. This package produces two executables: gsim_int, the interactive version and gsim_bat, the batch version. The Department of Physics at FIU uses this application for their experiments and computations.

This application is extremely computation intensive, as it has to predict the paths and simulate it for a large number of electrons. The computational requirement for this project prompted the study of the 3-Layer model and is the basic foundation for this thesis. This application was successfully run on the 3-Layer model setup.

5.2 GeneWise

The GeneWise algorithms combine statistical gene prediction methods with dynamic-programming searches which can compare protein based information to genomic DNA sequence, allowing for possible frame shifts and large intron gaps in the DNA sequence [19]. The merged algorithms compare a DNA sequence to a protein sequence on-the-fly parsing the gene structure and skipping introns while computing alignments.

The algorithms related to the GeneWise family have flexible computational load, and therefore produce different levels of accuracy in the biological model. More specifically, the GeneWise algorithms differ in the complexity of the gene parser used and in the consideration of how gaps in the protein alignment interact with introns.

The Genewise application package was installed and tested on the 3-Layer model. This application is quite suitable for the high-throughput computing environment provided by the 3-Layer model since it uses a dynamic programming algorithm

5.3 Wise2

Wise2 [15] is a package focused on a comparison of biopolymers, commonly DNA sequence and protein sequence. There are many other packages that do this, such as the BLAST package [17] and the Fasta package. Wise2's particular forte is the comparison of DNA sequence at the level of its protein translation. This comparison allows the simultaneous prediction of say gene structure with homology-based alignment. It uses the Smith-Waterman algorithm [16], among other algorithms. The Smith-Waterman algorithm is a means of searching protein databases to find those with the best alignment using dynamic programming.

Both parallel and non-parallel versions of the Wise2 package are available. Both versions were installed and successfully run on the 3-Layer model setup. The sub string-matching algorithm used here is a dynamic programming problem. Dynamic Programming problems compute the final solution based on the solutions of its sub-problems. Hence these are highly parallizable applications. Also, since the sub-string matching has to be done for a very long sequence, these applications consume a lot of time and computational power. These characteristics are quite ideal for the 3-layer model and they benefit by running on such a setup.

5.4 BLAST

BLAST (Basic Local Alignment Search Tool) is a set of similarity search programs designed to explore all of the available sequence databases regardless of whether the query is protein or DNA.

The BLAST programs have been designed for speed, with a minimal sacrifice of sensitivity to distant sequence relationships. The scores assigned in a BLAST search have a well-defined statistical interpretation, making real matches easier to distinguish from random background hits. BLAST uses a heuristic algorithm, which seeks local as opposed to global alignments and is therefore able to detect relationships among sequences that share only isolated regions of similarity [17].

This is again a sub-string matching application and is well suited for the High Throughput computing environment provided by the 3-Layer model. BLAST was successfully installed and tested on this setup. The application was submitted to the Condor Central manager through Globus. The sub-problems were computed on the cluster, or any available machine on the condor pool and put together by the main process running on the central node on the condor. This distribution of the processing improved the overall running time required for the application. Thus the 3-Layer model can be used for coputation intensive HTC jobs.

6 Conclusion

The 3-Layer model is meant for applications that need High Throughput Computing. It caters to their needs by providing a good resource management and a Beowulf cluster that co-ordinate the processing of jobs.

If the cluster is not available then the resource manager submits the jobs to other available machines in the resource pool. When the cluster is not available, the time taken to run the jobs on other available resources increases. But this is definitely better than having to wait till the cluster becomes available to process a job.

Hence there is a considerable improvement in the performance of the Globus node by using the 3-Layer model.

References

1. Miron Livny, Jim Basney, Rajesh Raman, Todd Tannenbaum, "*Mechanisms for High Throughput Computing*", SPEEDUP Journal, Vol. 11, No. 1, June 1997.

2. Miron Livny, Matt W. Mutka, "*The Available Capacity of a Privately Owned Workstation Environment*", Performance Evaluation, vol. 12, no. 4, pp. 269-284, July 1991.

3. Miron Livny, Matt W. Mutka, Michael Litzkow, "*Condor - A Hunter of Idle Workstations*", Proceedings of the 8th International Conference of Distributed Computing Systems, pages 104-111, June, 1988.

4. Miron Livny, D.H.J Epema, R. van Dantzig, X. Evers, Jim Pruyne, "*A Worldwide Flock of Condors: Load Sharing among Workstation Clusters*", Journal on Future Generations of Computer Systems, vol. 12, 1996.

5. Ian Foster, Carl Kesselman, "*Globus: A Metacomputing Infrastructure Toolkit*", International Journal of Supercomputer Applications, vol. 11, no. 2, pp. 115-128, 1997.

6. Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, Charles V. Packer, "*Beowulf: A Parallel Workstation for Scientific Computation*", Proceedings of the International Conference on Parallel Processing, 1995.

7. William Gropp, Ewing Lusk, and Anthony Skjellum – "*Using MPI : Portable Parallel Programming with Message-Passing Interface*", 2nd ed., ISBN 0-262-57134-X, QA76.642.G76 1999, The MIT Press.

8. William Gropp, Ewing Lusk, Rajeev Thakur – "*Using MPI-2: Advanced Features of the Message-Passing Interface*", ISBN 0-262-057133-1, QA76.642.G762 1999, The MIT Press.

9. Ian Foster, Carl Kesselman – "*The Grid: Blueprint for a New Computing Infrastructure*", ISBN 1-55860-475-8, 1999, Morgan Kaufmann Publishers Inc.

10. Karen Castagnera, Doreen Cheng, Rod Fatoohi, Edward Hook, Bill Kramer, Craig Manning, John Musch, Charles Niggley, William Saphir, Douglas Sheppard, Merritt Smith, Ian Stockdale, Shaun Welch, Rita Williams, and David Yip, *"Clustered Workstations and their potential role as high speed compute processors"*, Technical Report RNS-94-003, NAS Systems Division, NASA Ames Research Center, April 1994.

11. Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling, *"Beowulf: Harnessing the power of parallelism in a Pile-of-PCs"*, Proceedings of the 1997 IEEE Aerospace Conference, 1997.

12. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jck Dongarra, *"MPI: The complete reference"*, MIT Press, 1995.

13. World Wide Web document, *The Beowulf Project*, http://www.beowulf.org

14. World Wide Web document, *The Scyld Beowulf Reference Manual*, version 1.03, http://www.scyld.com/support/docs/

15. Ewan Birney, Richard Copley, *"Wise2 Documentation (version 2.1.20 stable)"*, Sanger Center, Wellcome Trust Genome Campus, Cambridge, England, July, 1999.

16. Smith TF, Waterman MS, *"Identification of Common Molecular Subsequences"*, Journal of Molecular Biology, 147(1): 195-7, March 1981.

17. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ, *"Basic local alignment search tool"*, Journal of Molecular Biology, 215(3): 403-10, October 1990.

18. World Wide Web document, *GSIM Info*, http://improv.unh.edu/maurik/gsim_info.shtml

19. World Wide Web document, *Compugen Products – Genewise*, http://www.cgen.com/products/genewise.htm

20. World Wide Web document, *Condor Overview*, http://www.cs.wisc.edu/condor/

21. World Wide Web document, *MPI Fourm*, http://www.mpi-forum.org

22. World Wide Web document, *Globus: Details: DUROC*, http://www.globus.org/duroc

23. Ian Foster, Carl Kesselman, G. Tsudik, S. Tuecke, *"A security Architecture for Computational Grids"*, Proceedings of the 5th ACM Conference on Computers and Communications Security Conference, pp 83-92, 1998.

24. Karl Czajkowski, Ian Foster, Nicholas Karonis, Carl Kesselman, Stuart Martin, Warren Smith, Steven Tuecke, *"A Resource Management Architecture for Metacomputing systems"*, Proceedings of the IPPS/SPDP '98 Workshop on Job scheduling Strategies for Parallel Processing, 1998.

25. Ian Foster, Carl Kesselman, Steven Tuecke, *"The Nexus Task Parallel Runtime System"*, 1st International Workshop on Parallel Processing, pp457-462, 1994.

Appendix

The following script determines the time taken to execute a program called *hello* on a Beowulf cluster and write it into a file called *runtimes.* For each measurement a sleep time of 10 seconds is given so as to eliminate the noise or other factors that could affect the runtime measurements.

```
#! /usr/bin/perl

# Authors:      Bangalore Guruprakash
#               Humberto R. Rivero

# this script runs issues a command to run several process in parallel,
# dumps the ouput of the command to a file called runtimes.proc,
# extracts the start and stop time, and dumps the results to screen or appends to a file
# called runtimes

# get command line ags, [0] is number of processes, [1] is how many times to loop
# if either argument is < 1, default values used
$np = $ARGV[0];
if( $np < 1 ) {
   $np = 8;
}
$loops = $ARGV[1];
if( $loops < 1 ) {
  $loops = 1;
}

# open the ouput file
open(OF,">>runtimes");

for($i = 1; $i <= $loops; $i++) {
  # issue command
  system("mpirun -np $np /root/mpi/hello/hello -p4dbg 10 > runtimes.proc");
  # now we extract the start and stop times from runtimes.proc
  open(IF,"runtimes.proc");
  @lines = <IF>;
  $found = 0;
  $starttime;
  $endtime;
  print OF ("----- JOB $i --------\n");
```

Contd.

```
  foreach $line (@lines) {
    # look for first occurence
    if( $line=~m/sent msg of type/ and $found == 0 ) {
      $found = 1;
      # extract time stamp
      @split_line = split(/ /,$line);
      $split_line[2]=~tr/()/ /;
      $starttime = $split_line[2];
      # write result to file
      print OF ("Start time of first sent: $starttime\n");
    } # end if

    # now for each occurence of "received type" store the time stamp
    # each subsequent occurence overwrites the previous, so at the end
    # we have captured the last occurence
    if( $line=~m/received type/ ) {
      @split_line = split(/ /,$line);
      $endtime = $split_line[2];
      $endtime=~tr/()/ /;
    } # end if
  } # end foreach

  print OF "Stop time of last received: $endtime\n";
  $time_elapsed = $endtime - $starttime;
  print OF "Elapsed time: $time_elapsed\n\n\n";
  close(IF);
  sleep(10);
} # end for
close(OF);
print ("done writing file\n");
exit 0;
```