Optimize Task Distribution in Grid Computing

A Thesis

Submitted to the Thesis supervisor

Syed Saiful Islam
Lecturer, BRAC University

By

Shyen Muhabbat Shikder

Student ID: 04101008

And

Md. Mahtab Uddin

Student ID: 04101002

In Partial Fulfillment of the

Requirements for the Degree

of

Bachelor of Science in Computer Science and Engineering

August, 2008

DECLARATION

I hereby declare that this thesis is based on the results found by myself. Materials of work found by other researcher are mentioned by reference. This thesis, neither in whole nor in part, has been previously submitted for any degree.

_____

Signature of

Shyen Muhabbat Shikder

_____

Signature of

Md. Mahtab Uddin

_____

Signature of Supervisor

Syed Saiful Islam
Lecturer, BRAC University

Acknowledgements

I would like to thank our thesis supervisor Saif Islam, Lecturer of BRAC University for his guidance and support throughout the thesis.

ABSTRACT

Grid computing is designed to use free cycles of computer to perform large calculations by using free cycles of computer. Grid computing does not need dedicated computers to perform calculations instead it uses free cycles of computer in a network. It works like a virtual super computer, but it doesn't involve extra hardware cost. As, it's a cost effective its popularity is increasing day by day. Now a day's popular applications are being made to support grid computing. For example Oracle database 10g is designed to support grid computing.

Although grid computing is so popular, but implementing software for grid based system is tough. It needs task distribution among computers. So it uses different programming techniques and needs to use special API. The software we are currently using may not support grid environment. Software companies need to convert their programs to support grid environment that involves development cost. On other hand users are not able to use their current software in grid environment. So, they need to buy another program that involves extra cost.

The objective of this project is to run traditional programs in grid system without any modification, so that we can run any executable program in grid system with parallel speeding performance. It will increase program compatibility and reduce cost.

Contents

List of Tables

Table Index: **page**

List of Figures

Figure Index: **page**

CHAPTER 1

Introduction

1.1 Background

Increased network bandwidth, more powerful computers, and the acceptance of the Internet have driven the on-going demand for new and better ways to compute. Commercial enterprises, academic institutions, and research organizations continue to take advantage of these advancements, and constantly seek new technologies and practices that enable them to seek new ways to conduct business. However, many challenges remain. Increasing pressure on development and research costs, faster time-to-market, greater throughput, and improved quality and innovation are always foremost in the minds of administrators - while computational needs are outpacing the ability of organizations to deploy sufficient resources to meet growing workload demands.

On top of these challenges is the need to handle dynamically changing workloads. The truth is, flexibility is key. In a world with rapidly changing markets, both research institutions and enterprises need to quickly provide compute power where it is needed most. Indeed, if systems could be dynamically created when they are needed, teams could harness these resources to increase innovation and better achieve their objectives. That is why grid computing is getting popular. It gives user the flexibility of processing, upgradeability and reliability in a cost effective way. It ensures the best use of idle resource in large network.

1.2 Aim

The aim of our project to increase compatibility of program that is used in non-grid environment to grid environment without hampering performance.

1.3 Objectives

There should be a specific objective of a project. The project objective consists of the benefits that an organization or person expects to achieve as a result of spending time and exerting effort to complete a project. As this is an academic project it has strict deadline in which the objective have to be fulfilled.

The objectives of the project are defined below:

- To establish a successful grid computing network based on existing system.

- Analyzing the system to identify its limitations.
- Develop our own grid based system that may be able to improve the system.
- Testing the system and determine its ability and limitations. That is if it is able to fulfill our goal.
- To produce detailed documentation and report on the project and the tool for future reference and use.

1.4 Motivation

In grid computing, the idea is to build an infrastructure that will make distributed computational resources available as easily as electric power is through the electricity distribution grid. Part of the original motivation for grid computing came from the problems in processing scientific data, where the use of dedicated supercomputers is expensive and frequently infeasible. Large networks of much cheaper and less powerful processors have long been touted as a natural alternative to such dedicated devices, but there has never been a technology capable of exploiting such distributed computational resources. The aim of grid computing is to provide such technologies. The "plumbing" for grid computing is essentially in place: we already have large-scale networks of distributed computers, connected by a (comparatively) reliable network using data communication protocols (TCP/IP etc) that are commonly agreed and widely used. The challenges in grid computing therefore lie in developing the software to drive the grid.

Grid applications (multi-disciplinary applications) couple resources that cannot be replicated at a single site even or may be globally located for other practical reasons. These are some of the driving forces behind the inception of grids. In this light, grids let users solve larger or new problems by pooling together resources that could not be coupled easily before.

But, implementing software for grid based system is costly. If any project can make it possible to run general purpose software to run on grid, it will be cost effective for developers and end users as well.

1.5 Scope

The scope of any successful project must be determined at the beginning of the project. Far too many projects fail to achieve success because of ill-defined scopes. Due to the fact that this is an

academic project the scope of the project is very closely related to the aim, objectives and deliverables of the project.

The scope of the project is defined by the following:

- Our project deals with task distribution in grid computing. It deals with incompatibility to run general software to run on grid by distributing its load.
- This project works on input files which are solvable by the program by splitting and possible to combine the outputs in to complete output.
- The tool developed only deals with programs that can be run as batch job and input and output method is file.
- Although the project is grid based, this project does not deal with management and security features in grid.

CHAPTER 2

Literature Review

2.1. What is Grid Computing

Grid computing is a method of harnessing the power of many computers in a network to solve problems requiring a large number of processing cycles and involving huge amounts of data.

In a grid network every node is a complete computer. Grid network is not limited with in a firewall or geographic region. The objective to use grid computing is to gain high throughput.

Sun defines a computational grid as "a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to computational capabilities." Grid computing can encompass desktop PCs, but more often than not its focus is on more powerful workstations, servers, and even mainframes and supercomputers working on problems involving huge datasets that can run for days. And grid computing leans more to dedicated systems, than systems primarily used for other tasks.[1]

Grid is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed "autonomous" resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements. It creates a "virtual supercomputer" by using a network of geographically dispersed computers. Volunteer computing, which generally focuses on scientific, mathematical, and academic problems, is the most common application of this technology.

2.2 Grid Vs Other System

There are different types of computing. Grid computing is one of those computing system. Now we discuss those systems and compare with Grid Computing below.

2.2.1 Distributed Computing

Distributed computing is a method of computer processing in which different parts of a program are run simultaneously on two or more computers that are communicating with each other over a network. Distributed computing is a type of segmented or parallel computing, but the latter term is most commonly used to refer to processing in which different parts of a program run simultaneously on two or more processors that are part of the same computer. While both types of processing require that a program be segmented—divided into sections that can run simultaneously, distributed computing also requires that the division of the program take into account the different environments on which the different sections of the program will be running. For example, two computers are likely to have different file systems and different hardware components.

An example of distributed computing is BOINC (Berkeley Open Infrastructure for Network Computing), a framework in which large problems can be divided into many small problems which are distributed to many computers. Later, the small results are reassembled into a larger solution.

Distributed computing is a natural result of using networks to enable computers to communicate efficiently. But distributed computing is distinct from computer networking or fragmented computing. The latter refers to two or more computers interacting with each other, but not, typically, sharing the processing of a single program. The World Wide Web is an example of a network, but not an example of distributed computing.

There are numerous technologies and standards used to construct distributed computations, including some which are specially designed and optimized for that purpose, such as Remote Procedure Calls (RPC) or Remote Method Invocation (RMI) or .NET remoting.[2]

2.2.2 Parallel Computing

Parallel computing is the simultaneous execution of some combination of multiple instances of programmed instructions and data on multiple processors in order to obtain results faster. The idea is based on the fact that the process of solving a problem usually can be divided into smaller tasks, which may be carried out simultaneously with some coordination. The technique was first put to practical use by ILLIAC IV in 1976, fully a decade after it was conceived.

A parallel computing system is a computer with more than one processor for parallel processing. In the past, each processor of a multiprocessing system always came in its own processor packaging, but recently-introduced multicore processors contain multiple logical processors in a single package. There are many different kinds of parallel computers. They are distinguished by the kind of interconnection between processors (known as "processing elements" or PEs) and memory. Flynn's taxonomy, one of the most accepted taxonomies of parallel architectures, classifies parallel (and serial) computers according to: whether all processors execute the same instructions at the same time (single instruction/multiple data—SIMD) or whether each processor executes different instructions (multiple instruction/multiple data—MIMD).

One major way to classify parallel computers is based on their memory architectures. Shared memory parallel computers have multiple processors accessing all available memory as global address space. They can be further divided into two main classes based on memory access times: Uniform Memory Access (UMA), in which access times to all parts of memory are equal, or Non-Uniform Memory Access (NUMA), in which they are not. Distributed memory parallel computers also have multiple processors, but each of the processors can only access its own local memory; no global memory address space exists across them. Parallel computing systems can also be categorized by the numbers of processors in them. Systems with thousands of such processors are known as massively parallel. Subsequently there are what are referred to as "large scale" vs. "small scale" parallel processors. This depends on the size of the processor, e.g. a PC based parallel system would generally be considered a small scale system. Parallel processor machines are also divided into symmetric and asymmetric multiprocessors, depending on whether all the processors are the same or not (for instance if only one is capable of running the operating system code and others are less privileged).

A variety of architectures have been developed for parallel processing. For example, ring architecture has processors linked by a ring structure. Other architectures include hypercubes, fat trees, systolic arrays, and so on.[3]

2.2.3 Super computers

"Distributed" or "grid computing" in general is a special type of parallel computing which relies on complete computers (with onboard CPU, storage, power supply, network interface, etc.) connected to a network (private, public or the Internet) by a conventional network interface, such as Ethernet. This is in contrast to the traditional notion of a supercomputer, which has many CPUs connected by a local high-speed computer bus.

The primary advantage of distributed computing is that each node can be purchased as commodity hardware, which when combined can produce similar computing resources to a many-CPU supercomputer, but at lower cost. This is due to the economies of scale of producing commodity hardware, compared to the lower efficiency of designing and constructing a small number of custom supercomputers. The primary performance disadvantage is that the various CPUs and local storage areas do not have high-speed connections. This arrangement is thus well-suited to applications where multiple parallel computations can take place independently, without the need to communicate intermediate results between CPUs.

The high-end scalability of geographically dispersed grids is generally favorable, due to the low need for connectivity between nodes relative to the capacity of the public Internet. Conventional supercomputers also create physical challenges in supplying sufficient electricity and cooling capacity in a single location. Both supercomputers and grids can be used to run multiple parallel computations at the same time, which might be different simulations for the same project, or computations for completely different applications. The infrastructure and programming considerations needed to do this on each type of platform are different, however.[4]

There are also differences in programming and deployment. It can be costly and difficult to write programs so that they can be run in the environment of a supercomputer, which may have a custom operating system, or require the program to address concurrency issues. If a problem can

be adequately parallelized, a "thin" layer of "grid" infrastructure can cause conventional, standalone programs to run on multiple machines (but each given a different part of the same problem). This makes it possible to write and debug programs on a single conventional machine, and eliminates complications due to multiple instances of the same program running in the same shared memory and storage space at the same time.

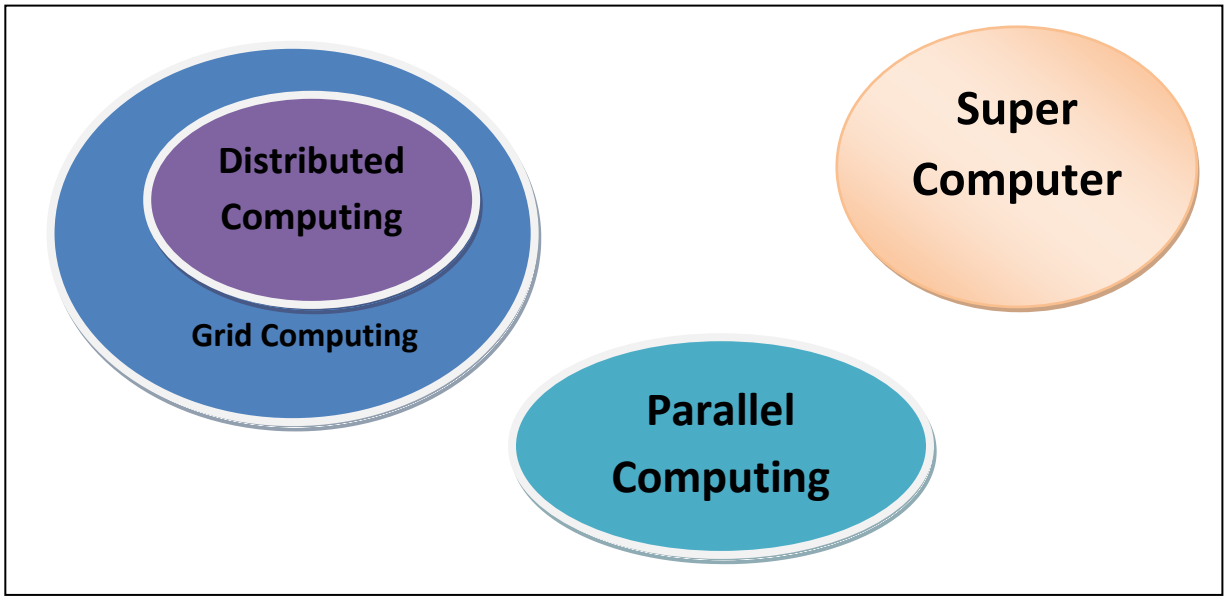


Figure 1: Grid Vs Others

In short we can say, grid and distributed computing either overlap, or distributed computing is a subset of grid computing. Parallel computing uses multiple processors in a single computer and super computer uses array of processors connected with high speed buses and needs custom operating system and applications.

2.3 Advantages of Grid Computing

2.3.1 **Cost effective:** No need to buy large six figure SMP servers for applications that can be split up and farmed out to smaller commodity type servers. Results can then be concatenated and analyzed upon job(s) completion

2.3.2 **Use of idle resource:** Much more efficient use of idle resources. Jobs can be farmed out to idle servers or even idle desktops. Many of these resources sit idle especially during off business hours. Policies can be in place that allows jobs to only go to servers that are lightly loaded or have the appropriate amount of memory/CPU characteristics for the particular application.

2.3.3 **Modularity:** Grid environments are much more modular and don't have single points of failure. If one of the servers/desktops within the grid fails there are plenty of other resources able to pick the load. Jobs can automatically restart if a failure occurs.

2.3.4 **Failsafe:** Don't have single points of failure. If one of the node (each computer) crash, then rest of the system will work as before.

2.3.5 **Easy management:** Policies can be managed by the grid software. The software is really the brains behind the grid. A client will reside on each server which sends information back to the master telling it what type of availability or resources it has to complete incoming jobs.

2.3.6 **Plug n play:** This model scales very well. Need more compute resources? Just plug them in by installing grid client on additional desktops or servers. They can be removed just as easily on the fly. This modular environment really scales well.

2.3.7 **No downtime:** Upgrading can be done on the fly without scheduling downtime. Since there are so many resources some can be taken offline while leaving enough for work to continue. This way upgrades can be cascaded as to not affect ongoing projects.

2.3.8 **Performance:** Jobs can be executed in parallel speeding performance. Grid environments are extremely well suited to run jobs that can be split into smaller chunks and run concurrently on many nodes. Using things like MPI will allow message passing to occur among compute resources.

2.4 Reliability Of Grid Computing In Real Time

In grid system every node is a complete computer and every computer has its own work to do, so it's unreliable for real time systems.

Its reliability doesn't depend on hardware. It is not reliable because it only uses the free cycles. When the machine is busy, then it has to be waiting.

In Table 1 shows the reliability for the non grid system. There, servers or small clusters are reliable for approximately hundred percent. In Table 2 shows the grid system. Which shows, three grid systems reliability. There are more than 10% times jobs are fails in DAS-2, 20 to 44 % time jobs are fails in TerraGrid and in grid3 system 27% time fails after 5 to 10 retries.

System Type	Reliability

Server	99.99999%
Small Cluster	99.999%

Table 1: Non-grid

System Name	Reliability
DAS-2	>10% Job Fails
TerraGrid	20-44% Failures
Grid3	27% Failures, 5-10 Retries

Table 2: Grid systems

So, grid based system are not suitable for real time system, but we don't always need real time system. For example if we need to run hundreds or thousands of simulations within small time then if we use single pc then it'll take several days may be months, but if we submit the same problems to a grid based system then it will use free cycles of multiple computer and complete the task in less time.[5]

2.5 **Disadvantages**

- **Incompatibility:** All programs are not compatible with grid. For example if we want to run a simple program that is designed to run in a single computer in grid, then the grid system cannot split the program and distribute it among its other nodes. Programs written for grid based system are complex and multithreaded. Often it needs extra library functions to write the code.
- **Slower for smaller tasks:** Small tasks may take more time than usual processing.

$$T = \frac{T_{Exec}}{N} + (T_{Split} + T_{Idle} + T_{Com.} + T_{Comb.})$$

$$T = T_{Exec}; [N = 1, T_{Split} = T_{Idle} = T_{Com.} = T_{Comb.} = 0]$$

In the above equations,

T = total time,

T_{Exec} = execution time,

N = number of computers connected in execution time,

T_{Split} = time to split the task,

T_{Idle} = time to find out the idle resources,

$T_{Com.}$ = time to communication like TCP connection and

$T_{Comb.}$ = time to combine the total project.

When the task is small, in the single computer (where n=1) and four additional time, $T_{Split} = T_{Idle} = T_{Com.} = T_{Comb.} = 0$, execution time will take less than in the grid system. Here overhead will increase and total execution time will increase as well as. But, when the task is large then grid system's execution time will be less. It depends on number of computers connected at a time, if it increases (N), time will decrease.

- **License restriction:** licensing across many servers may make it prohibitive for some applications.

2.6 Works on grid computing

2.6.1 Hungarian Cluster Grid and Super Grid systems

As a result of the ClusterGrid project a significant high-performance computing Grid infrastructure has been created in Hungary. The ClusterGrid currently connects about 600 PCs of 13 higher educational institutions. The system is constantly grows and the final goal is to connect more than 2000 PCs by the end of 2004. Although the ClusterGrid already works as a regular infrastructure service it raises several problems to be solved.
The basic goal of this project to solve the problems by providing new functions for the ClusterGrid like

- checkpoint handling for PVM and MPI programs,
- Grid brokering,
- workflow management,
- high-level user interfaces,
- Grid monitoring,
- handling large data files and databases in the Grid,
- accounting system support,
- Application of VPN and IPv6 technologies.

In order to provide these new features we adapt the results of the Hungarian SuperGrid project.[12]

2.6.2 Project Ganglia

Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It is based on a hierarchical design targeted at federations of clusters. It leverages widely used technologies such as XML for data representation, XDR for compact, portable data transport, and RRD tool for data storage and visualization. It uses carefully engineered data structures and algorithms to achieve very low per-node overheads and high concurrency. The implementation is robust, has been ported to an extensive set of operating systems and processor architectures, and is currently in use on thousands of clusters around the world. It has been used to link clusters across university campuses and around the world and can scale to handle clusters with 2000 nodes.[7]

Ganglia is an open-source project that grew out of the University of California, Berkeley Millennium Project which was initially funded in large part by the National Partnership for Advanced Computational Infrastructure (NPACI) and National Science Foundation RI Award EIA-9802069. NPACI is funded by the National Science Foundation and strives to advance science by creating a ubiquitous, continuous, and pervasive national computational infrastructure: the Grid. Current support comes from Planet Lab: an open platform for developing, deploying, and accessing planetary-scale services.

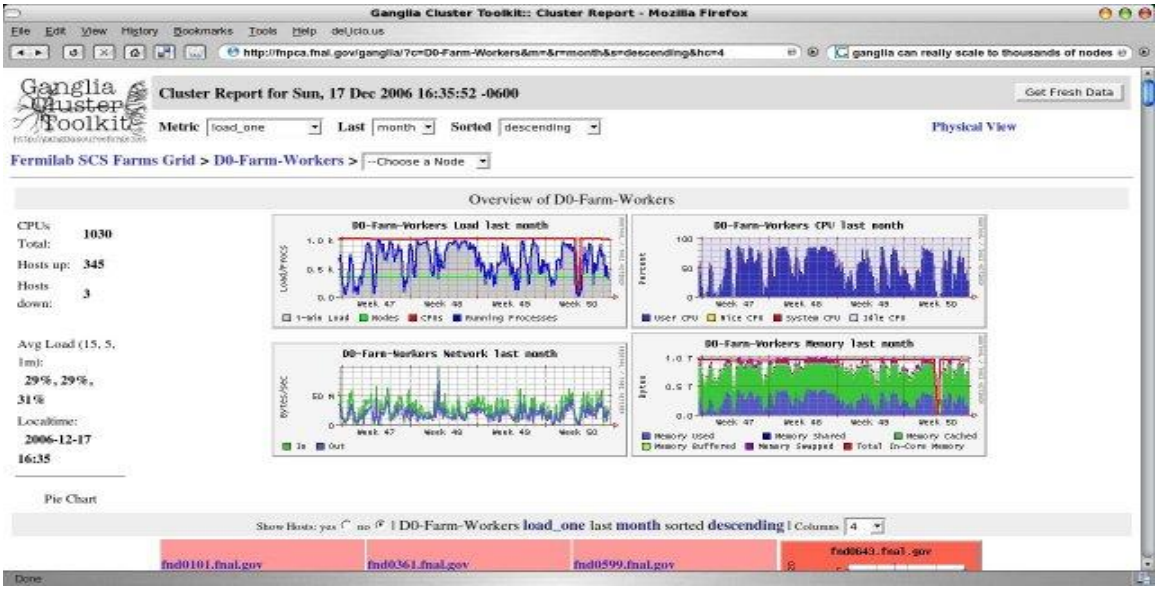


Figure 2 : analysis of **Project Ganglia**

2.6.3 TeraGrid

The TeraGrid is the largest cyber infrastructure facility available for non-classified use in the US. The TeraGrid is a centerpiece of the efforts of the US National Science Foundation (NSF) to enable new, 21st century science innovations. The TeraGrid provides a network of supercomputers with more than 250 teraflops of computing power, data storage facilities to store more than 30 peta bytes of data, high-resolution visualization environments, and toolkits for grid computing, all connected through a very high-capacity network. If you need access to more computing power, storage capabilities, or advanced consulting support to advance your scientific research, consider getting a TeraGrid account.[13]

Figure 3: This image shows the geographic locations of current TeraGrid Resource Providers and the 10Gb/s network links that interconnect them. The TeraGrid is a virtual facility for scientific research that integrates computational, storage, information, and data analysis resources at the San Diego Supercomputer Center, the Texas Advanced Computing Center, the University of Chicago/Argonne National Laboratory, the National Center for Supercomputing Applications, Purdue University, Indiana University, Oak Ridge National Laboratory, the Pittsburgh Supercomputing Center, and the National Center for Atmospheric Research. As a TeraGrid Resource Provider, NCAR is committed to offering a highly distributed network of computational, data, and knowledge resources to multidisciplinary groups of researchers, students, educators, impact and assessment communities, and policy makers around the world.

2.6.4 SETI@home

SETI (Search for Extraterrestrial Intelligence) is a scientific area whose goal is to detect intelligent life outside Earth. One approach, known as radio SETI, uses radio telescopes to listen for narrow-bandwidth radio signals from space. Such signals are not known to occur naturally, so detection would provide evidence of extraterrestrial technology.

Radio telescope signals consist primarily of noise (from celestial sources and the receiver's electronics) and man-made signals such as TV stations, radar, and satellites. Modern radio SETI projects analyze the data digitally. More computing power enables searches to cover greater frequency ranges with more sensitivity. Radio SETI, therefore, has an insatiable appetite for computing power.

Previous radio SETI projects have used special-purpose supercomputers, located at the telescope, to do the bulk of the data analysis. In 1995, David Gedye proposed doing radio SETI using a virtual supercomputer composed of large numbers of Internet-connected computers, and he organized the SETI@home project to explore this idea. SETI@home was originally launched in May 1999.

With over 5.2 million participants worldwide, the project is the distributed computing project with the most participants to date. The original intent of SETI@home was to utilize 50,000-100,000 home computers. Since its launch on May 17, 1999, the project has logged over two million years of aggregate computing time. On September 26, 2001, SETI@home had performed a total of 1021 floating point operations. It is acknowledged by the Guinness World Records as the largest computation in history. With over 334,155 active computers in the system (1.8 million total) in 210 countries, as of August 04, 2008, SETI@home has the ability to compute over 528 Teraflops. For comparison, Blue Gene (one of the world's fastest supercomputers) peaks at just over 596 Teraflops with sustained rate of 478 Teraflops.[14]

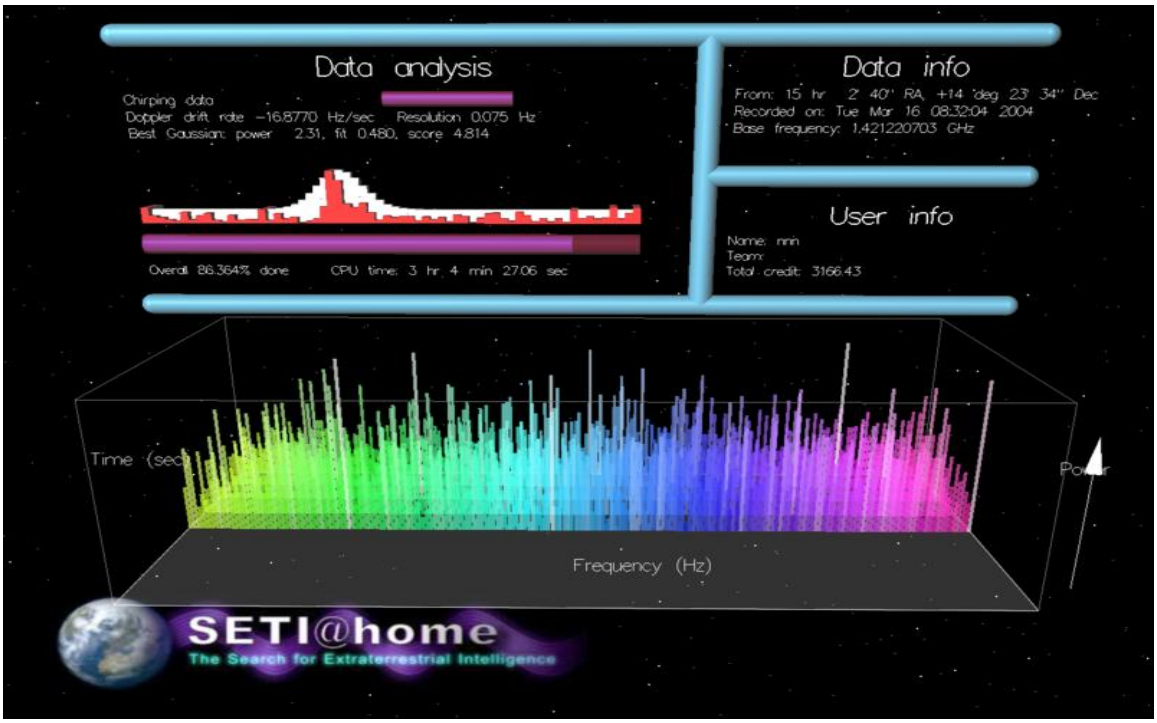


Figure 4: data analysis

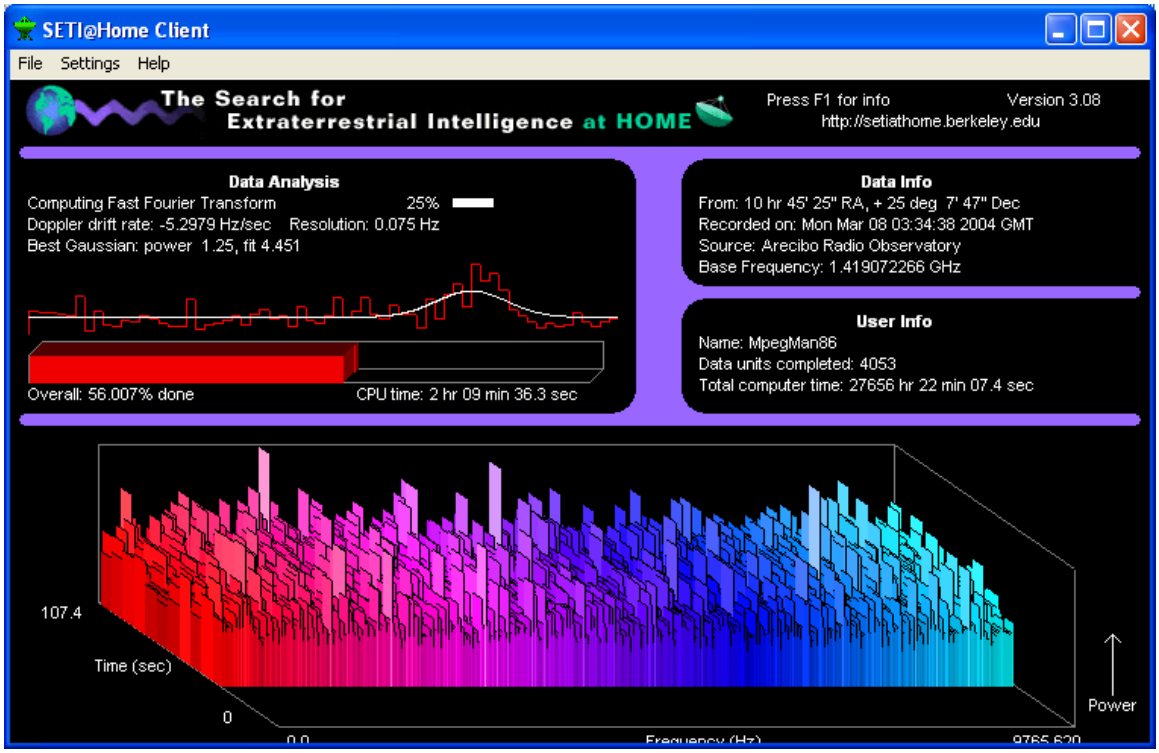


Figure 5: Seti@home Clients

2.6.5 Grid3/Grid 2003

The Grid3 collaboration has deployed an international Data Grid with dozens of sites and thousands of processors. The facility is operated jointly by the U.S. Grid projects iVDGL, GriPhyN and PPDG, and the U.S. participants in the LHC experiments ATLAS and CMS. [15]

Project highlights include:

- Participation by more than 25 sites across the US and Korea which collectively provide more than 2000 CPUs
- Resources used by 7 different scientific applications, including 3 high energy physics simulations and 4 data analyses in high energy physics, bio-chemistry, astrophysics and astronomy
- More than 100 individuals are currently registered with access to the Grid
- A peak throughput of 500-900 jobs running concurrently with a completion efficiency of approximately 75%

2.6.6 IBM and grid

IBM has a long and thorough involvement with both the technology and the business issues that have led to the grid computing evolution. "Virtualization" — the driving force behind grid computing — has been a key factor since the earliest days of electronic business computing. IBM put the main in mainframe, in part, by creating virtual memory, virtual storage and the virtual processor. This development enabled the computer to do many processing jobs simultaneously for hundreds and eventually thousands of users. Users got mainframe-strength computing; businesses got greater leverage from an expensive and powerful asset.

Fast forward to today. Almost every organization is sitting on top of enormous, unused computing capacity, widely distributed. This is an intolerable situation for customers. (Imagine an airline with 90% of its fleet on the ground, an automaker with 40% of its assembly plants idle, a hotel chain with 95% of its rooms unoccupied.) Once again, virtualization can help.[8]

Grid computing represents this advanced development in virtualization — and IBM Grid Computing continues IBM's history of IT innovation for business. Taking a significant role in the growing grid community, IBM offers a full line of products and services that continue to be developed for both grid customers and those ready for next steps.

2.6.7 Search for Extraterrestrial Intelligence

SETI (the Search for Extraterrestrial Intelligence) is a scientific effort to discover intelligent life elsewhere in the universe, primarily by attempting to discover radio signals that indicate intelligence. Cornell astronomer Frank Drake is credited with being the first to "listen" for intelligent signals with a radio telescope in 1960. Although NASA has funded some study in the past, current efforts are privately funded, in part by Arthur C. Clarke, Microsoft co-founder Paul Allen, Intel founder Gordon Moore, and Hewlett-Packard cofounders David Packard and William Hewlett.

The SETI Institute's Project Phoenix is using computers to search about 1,000 stars within 200 light-years of our solar system for radio signals beamed toward us or any other location. Project Phoenix's 140-foot radio telescope in Green Bank, West Virginia aims at one star at a time while astronomer-monitored computers search each 1,000 band from 1,000 to 3,000 MHz for a signal limited to a narrowband range. Scientists believe that a signal focused within a narrow frequency band would suggest an intelligent source.

About two-thirds of the first 1,000 stars have been searched with no success yet reported. There are, however, over 400 billion stars in our own galaxy so the study may last quite a long time. The directors of the project are soliciting volunteers to help analyze the radio telescope data at their home computers.

2.6.8 Grid Computing At Hartford Life

Grid computing, which is already being used in some academic and research communities, is making its way to the life insurance industry. Essentially grid computing involves sharing computing, data, storage, application, or network resources, to ultimately allow companies to solve large-scale, complex computational problems.

Hartford Life is among the first life insurance companies to implement grid computing. Resource recently talked with Vic Severino, senior vice president and CIO, Hartford Life, about how The Hartford is using grid computing to help manage the risk for income protection benefits associated with its variable annuities.

Resource: Why did The Hartford decide to implement grid computing technology?

Scenario: It really was out of pure necessity. We have some complex products that require a high level of computing power. Essentially we had a very pressing business need—and I would say this was probably one of our most important business initiatives—to essentially manage the risk as a result of these complex products. We needed a lot of accessible and stable computing power. So we did some research and talked to some of our investment banking partners; the investment banks have been using grid computing for a few years. We decided to embrace grid computing for ourselves. What got us into it was just the fact that we had a very pressing business need.

2.6.9 Condor Project

This is a perfect example of virtual supercomputing service using academic network. This is implemented by University of Wisconsin, Madison.

Condor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the

jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion.

Condor can be used to build Grid-style computing environments that cross administrative boundaries. Condor's "flocking" technology allows multiple Condor compute installations to work together. Condor incorporates many of the emerging Grid-based computing methodologies and protocols. For instance, Condor-G is fully interoperable with resources managed by Globus.[9]

2.7 Implementing Grid

In grid computing a single or multiple tasks is divided among computers to get maximum output in less time. There are three ways to implement a grid:

1. **Run application on an available machine in grid:** In this method different task divided among computers and all computers execute each task during their free cycle. Here each one is different and complete task.

2. **Use an application that will divide the work so that all process can be distributed**: Here central servers divide a single task among computers in a grid. In this method the task have to be divisible and the program have to be written to support grid environment. Here a task contains multiple threads, so it is possible to divide among computers.

3. **Application executed many times on different machine in grid:** This method is only applicable for the task where the instruction is same but need to process with different data. In this method same program is run in multiple computers with different data.

2.8 Widely Used tools available for grid computing

There are many tools available which provides API to program for grid environment. The widely used products are.

- **JPPF**:Java Parallel Processing Framework (JPPF) is an open source Grid Computing platform written in Java that makes it easy to run applications in parallel, and speed up

their execution by orders of magnitude. Write once, deploy once and execute everywhere. [http://www.jppf.org]

- **Globus Toolkit:** The open source Globus Toolkit (GTK) is a fundamental enabling technology for the "Grid," letting people share computing power, databases, and other tools securely online across corporate, institutional, and geographic boundaries without sacrificing local autonomy. The toolkit includes software services and libraries for resource monitoring, discovery, and management, plus security and file management. [http://www.globus.org/toolkit/]

- **Gridbus:** Gridbus is developed by The Grid Computing and Distributed Systems (GRIDS) Laboratory, University of Melbourne, Australia. The Gridbus project is engaged in the creation of open-source specifications, architecture and a reference Grid toolkit implementation of service-oriented grid and utility computing technologies for eScience and eBusiness applications. The Gridbus software is being used in Grid-enabling a number of applications in science, engineering, and commerce. [www.gridbus.org]

CHAPTER 3

Condor

We analyze condor project for understand grid system. We install it and execute this open source project in our lab. We try to find out where is lacking of this system. So we had to understand this project.

3.1 Overview

Condor is a software system that creates a High-Throughput Computing (HTC) environment. It effectively utilizes the computing power of workstations that communicate over a network. Condor can manage a dedicated cluster of workstations. Its power comes from the ability to effectively harness non-dedicated, preexisting resources under distributed ownership.

A user submits the job to Condor. Condor finds an available machine on the network and begins running the job on that machine. Condor has the capability to detect that a machine running a Condor job is no longer available (perhaps because the owner of the machine came back from lunch and started typing on the keyboard). It can checkpoint the job and move (migrate) the jobs to a different machine which would otherwise be idle. Condor continues job on the new machine from precisely where it left off.

In those cases where Condor can checkpoint and migrate a job, Condor makes it easy to maximize the number of machines which can run a job. In this case, there is no requirement for machines to share file systems (for example, with NFS or AFS), so that machines across an entire enterprise can run a job, including machines in different administrative domains.

Condor can be a real time saver when a job must be run many (hundreds of) different times, perhaps with hundreds of different data sets. With one command, all of the hundreds of jobs are submitted to Condor. Depending upon the number of machines in the Condor pool, dozens or even hundreds of otherwise idle machines can be running the job at any given moment.

Condor does not require an account (login) on machines where it runs a job. Condor can do this because of its remote system call technology, which traps library calls for such operations as reading or writing from disk files. The calls are transmitted over the network to be performed on the machine where the job was submitted.

Condor provides powerful resource management by match-making resource owners with resource consumers. This is the cornerstone of a successful HTC environment. Other compute cluster resource management systems attach properties to the job queues themselves, resulting in user confusion over which queue to use as well as administrative hassle in constantly adding and editing queue properties to satisfy user demands. Condor implements ClassAds, a clean design that simplifies the user's submission of jobs.

ClassAds work in a fashion similar to the newspaper classified advertising want-ads. All machines in the Condor pool advertise their resource properties, both static and dynamic, such as available RAM memory, CPU type, CPU speed, virtual memory size, physical location, and current load average, in a resource offer ad. A user specifies a resource request ad when submitting a job. The request defines both the required and a desired set of properties of the resource to run the job. Condor acts as a broker by matching and ranking resource offer ads with resource request ads, making certain that all requirements in both ads are satisfied. During this match-making process, Condor also considers several layers of priority values: the priority the user assigned to the resource request ad, the priority of the user which submitted the ad, and desire of machines in the pool to accept certain types of ads over others.[9]

3.2 The Different Roles of a computer in a network

3.2.1 Central Manager

There can be only one central manager for your pool. The machine is the collector of information, and the negotiator between resources and resource requests. These two halves of the central manager's responsibility are performed by separate daemons, so it would be possible to have different machines providing those two services. However, normally they both live on the same machine. This machine plays a very important part in the Condor pool and should be reliable. If this machine crashes, no further matchmaking can be performed within the Condor system (although all current matches remain in effect until they are broken by either party involved in the match). Therefore, choose for central manager a machine that is likely to be up and running all the time, or at least one that will be rebooted quickly if something goes wrong. The central manager will ideally have a good network connection to all the machines in your pool, since they all send updates over the network to the central manager. All queries go to the central manager.

3.2.2 Execute

Any machine in your pool (including your Central Manager) can be configured for whether or not it should execute Condor jobs. Obviously, some of your machines will have to serve this function or your pool won't be very useful. Being an execute machine doesn't require many resources at all. About the only resource that might matter is disk space, since if the remote job dumps core, that file is first dumped to the local disk of the execute machine before being sent back to the submit machine for the owner of the job. However, if there isn't much disk space, Condor will simply limit the size of the core file that a remote job will drop. In general the more resources a machine has (swap space, real memory, CPU speed, etc.) the larger the resource requests it can serve. However, if there are requests that don't require many resources, any machine in your pool could serve them.

3.2.3 **Submit**

Any machine in your pool (including your Central Manager) can be configured for whether or not it should allow Condor jobs to be submitted. The resource requirements for a submit machine are actually much greater than the resource requirements for an execute machine. First of all, every job that you submit that is currently running on a remote machine generates another process on your submit machine. So, if you have lots of jobs running, you will need a fair amount of swap space and/or real memory. In addition all the checkpoint files from your jobs are stored on the local disk of the machine you submit from. Therefore, if your jobs have a large memory image and you submit a lot of them, you will need a lot of disk space to hold these files. This disk space requirement can be somewhat alleviated with a checkpoint server, however the binaries of the jobs you submit are still stored on the submit machine.

3.2.4 **Checkpoint Server**

One machine in your pool can be configured as a checkpoint server. This is optional, and is not part of the standard Condor binary distribution. The checkpoint server is a centralized machine that stores all the checkpoint files for the jobs submitted in your pool. This machine should have lots of disk space and a good network connection to the rest of your pool, as the traffic can be quite heavy.

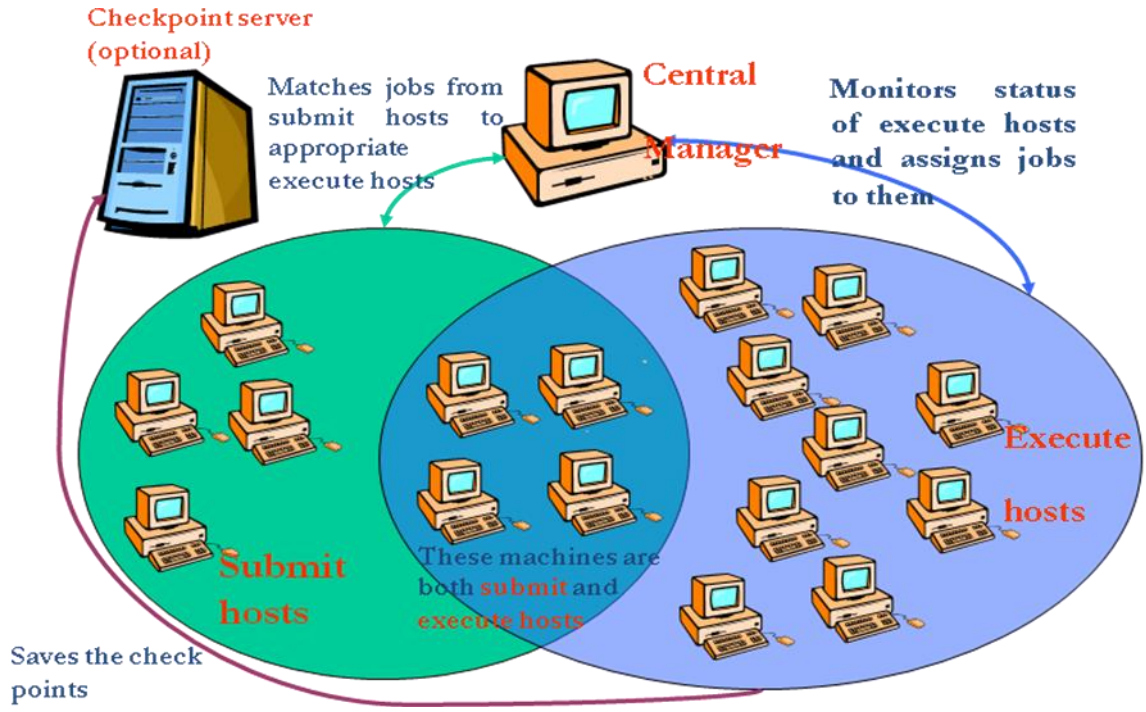


Figure 6: Condor V7.1.0

3.3 Our Installed system:

We installed condor in both Windows and Linux PC. In both cases we use one central manager and two execute and submit hosts. We didn't install any checkpoint server because of limitations of resources. The configurations of the PCs are as follows:

Processor	Chipset	Memory	OS
Pentium 4 2.4GHz	Intel 845G	512MB	Fedora core 5

Table 3: Linux

Processor	Chipset	Memory	OS
Pentium D 2.4GHz	Intel 945G	512MB	Windows 2000

Table 4: Windows

3.4 The Condor Daemons

The following list describes all the daemons and programs that could be started under Condor and what they do:

3.4.1 Condor master: This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the master will restart the affected daemons. In addition, if any daemon crashes, the master will send e-mail to the Condor Administrator of your pool and restart the daemon. The condor master also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The condor master will run on every machine in your Condor pool, regardless of what functions each machine are performing.

3.4.2 **Condor startd:** This daemon represents a given resource (namely, a machine capable of running jobs) to the Condor pool. It advertises certain attributes about that resource that are used to match it with pending resource requests. The startd will run on any machine in your pool that you wish to be able to execute jobs. It is responsible for enforcing the policy that resource owners configure which determines under what conditions remote jobs will be started, suspended, resumed, vacated, or killed. When the startd is ready to execute a Condor job, it spawns the condor starter, described below.

3.4.3 **Condor starter:** This program is the entity that actually spawns the remote Condor job on a given machine. It sets up the execution environment and monitors the job once it is running. When a job completes, the starter notices this, sends back any status information to the submitting machine, and exits.

3.4.4 **Condor schedd:** This daemon represents resource requests to the Condor pool. Any machine that you wish to allow users to submit jobs from needs to have a condor schedd running. When users submit jobs, they go to the schedd, where they are stored in the job queue, which the schedd manages. Various tools to view and manipulate the job queue (such as condor submit, condor q, or condor rm) all must connect to the schedd to do their work. If the schedd is down on a given machine, none of these commands will work. The schedd advertises the number of waiting jobs in its job queue and is responsible for claiming available resources to serve those requests. Once a schedd has been matched with a given resource, the schedd spawns a condor shadow to serve that particular request.

3.4.5 **Condor shadow:** This program runs on the machine where a given request was submitted and acts as the resource manager for the request. Jobs that are linked for Condor's standard universe, which perform remote system calls, do so via the condor shadow. Any system call performed on the remote execute machine is sent over the network, back to the condor shadow which actually performs the system call (such as file I/O) on the submit machine, and the result is sent back over the network to the remote job. In addition, the shadow is responsible formaking decisions about the request (such as where checkpoint files should be stored, how certain files should be accessed, etc).

3.4.6 **Condor collector:** This daemon is responsible for collecting all the information about the status of a Condor pool. All other daemons periodically send ClassAd updates to the collector. These ClassAds contain all the information about the state of the daemons, the resources they

represent or resource requests in the pool (such as jobs that have been submitted to a given schedd). The condor status command can be used to query the collector for specific information about various parts of Condor. In addition, the Condor daemons themselves query the collector for important information, such as what address to use for sending commands to a remote machine.

3.4.7 **Condor negotiator:** This daemon is responsible for all the match-making within the Condor system. Periodically, the negotiator begins a negotiation cycle, where it queries the collector for the current state of all the resources in the pool. It contacts each schedd that has waiting resource requests in priority order, and tries to match available resources with those requests. The negotiator is responsible for enforcing user priorities in the system, where the more resources a given user has claimed, the less priority they have to acquire more resources. If a user with a better priority has jobs that are waiting to run, and resources are claimed by a user with a worse priority, the negotiator can preempt that resource and match it with the user with better priority.

3.4.8 **Condor kbdd:** This daemon is only needed on Digital Unix. On that platforms, the condor startd cannot determine console (keyboard or mouse) activity directly from the system. The condor kbdd connects to the X Server and periodically checks to see if there has been any activity. If there has, the kbdd sends a command to the startd. That way, the startd knows the machine owner is using the machine again and can perform whatever actions are necessary, given the policy it has been configured to enforce.

3.4.9 **Condor ckpt:** server This is the checkpoint server. It services requests to store and retrieve checkpoint files. If your pool is configured to use a checkpoint server but that machine (or the server itself is down) Condor will revert to sending the checkpoint files for a given job back to the submit machine.

3.4.10 **Condor quill:** this daemon builds and manages a database that represents a copy of the Condor job queue. The condor q and condor history tools can then query the database.

3.4.11 **Condor dbmsd:** This daemon assists the condor quill daemon.

3.4.12 **Condor gridmanager:** This daemon handles management and execution of all grid universe jobs. The condor schedd invokes the condor gridmanager when there are grid universe jobs in the queue, and the condor gridmanager exits when there are no more grid universe jobs in the queue.

3.4.13 **Condor had:** This daemon implements the high availability of a pool's central manager through monitoring the communication of necessary daemons. If the current, functioning, central manager machine stops working, then this daemon ensures that another machine takes its place, and becomes the central manager of the pool.

3.4.14 **Condor replication:** This daemon assists the condor had daemon by keeping an updated copy of the pool's state. This state provides a better transition from one machine to the next, in the event that the central manager machine stops working.

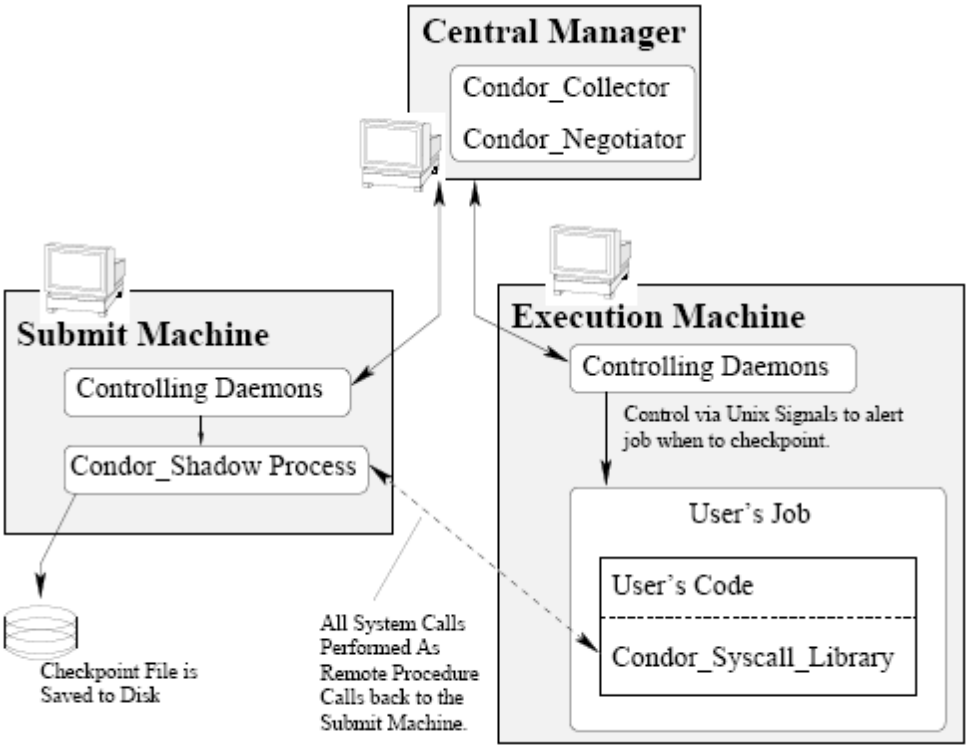


Figure 7: Graphical representation Pool Architecture

3.5 **Installation Preparation**

Before installation, make a few important decisions about the basic layout of your pool. The decisions answer the questions:

1. What machine will be the central manager?

2. What machines should be allowed to submit jobs?

3. Will Condor run as root or not?

4. Who will be administering Condor on the machines in your pool?

5. Will you have a Unix user named condor and will its home directory be shared?

6. Where should the machine-specific directories for Condor go?

7. Where should the parts of the Condor system be installed?

• Configuration files

• Release directory

– user binaries

– system binaries

– lib directory

– etc directory

• Documentation

8. Am I using AFS?

9. Do I have enough disk space for Condor?

3.6 Platform-Specific Information

The Condor Team strives to make Condor work the same way across all supported platforms. However, because Condor is a very low-level system which interacts closely with the internals of the operating systems on which it runs. This system supports Linux, Windows, and UNIX.

3.6.1 Linux

This section provides information specific to the Linux port of Condor. Linux is a difficult platform to support. It changes very frequently, and Condor has some extremely system-dependent code (for example, the check pointing library).

Condor is sensitive to changes in the following elements of the system:

• The kernel version
• The version of the GNU C library (glibc)
• the version of GNU C Compiler (GCC) used to build and link Condor jobs (this only matters for Condor's Standard universe which provides check pointing and remote system calls)

The Condor Team tries to provide support for various releases of the distribution of Linux. Red Hat is probably the most popular Linux distribution, and it provides a common set of versions for the above system components at which Condor can aim support. Condor will often work with Linux distributions other than Red Hat (for example, Debian or SuSE) that have the same versions of the above components. However, we do not usually test Condor on other Linux distributions and we do not provide any guarantees about this.
New releases of Red Hat usually change the versions of some or all of the above system-level components. A version of Condor that works with one release of Red Hat might not work with newer releases. The following sections describe the details of Condor's support for the currently available versions of Red Hat Linux on x86 architecture machines.

3.6.1.1 Linux Kernel-specific Information

Distributions that rely on the Linux 2.4.x and all Linux 2.6.x kernels through version 2.6.10 do not modify the time of the input device file. This leads to difficulty when Condor is run using one of these kernels. The problem manifests itself in that Condor cannot properly detect keyboard or mouse activity. Therefore, using the activity in policy setting cannot signal that Condor should stop running a job on a machine. Condor version 6.6.8 implements a workaround for PS/2 devices. A better fix is the Linux 2.6.10 kernel patch linked to from the directions posted at This patch works better for PS/2 devices, and may also work for USB devices. A future version of Condor will implement better recognition of USB devices, such that the kernel patch will also definitively work for USB devices.

3.6.1.2 Red Hat Version 9.x

Red Hat version 9.x is fully supported in Condor Version 7.0.1. condor compile works to link user jobs for the Standard universe with the versions of gcc and glibc that come with Red Hat 9.x.

3.6.1.3 Red Hat Fedora 1, 2, and 3

Redhat Fedora Core 1, 2, and 3 now support the check pointing of statically linked executables just like previous revisions of Condor for Red Hat. Condor compiles works to link user jobs for the Standard universe with the versions of gcc that are distributed with Red Hat Fedora Core 1, 2, and 3.

However, there are some caveats: A) You must install and use the dynamic Red Hat 9.x binaries on the Fedora machine and B) if you wish to do run a condor compiled binary in standalone mode(either initially or in resumption mode), then you must pretend the execution of said binary with setarch i386. Here is an example: suppose we have a Condor-linked binary called myapp, running this application as a standalone executable will result in this command: setarch i386 myapp. The subsequent resumption command will be: setarch i386 myapp - condor restart myapp.ckpt. When standard universe executables condor compiled under any currently supported Linux architecture of the same kind (including Fedora 1, 2, and 3) are running inside Condor, they will automatically execute in the i386 execution domain. This means that the exec shield functionality (if available) will be turned off and the shared segment layout will default to Red Hat 9 style. There is no need to do the above instructions concerning setarch if the executables are being submitted directly into Condor via condor submit.

3.7 Microsoft Windows

Windows is a strategic platform for Condor, and therefore we have been working toward a complete port to Windows. Our goal is to make Condor every bit as capable on Windows as it is on UNIX – or even more capable.

Porting Condor from UNIX to Windows is a formidable task, because many components of Condor must interact closely with the underlying operating system. Instead of waiting until all components of Condor are running and stabilized on Windows, we have decided to make a clipped version of Condor for Windows. A clipped version is one in which there is no check pointing and there are no remote system calls.

3.7.1 Limitations under Windows

In general, this release for Windows works the same as the release of Condor for UNIX. However, the following items are not supported in this version:

• The standard job universe is not present. This means transparent process checkpoint/migration and remote system calls are not supported.

• For **grid** universe jobs, the only supported grid type is **condor**.

• Accessing files via a network share that requires a Kerberos ticket (such as AFS) is not yet supported.

3.8 Macintosh OS X

This section provides information specific to the Macintosh OS X port of Condor. The Macintosh port of Condor is more accurately a port of Condor to Darwin, the BSD core of OS X. Condor uses the Carbon library only to detect keyboard activity, and it does not use Cocoa at all. Condor on the Macintosh is a relatively new port, and it is not yet well-integrated into the Macintosh environment.

Condor on the Macintosh has a few shortcomings:

• Users connected to the Macintosh via SSH are not noticed for console activity.

• The memory size of threaded programs is reported incorrectly.

• No Macintosh-based installer is provided.

• The example start up scripts do not follow Macintosh conventions.

• Kerberos is not supported.

Condor does not yet provide Universal binaries for Mac OSX. There are separate downloadable packages for both PowerPC (PPC) and Intel (x86) architectures, so please ensure you are using the right Condor binaries for the platform you are trying to run on.

3.9 Limitations of condor:

- **Compatibility:** There are a few jobs which can really run on Condor. They have to be batch; they have to seek input from an input file.

- **Job distribution:** Another major problem is that it doesn't use the capabilities of the Cluster to its full potential. What it simply does is just execute a job at the most powerful workstation. And even if the job is multithreaded the entire job is run on a single PC. This is tantamount to just taking the job to the most powerful computer and take the result. Cluster software's should break up multi threaded jobs in order to achieve computational speed up.

It also includes other limitations of grid computing.

CHAPTER 4

Research Questions

4.1 Can grid computing provide better performance than other computing in a lab environment?

It depends on our need. If we want a real time system or a dedicated server then grid computing is not for this kind of work. Grid computing used in such cases when we want to use free resources in a network. For example, if we need to run hundreds of large scale simulation or calculation it will take weeks, may be months if we run it on a single computer. But if we have a large network with hundreds of computers we can setup a grid and submit the tasks into it. It will run these tasks on idle computers and reduce the running time to several hours or days. After completing the job it may notify the user by mail automatically. We can use mainframe or super computer for this kind of job but it will not be cost effective.

4.2 Is there a breakpoint before which Grid computing is not efficient?

Besides common limitations of grid computing different grid computing system has different limitations. As we are focusing on task distribution, in this aspect different grid computing system has different task distribution method. For example Condor doesn't split any task. Condor Central manager just assign each task to a execution host. The problem of this system is, if any task is too big the central manager will assign it to a single host and it will take huge time to process. In this scenario it will not show any performance improvement.

4.6. What are the different techniques used in different grid based system to split task?

Different grid based system uses different methods to split task. For some system programs are especially designed to run on grid system. The programs are written using special APIs and support multiple threads. This type of system is effective when we execute only one type of job. For example SETI@home is designed to analyze only radio telescopic data to search extraterrestrial intelligence.

If the grid is designed to execute different types of job then it's not cost effective to convert all programs for grid based system. Condor is designed to execute any type of program in grid. It takes a list of job and run each job in different host. It doesn't split any job. So, if there is a few job and jobs are big enough it's not better than run jobs in a single computer.

4.4 What are the techniques to improve the job distribution system?

Apparently when we assign small task to different hosts it takes less time to finish the job. So in our approach, if the split and combine process does not take more time in theory it should take less time to complete bigger job.

4.5 Are all types of input possible to split?

No, it's not possible to split all types of input. First we need to find which types of inputs are split able, can be run independently in different host and finally it's possible to combine the output files. There are a lot of programs that can support this strategy. Out project is for those programs.

CHAPTER 5

Design and Implementation

After analyzing different grid based system what we found is, write a program for a grid based system is costly and it needs more resources. The three major problems that we are focusing on are:

1. **Compatibility:** Different non- grid programs that used in desktop environment are not compatible with grid. We need to buy programs that support grid computing. On other hand if we already buy this program then we need to buy another program for grid system.

2. **Development:** For developers, when they already have a working program, then it is needed to convert these programs so that it has multiple threads, Converting is not an easy process and it needs extra cost and time.

3. **Backward compatibility of grid programs:** A program developed for grid system is not compatible with regular desktop environment. So, if we need the program for both systems we have to use different programs.

5.1 Our objective:

5.1.1 **Increase Compatibility:** Run traditional programs in grid system without any modification, so that we can run any executable program in grid system. It will increase program compatibility and reduce cost.

5.1.2 Optimization: Optimize program execution time by dividing the program among different nodes in grid. It will reduce the execution time. For example:

If execution time of a program is : T
Number of nodes in grid is: N
Then execution time will be: T/N

5.1.3 High Throughput: Run task in optimized way so that we can gain high throughput. Like other grid system our objective is to maximize output within a certain amount of time. As grid system is not for real time work so, performance is not important here. What our objective is to reduce execution time for larger task without using powerful dedicated system.

5.2 How Our System Works:

5.2.1 Architecture of our system:

Our designed system consists of three kinds of computer.

- **Server/Root:** It's the main server of our system we submit all tasks to this computer. Its job is to split and distribute jobs to each and every node in grid.
- **Node:** Nodes are executing hosts. They collect job from server execute then and submit the output to server. There are many nodes in a grid. If we increase number of nodes then execution time will reduce.
- **Shared storage (Future improvement):** It's the computer that stores the input and output file to reduce communication overhead to root. We didn't implement it yet, but it's in our design and will be added in future improvement. It's not compulsory but it increase performance in larger grid.

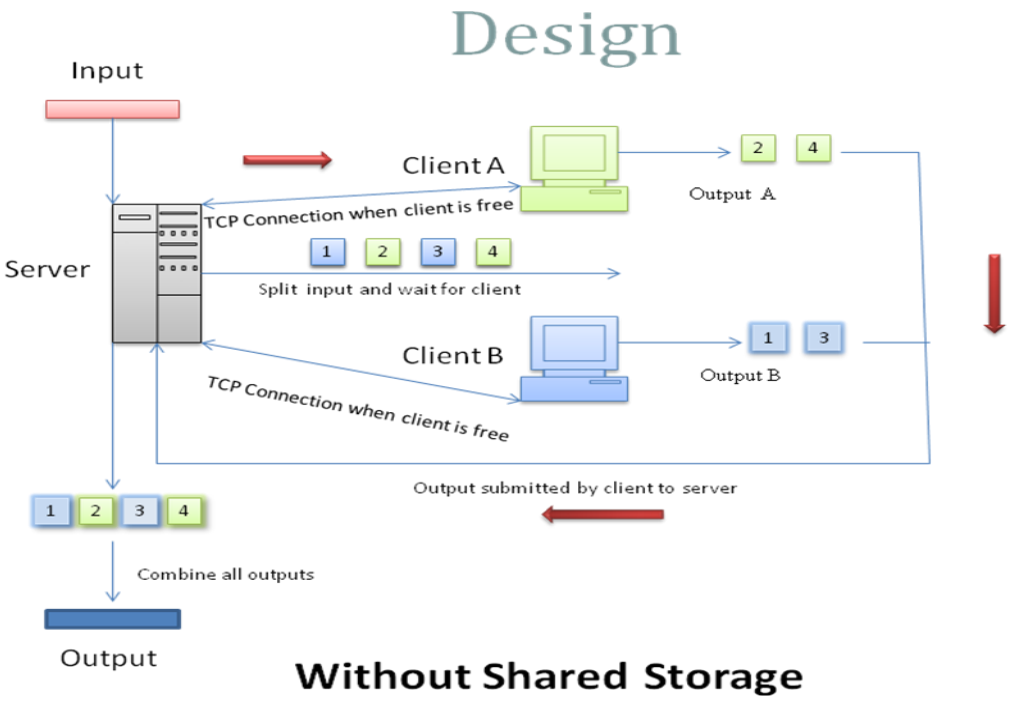


Figure 8

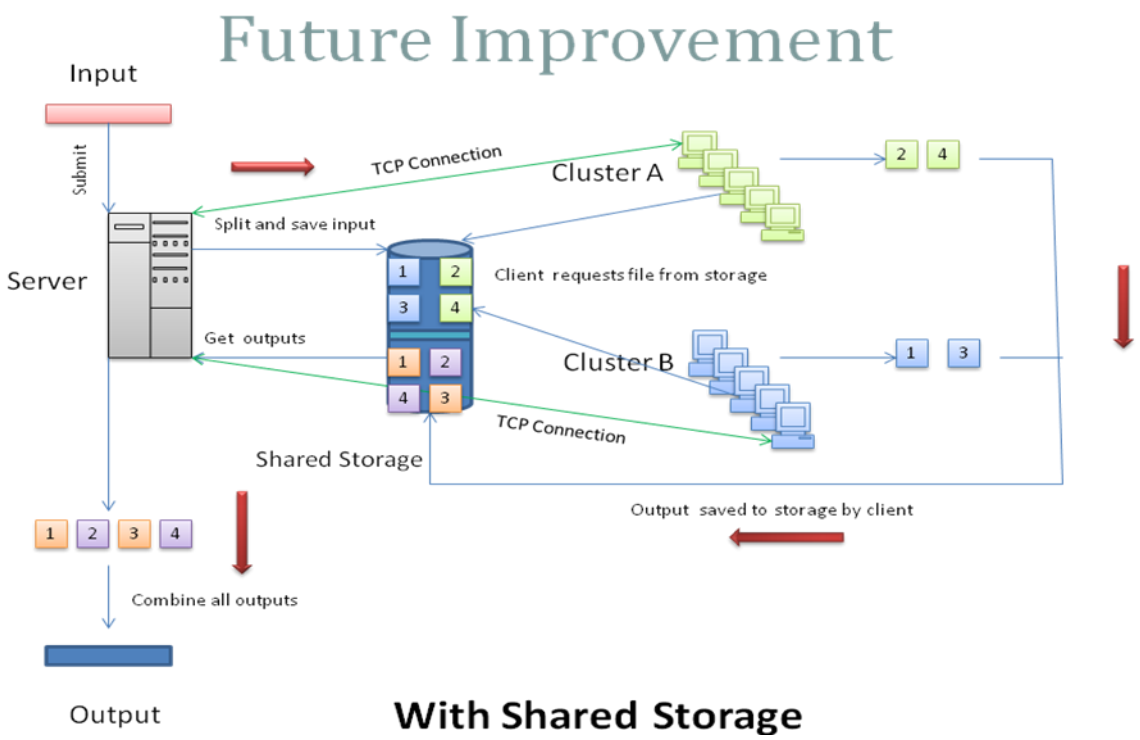

Future Improvement
Input
Submit
TCP Connection
Cluster A
2 4
Split and save input
Server
1 2
3 4
Client requests file from storage
Get outputs
1 2
4 3
Cluster B
1 3
TCP Connection
Shared Storage
Output saved to storage by client
1 2 3 4
Combine all outputs
Output
With Shared Storage

Figure 9

5.3 Steps of solving a problem:

Our designed system works in five different steps to execute traditional executable files to run on a grid system.

5.3.1 **Split Input:** Our designed system doesn't modify program's executable file instead it splits the input file. It is not possible to modify a binary file. So, what we do is we use a splitter to split the input file in such way that we can easily execute each split in different computers. We only need to write code for the splitter or we can use third party splitter program like video splitter to split a large video file.

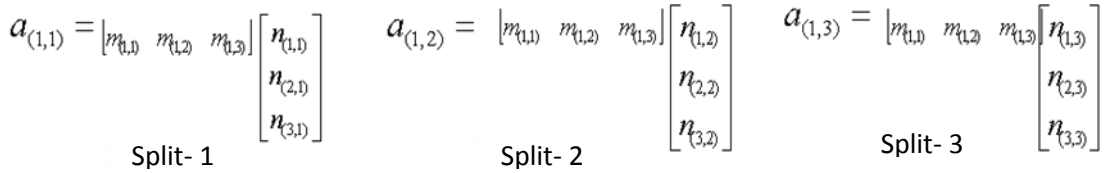
Example:

There are two 3X3 matrices and a answer matrix which is also 3X3. We can solve it by using a generic matrix multiplication algorithm. Let's consider the program that doesn't support grid can solve this problem.

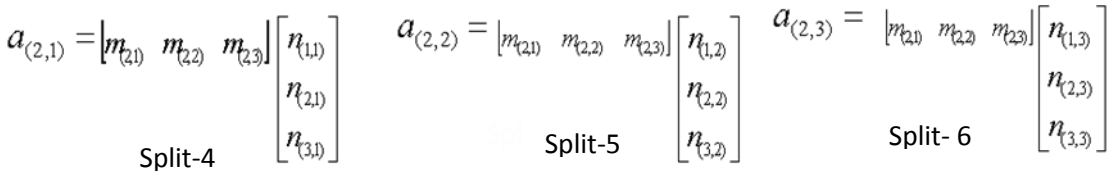
$$\begin{bmatrix} m_{(1,1)} & m_{(1,2)} & m_{(1,3)} \\ m_{(2,1)} & m_{(2,2)} & m_{(2,3)} \\ m_{(3,1)} & m_{(3,2)} & m_{(3,3)} \end{bmatrix} \quad X \quad \begin{bmatrix} n_{(1,1)} & n_{(1,2)} & n_{(1,3)} \\ n_{(2,1)} & n_{(2,2)} & n_{(2,3)} \\ n_{(3,1)} & n_{(3,2)} & n_{(3,3)} \end{bmatrix}$$

$$= \begin{bmatrix} a_{(1,1)} & a_{(1,2)} & a_{(1,3)} \\ a_{(2,1)} & a_{(2,2)} & a_{(2,3)} \\ a_{(3,1)} & a_{(3,2)} & a_{(3,3)} \end{bmatrix}$$

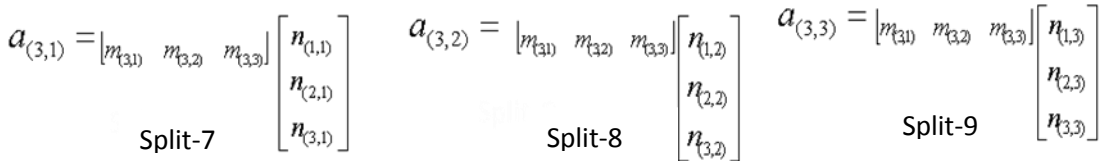
Now if we want to run this program into grid system we need to split it to run each part on each computer. We also have to consider that each part of input is solvable by the multiplication program.



Split- 1 Split- 2 Split- 3



Split-4 Split-5 Split- 6



Split-7 Split-8 Split-9

After splitting the input file we get total 9 split. Each one of them is a complete matrix multiplication program and solvable by previous matrix multiplication program designed to run in non-grid system.

5.3.2 **Distribution:** The input files are the distributed among the computers in grid along with unmodified binary file so that each node can execute each task separately. When a node of grid is free it contacts with server. Server then starts distribution process. In this process server also checks if all outputs are collected. After submitting a task if output is not returned for any task then server submits the task again to another node.

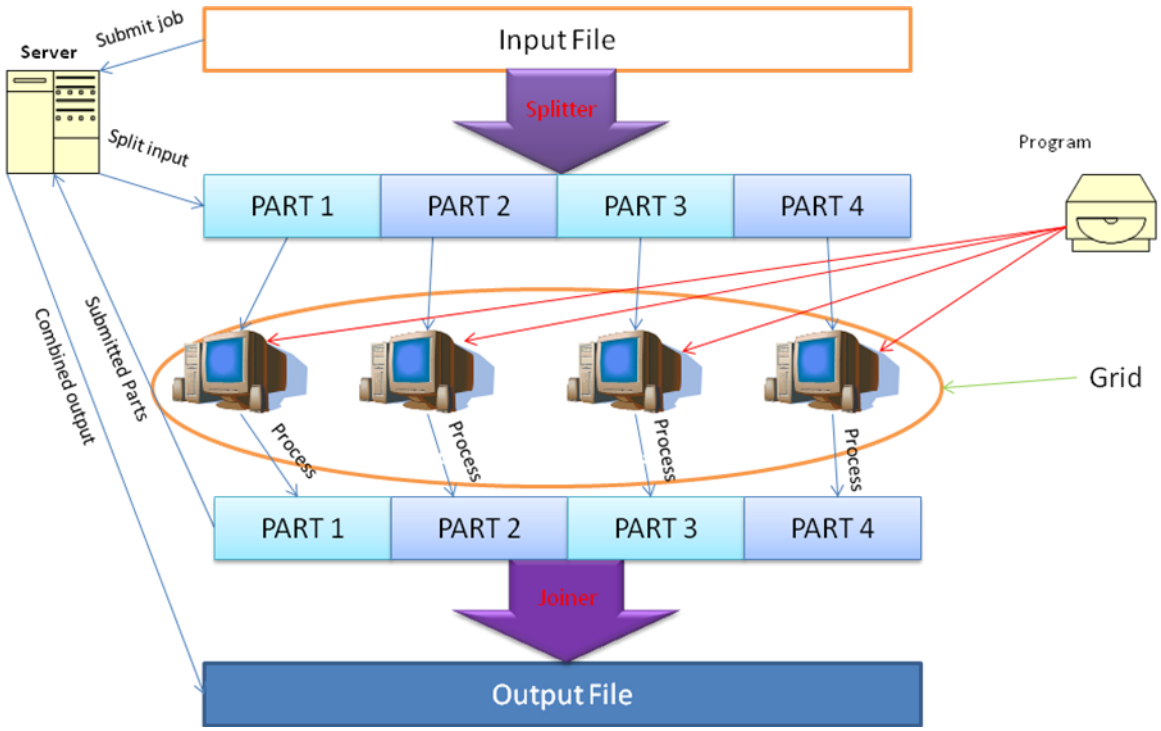
If shared storage is used then when nodes connect to server then server only sends location of shared storage and file list. After those nodes collects inputs and programs from that shared storage.

5.3.3 **Execution:** Each system runs the program with its provided input file and submits the output to server.

5.3.4 **Collection:** Server collects all the input files and stores them into its memory. It collects data until all output is collected.

If shared storage is used then nodes submit its output to shared storage any notifies server. After getting notifications for all task server downloads the result from shared storage to its memory.

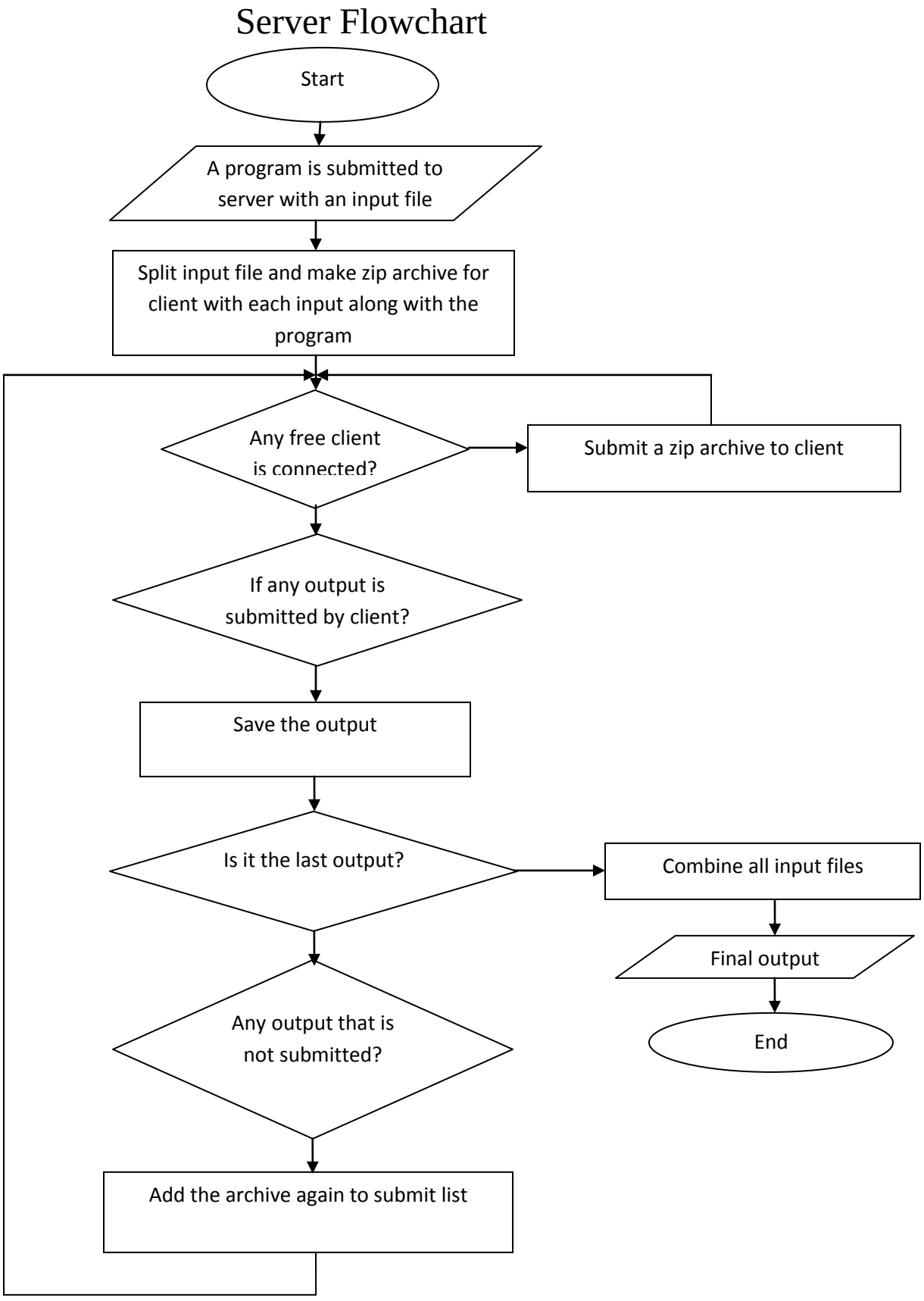
5.3.5 **Combine:** This is the final step of our design. In this step all results are in root/server's memory. Now server runs the joiner program to combine all results and give final output. Joiner program may be any third party program like video joiner.



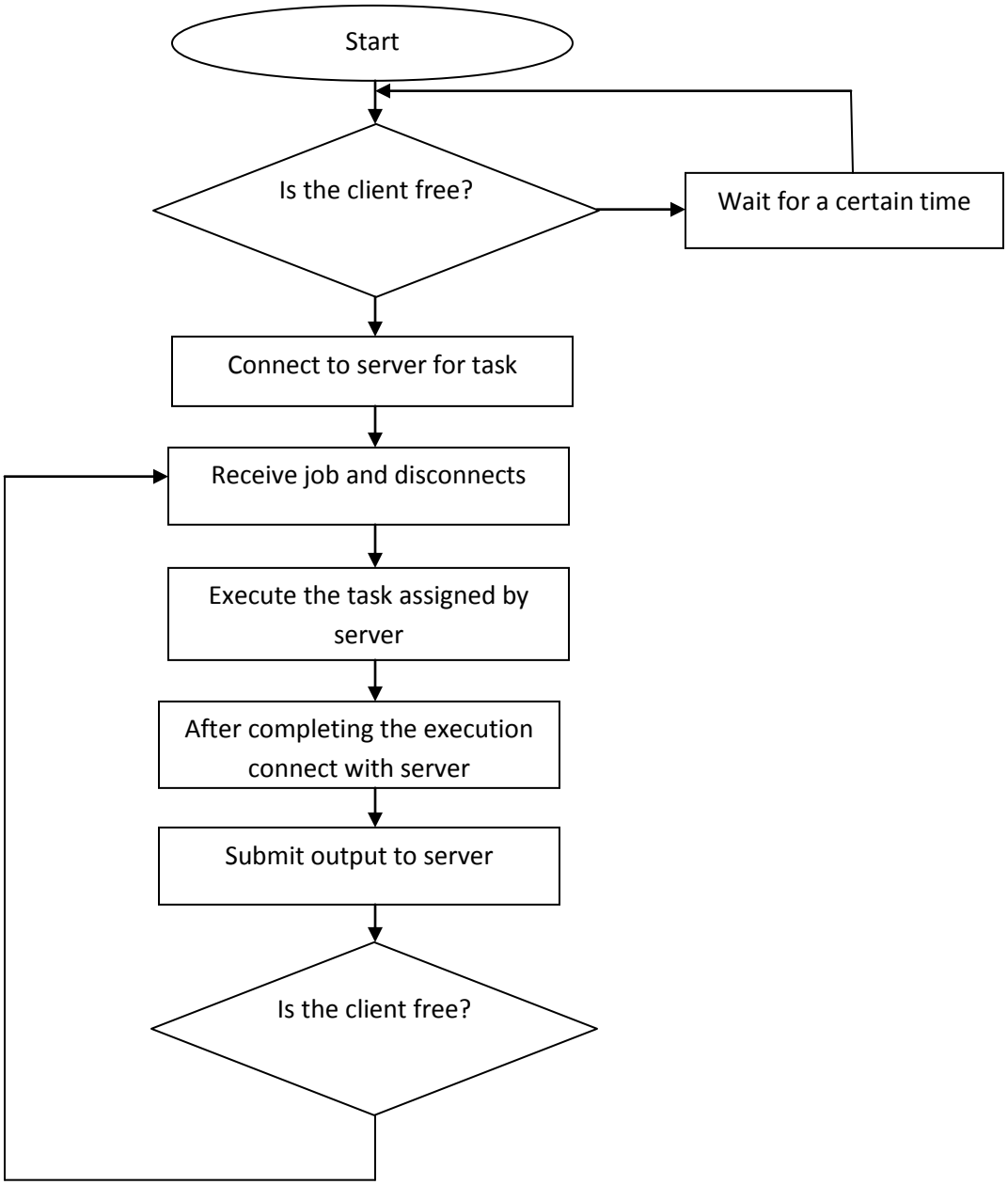
Block diagram of our system

Figure 10

5.4 Flowchart:



Client Flowchart



5.5 A practical scenario:

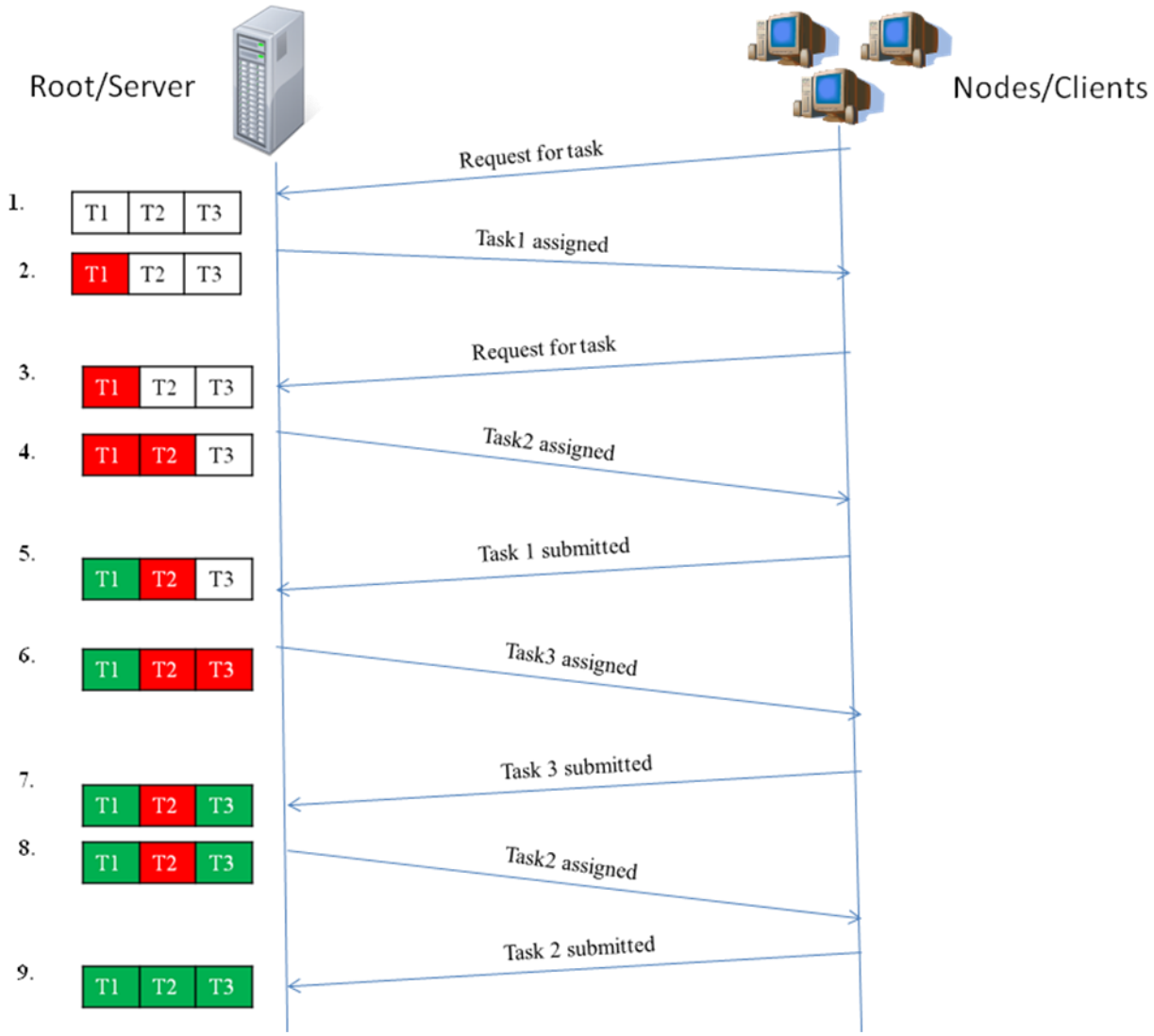


Figure 11

On the above diagram there is a root/server and several nodes/clients. Every node connects with server when it is available (i.e. when nodes are idle). Each time every node connects to server assigns a task to that node. The job of each node is to execute the assigned job and submit the output file to server. When any node submits output then server assigns another task to that node.

Now let's consider the scenario describe above:

1. A task is submitted to server and server split it into three parts. Now a node connects with server.
2. Server assigns 'Task1' to that client. Client got disconnect and start to execute the task.
3. Another client connects with server.
4. Server assigns 'Task2' to that client.
5. First client submit 'Task 1'
6. Server assigns 'Task3' to that client.
7. 'Task3' is submitted.
8. Now server find there is no other task but 'Task2' is pending, so it assigns 'Task2' to that client.
9. 'Task2' is submitted, so job is finished.

5.6 Implementation:

Our designed system is implemented using java. There are two individual programs in this system, one is root and another is node. Root is the main server it is always running and node program run in each nodes/clients in a grid.

5.6.1 Root:

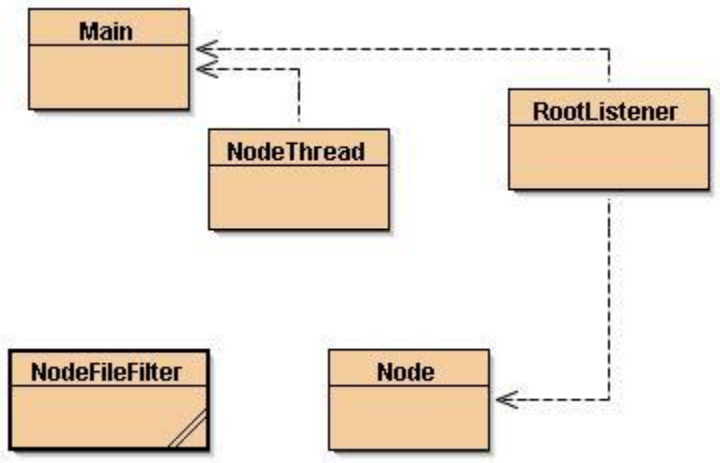


Figure 12: Root

5.6.2 Node:

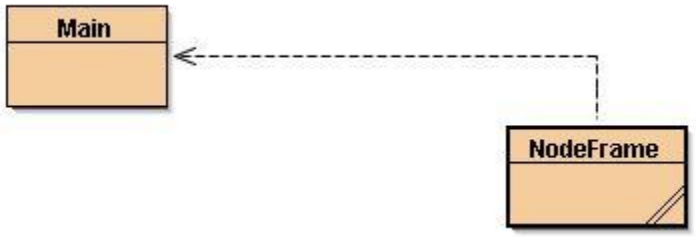


Figure 13: Node

CHAPTER 6

Empirical Study

6.1 Performance Analysis:

6.1.1 Matrix Multiplication:

We've run matrix multiplication program in single computer and grid computing environment. In each case we've used square matrix of different sizes with double values. We use general matrix multiplication algorithm [i.e. time complexity is $O(n^3)$]

6.1.2 **Grid System:** We run the matrix multiplication in grid environment with various numbers of computers and various sizes of matrixes. The result we find is:

Processor	Clock Speed	Memory	OS
Intel Pentium D	2.4 GHz	512 MB	Windows 2000/XP

Table 5: PC Configuration

No.	Matrix	Pc (Nodes)	Data Type	No. of split	Time (S)
1.	10 X 10	27	Double	100	13.031
2.	50 X 50	34	Double	2500	287.047
3.	100 x 100	27	Double	10000	1167.281
4.	100 X 100	36	Double	10000	1112.575
5.	150 X 150	37	Double	22500	2497.505

Table 6: Timing Calculation

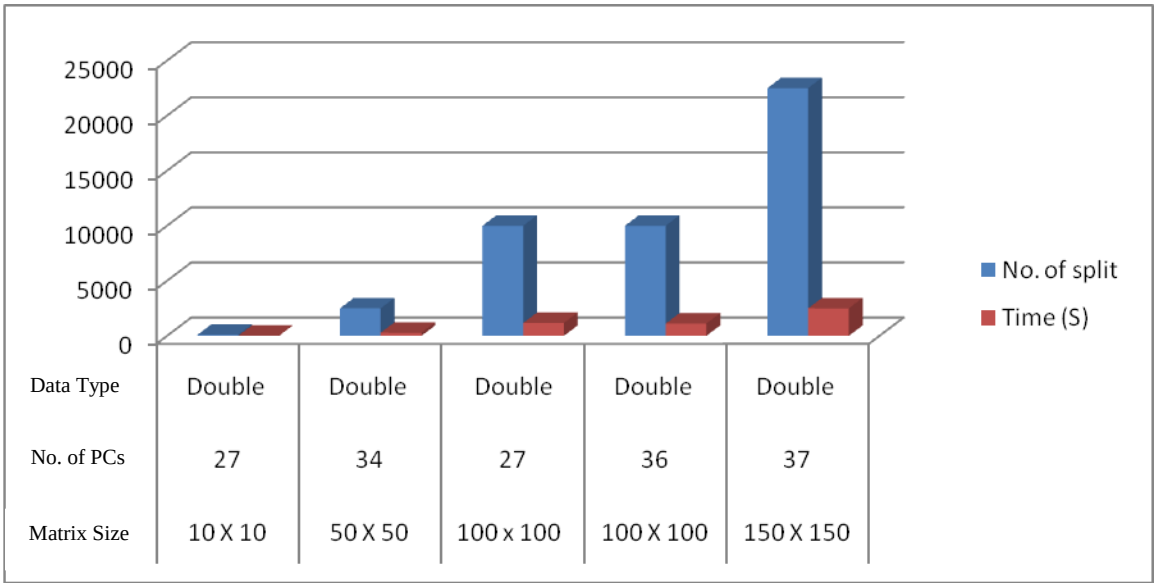


Figure 14: Performance Graph

From the above data table and graph we find a rough measurement of performance of our implemented grid system.

In observation 3&4 we use 100X100 size matrix in a grid containing 27 and 36 nodes. When we used 27 PCs it took 1167.281 second and when we used 36 PCs it took 1112.575 second. That is increase number of PCs slightly reduces processing time. So, we can say, the grid is working, because if we increase the number of nodes then it takes less processing time.

6.1.3 **Resource utilization in grid system:** Here are the data tables of resource utilization of grid system.

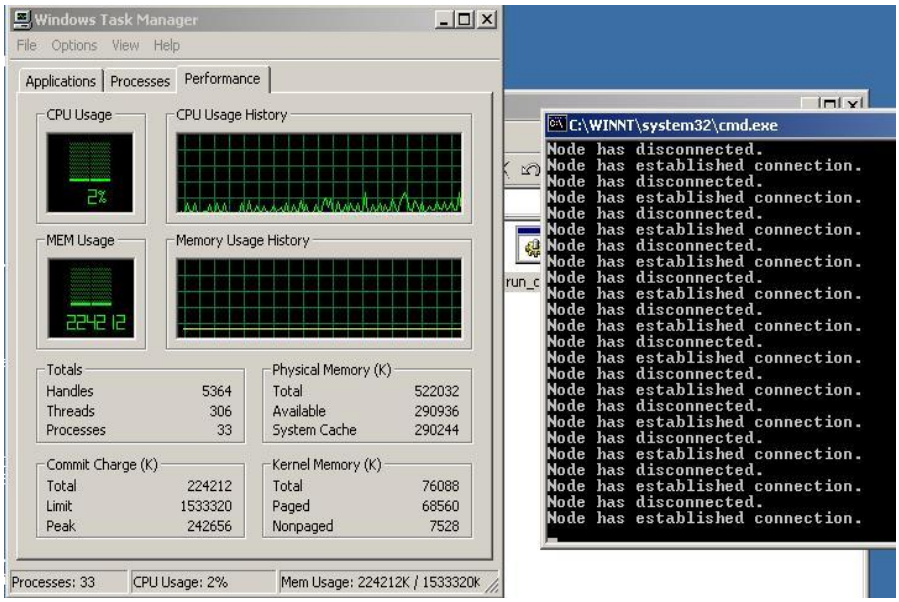


Figure 15: CPU

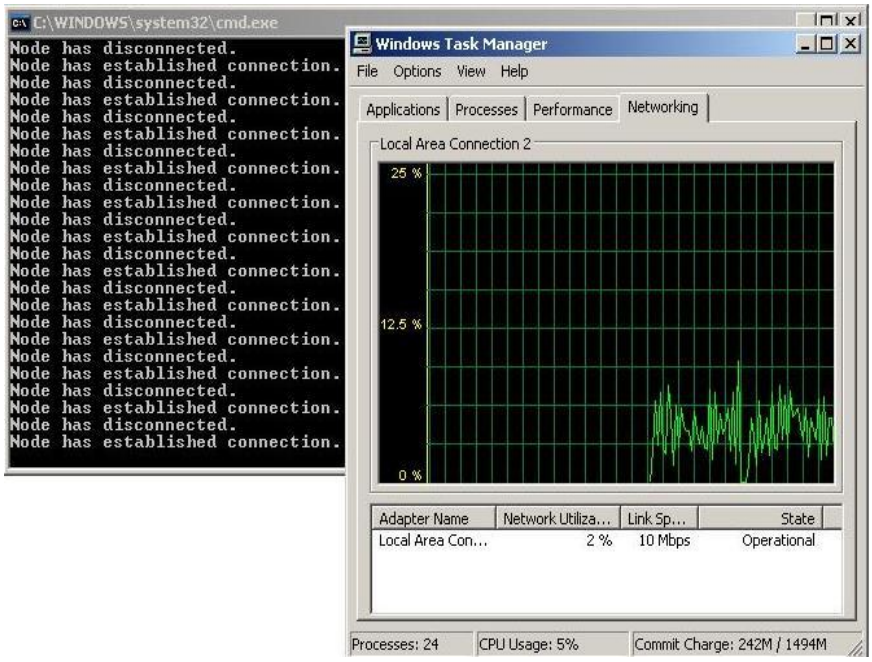


Figure 16: Network

CPU Utilization	Network Utilization	Client/server
>30%	>10%	Client
>50%	>40%	Server

Table 7: Resource Utilization

Matrix	Data Type	No. of split	Time (S)
10 X 10	**Double**	**100**	**0.000406**
50 X 50	**Double**	**2500**	**0.000656**
100 x 100	**Double**	**10000**	**0.000843**
150 X 150	**Double**	**22500**	**0.000928**

Table 8:

From the above data table we find that our designed system works and it is light weight on network, but it cannot use the full processing power of node computers.

Since the task we used is small, it needs less data transfer and less processing time to split. For larger task server activity and network utilization may increase, but we can reduce it by using shared storage. Shared storage will reduce data transfer and communication to node computers of server.

6.2 **Non-Grid System:** In this stage we test the same matrix multiplication program to multiply different size of matrixes in single computer.

Processor	Clock Speed	Memory	OS
Intel Pentium D	2.4 GHz	512 MB	Windows 2000/XP

Table 9: PC Configuration

Timing Calculation

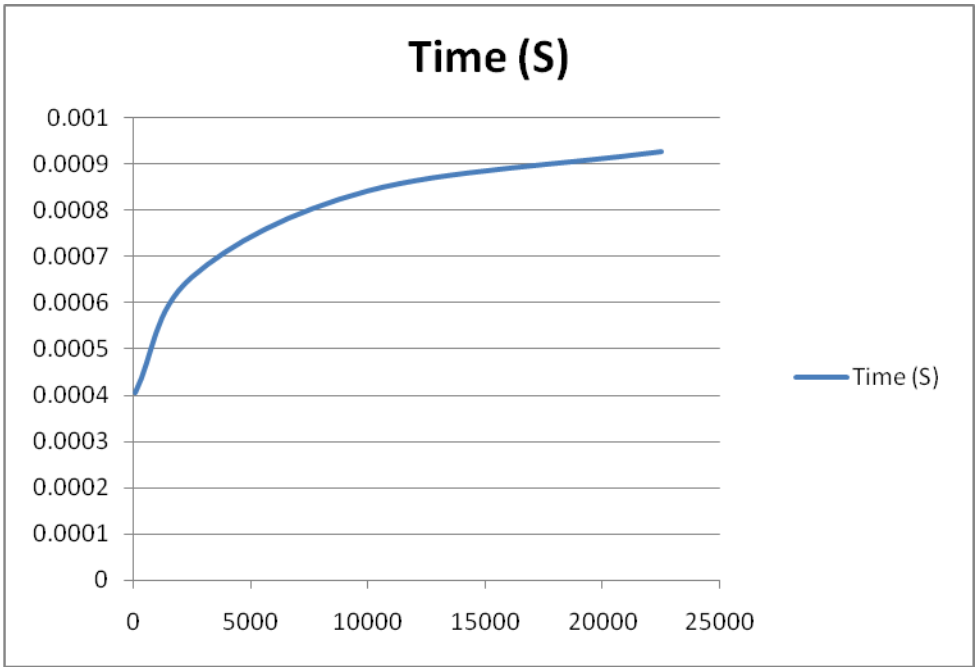


Figure 17: Performance Graph

From the above data table and graph we find that computer is faster, because it uses its full capacity to process data and it doesn't need communication time.

6.3 **How to optimize the system:** Hence the system we developed is in testing stage and no improvement is implemented, our primary result shows single PC is far faster than our designed system. However it is possible to improve the system improve the system. These are:

6.3.1 Increase CPU thread: The computers we used are all multiprocessor system (N.B Intel Pentium D has two CPUs) it actually have multiple slots to execute more than one job at

a time. So we can assign more than one task to node computers and execute them parallel in nodes to use the processing power of nodes efficiently.

6.3.2 Optimized communication: In our current design we send one input file to node along with the program that needs to be executed. This communication is not efficient, because we are sending the program each time server assigns job to a node. This is not efficient because for each task the executable file should send only once.

6.3.3 Send and Receive multiple I/O files together: In our system nodes connect to server receive one input file then disconnects. After processing is done each node connects to server, submit the output and receive a new task. It works fine when each part needs long processing time. But for smaller task it is overhead, because communication time may take more than processing time. On other hand CPU becomes idle during communication, because at that time it has no task to process. So, more communication means idler CPU. We can optimize the system in such way than server will send multiple input files together for smaller task and nodes will submit multiple outputs together. We can also re design the system in such way that during execution a node can receive task and save it for later execution. It will reduce execution time.

6.3.4 Using native code: We use Java for implementing our system. Java uses virtual machine which is slower than native code written in C or C++. So, rewriting the source code using native language will increase its performance.

6.3.5 Use larger task: The programs we tested are not large enough for grid based system. As testing needs huge resource, it was not possible for us to run the system for larger tasks that takes enough time to compare with non-grid system. So, using larger task will make the system better than single computer.

6.3.6 Adding more nodes: In our test we found that, adding more node decrease processing time. We can add more computers to system to reduce processing time.

6.4 Limitations of our system:

As we tried to improve conventional grid system, our system also has some limitations. It also includes the limitations of grid based system. The limitations of our system are as follows:

6.4.1 **Compatibility:** We are dividing task by splitting the input files to maintain compatibility with non-grid system, but all inputs are not possible to split. So, for some jobs that are not possible to split, we have to follow traditional method.

6.4.2 **Cross platform support:** As our program is written in Java, it virtually supports all platforms, but it has a limitation. This system supports cross platform only when same executable file is available for each platform. For example matrix multiplication will work on cross platform only when the same executable file is available for both platforms.

6.4.3 **Security:** In our system security is not yet implemented. It does not support any kind of encryption for communication. It needs to be modified if we want to add security features.

CHAPTER 7

Project Review

7.1 Future Work

Future work for this project can be focused on the different areas as detailed below.

7.1.1 Designed more optimized system:

Our current system is not optimized. For example it cannot use advantages of multiprocessor based system and communication between server and client is not optimized. In future our objective is to optimize these processes. Another optimization can be done by converting the program into a native language. Our current system is developed using Java. It's run on JVM that makes it slower than native program.

7.1.2 Make the program more usable:

Adding user interface in our current system is one of our future plans. Current system doesn't have any GUI. So, it is tough for general user to use this system. To make the system more usable and more interactive developing GUI is an important part.

7.1.3 Adding more features:

Presently our program doesn't have any monitoring tool or controlling tool to monitor and control server and its corresponding nodes. Monitoring and controlling tool is necessary for a grid system. It brings more usability and flexibility to grid.

7.1.4 More testing:

Testing a grid computing system is difficult task. It needs huge resource and plenty of time. As, it is in academic project and we have limitations of resources, we could not test our system properly. Our future plan is to test the system more efficiently and execute much more sample tasks.

7.1.5 Make the system more secure:

Our current system doesn't have any security features. Communication between server and clients needs to me encrypted to avoid man in the middle attack. It also needed to add different access level for different types of grid system users.

7.2 Conclusion

With the enormous flexibility and reliability afforded by computing grids, it may seem surprising that they not more pervasive today. The primary explanation for that fact is that grids exist in the context of a large ecosystem. It is not possible to go to a store and purchase a grid. Roadblocks to wider adoption are both technical and business-oriented in nature. From a technical perspective, it is safe to assume that applications not designed in multiprocessor environments are by default uni-processor applications. They can be executed on a multiprocessor node, but they will not use more than one processor, even if more are available, and hence the total run time won't be shorter.

From a cost perspective, it might be attractive to share resources across organizations, including different companies, even in different countries. Doing so implies additional overhead to ensure data integrity, security, and resource billing. The technology to support these functions is still evolving. The lack of precedents makes potential users squeamish about trusting their code and data to be executed by someone else in a shared resource environment represented by a grid. Therefore, few grids today cross company boundaries. The largest user communities today for grids belong to government and academic research.

This challenge translates directly into opportunity for solution providers and system integrators that can overcome them. As the ecosystem of solutions for grid computing continues to evolve, adoption is likely to increase by private companies that seek to harness the power and cost advantages of grid computing. The project provides background both for those who seek to create those solutions and for those who wish to implement them.

Glossary

Grid Computing - is a form of distributed computing

CPU – Central Processing Unit

Source Code- Lines of code that make up a program.

Source Code Analysis- Analysis of properties of source code.

Cluster - a group of loosely coupled computers that work together closely

NPACI - National Partnership for Advanced Computational Infrastructure

Flops – Floating point operation per second

Virtualization - In computing, virtualization is a broad term that refers to the abstraction of computer resources

Extraterrestrial Intelligence - scientific effort to discover intelligent life elsewhere in the universe

References and Bibliography

- [1] http://www.extremetech.com/article2/0,1558,1153023,00.asp
- [2] http://en.wikipedia.org/wiki/List_of_distributed_computing_projects
- [3] https://computing.llnl.gov/tutorials/parallel_comp/
- [4] http://www.cs.cmu.edu/~scandal/research-groups.html
- [5] Reliability of grid http://www.springerlink.com/index/w77178778r2h2jj3.pdf
- [6] Reliability http://www.ogf.org/OGF_Special_Issue/GridReliabilityDabrowski.pdf
- [7] **Project Ganglia** University of California, Berkeley, Project Ganglia: http://www.millennium.berkeley.edu/
- [8] Introduction to grid computing with globus [IBM Red book]
- [9] University of Wisconsin, Madison. Condor Project http://www.cs.wisc.edu
- [10]Laboratory of parallel and distributed systems, Hungary, http://www.lpds.sztaki.hu/
- [11] University of Cyprus, GridBench, http://grid.ucy.ac.cy/gridbench/
- [12] http://www.lpds.sztaki.hu/projects/current/ikta4-075/ikta4-075.ppt
- [13] http://en.wikipedia.org/wiki/TeraGrid
- [14] http://setiathome.berkeley.edu/
- [15] www.gridcomputing.org/grid2003/

Appendices

Root:

Main.java

```
/**
 * @author Md. Mahtab Uddin & Shyen Muhabbat Shikhder
 * @author 04101002 & 04101008, repectively
 */

package grid.thesis.root;

import java.io.*;
import java.net.*;
import java.util.*;

public class Main {

    public static final int ROOT_PORT = 10051;

    public static int nodeCount = 0;

    private static Vector<Node> nodes = null;

    synchronized public static Vector<Node> getNodes() {
      return nodes;
    }

    private static int jobsDone = 0;

    synchronized public static boolean jobsComplete(boolean nodeThread) {
      if (nodeThread) {
          ++jobsDone;
      }
      return (jobsDone == inputFiles.length);
    }

    public static File[] inputFiles;

    public static File[] programFiles;

    public static File fromNodes;
```

```
public static void main(String[] args) {
  nodes = new Vector<Node>();
  ServerSocket aServerSocket = null;
  Thread rootListener = new Thread(new RootListener(aServerSocket, ROOT_PORT));
  rootListener.start();

  splitInput();
  sendToNodes();

  while (!jobsComplete(false)) {
     try {
          Thread.sleep(1000);
     } catch (InterruptedException e) {
          e.printStackTrace();
     }
  }

  System.out.println("End of ROOT!");

  try {
     rootListener.join();
  } catch (InterruptedException e) {
     e.printStackTrace();
  }

  try {
     Runtime.getRuntime().exec("combine.exe");
  } catch(IOException e) {
     System.err.println("Could not combine split files");
     System.exit(1);
  }
}

private static void splitInput() {
  try {
     Runtime.getRuntime().exec("split.exe");
  } catch (IOException e) {
     System.err.println("Could not invoke input splitter.");
     e.printStackTrace();
     System.exit(1);
  }

  try {
     Thread.sleep(3000);
  } catch (InterruptedException e) {
     e.printStackTrace();
```

```
        }

        inputFiles = new File(System.getProperty("user.dir"))
                .listFiles(new NodeFileFilter("out"));
        programFiles = new File(System.getProperty("user.dir"))
                .listFiles(new NodeFileFilter("matrix"));
    }

    private static void sendToNodes() {
        fromNodes = new File("fromNodes");
        if (!fromNodes.exists()) {
            fromNodes.mkdir();
        }
        PriorityQueue<Integer> jobList = new PriorityQueue<Integer>();
        for (int i = 0; i < inputFiles.length; ++i) {
            jobList.add(i);
        }
        while (true) {
            Integer jobNumber = jobList.poll();
            if (null == jobNumber) {
                //System.out.println("All inputs sent to nodes.");
                break;
            }
            //System.out.println("about to get a node.");
            Node aNode = getAvailableNode();
            aNode.startJob();
            new Thread(new NodeThread(jobNumber, aNode)).start();
        }
    }

    private static Node getAvailableNode() {
        while (true) {
            //System.out.println("about to make a call to getNodes.");
            for (Node aNode : getNodes()) {
                //System.out.println("inside for loop.");
                if (!aNode.isBusy()) {
                    return aNode;
                } else {
                    Thread.yield();
                }
            }
            Thread.yield();
        }
    }
}
```

Node.java

```
/**
 * @author Md. Mahtab Uddin & Shyen Muhabbat Shikhder
 * @author 04101002 & 04101008, repectively
 */
package grid.thesis.root;

import java.net.*;
import java.util.*;

class Node {

    private String id = null;
    private boolean busy;
    private Socket aSocket= null;

    public Node(String id, Socket aSocket) {
      this.id = id;
      this.aSocket = aSocket;
      busy = false;
    }

    public boolean isBusy() {
      return busy;
    }

    public void jobDone() {
      busy = false;
    }

    public void startJob() {
      busy = true;
    }

    public String getId() {
      return id;
    }

    public Socket getSocket() {
      return aSocket;
    }
}
```

NodeFileFilter.java

```
/**
 * @author Md. Mahtab Uddin & Shyen Muhabbat Shikhder
 * @author 04101002 & 04101008, repectively
 */
package grid.thesis.root;

import java.io.File;
import java.io.FilenameFilter;

class NodeFileFilter implements FilenameFilter {

   public NodeFileFilter(String name) {
     this.name = name;
   }

   private String name;

   public boolean accept(File aFile, String name) {
     return name.startsWith(this.name);
   }
}
```

NodeThread.java

```
/**
 * @author Md. Mahtab Uddin & Shyen Muhabbat Shikhder
 * @author 04101002 & 04101008, repectively
 */
package grid.thesis.root;

import java.util.Observable;
import java.io.*;
import java.util.zip.*;

public class NodeThread implements Runnable {

   private int jobNumber;

   private Node aNode;

   public NodeThread(int jobNumber, Node aNode) {
     this.jobNumber = jobNumber;
     this.aNode = aNode;
   }

   public void run() {
     File input = Main.inputFiles[0];
     File program = Main.programFiles[0];
```

```
File output = new File("in" + jobNumber + ".txt");

File zippedFile = zipFilesForSending(input, program);
BufferedInputStream outfile = null;
try {
   outfile = new BufferedInputStream(new FileInputStream(zippedFile));
} catch (FileNotFoundException e) {
   e.printStackTrace();
}

BufferedOutputStream toNode = null;
try {
   byte [] tempBuffer = new byte[4*1024];
   int length = 0;
   toNode = new BufferedOutputStream(aNode.getSocket().getOutputStream());
   while ((length = outfile.read(tempBuffer)) > 0) {
         toNode.write(tempBuffer, 0, length);
   }
   outfile.close();
   toNode.close();
} catch (IOException e) {
   e.printStackTrace();
}

try {
   Thread.sleep(2000);
} catch (InterruptedException e) {
   e.printStackTrace();
}

BufferedInputStream fromNode = null;
try {
   fromNode = new BufferedInputStream(aNode.getSocket().getInputStream());
} catch (IOException e) {
   e.printStackTrace();
}

BufferedOutputStream infile = null;
try {
   byte [] tempBuffer = new byte[4*1024];
   int length = 0;
   infile = new BufferedOutputStream(new FileOutputStream(output));
   while ((length = fromNode.read(tempBuffer)) > 0) {
         infile.write(tempBuffer, 0, length);
   }
   fromNode.close();
   infile.close();
} catch (IOException e) {
   e.printStackTrace();
```

```
        }

        aNode.jobDone();
        Main.jobsComplete(true);

//      FileChannel inChannel = infile.getChannel();
//      ByteBuffer aBuffer = ByteBuffer.allocate((int) zippedFile.length());
//      try {
//          inChannel.read(aBuffer);
//      } catch (IOException e) {
//          e.printStackTrace();
//      }

//      FileOutputStream outfile = null;
//      try {
//          outfile = new FileOutputStream("test.zip");
//      } catch (FileNotFoundException e) {
//          e.printStackTrace();
//      }
//      FileChannel outChannel = outfile.getChannel();
//      aBuffer.flip();
//      try {
//          outChannel.write(aBuffer);
//          outfile.close();
//      } catch (IOException e) {
//          e.printStackTrace();
//      }
    }

    public File zipFilesForSending(File input, File program) {
        FileInputStream infile = null;
        FileInputStream progfile = null;
        try {
            infile = new FileInputStream(input);
            progfile = new FileInputStream(program);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        File zippedFile = new File("out" + jobNumber + ".zip");
        ZipOutputStream out = null;
        try {
            out = new ZipOutputStream(new FileOutputStream(zippedFile));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        try {
            byte[] buffer = new byte[1024];
            int length;
```

```
            out.putNextEntry(new ZipEntry(input.getName()));
            while ((length = infile.read(buffer)) > 0) {
                    out.write(buffer, 0, length);
            }
            out.closeEntry();
            infile.close();
            out.putNextEntry(new ZipEntry(program.getName()));
            while ((length = progfile.read(buffer)) > 0) {
                    out.write(buffer, 0, length);
            }
            out.closeEntry();
            progfile.close();

            out.close();
          } catch (IOException e) {
            e.printStackTrace();
          }

          return zippedFile;
        }

}
```

RootListener.java

```
/**
 * @author Md. Mahtab Uddin & Shyen Muhabbat Shikhder
 * @author 04101002 & 04101008, repectively
 */
package grid.thesis.root;

import java.io.IOException;
import java.net.*;

class RootListener implements Runnable {

    private ServerSocket aServerSocket= null;
    private int rootPort = 0;

    public RootListener(ServerSocket aServerSocket, int rootPort) {
      this.aServerSocket = aServerSocket;
      this.rootPort = rootPort;
    }

    public void run() {
      try {
        aServerSocket = new ServerSocket(rootPort);
      } catch (IOException e) {
        System.err.println("Could not listen on port: " + rootPort);
        e.printStackTrace();
```

```
            System.exit(1);
        }

        while (true) {
            try {
                Socket aSocket = aServerSocket.accept();
                Main.getNodes().add(new Node("Node" + ++Main.nodeCount, aSocket));
            } catch (IOException e) {
                System.err.println("Could not connect to Node.");
                e.printStackTrace();
                System.exit(1);
            }

        }
      }

}
```

Node:

Main.java

```
/**
 * @author Md. Mahtab Uddin & Shyen Muhabbat Shikhder
 * @author 04101002 & 04101008, repectively
 */

package grid.thesis.node;

import javax.swing.*;
import java.net.*;
import java.util.zip.*;
import java.io.*;

public class Main {

    public static final String ROOT_ADDRESS = "192.168.0.182";

    public static final int ROOT_PORT = 10051;

    static JFrame nodeWindow = new NodeFrame("Node running");

    private static Socket aSocket = null;

    public static void main(String[] args) {
      nodeWindow.setVisible(true);
```

```
  try {
     aSocket = new Socket(ROOT_ADDRESS, ROOT_PORT);
  } catch (UnknownHostException e) {
     System.err.println("Could not find root server.");
     e.printStackTrace();
     System.exit(1);
  } catch (IOException e) {
     e.printStackTrace();
  } catch (Exception e) {
     e.printStackTrace();
     System.exit(1);
  }



  while (true) {
     mainLoop();
  }
}

public static void mainLoop() {
  try {
     BufferedOutputStream infile = new BufferedOutputStream(new FileOutputStream("out.zip"));
     BufferedInputStream fromRoot = new BufferedInputStream(aSocket.getInputStream());

     byte[] tempBuffer = new byte[4*1024];
     int length = 0;

     while ((length = fromRoot.read(tempBuffer)) > 0) {
           infile.write(tempBuffer, 0, length);
     }
     fromRoot.close();
     infile.close();
  } catch (IOException e) {
     e.printStackTrace();
  } catch (Exception e) {
     System.exit(1);
  }

  unzipFilesForReading(new File("out.zip"));
  runProgram();

  try {

     BufferedOutputStream toRoot = new BufferedOutputStream(aSocket.getOutputStream());
     BufferedInputStream outfile = new BufferedInputStream(new FileInputStream("output.txt"));

     byte[] tempBuffer = new byte[4*1024];
```

```
        int length = 0;

        while ((length = outfile.read(tempBuffer)) > 0) {
                toRoot.write(tempBuffer, 0, length);
        }
        outfile.close();
        toRoot.close();


    } catch (IOException e) {
        e.printStackTrace();
    } catch (Exception e) {
        System.exit(1);
    }

}

public static void runProgram() {
    try {
        Thread.sleep(3000);

        System.out.println(System.getProperty("user.dir"));

        Runtime.getRuntime().exec("matrixmulfinal.exe");
    } catch (IOException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void unzipFilesForReading(File zippedFile) {
    String inputFilename = "input.txt";
    String progFilename = "matrixmulfinal.exe";

    FileOutputStream outfile = null;
    FileOutputStream progfile = null;
    try {
        outfile = new FileOutputStream(inputFilename);
        progfile = new FileOutputStream(progFilename);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    ZipInputStream in = null;
    try {
        in = new ZipInputStream(new FileInputStream(zippedFile));
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
```

```
            try {
                byte[] buffer = new byte[1024];

                int length = 0;
                in.getNextEntry();
                while ((length = in.read(buffer, 0, buffer.length)) > 0) {
                        outfile.write(buffer, 0, length);
                }
                outfile.close();
                in.closeEntry();

                length = 0;
                in.getNextEntry();
                while ((length = in.read(buffer, 0, buffer.length)) > 0) {
                        progfile.write(buffer, 0, length);
                }
                progfile.close();
                in.closeEntry();

                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }


        static void cleanup() {
            System.out.println("Cleanup running ...");
            try {
                aSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

}
```

NodeFrame.java

```
    /**
     * @author Md. Mahtab Uddin & Shyen Muhabbat Shikhder
     * @author 04101002 & 04101008, repectively
     */


    package grid.thesis.node;

    import javax.swing.JFrame;
    import java.awt.event.*;
```

```
    public class NodeFrame extends JFrame {

      public NodeFrame(final String title) {
        setTitle(title);
        enableEvents(WindowEvent.WINDOW_EVENT_MASK);
      }

      protected void processWindowEvent(final WindowEvent e) {
        if (e.getID() == WindowEvent.WINDOW_CLOSING) {
          Main.cleanup();
          dispose();
          System.exit(0);
        }
        super.processWindowEvent(e);
      }
}
```