

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem Statement . . . . .	1
1.2	Project Scope . . . . .	1
1.3	Methodology . . . . .	2
<b>2</b>	<b>Review of Literature</b>	<b>5</b>
2.1	Introduction to Grid and Grid computing . . . . .	5
2.2	Introduction to Linux . . . . .	6
2.3	Networking in Linux . . . . .	6
2.3.1	Socket Api functions . . . . .	6
2.3.2	The Client Server Model . . . . .	7
2.4	Compiling a Single Source File . . . . .	8
2.5	Threads . . . . .	9
2.5.1	Thread Creation . . . . .	9
2.5.2	Joining Threads . . . . .	10
2.5.3	Compiling and Executing a C++ program with pthread . . . . .	10
2.5.4	Thread Synchronization . . . . .	10
2.6	Graphical User Interface . . . . .	11
2.6.1	Glade Interface Designer . . . . .	12
2.6.2	Building A GTK+ Project using glade . . . . .	13
2.6.3	Compiling GTK+ source files . . . . .	14
2.7	Grid Scheduling Algorithm . . . . .	15
2.7.1	Introduction to Grid Scheduling . . . . .	15
2.7.2	Job Scheduling Framework . . . . .	15
2.7.3	Terms related to scheduling algorithms . . . . .	17

2.7.4	Heuristics . . . . .	17
2.8	GIT . . . . .	22
2.9	Autotools . . . . .	22
2.10	Latex . . . . .	23
<b>3</b>	<b>Design</b>	<b>24</b>
3.1	XML Parser . . . . .	24
3.1.1	Format of Problem Solving Schema(PSS) . . . . .	25
3.2	Networking Protocols . . . . .	28
3.2.1	Connection Establishment Protocol . . . . .	28
3.2.2	Task Transfer and Execution Protocols . . . . .	28
3.3	Execution Monitor Program . . . . .	29
3.4	Graphical User Interface . . . . .	30
3.4.1	Supervisor . . . . .	30
3.4.2	Worker . . . . .	32
3.5	DataFlow Diagrams(DFD) . . . . .	33
3.6	Deployment Diagram . . . . .	36
3.7	Sequence Diagram . . . . .	37
3.8	Collaboration Diagram . . . . .	38
<b>4</b>	<b>Implementation</b>	<b>39</b>
4.1	Networking . . . . .	39
4.2	Proposed Grid Scheduling Algorithms . . . . .	40
4.2.1	Simulation . . . . .	41
4.2.2	Results . . . . .	42
4.3	Execution Monitor Program (EMP) . . . . .	43
4.4	Interfaces . . . . .	44
4.4.1	Command Line Interface (CLI) . . . . .	44
4.4.1.1	Supervisor . . . . .	44
4.4.1.2	Worker . . . . .	45
4.4.2	Graphical User Interface (GUI) . . . . .	45
4.4.2.1	Supervisor . . . . .	46
4.4.2.2	Worker . . . . .	48

<b>5</b>	<b>Further Work</b>	<b>49</b>
<b>6</b>	<b>References</b>	<b>50</b>
<b>7</b>	<b>Acknowledgement</b>	<b>52</b>
<b>8</b>	<b>Summary</b>	<b>53</b>

# List of Figures

2.1	Client Server Model . . . . .	8
2.2	Glade Interface Designer . . . . .	12
2.3	Layered Architecture of Service Grid . . . . .	16
3.1	Grid Computing Framework . . . . .	30
3.2	File Selector . . . . .	30
3.3	Supervisor Execution and Monitoring of Workers . . . . .	31
3.4	Worker side GUI . . . . .	32
3.5	Level 0 DFD . . . . .	33
3.6	Level 1 DFD . . . . .	33
3.7	Level 2 DFD for Supervisor . . . . .	34
3.8	Level 2 DFD for Worker . . . . .	35
3.9	Deployment Diagram . . . . .	36
3.10	Sequence Diagram . . . . .	37
3.11	Collaboration Diagram . . . . .	38
4.1	Simulation Results Graph . . . . .	43
4.2	Supervisor Command line window . . . . .	44
4.3	Worker command line window . . . . .	45
4.4	Supervisor - Task Information . . . . .	46
4.5	Supervisor - Worker Node List . . . . .	47
4.6	Supervisor - Worker Node Information . . . . .	47
4.7	Worker side(Execution Monitor) GUI . . . . .	48
4.8	Worker side(Command window) GUI . . . . .	48

# 1 Introduction

## 1.1 Problem Statement

In a large computer network, such as in Universities, we have a large number of machines running at the same time, but not being used to their full potential. This waste of CPU cycles can be put to use by connecting these machines via a Grid Computing Framework, which can use this vast amount of untapped computing power to solve scientific problems with a divide-and-conquer approach, which might take a long time to complete on a single machine.

## 1.2 Project Scope

The objectives of the project are:

- Accept the problem to be solved from the user, consisting of parallel code units called Tasks, dependency matrix of tasks, etc.
- Distribute these tasks while taking in consideration the inter-dependency of tasks, and using a load balancing algorithm.
- Solve tasks at workers; record the output and errors (if any). Send the output and the error and performance logs to the supervisor.
- Collect outputs and logs from workers. Update worker performance statistics.
- Arrange outputs as desired by the user and present it to the user.
- Provide the user with the capability to modify the execution flow on-the-fly.

## 1.3 Methodology

The project primarily involves three entities.

- User
- Supervisor
- Worker Node(s)

The user is concerned with solving a problem, which is composed of a set of independent / interdependent tasks with separate input sets. The user submits the complete problem to the supervisor node.

The problem consists of the following:

- Problem Solving Schema (PSS): It is an XML document which describes the problem by giving a short summary of the problem, the name of the task files, priority of the tasks, a dependency matrix of the tasks.
- Task File(s): These are C/C++ source code files
- Task File Input Sets: The input to be provided to the respective task files.
- Auxilliary Files: Other files which are needed by the tasks.
- Result Compilation Program: This program collates all the output generated by the independent task files when they are remotely executed on worker nodes. The above set of files is collectively known as the Problem.
- Execution Monitor Program: This program is executed after certain 'checkpoints' (which the user specifies in the PSS). Here, the Supervisor stops issuing new tasks to clients, executes the EMP as per the execution commands in the PSS, and waits for the EMP's commands. The EMP can either ask the Supervisor to stop all or specific tasks, or to redo all or specific tasks, or continue distributing tasks normally.
- The supervisor begins the execution by parsing the PSS. It then uses the dependency matrix to topologically sort the tasks to be performed and then queues as per their priority. The supervisor then distributes the tasks amongst the workers using

a specialized Grid Scheduling Algorithm, which intends to maintain the economy of resource consumption as well as the speed of task solving. As explained, the EMP is executed as an when checkpoint tasks are completed.

The supervisor provides the worker with the following:

- Task Source File(s): These are the source files of the tasks.
- Task Input File(s) and other Auxilliary file(s): This collection of files contains the input set provided to a task and any auxilliary files if required.
- Task Compilation Commands: These commands are used to compile the task source files into executables.
- Task Execution Commands: These commands are used to specify how the executable file(s) have to be run.
- Task Timeout: This is the time limit for the task execution.
- Task Priority: This priority can be used by the Worker to set the niceness of the task executable.

This collection of information is known as a Task. The supervisor packs these files and information into a single tar-gzip archive, and sends it to the respective worker nodes when required. The worker upon receiving the archive, unzips it and compiles the task file. The compiled executable is executed on the worker machine with the specified priority up till either the task completes execution (successfully or unsuccessfully) or the task time outs. Whatever be the result, the worker is obligated to send the following to the supervisor:

- Task Output Files and other Auxilliary files
- Error Logs: Any errors (compile-time or run-time)

This packet of information is collectively known as a Task Execution Result. The supervisor continuously keeps receiving the Task Execution Results. However, some of them might be solved and some might have generated errors. If the task has generated errors, it might be compile-time or run-time errors. But since, a worker node on the grid does not guarantee a perfect environment always, the supervisor retries sending the same Task

to other worker nodes. Repeated failure to get a valid Task Execution Result leaves the supervisor with no choice, but to abandon the task as well as the problem. However, if it receives a valid Task Execution Result, it stores the result and continues with its work. If the completed task was a checkpoint, the EMP is executed and the Supervisor waits for its commands, and takes actions accordingly.

When all the tasks have been completed, the supervisor executes the Result Compilation Program to collate all the output as per the user. After the RCP produces its output, the problem is said to be successfully solved.

Regardless of whether the problem was solved successfully or not, the following information is available with the Supervisor:

- Problem Output: The output generated by the RCP. None, if the problem was abandoned.
- Task Execution Result(s): All the Task Execution Result(s) are stored here so that the user can debug and check the outputs.
- Statistics: Some statistics regarding the problem, like the total computing resources consumed, etc.



## 2 Review of Literature

### 2.1 Introduction to Grid and Grid computing

Grid is basically a network of computers. Grid computing (or the use of computational grids) is the combination of computer resources from multiple administrative domains applied to a common task that requires a great number of computer processing cycles or the need to process large amounts of data. [2]

One of the main strategies of grid computing is using software to divide and apportion pieces of a program among several computers, sometimes up to many thousands. Grid computing is distributed, large scale cluster computing, as well as a form of network distributed parallel processing. The size of grid computing may vary from being small confined to a network of computers. It is a form of distributed computing whereby a “super and virtual computer” is composed of a cluster of networked loosely coupled computers acting in concert to perform very large tasks. This technology has been applied to computationally intensive scientific, mathematical, and academic problems through volunteer computing, and it is used in commercial enterprises for such diverse applications as drug discovery, economic forecasting, seismic analysis, and back-office data processing in support of e-commerce and Web services.

In grid computing, network programming, identical to socket programming or client-server programming, involves writing computer programs that communicate with other programs across a computer network. The program or process initiating the communication is called a worker process, and the program waiting for the communication to be initiated is the supervisor process. The worker and supervisor processes together form a distributed system. The communication between the worker and supervisor process may either be connection-oriented (such as an established TCP virtual circuit or session), or connectionless (based on UDP datagram).

## 2.2 Introduction to Linux

The name "Linux" comes from the Linux kernel, originally written in 1991 by Linus Torvalds. Linux is an implementation of the UNIX design philosophy, which means that it is a multiuser system. This has numerous advantages, even for a system where only one or two people will be using it. Security, which is necessary for protection of sensitive information, is built into Linux at selectable levels. More importantly, the system is designed to multitask. Whether one user is running several programs or several users are running one program, Linux is capable of managing the traffic.

## 2.3 Networking in Linux

### 2.3.1 Socket Api functions

Sockets are usually implemented by an API library such as Berkeley sockets. [4,16]

This list is a summary of functions or methods provided by the Berkeley sockets API library:

- `socket()` creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it.
- `bind()` is typically used on the supervisor side, and associates a socket with a socket address structure, i.e. a specified local port number and IP address.
- `listen()` is used on the supervisor side, and causes a bound TCP socket to enter listening state.
- `connect()` is used on the worker side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.
- `accept()` is used on the supervisor side. It accepts a received incoming attempt to create a new TCP connection from the remote worker, and creates a new socket associated with the socket address pair of this connection.
- `send()` and `recv()`, or `write()` and `read()`, or `recvfrom()` and `sendto()`, are used for sending and receiving data to/from a remote socket.

- `close()` causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.

### 2.3.2 The Client Server Model

Most interprocess communication uses the client server model. These terms refer to the two processes which will be communicating with each other. One of the two processes, the worker, connects to the other process, the server, typically to make a request for information. A good analogy is a person who makes a phone call to another person. <sup>[1]</sup>

The worker needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the worker prior to the connection being established. Once a connection is established, both sides can send and receive information. The system calls for establishing a connection are somewhat different for the worker and the server, but both involve the basic construct of a socket. A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

The steps involved in establishing a socket on the client side are as follows:

1. Create a socket with the `socket()` system call.
2. Connect the socket to the address of the server using the `connect()` system call.
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the `read()` and `write()` system calls.

The steps involved in establishing a socket on the server side are as follows:

1. Create a socket with the `socket()` system call.
2. Bind the socket to an address using the `bind()` system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the `listen()` system call.
4. Accept a connection with the `accept()` system call. This call typically blocks until a worker connects with the supervisor.
5. Send and receive data

The following figure illustrates the example of client/server relationship of the socket APIs for connection-oriented protocol (TCP):

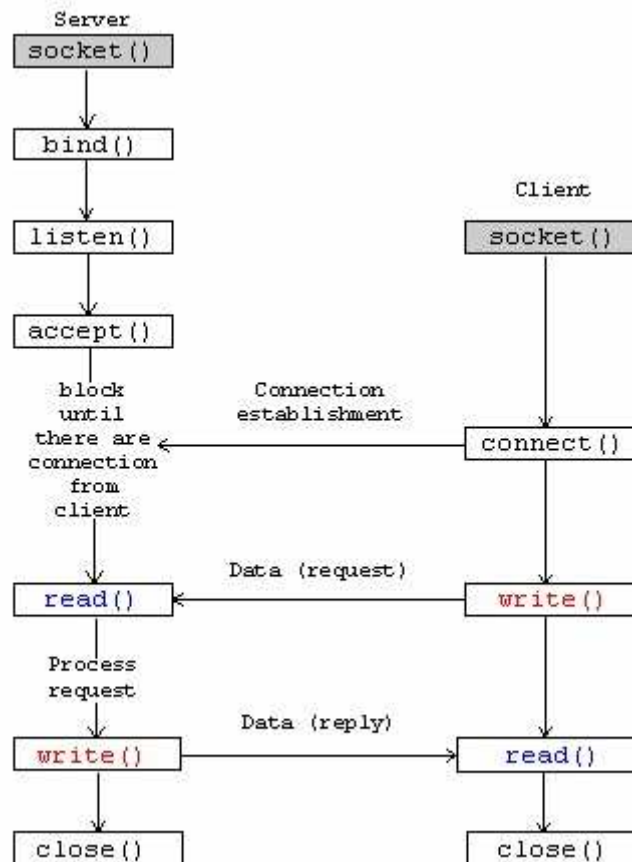


Figure 2.1: Client Server Model

## 2.4 Compiling a Single Source File

The name of the C++ compiler is `g++`. To compile a C++ source file, you use the `-c` option. So, for example, entering this at the command prompt compiles `main.cpp` as

```
% g++ -c main.cpp
```

The resulting object file is named `main.o`

The `-o` option gives the name of the file to generate as output from the link step.

```
% g++ -o main main.o
```

Now you can run `main` like this:

```
. /main
```

## 2.5 Threads

Threads, like processes, are a mechanism to allow a program to do more than one thing at a time. As with processes, threads appear to run concurrently; the Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute. Conceptually, a thread exists within a process. Threads are a finer-grained unit of execution than processes. While invoking a program, Linux creates a new process and in that process creates a single thread, which runs the program sequentially. That thread can create additional threads; all these threads run the same program in the same process, but each thread may be executing a different part of the program at any given time. <sup>[3]</sup>

GNU/Linux implements the POSIX standard thread API (known as pthreads). All thread functions and data types are declared in the header file `<pthread.h>`. The pthread functions are not included in the standard C library. Instead, they are in `lib pthread`, so it is required to add `-lpthread` to the command line while linking a program.

### 2.5.1 Thread Creation

Each thread in a process is identified by a thread ID. When referring to thread IDs in C or C++ programs, type `pthread_t` is used. Upon creation, each thread executes a thread function. This is just an ordinary function and contains the code that the thread should run. When the function returns, the thread exits. On GNU/Linux, thread functions take a single parameter, of type `void*`, and have a `void*` return type. The parameter is the thread argument: GNU/Linux passes the value along to the thread without looking at it. The program can use this parameter to pass data to a new thread. Similarly, the program can use the return value to pass data from an exiting thread back to its creator.

The `pthread_create` function creates a new thread. It is provided with the following:

1. A pointer to a `pthread_t` variable, in which the thread ID of the new thread is stored.
2. A pointer to a thread attribute object. This object controls details of how the thread interacts with the rest of the program. If you pass `NULL` as the thread attribute, a thread will be created with the default thread attributes.

3. A pointer to the thread function. This is an ordinary function pointer, of this type:  
`void* (*) (void*).`
4. A thread argument value of type `void*`. Whatever you pass is simply passed as the argument to the thread function when the thread begins executing.

A call to `pthread_create` returns immediately, and the original thread continues executing the instructions following the call. Meanwhile, the new thread begins executing the thread function. Linux schedules both threads asynchronously, and program must not rely on the relative order in which instructions are executed in the two threads.

### 2.5.2 Joining Threads

One solution is to force `main` to wait until the other two threads are done. What is needed is a function similar to `wait` that waits for a thread to finish instead of a process. That function is `pthread_join`, which takes two arguments: the thread ID of the thread to wait for, and a pointer to a `void*` variable that will receive the finished thread's return value. If you don't care about the thread return value, pass `NULL` as the second argument.

### 2.5.3 Compiling and Executing a C++ program with pthread

Compiling of a C++ source file `thread_demo.cpp` which uses `pthread` is done as follows:

```
% g++ -c thread_demo.cpp -lpthread
```

Linking the `thread_demo.o` object file :

```
% g++ -o thread_demo thread_demo.o -lpthread
```

Executing the `thread_demo.out` :

```
./thread_demo
```

### 2.5.4 Thread Synchronization

Programming with threads is very tricky because most threaded programs are concurrent programs. In particular, there's no way to know when the system will schedule one thread to run and when it will run another. One thread might run for a very long time, or the

system might switch among threads very quickly. On a system with multiple processors, the system might even schedule multiple threads to run at literally the same time.

To allow threads to cooperate and compete for resources, it is necessary to provide mechanisms for synchronization and communication. Semaphores are used for the synchronization of threads.

A **semaphore** is a counter that can be used to synchronize multiple threads. Each semaphore has a counter value, which is a non-negative integer.

A **semaphore** supports two basic operations:

- A *wait* operation decrements the value of the semaphore by 1. If the value is already zero, the operation blocks until the value of the semaphore becomes positive (due to the action of some other thread). When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns.
- A *post* operation increments the value of the semaphore by 1. If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore, one of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero).

To use semaphores in Linux with C++, include `<semaphore.h>`

A semaphore is represented by a *sem\_t* variable. Before using it, it is initialized using the *sem\_init* function, passing a pointer to the *sem\_t* variable. The second parameter should be zero and the third parameter is the semaphore's initial value. If no longer needed, it is deallocated with *sem\_destroy*.

To wait on a semaphore, use *sem\_wait*. To post to a semaphore, use *sem\_post*. To retrieve the current value of a semaphore, use *sem\_getvalue*.

## 2.6 Graphical User Interface

GTK+ is a library for creating graphical user interfaces. It works on many UNIX-like platforms, Windows, and on framebuffer devices. GTK+ is released under the GNU Library General Public License (GNU LGPL), which allows for flexible licensing of worker applications. GTK+ has a C++-based object-oriented architecture that allows for maximum flexibility.<sup>[6,13]</sup>

GTK+ depends on the following libraries:

- GLib: A general-purpose utility library, not specific to graphical user interfaces. GLib provides many useful data types, macros, type conversions, string utilities, file utilities and a main loop abstraction.
- GTK+: The GTK+ library itself contains widgets, that is, GUI components such as #GtkButton or #GtkTextView.

### 2.6.1 Glade Interface Designer

Glade Interface Designer is a graphical user interface builder for GTK+, with additional components for GNOME. The Designer enables to create and edit user interface designs for GTK+ applications.

The GTK+ library provides an extensive collection of user interface building blocks such as text boxes, dialog labels, numeric entries, check boxes, and menus. These building blocks are called *widgets*. Glade can be used to place widgets in a GUI. Glade allows to modify the layout and properties of these widgets. Glade can be used to add connections between widgets and application source code. [17]

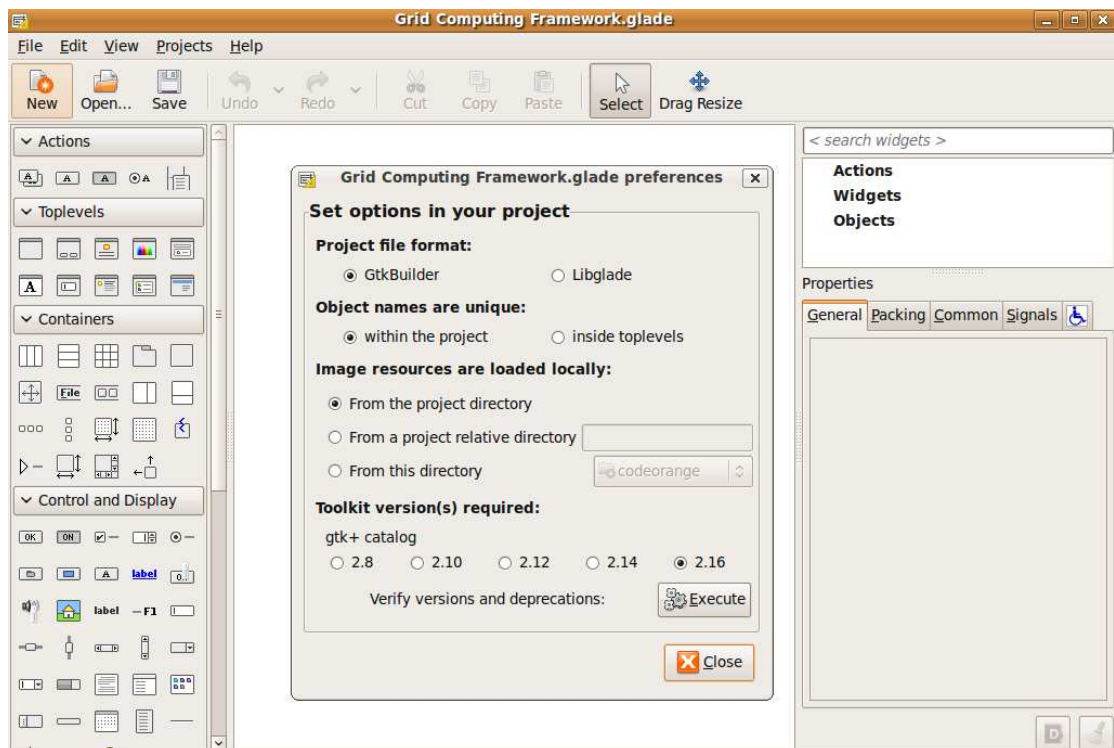


Figure 2.2: Glade Interface Designer



## 2.6.2 Building A GTK+ Project using glade

Glade builds all the files needed to generate the GUI for an application when *Build* is clicked in the main glade window.

- In the `project#` directory

### **acconfig.h/autogen.sh/configure.in/Makefile.am/stamp-h.in**

These files are all necessary for the autoconf/autogen/automake systems which automate compiling and library dependencies in Linux. Generally Glade users just need to type `./autogen.sh` to configure their application and generate the Makefiles needed.

- In the `project#/src` directory

### **main.cpp**

This file contains `main()`, the initial function for all C++ programs.

### **support.c**

This file contains glade support functions and should not be edited. In this file is included `lookup_widget()` function, which allows to find a particular widget by name given another widget.

### **callbacks.c**

This file is where glade writes all the callbacks and signal handlers. Glade does not rewrite this file, but only appends to it as more callbacks are required.

### **interface.cpp**

This is another file, like `support.c`, which should not be edited. Here Glade writes out the functions to create the GUI.

### **Makefile.am/Makefile.in/Makefile**

These files are created by the automake/autoconf packages and provide the guidelines used by `gcc` to compile the application.

### 2.6.3 Compiling GTK+ source files

Compiling gtk+ source code files can be difficult, specially when it is required to include several source files and have to type the compiling command everytime to do it.

Makefiles are special format files that together with the make utility help to automatically build and manage the projects.

On running *make*, the program will look for a file named *makefile* in your directory, and then execute it.

After this the build process proceeds as:

1. Compiler takes the source files and outputs object files
2. Linker takes the object files and creates an executable

The trival way to compile a file `main.cpp` and obtain an executable, is by running the command:

```
g++ -c main.cpp `pkg-config --cflags gtk+-2.0` `pkg-config --libs gtk+-2.0`
```

To link the object file `main.o`:

```
g++ -o main main.o `pkg-config --cflags gtk+-2.0` `pkg-config --libs gtk+-2.0`
```

## 2.7 Grid Scheduling Algorithm

### 2.7.1 Introduction to Grid Scheduling

A Grid computing system is a system which has various machines to execute a set of tasks. High performance is essential for Grid computing systems in the field of natural science and engineering for large scale simulation. Therefore scheduling algorithms are required which assign tasks to machines in a heterogeneous grid computing system. The scheduling algorithm determines the execution order of the tasks which will be assigned to machines. Since the problem of allocating independent jobs in heterogeneous computational resources is known as NP-complete, an approximation or heuristic algorithm is highly desirable.

In the scheduling algorithms, we consider that the tasks randomly arrive the system. We assume that the scheduling algorithms are nonpreemptive, i.e., tasks must run to completion once they start, and the tasks will meet the deadlines on machines they are allocated. All the tasks are independent, i.e., there is no synchronization or communication among the tasks. In the on-line mode, a task is assigned to a machine as soon as it arrives at the scheduling system.

### 2.7.2 Job Scheduling Framework

Grid takes the following characteristics into account:

- **Dynamics of Grid:** The capacity of computation or services is distributed and dynamic. The available resources of each node vary over time and the changes of resources affect the Grid system. For example, when a new service is added in a node, other nodes in the same Grid must be aware of the change at once. However, this change shall have very little impact on other Grid nodes. A hierarchical architecture can meet these requirements. The capacity of resource management is different at different levels.
- **Autonomy of Grid nodes:** Each node is independent. A node itself determines whether or not to run a new job, how to schedule it and how to optimize resource utilization. The strength of this autonomy varies at different levels of the Grid

architecture. Generally higher a level, the more autonomous it is.

- **Diversity of tasks:** Various tasks consume different resources, including computing power and applications. The constraints on each task are also different. Some tasks require to be done in the shortest execution time while others require the lowest cost. Even some tasks don't specify any constraint. Since the resources required by tasks could be fully distributed in the entire Grid system, it is necessary to classify tasks and design diversified scheduling policies and algorithms for different task categories.
- **Adaptation of Grid:** The functions of Grid are realized through middle services. In order to increase the resource utilization and support QoS(Quality of Service), the services should be adaptive to historical and predictive information.

Logically, the architecture is divided into three levels: the Grid Manager level, the Grid level and the Node level. The Node level can be a computer, a service-provider and a storage resource.

The Grid level consists of machine groups, each of which are called Worker Nodes, in which all nodes usually belong to one organization. For example in the figure below, Grid 1 may belong to the Computer Science Department Lab 1 and Grid 2 may belong to the Computer Science Department Lab 2. Each grid can be regarded as a unified entity, and all nodes in it have a common objective. Though, a grid can fully centrally control the resources of its nodes, it cannot utilize the resources of nodes in other grids directly.

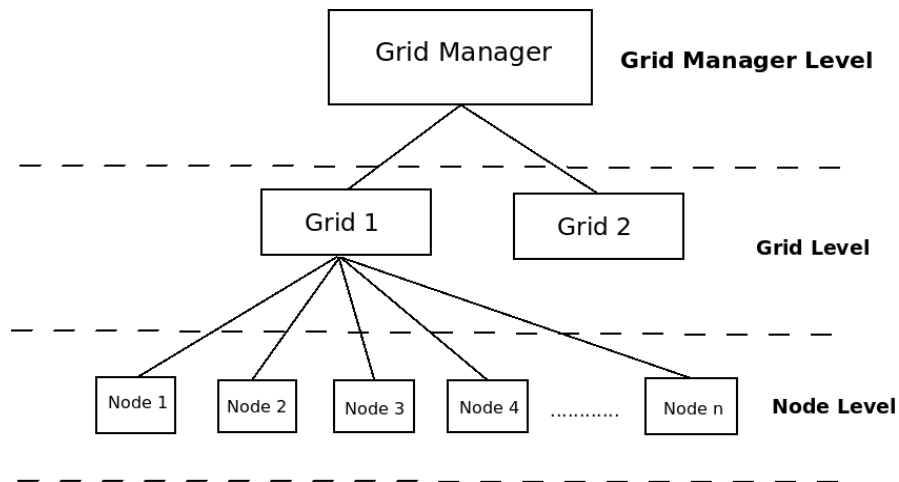


Figure 2.3: Layered Architecture of Service Grid

The Grid Manager (GM) level can make multiple Grids work in concert. GM must maintain a unique entry address list of the supervisors of all Grids in order to locate them. Besides the entry address list, an overall resource list is also kept in GM and updated asynchronously. The actual resource management and scheduling services are distributed in the GM.

### 2.7.3 Terms related to scheduling algorithms

- **Metatask** be defined as a collection of independent tasks with no intertask data dependencies.
- The size of the metatask (i.e., the number of tasks to execute),  $\tau$
- The number of machines in the HC (Heterogeneous Computing) suite,  $\mu$   
( $\tau$  and  $\mu$  are static and known beforehand)
- An accurate estimate of the expected execution time for each task on each machine is known prior to execution and contained within a  $\tau \times \mu$  **ETC** (expected time to compute) **matrix**.
- $\tau \times 1$  **baseline column vector**,  $B$ , of floating point values is created.
- The amount of variance among the execution times of tasks in the metatask for a given machine is defined as *task heterogeneity*,  $\Phi_b$ .
- *Machine heterogeneity*,  $\Phi_r$ , represents the variation that is possible among the execution times for a given task across all the machines.

### 2.7.4 Heuristics

A lot of research has been done in the field of scheduling tasks on a set of heterogeneous machines. The following heuristics are some of the commonly known<sup>[7, 8, 9, 10, 11, 12]</sup>

- **OLB**: Opportunistic Load Balancing (OLB) assigns each task, in arbitrary order, to the next machine that is expected to be available, regardless of the task's expected execution time on that machine. The intuition behind OLB is to keep all machines as busy as possible. One advantage of OLB is its simplicity, but because

OLB does not consider expected task execution times, the mappings it finds can result in very poor makespans.

- **MET:** Minimum Execution Time (MET) assigns each task, in arbitrary order, to the machine with the best expected execution time for that task, regardless of that machine's availability. The motivation behind MET is to give each task to its best machine. This can cause a severe load imbalance across machines.
- **MCT:** Minimum Completion Time (MCT) assigns each task, in arbitrary order, to the machine with the minimum expected completion time for that task. This causes some tasks to be assigned to machines that do not have the minimum execution time for them. The intuition behind MCT is to combine the benefits of OLB and MET.
- **Min min:** The Min min heuristic begins with the set  $U$  of all unmapped tasks. Then, the set of minimum completion times, task  $t_i \in U$  is found. Next, the task with the overall minimum completion time from  $M$  is selected and assigned to the corresponding machine (hence the name Min min). Last, the newly mapped task is removed from  $U$ , and the process repeats until all tasks are mapped (i.e.,  $U$  is empty). Min min is based on the minimum completion time, as is MCT. However, Min min considers all unmapped tasks during each mapping decision and MCT only considers one task at a time.
- **Max min:** The Max min heuristic is very similar to Min min. The Max min heuristic also begins with the set  $U$  of all unmapped tasks. Then, the set of minimum completion times,  $M$ , is found. Next, the task with the overall maximum completion time from  $M$  is selected and assigned to the corresponding machine (hence the name Max min). Last, the newly mapped task is removed from  $U$ , and the process repeats until all tasks are mapped (i.e.,  $U$  is empty)
- **Sufferage:** This heuristic, as given in <sup>[12]</sup> aims to map that task to the most suitable machine, which will 'suffer' the most if that machine is not assigned to it. In other words, the task whose difference between its best completion time and the second best completion time amongst all the yet unmapped tasks is the greatest, is mapped on to the machine where it gets its best completion time. This process

is repeated till all tasks are mapped.

- **GA:** Genetic Algorithms (GAs) are a technique used for searching large solution spaces. Consider  $X$  chromosomes where each chromosome is a  $\tau \times 1$  vector.

```
initial population generation;  
evaluation;  
while(stopping criteria not met) {  
    selection;  
    crossover;  
    mutation;  
    evaluation;  
}  
output best solution;
```

The initial population is generated using two methods: (a)  $X$  randomly generated chromosomes from a uniform distribution, or (b) one chromosome that is the Min min solution and  $X-1$  random solutions. The GA actually executes eight times (four times with initial populations from each method), and the best of the eight mappings is used as the final solution.

Each chromosome has a fitness value, which is the makespan that results from the matching of tasks to machines within that chromosome. After the generation of the initial population, all of the chromosomes in the population are evaluated based on their fitness value, with a smaller fitness value being a better mapping.

Next, the crossover operation selects a random pair of chromosomes and chooses a random point in the first chromosome. For the sections of both chromosomes from that point to the end of each chromosome, crossover exchanges machine assignments between corresponding tasks.

After crossover, the mutation operation is performed. Mutation randomly selects a chromosome, then randomly selects a task within the chromosome, and randomly reassigns it to a new machine.

Finally, the chromosomes from this modified population are evaluated again. This completes **one iteration** of the GA. The GA stops when any one of three conditions are met: (a) 1000 total iterations, (b) no change in the elite chromosome for, say, 150 iterations, or (c) all chromosomes converge to the same mapping. Until the stopping criterium is met, the loop repeats, beginning with the selection step. This is the algorithm, as proposed by [8].

- **SA:** Simulated Annealing (SA) is an iterative technique that considers only one possible solution (mapping) for each metatask at a time. This solution uses the same representation as the chromosome for the GA. SA uses a procedure that probabilistically allows poorer solutions to be accepted to attempt to obtain a better search of the solution space. This probability is based on a system temperature that decreases for each iteration. As the system temperature cools, it is more difficult for poorer solutions to be accepted. The initial system temperature is the makespan of the initial (random) mapping.

The initial SA procedure implemented here is as follows. The first mapping is generated from a uniform random distribution. The mapping is mutated in the same manner as the GA, and the new makespan is evaluated. If the new makespan is better, the new mapping replaces the old one. If the new makespan is worse (larger), a uniform random number  $z \in [0, 1)$  is selected. Then,  $z$  is compared with  $y$ , where

$$y = \frac{1}{1 + e^{\left(\frac{oldmakespan - newmakespan}{temperature}\right)}}$$

If  $z > y$  the new (poorer) mapping is accepted; otherwise it is rejected, and the old mapping is kept.

- **KPB**(K-Percent Best) first finds  $(k/100)*m$  (where  $m$  is no of machines) best machines based on the execution time for the task, and the algorithm calculates the minimum one among selected machine completion times, and then the algorithm assigns the task to the machine which has the minimum. The time to get the subset of machines is  $O(m \log m)$  because the time for sorting  $m$  execution times is needed. The time to determine the machine which has minimum completion time is  $O(m)$ . Overall time complexity of KPB is  $O(m \log m)$ .



- **Tabu:** Tabu search is a solution space search that keeps track of the regions of the solution space which have already been searched so as not to repeat a search near these areas. A solution (mapping) uses the same representation as a chromosome in the GA approach.

The implementation of Tabu search used here begins with a random mapping as the initial solution, generated from a uniform distribution. To manipulate the current solution and move through the solution space, a short hop is performed. The intuitive purpose of a short hop is to find the nearest local minimum solution within the solution space. The basic procedure for performing a short hop is to consider, for each possible pair of tasks, each possible pair of machine assignments, while the other  $\tau - 2$  assignments are unchanged. This is done for every possible pair of tasks.

The pseudocode for the short hop procedure is shown as follows:

```

LOOP:      /*begin short hop procedure*/
  for ti = 0 to  $\tau - 1$  /* first task in pair */
    for mi = 0 to  $\mu - 1$  /* first machine in pair */
      for tj = ti to  $\tau - 1$  /* second task in pair */
        for mj = 0 to  $\mu - 1$  /* second machine in pair */
          if (ti == tj)
            evaluate new solution
              with task tj on machine mj;
          else
            evaluate new solution with
              task ti on machine mi and
              task tj on machine mj;
          if (new solution is better)
          {
            replace old solution with new solution;
            successful_hops = successful_hops + 1 ;
            goto LOOP; /*restart from initial state*/
          }
        if (successful_hops == limithops)

```

```

                                goto END; /*end all searching*/
                                end for
                                end for
                                end for
                                end for
END:

```

The important heuristics were studied and compared with our proposed algorithms in the section 4.2.

## 2.8 GIT

git is a free & open source, distributed version control system designed to handle everything from small to very large projects with speed and efficiency. <sup>[15]</sup>

Every Git clone is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server. Branching and merging are fast and easy to do.

Git is used for version control of files, much like tools such as Mercurial, Bazaar, Subversion, CVS, Perforce, and Visual SourceSafe.

Some of the projects that are using GIT are Linux Kernel, Gnome, Fedora, Qt , Debian etc.

We used GIT to track the development of Grid Computing Framework the repository can be found at <http://github.com/reddragon/Grid-Computing-Framework>

## 2.9 Autotools

Autotools is the linux standard for software delivery . Autotools makes use of the **make** utility for compiling the source packages for a certain target machine.It also makes use of various shell scripts to compile and configure the source .The advantage of using autotools is that we can allow the user to modify the configuration for building the files according to his machine preferences.

It also makes it easy to uninstall the software by giving a simple script which is defined by the software provided.

## 2.10 Latex

L<sup>A</sup>T<sub>E</sub>X is a document preparation system for high-quality typesetting <sup>[18]</sup>. It is most often used for medium-to-large technical or scientific documents but it can be used for almost any form of publishing. It is based on the idea that authors should be able to focus on the content of what they are writing without being distracted by its visual presentation. In preparing a L<sup>A</sup>T<sub>E</sub>X document, the author specifies the logical structure using familiar concepts such as chapter, section, table, figure, etc., and lets the L<sup>A</sup>T<sub>E</sub>X system worry about the presentation of these structures. It therefore encourages the separation of layout from content while still allowing manual typesetting adjustments where needed.

Sample L<sup>A</sup>T<sub>E</sub>X code:

```
\documentclass[12 pt]{ article }
\usepackage{amsmath}
\title{\LaTeX}
\date{}
\begin{document}
  \maketitle
  \LaTeX{} is a document preparation system for the \TeX{}
  typesetting program.
  % This is a comment; it is not shown in the final output.
  % The following shows a little of the typesetting power of LaTeX
  \begin{align}
    E &= mc^2 && \\
    m &= \frac{m_0}{\sqrt{1-\frac{v^2}{c^2}}}
  \end{align}
\end{document}
```

We used the L<sup>A</sup>X Document Processor for working with the L<sup>A</sup>T<sub>E</sub>X system. <sup>[19]</sup>

## 3 Design

### 3.1 XML Parser

XML (Extensible Markup Language) is a set of rules for encoding documents electronically. Some of the key elements in XML document are as follows:

- **(Unicode) Character**

By definition, an XML document is a string of characters. Almost every legal Unicode character may appear in an XML document.

- **Processor and Application**

The processor analyzes the markup and passes structured information to an application. The specification places requirements on what an XML processor must do and not do, but the application is outside its scope. The processor (as the specification calls it) is often referred to colloquially as an XML parser.

- **Markup and Content**

The characters which make up an XML document are divided into markup and content. Markup and content may be distinguished by the application of simple syntactic rules. All strings which constitute markup either begin with the character "<" and end with a ">", or begin with the character "&" and end with a ";". Strings of characters which are not markup are content.

- **Tag**

A markup construct that begins with "<" and ends with ">".

- **Element**

A logical component of a document which either begins with a start-tag and ends with a matching end-tag, or consists only of an empty-element tag. The characters

between the start- and end-tags, if any, are the element's content, and may contain markup, including other elements, which are called child elements.

### 3.1.1 Format of Problem Solving Schema(PSS)

The problem solving schema for our project :

```
<problem>
  <description>
    <name>Distributed Sort of a Large Array</name>
    <problemid>p00001</problemid>
    <userid>User 1</userid>
    <purpose>
      To do a distributed sort on a large array
    </purpose>
  </description>
</tasks>
  <task>
    <taskid> t1 </taskid>
    <taskpriority> 19 </taskpriority>
    <dependencies>    </dependencies>
    <tasktimeout> 1 </tasktimeout>
    <networklatencytime> 0.2 </networklatencytime>
    <tasksourcepath>
      <ts>t1.cpp</ts>
    </tasksourcepath>
    <taskinputsetpath>
      <ti>t1_inp.inp</ti>
    </taskinputsetpath>
    <taskoutputsetpath>
      <to>t1_out.out</to>
    </taskoutputsetpath>
    <taskcompilecommand>
      <tc>g++ -o t1 t1.cpp</tc>
```

```

        </taskcompilecommand>

        <taskexecutioncommand> ./t1 </taskexecutioncommand>

    </task>
</tasks>

<rcp> <rcpsourcepath>./rcp</rcpsourcepath> </rcp>

<emp>

    <emptimeout>2</emptimeout>

    <checkpoints> <cp>t1</cp> </checkpoints>

    <executioncommands><ec>./emp</ec> </executioncommands>

</emp>
</problem>

```

**The tags used in the above PSS :**

- <problem>: It is the root tag. It includes the child tags <description>, <tasks>, <rcp> and <emp>.
- <description>: This tag describes the problem to be solved. It includes the following tags as its child tags <name>,<problemid>,<purpose>,<userid>
- <name>: It contains the name of the problem.
- <problemid>: It uniquely identifies the problem.
- <purpose>: It defines the purpose of the problem.
- <userid>: It is used to identify which user has submitted the problem.
- <tasks>: It includes all the tasks in the problem.
- <task>: It defines a single task,its input file,output file,compile command,execution command etc.
- <taskid>: It uniquely defines a task wrt the current problem.
- <taskpriority>: It defines the priority of the current task.
- <dependencies>: It defines list of tasks dependent on the current task, using the <do> </do> tags and the task-id of the task on which the task under question is dependent, written between the tags.

- `<tasktimeout>`: It defines the maximum time allocated to complete the execution of the current task.
- `<networklatencytime>`: It defines the maximum delay in network that can be tolerated by the supervisor.
- `<tasksourcepath>`: It defines the source code files for the current task.
- `<taskinputsetpath>`: It defines the input files for the execution of the current task.
- `<taskoutputsetpath>`: It defines the output files for the current task.
- `<taskcompilecommand>`: It defines the compile command for the current task.
- `<taskexecutioncommand>`: It defines the compile command for the current task.
- `<rcp>`: RCP stands for Result compilation Program. This program collates all the output generated by the independent task files when they are remotely executed on worker nodes.
- `<rcpsourcepath>`: It defines the execution command for RCP.
- `<emp>`: Emp stands for Execution Monitor Program. It defines the checkpoints and execution command. It is used to monitor the execution of tasks on worker machines.
- `<checkpoints>`: It is used to specify the tasks which are checkpoints, i.e. the tasks, whose completion will be followed by the execution of the EMP.
- `<executioncommands>`: It defines the execution command for the EMP.

The parser scans the input PSS file and checks the elements for any incorrect tags, markup and content. If any errors are found, they are notified to the user and the user needs to modify the PSS file he/she has submitted. If no errors are found, the supervisor starts the execution as explained.

## 3.2 Networking Protocols

### 3.2.1 Connection Establishment Protocol

- **SUPERVISORPING**: This command is used by the supervisor to broadcast its address.
- **WORKERRESP**: This command is used by a worker to acknowledge the **SUPERVISORPING** from the supervisor. The worker also sends its metrics piggybacked with this message.

### 3.2.2 Task Transfer and Execution Protocols

- **COLLECTFILE**: It is sent by the supervisor to the worker on which the task is scheduled. It tells the worker the name of the task file and its size, and the port id of the port to be used.

eg. **COLLECTFILE**                      task.tar.gz      14168978              2

file_send_command	file_name	file_size (bytes)	port_id
-------------------	-----------	-------------------	---------

- **COLLECTFILEACK**: On receiving the **COLLECTFILE** command the worker understands that it has been chosen for the next task and sends a **COLLECTFILEACK** if it is not busy.

eg. **COLLECTFILEACK**                      task.tar.gz

file_send_command	file_name
-------------------	-----------

- **COLLECTRES**: This command is sent by the worker to the supervisor after it has finished computing the tasks and is about to send the task execution results.

eg. **COLLECTRES**                      task\_op.tar.gz              14168978

file_send_command	file_name	file_size (bytes)
-------------------	-----------	-------------------

- **COLLECTRESACK**: This command is sent by the Supervisor to the worker which has sent the **COLLECTRES** command when it is free to receive the file.

eg. **COLLECTRESACK**                      task\_op.tar.gz



### 3.3 Execution Monitor Program

The execution monitor program specifications are defined in the <emp> tag of PSS. The main purpose of the EMP is to add flexibility to the system, so that a wider variety of problems can be solved. Whenever a checkpoint task is completed, the Supervisor writes the task-ids of the completed tasks onto a file, which is read by the EMP. The EMP in turn, responds by writing the appropriate commands on to a specific file. The supervisor suspends any further task execution until the commands can be read from that file.

The following are the commands, which form a part of the protocol used with EMP :

- CONTINUE ALL : It is used when EMP does not want to interfere with the normal execution.
- STOP ALL : It is used when EMP detects that no further execution of all tasks should be continued.
- STOP task : It is used when EMP detects that no further execution of a particular task should be continued.
- REDO task : EMP issues this command when it deems it appropriate to redo a task which has already completed. The task may now have a different input set, or other changes.
- REDO ALL : This command is issued when all the tasks completed till now have to be redone.

## 3.4 Graphical User Interface

### 3.4.1 Supervisor

The Supervisor consists of 3 screens

- Grid Computing Framework, File Selector, Supervisor Execution and Monitoring of Workers.



Figure 3.1: Grid Computing Framework

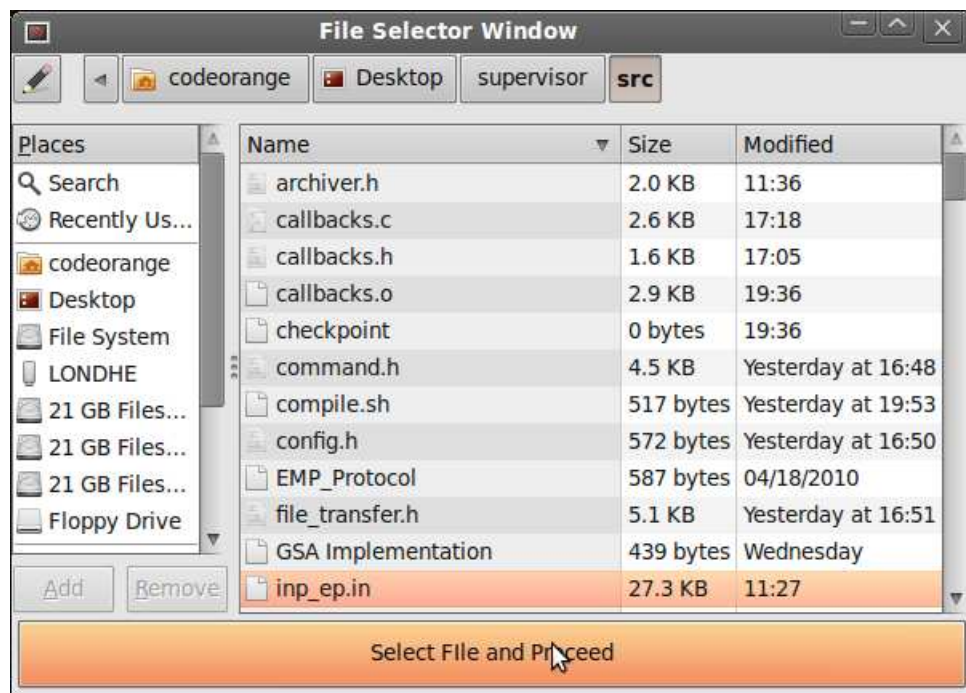


Figure 3.2: File Selector

The figure 3.1 is the first screen in Supervisor side GUI. It includes two buttons, the first is used to recover from a failure and second is the start execution of new tasks. If user selects the first button, the problem begins execution, but the tasks which had been

completed before are not executed again. If the user clicks on the second button, a fresh problem can be submitted. The File Selector in figure 3.2 helps in selecting the PSS file.

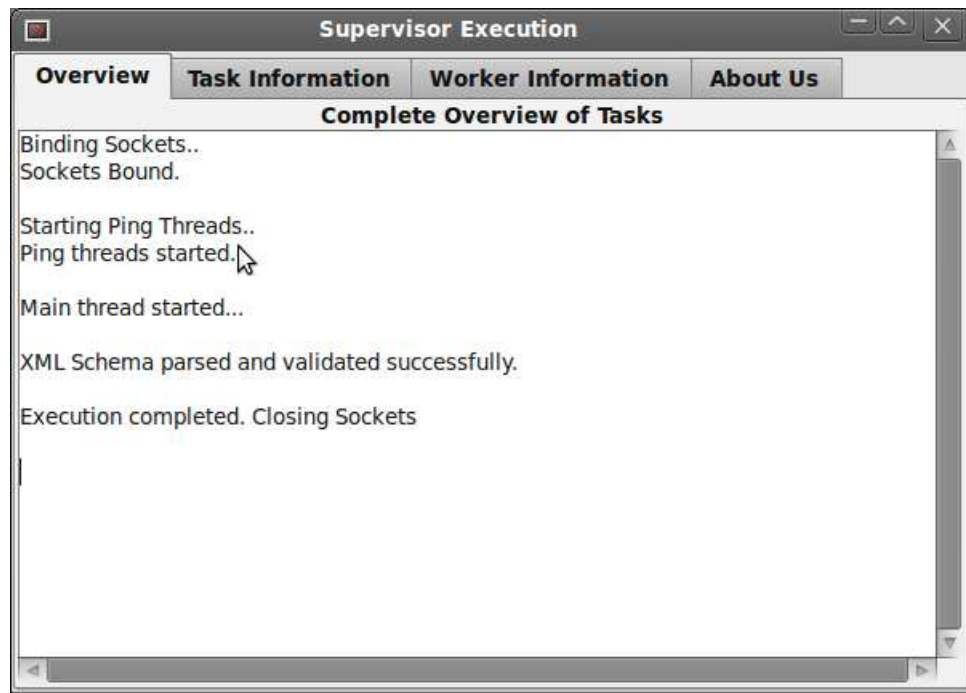


Figure 3.3: Supervisor Execution and Monitoring of Workers

The figure shows the Supervisor's GUI. The 'Overview' tab displays important messages related to the overall execution of the problem. The 'Task Information' tab displays commands / messages pertaining to the execution of the tasks. The 'Worker Information' tab allows the user to see information related to a specific worker, like the IP Address, Performance Metric (related to the computational capacity of the node) and Network Metric (related to the cloggedness of the Node - Supervisor network link).

### 3.4.2 Worker

The following figure represents the worker side GUI

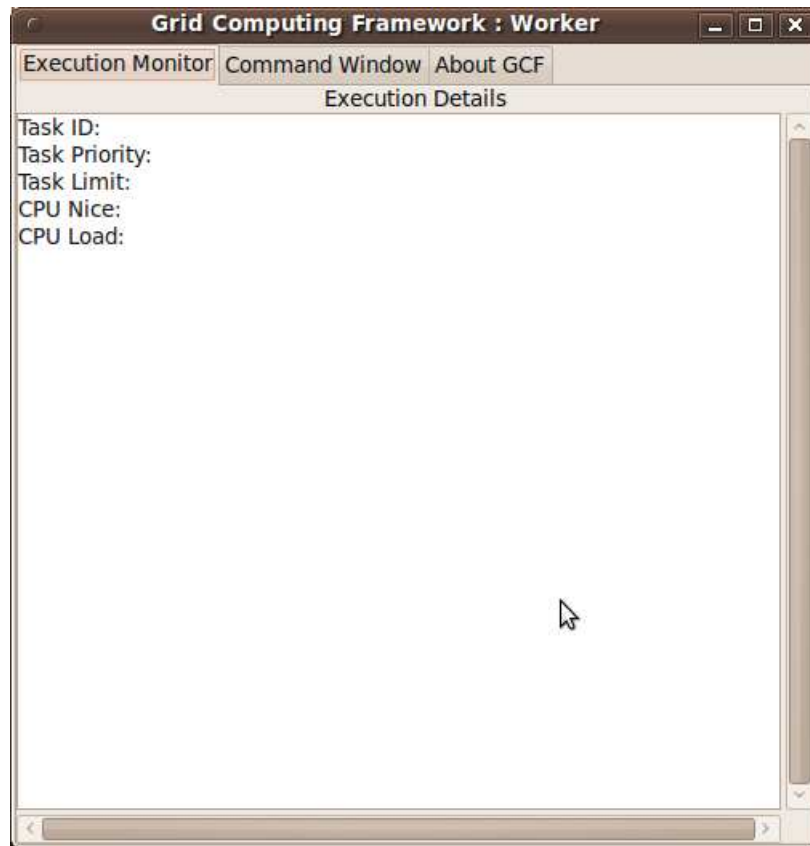


Figure 3.4: Worker side GUI

The 'Execution Monitor' describes the Task-Id, Priority, Time Limit, CPU Nice and the CPU Load of the task being currently executed. The second tab is the command window, which displays important messages pertaining to the execution of tasks on the worker.

### 3.5 DataFlow Diagrams(DFD)

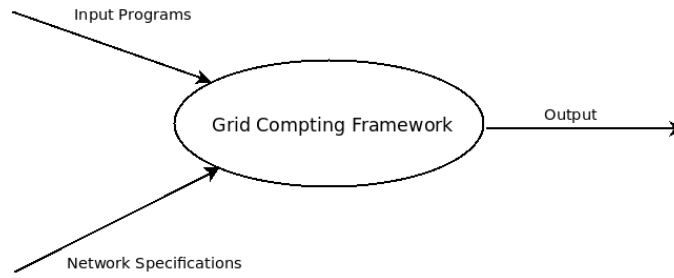


Figure 3.5: Level 0 DFD

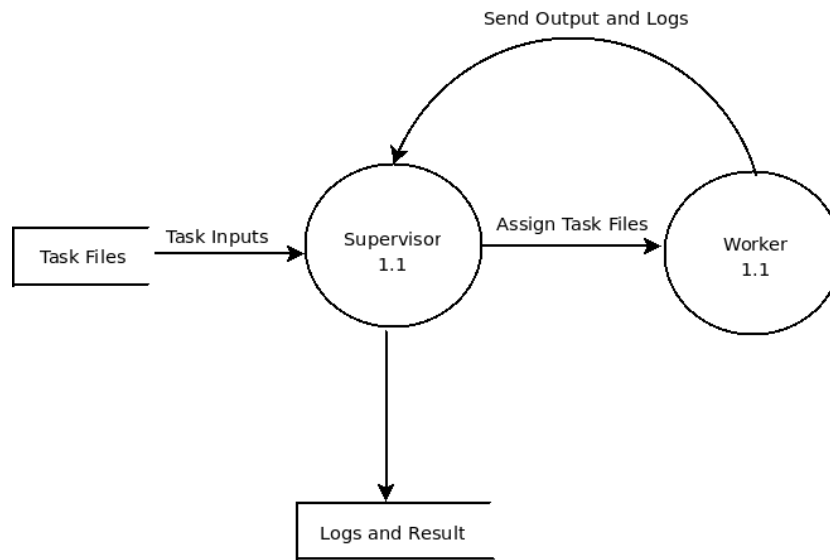


Figure 3.6: Level 1 DFD

- **Level 0 DFD:** The level 0 DFD consists of three main things, namely, the input program(s), the Network specifications and the output file. Initially the user would specify the LAN attributes as of who would be the supervisor and connect the workers to the supervisor via a LAN. Next, the user will give as input the program file(s) which need to be processed by the Grid Computing Framework. The final output would be given by the supervisor in a fashion desired by the user.
- **Level 1 DFD:** The level 1 DFD describes the functioning of the system in more detail. Initially, as mentioned in level 0 DFD, the user would decide which machine to use as a supervisor and then would specify the LAN configurations. These LAN specifications would be known only to the supervisor machine. Also the user would give as an input the program file(s) to the supervisor. But before actually reading

the files, they would be parsed, using a XML parser on the supervisor, in a format which can be understood by the supervisor programs.

Then the supervisor would use this parsed file and distribute tasks to the “active” workers based on the Grid Scheduling Algorithm and also the metrics of the worker. This worker would then perform the task assigned to it and generate some output. This output would then be sent to the supervisor.

The supervisor would, thus, receive the output from each and every worker. It then runs RCP to finally arrange them all in a format wanted by the user.

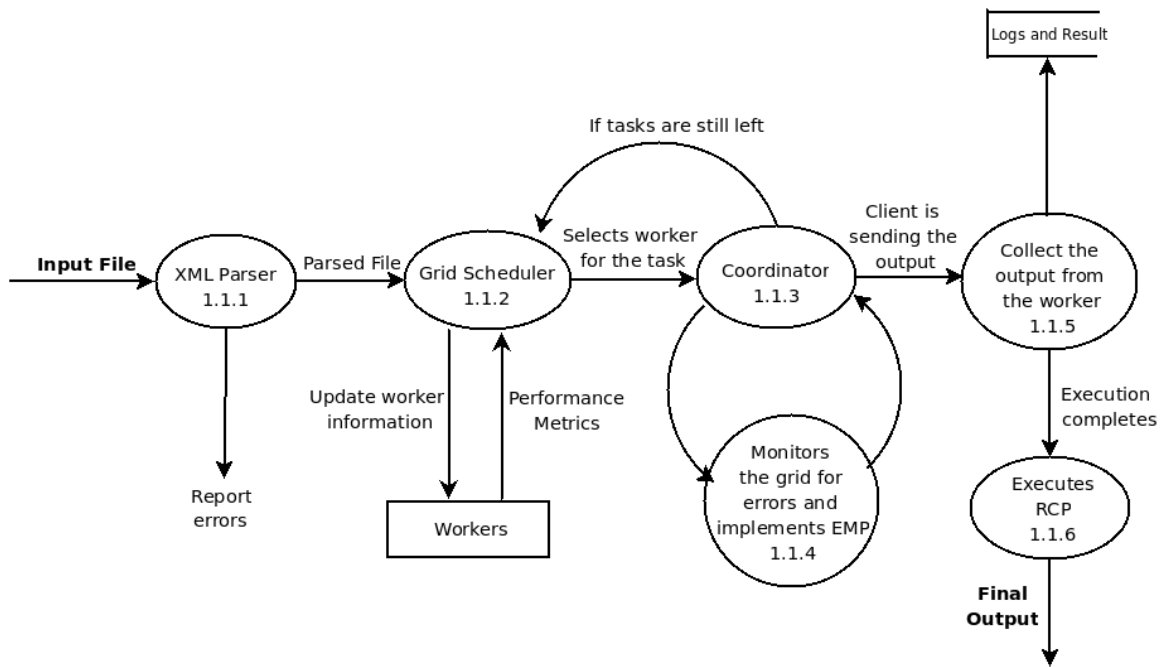


Figure 3.7: Level 2 DFD for Supervisor

- Level 2 DFD:** It gives a broader or magnified view of what the the supervisor, would be performing. As we can see the supervisor selects one of the task files to be executed, then executes the grid scheduling algorithm to determine the worker to which this task file is to be assigned. Once the job scheduling is done the supervisor sends the task file to the selected worker for further processing, till the supervisor gets the result its continuously monitors the grid to check for any network or logical errors and executes EMP. Once the worker sends the response, i.e., the output the supervisor collects it and stores it in the form of Logs and Results. After all the tasks are executed, RCP is implemented to produce the output in a format required by the user.

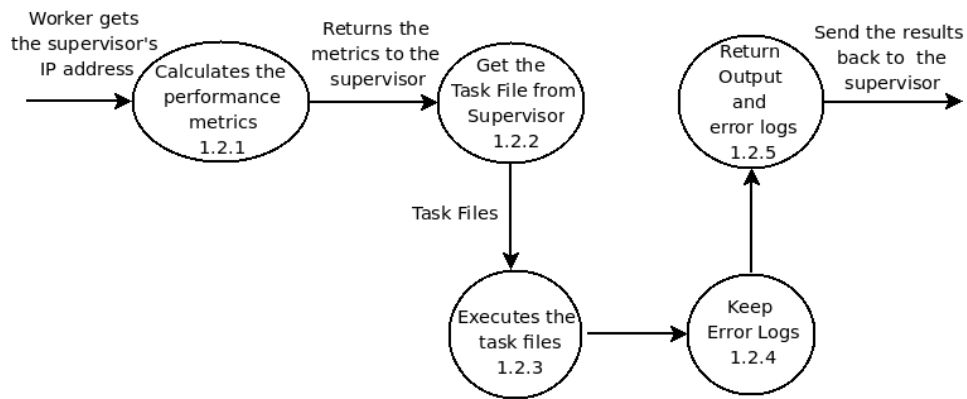


Figure 3.8: Level 2 DFD for Worker

Above is the Level 2 DFD for the worker's module, the job of the worker module is to accept the Task File assigned to it by the server. Once the task file is obtained the worker executes the Task Script which basically contains the compiling and other options related to the execution of the task file. After this step the worker executes the compiled task file and keeps track of the errors and stores it in a error log .When the execution is over the worker collects all the output files and error logs , and sends this back to the supervisor .

### 3.6 Deployment Diagram

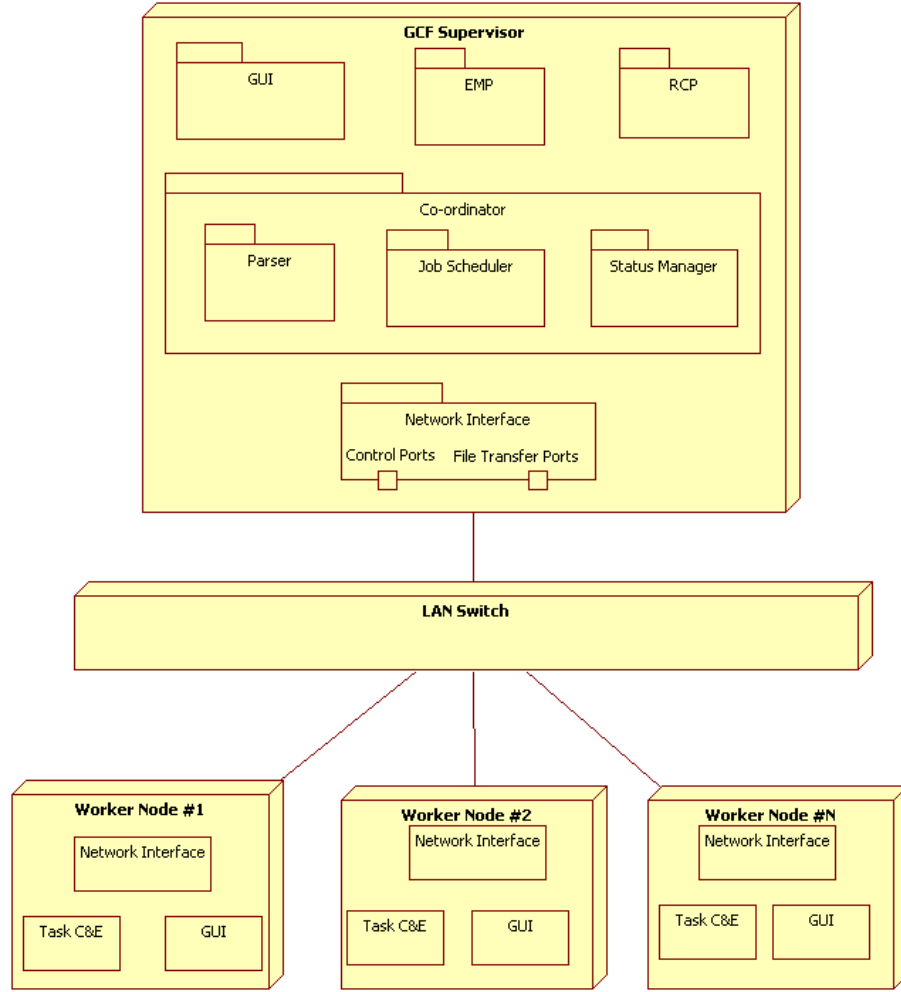


Figure 3.9: Deployment Diagram

From above diagram, we can see that there are two main components, namely the Grid Supervisor and Worker Nodes. In the Grid supervisor, the coordinator manages the overall Supervisor objectives such as distributing task files, monitoring task executions and worker nodes along with their execution time. The parser is responsible for parsing the PSS file for handling the errors. The Job scheduler maps the tasks onto the worker nodes and the Status Manager keeps track of the Worker Nodes and Tasks. The Network Interface is responsible for establishing the connection and data transfer between the supervisor and the worker nodes through the LAN switch. Similarly, each worker nodes has a network interface, which performs similar jobs. In addition to this, the Task C&E unit is responsible for task compilation and execution.



### 3.7 Sequence Diagram

The below diagram represents the exact sequence starting from the Problem submission to the execution of the RCP.

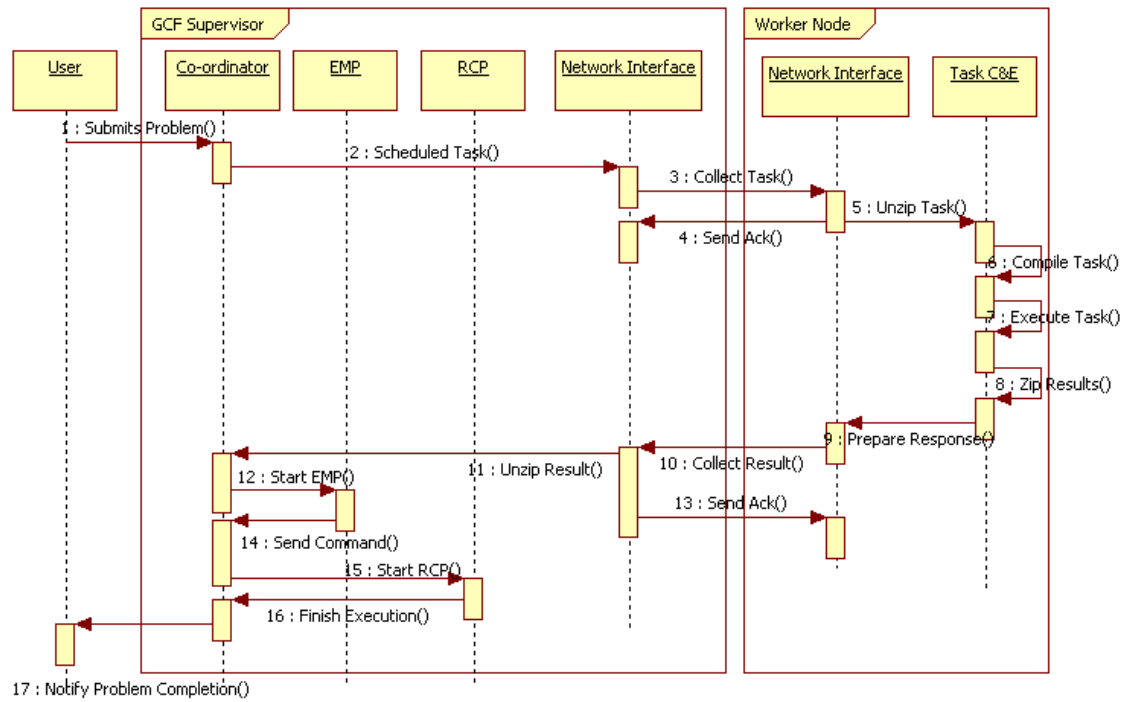


Figure 3.10: Sequence Diagram

### 3.8 Collaboration Diagram

The figure below illustrates the work done by each component in Grid Computing Framework.

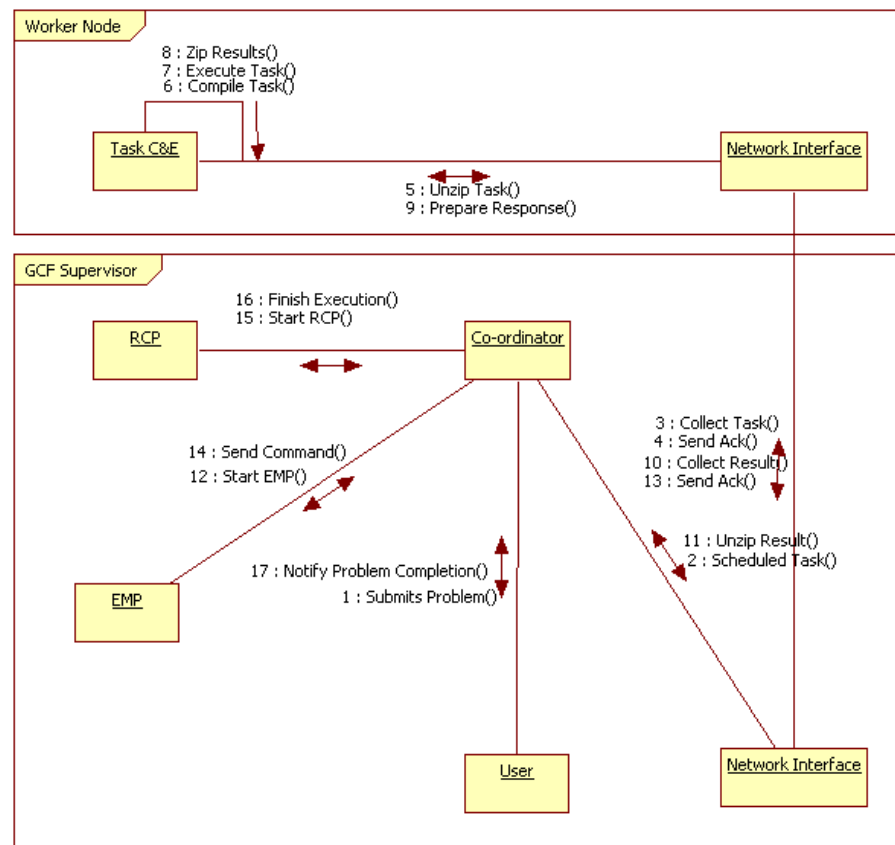


Figure 3.11: Collaboration Diagram

## 4 Implementation

### 4.1 Networking

The protocol that is used for the networking module of our project has been explained and analyzed in the design section. For network programming, Berkeley Sockets have been used. The networking module is multithreaded on both the Worker and Supervisor side .

On the server side there are mainly three threads

1. Ping thread
2. Ping Listener thread
3. Co-ordinator thread
4. File Transfer thread

The ping thread is responsible for pinging the active workers so as to allow them to know that the supervisor is active. The ping listener thread continuously listens for pings coming from the active workers. This helps the server keep a check on the active clients, and remove dead workers from its list. The clients piggyback their current performance and network metrics for them along with the ping. This information is then used by the server to judge the load of each client and also the congestion in the network.

The co-ordinator thread is responsible for establishing TCP connections with workers and maintaining a communication protocol by the means of various commands as explained. The file transfer thread is responsible for handling various file transfer requests which are to be executed.

On the worker side there are mainly four threads

1. Ping thread
2. Ping Listener thread
3. Command Listener thread
4. File Transfer thread

These threads have jobs similar to their counterparts on the Supervisor side.

## 4.2 Proposed Grid Scheduling Algorithms

For the purpose of choosing a suitable algorithm for the Framework, we studied the heuristics mentioned previously. However, MCT, Min-Min, Sufferage, Genetic Algorithms and Simulated Annealing Algorithms were found to be efficient and hence were tested further.

MCT and Sufferage are Online mode heuristics and were tested with others which are Batch mode heuristics by mapping tasks onto machines in a fixed arbitrary order.

Simulated Annealing in its simplest form, performs very poorly across all ETC matrices since it allows poor solutions to be accepted. **We propose a modified Simulated Annealing algorithm**, which is similar to Simulated Annealing, except that it is seeded with the solution of the Min-Min heuristic. This improves the solution quality and it is a good alternative, if there are strict time constraints.

The Genetic Algorithm implementation by Braun et al. <sup>[8]</sup>, was found to be one of the best heuristics in all type of ETC matrices. However, the results and especially the execution time can be improved with some modifications. **Our research led us to a modified Genetic Algorithm** which differs from the implementation in <sup>[8]</sup> with respect to seven major aspects.

1. Population Size: The size of the population was reduced to 100 from the original 200.
2. No unseeded iterations: It was observed that unseeded iterations with all random populations do not produce schedules with makespans comparable to Min-Min. Hence, only four seeded iterations were conducted. Thus, removing them **reduced the execution time by almost half**.

3. Seeding: The algorithm was seeded with the solutions of Suffrage, MCT and Min-Min instead of just Min-Min in the original implementation. This gives variety to the population.
4. Elitism: The algorithm allows the top 5% solutions to survive into the next generation, instead of just the best solution. This is required to let the solutions which are close to global makespan minima get closer in the future generations.
5. Swap & Fitness based Crossover: The crossover is implemented in two steps. A one bit swap operation with 40% probability, swaps the machine mapping of a single task between two chromosomes. Next, a crossover which is based upon fitness with 20% probability takes place. In this, two parents produce a single offspring. The new chromosome has machine assignments based upon the fitness of the parents. If one of the parents is more fit than the other, the offspring is more likely to have the assignments according to that parent.
6. Stopping Condition: The stopping condition was changed to 100 steps without a change in the elite chromosome.
7. Local Search: It was observed that the algorithm is not good in finding local optimum solutions. Thus, to improve the solutions returned by the algorithm, a final local search was performed on the chromosome with the best makespan. This was done by swapping task assignments between the more loaded half of the machines and lesser loaded half of the machines. This often led to about **0.2 - 0.3% improvement** in the makespan.

#### 4.2.1 Simulation

A simulation was conducted to find the most efficient heuristics amongst the chosen few. The performance metric under inspection was the makespan. The simulation was conducted as specified by Braun et al. [8]. Three types of ETC matrices, Consistent, Semi-Consistent and Inconsistent were created. Each with four types of heterogeneity, low task - low machine, low task - high machine, high task - low machine and high task - high machine heterogeneities. Each type of matrix had 10 meta-tasks.

After the execution of all heuristics on a particular task, the makespan of the solutions

provided were noted. Each heuristic was given a penalty for its performance, which was evaluated as follows:

$$P_{it} = \frac{makespan_{it}}{\max_{j \in H} (makespan_{jt})}$$

where  $H$  is the set of heuristics and  $t$  is the task to be executed. Thus, the penalty given to a heuristic  $i$  for the task  $t$ , lies in the range  $0 < P_{it} \leq 1$ .

#### 4.2.2 Results

The simulation was performed. After the completion of the execution of the heuristics on the entire set of matrices, the following results were obtained.

Heuristics \ Matrices	Consistent	Semi Consistent	Inconsistent
SA Min-Min	6.70564	3.54590	2.94299
Min-Min	6.88361	3.65915	3.00475
MCT	6.95000	4.27075	3.57312
GSA	6.80507	4.30103	3.49500
Sufferage	6.97664	4.26640	3.58922
Braun GA	6.59389	3.46103	2.88257
Modified GA	6.46727	3.40758	2.86905
Simualted Annealing	40.00000	40.00000	40.00000

Table 4.1: Simulation Results

The results of all four types of heterogenities of the same type of matrices were combined. It was pretty clear that Simulated Annealing performed the worst, and the Modified Genetic Algorithm provided makespans upto 14.28 times better than it (in the Inconsistent Matrices).

**Modified Simulated Annealing was better than Min-Min by about 2.7%, 3.0% and 3.28%** in Consitent, Semi Consistent and Inconsistent matrices.

**Modified Genetic Algorithm outperformed the Braun Genetic Algorithm by 1.91% and 1.73%** in Consistent and Semi-Consistent matrices. However, in the case of Inconsistent matrices, the results were quite similar. The important advantage offered by the Modified Genetic Algorithm was in terms of speed, where **it was faster than the Braun GA by roughly 45 - 51% in each case.**

Hence, the Modified Simulated Annealing was chosen as the heuristic of choice if

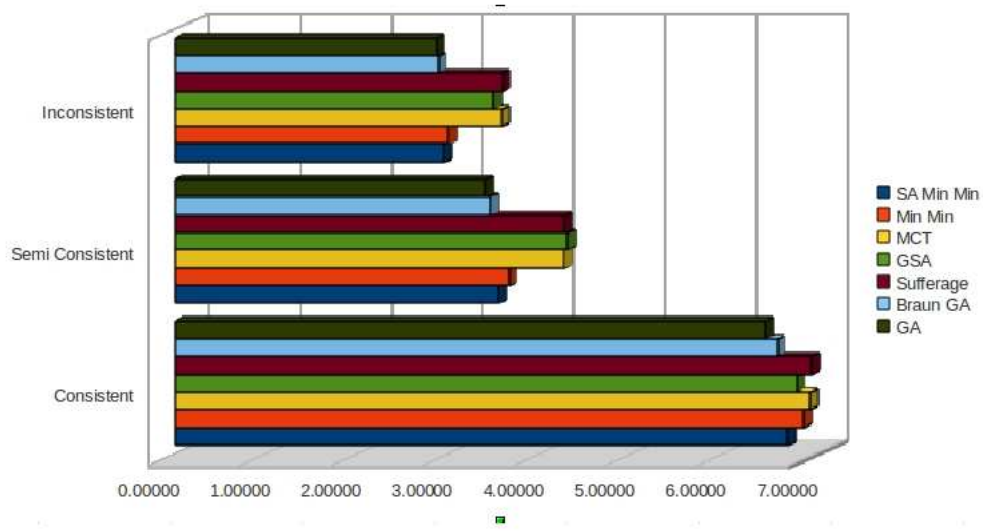


Figure 4.1: Simulation Results Graph

the speed of the heuristic is more important, and the Modified Genetic Algorithm, if improvement in makespan was a priority.

### 4.3 Execution Monitor Program (EMP)

The Execution Monitor Program (EMP) is executed after completion of a task which is a checkpoint. The Supervisor writes the task-ids of completed tasks onto the `tasks_done.emp` file in the format:

<taskid-1>

<taskid-2>

....

The EMP is then executed, which writes the commands to be executed in the file `emp_response.emp` file. In the following format

<numberofcommands>

<command-1>

...

<command-n>

After the EMP is done with writing, it writes "OK" onto the file `emp_status.emp`. When the Supervisor finds the "OK", it begins parsing the commands and executing

them as explained earlier

## 4.4 Interfaces

### 4.4.1 Command Line Interface (CLI)

The CLI displays the various information and messages that the supervisor and worker receive or generate. Also, the user can start the Supervisor in either in 'new task' mode or in 'recovery' mode.

#### 4.4.1.1 Supervisor

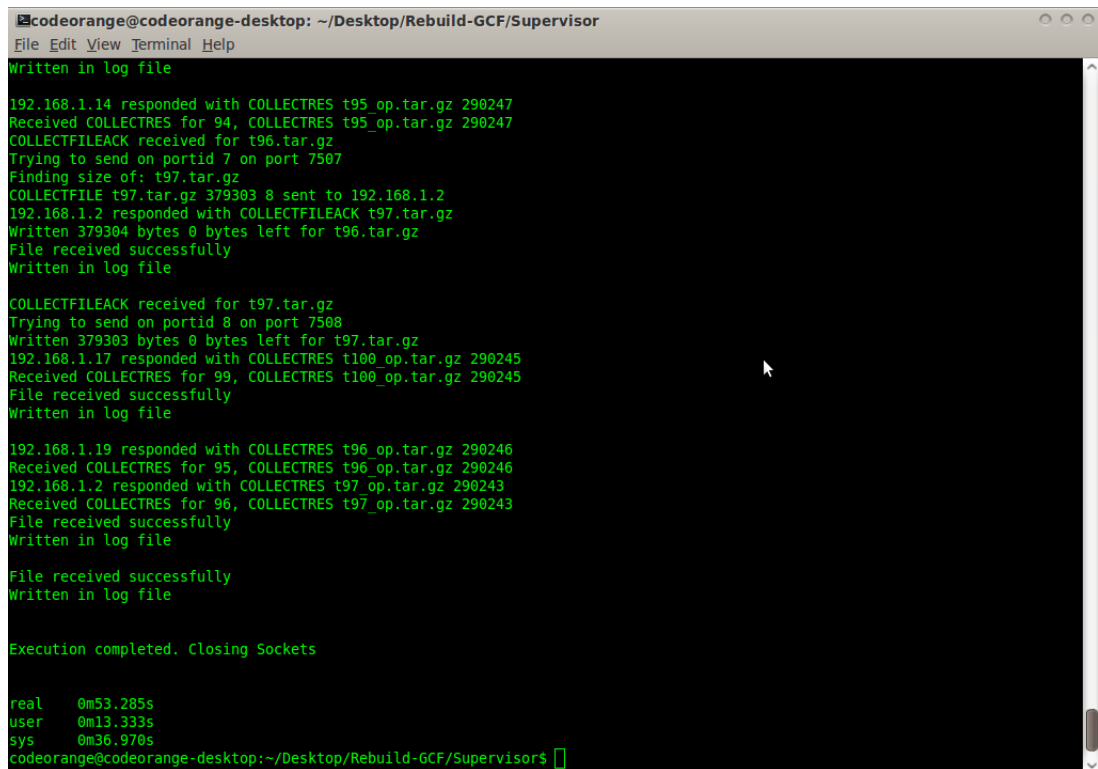
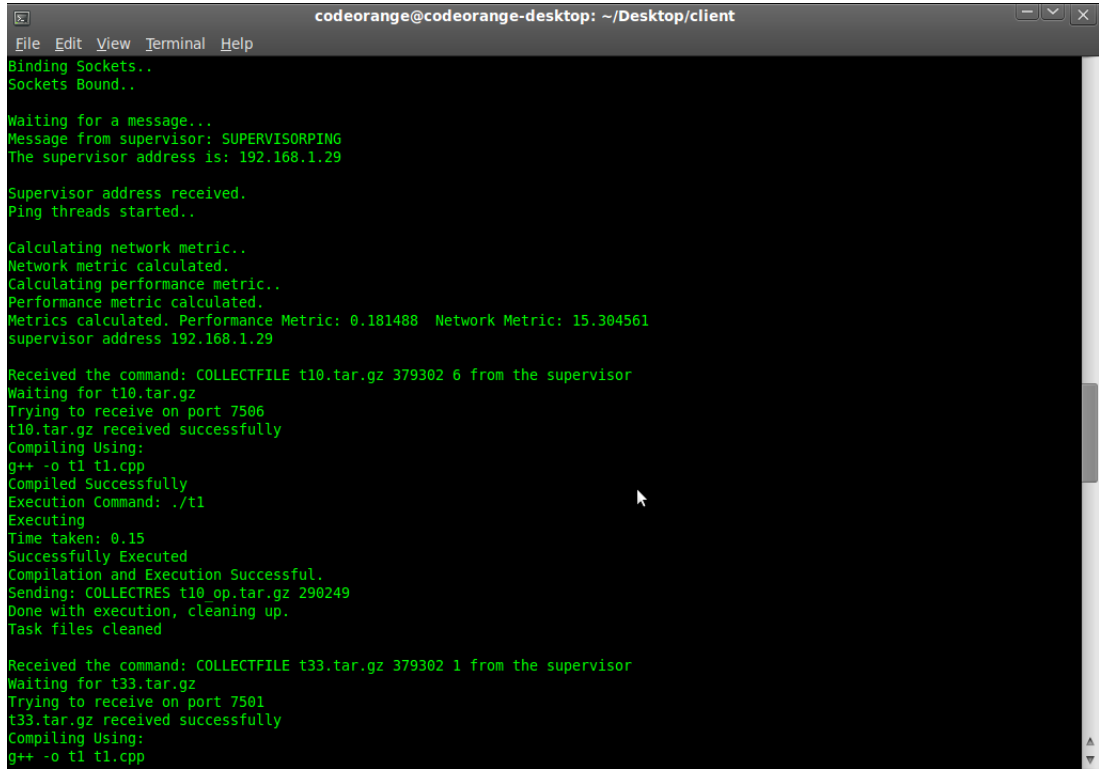
A screenshot of a terminal window titled 'codeorange@codeorange-desktop: ~/Desktop/Rebuild-GCF/Supervisor'. The terminal shows a series of log messages in green text on a black background. The messages include file collection status for various tasks (t95, t96, t97, t100) and their corresponding output files. At the bottom, it shows 'Execution completed. Closing Sockets' and a summary of execution times: real 0m53.285s, user 0m13.333s, sys 0m36.970s. The prompt is 'codeorange@codeorange-desktop:~/Desktop/Rebuild-GCF/Supervisor\$'.

Figure 4.2: Supervisor Command line window

The figure below shows the Supervisor's window of 100 tasks. The size of each input and output file was roughly 700 KiB, with 1000000 integers. The total execution time for 100 tasks i.e. roughly 70 MiB of data was 53.285sec (real time).



#### 4.4.1.2 Worker



```
codeorange@codeorange-desktop: ~/Desktop/client
File Edit View Terminal Help
Binding Sockets..
Sockets Bound..

Waiting for a message...
Message from supervisor: SUPERVISORPING
The supervisor address is: 192.168.1.29

Supervisor address received.
Ping threads started..

Calculating network metric..
Network metric calculated.
Calculating performance metric..
Performance metric calculated.
Metrics calculated. Performance Metric: 0.181488 Network Metric: 15.304561
supervisor address 192.168.1.29

Received the command: COLLECTFILE t10.tar.gz 379302 6 from the supervisor
Waiting for t10.tar.gz
Trying to receive on port 7506
t10.tar.gz received successfully
Compiling Using:
g++ -o t1 t1.cpp
Compiled Successfully
Execution Command: ./t1
Executing
Time taken: 0.15
Successfully Executed
Compilation and Execution Successful.
Sending: COLLECTRES t10_op.tar.gz 290249
Done with execution, cleaning up.
Task files cleaned

Received the command: COLLECTFILE t33.tar.gz 379302 1 from the supervisor
Waiting for t33.tar.gz
Trying to receive on port 7501
t33.tar.gz received successfully
Compiling Using:
g++ -o t1 t1.cpp
```

Figure 4.3: Worker command line window

The above screenshot is the CLI of the worker part which is currently executing tasks which are being recieved from the server.

#### 4.4.2 Graphical User Interface (GUI)

We have used the Gnome Window Toolkit (GTK) for constructing the GUI for our project. GTK is written in C and provides stable libraries which can be used for generating GUI. The major advantage of GTK is its excellent integration with the Gnome Desktop Manager (GDM) , which is the default desktop manager for major distributions like Ubuntu, Fedora, Open Suse, Debian etc. Thus the GUI blends in with the desktop color schemes and themes in use on the machine.

The problem with GTK is that it is **thread aware** and not **thread safe** i.e one cannot call a GTK function without using a mutex (so as to protect the glib library from memory access errors). To fix this, we implemented our own mutex by using a global lock variable, this lock has to be obtained by any thread which wishes to update the GUI. Along with the lock we have also established a producer-consumer relation among

threads which generate the updates for the GUI and the threads which are responsible for calling GTK functions.

#### 4.4.2.1 Supervisor

The figure below shows the Supervisor's window of 100 tasks. The size of each input and output file was roughly 700 KiB, with 1000000 integers. The total execution time for 100 tasks i.e. roughly 70 MiB of data was 53.285sec (real time).

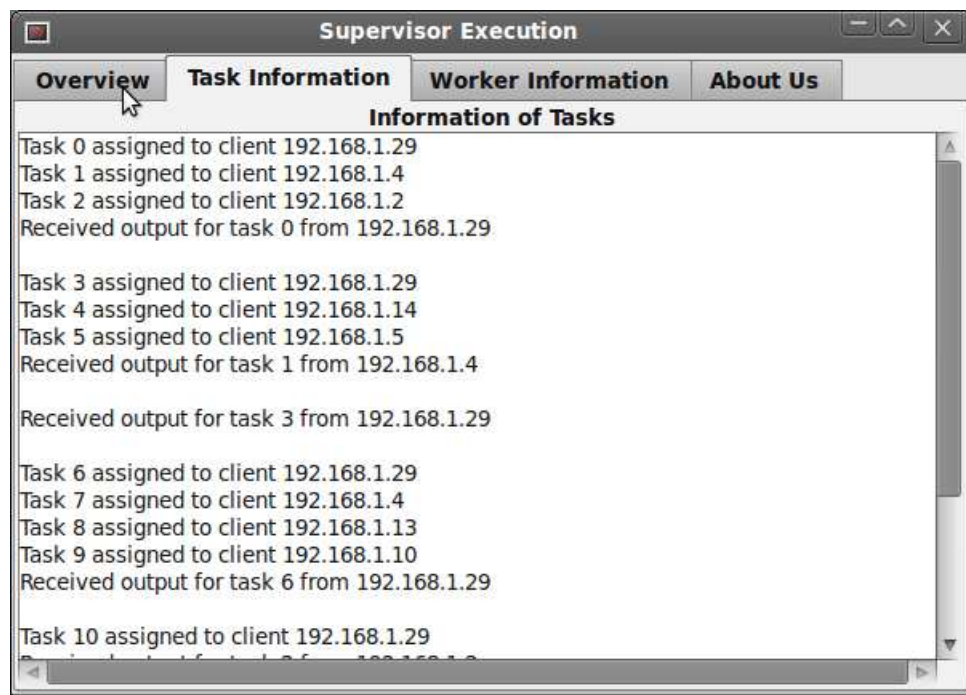


Figure 4.4: Supervisor - Task Information

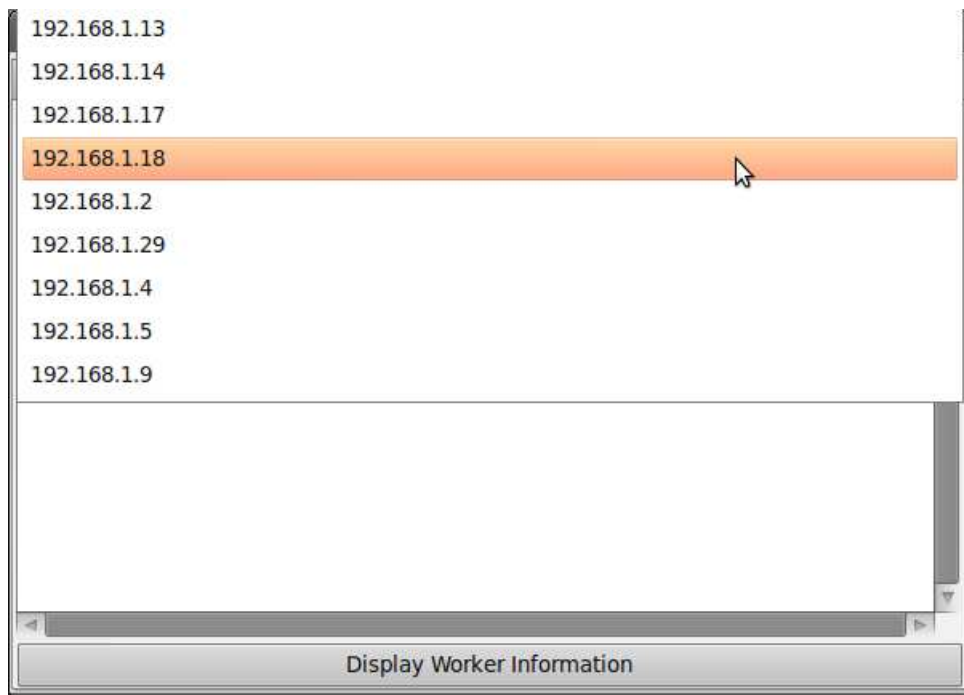


Figure 4.5: Supervisor - Worker Node List

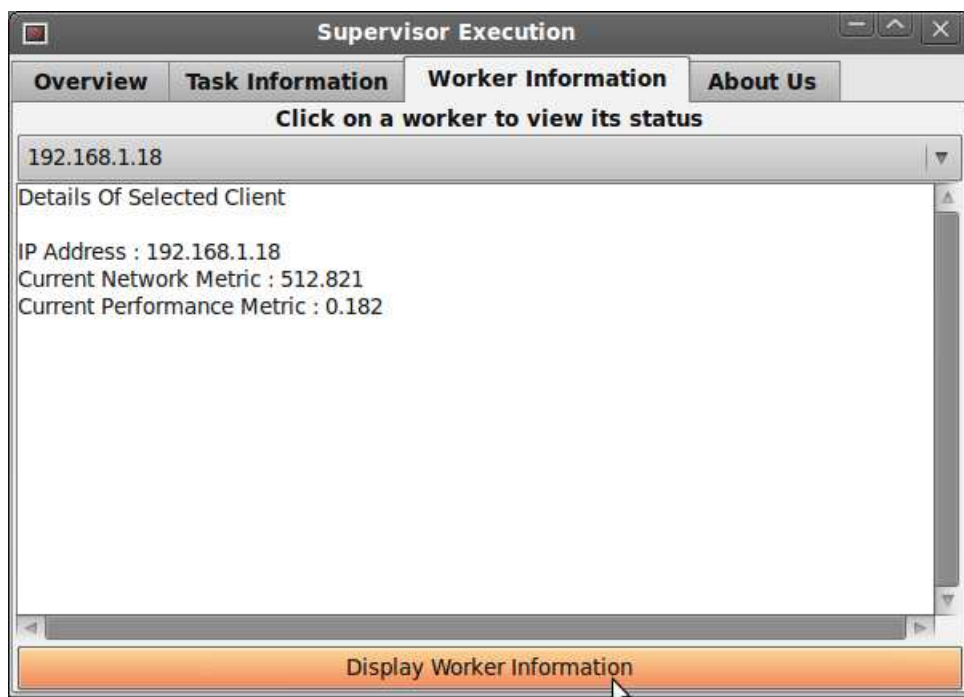


Figure 4.6: Supervisor - Worker Node Information

#### 4.4.2.2 Worker

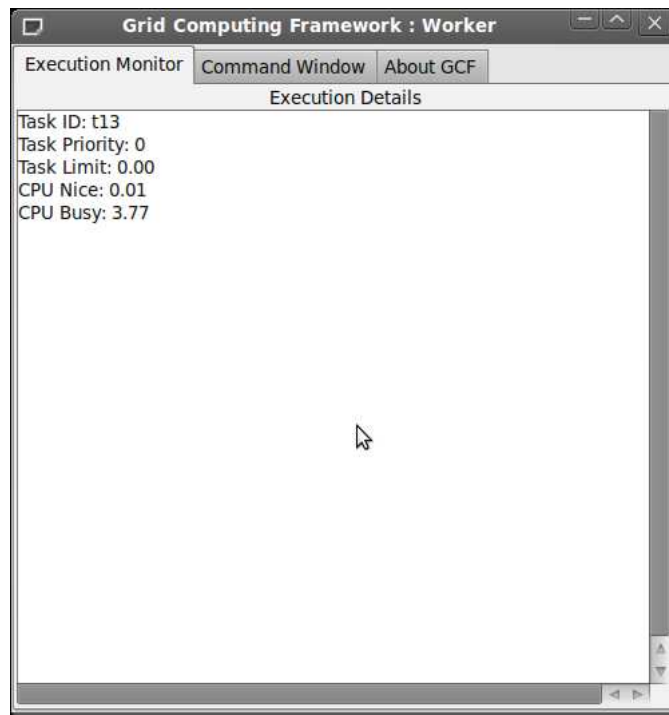


Figure 4.7: Worker side(Execution Monitor) GUI

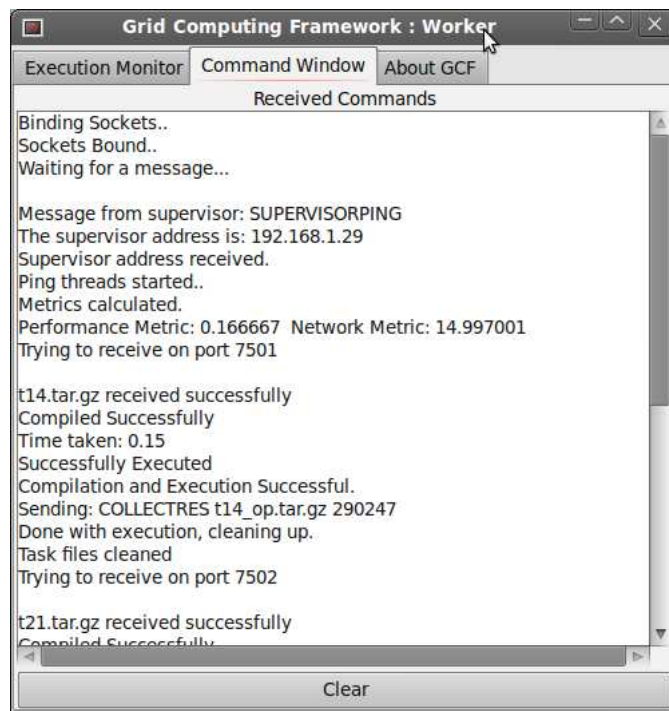


Figure 4.8: Worker side(Command window) GUI

The above screenshot shows the worker which is executing task files given by the supervisor and also the GUI implementation.

## 5 Further Work

In the past two semesters, we have implemented all the functionalities that we had envisioned in the Grid Computing Framework. The Execution Monitor Program was added to the Design at a later stage, to add variety to the problems that can be solved with the framework.

However, further work can be done to scale the framework to a bigger network. We propose the following follow-up to our current work:

- Implement the Grid Manager Level to handle multiple Supervisors and distribute problems to them.
- Add security to the communication protocol for unsecure networks.
- Allow multiple servers to function in harmony on the same subnet.
- Add capability to handle extreme network load.
- Implement a Virtual Private Network (VPN) so that the Grid can be implemented over Internet as well.

## 6 References

For conceiving and implementing Grid Computing Framework a lot of research was done for getting the concepts as well as the help needed for implementation , here are a collection of books and links used for this purpose.

- [1] Andrew S Tennenbaum “Computer Networks”
- [2] C.S.R.Prabhu “Grid and Cluster Computing”
- [3] Mark Mitchell, Jeffrey Oldham and Alex Samuel “Advanced Linux Programming”
- [4] Socket Programming in Linux - <http://www.tenouk.com/cnlinuxsockettutorials.html>
- [5] [http://en.wikipedia.org/wiki/Grid\\_computing](http://en.wikipedia.org/wiki/Grid_computing)
- [6] GTK+ documentation: <http://www.gtk.org/documentation.html>
- [7] Yang Gao, Hongqiang Rong, Joshua Zhexue Huang “Adaptive grid job scheduling with genetic algorithms”
- [8] Tracy D. Braun, Howard Jay Siegel, Noah Beck, et al. “A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems”
- [9] Fangpeng Dong, Selim G. Akl “Scheduling Algorithms for Grid Computing: State of the Art and Open Problems”
- [10] Hak Du Kim, Jin Suk Kim “An On-line Scheduling Algorithm for Grid Computing Systems”
- [11] Abhishek Kumar, Navneet Chaubey, Sireesha Yakkali ”Immediate Mode Scheduling Methods for Independent Jobs on Open Online Heterogeneous Systems”
- [12] Muthucumaru Maheshwaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, Richard F Freund “Dynamic Matching and Scheduling of a Class of a Independent onto Heterogeneous Computing Systems”
- [13] Tadej Borovšak - <http://tadeboro.blogspot.com/2009/06/multi-threaded-gtk-applications.html>

- [14] Javier Carretero, Fatos Xhafa, Ajith Abraham - “Genetic Algorithm Based Schedulers for Grid Computing Systems”
- [15] M Srinivas, L M Patnaik - “Genetic Algorithms: A Survey”
- [16] Jefferey Dean, Sanjay Ghemawat - “MapReduce: Simplified Data Processing on Large Clusters”
- [15] Buddhika Siddhisena - “The smart GIT”
- [16] Brian Hall - “Beej’s Guide to Network Programming Using Internet Sockets”
- [17] Glade Tutorials - “<http://live.gnome.org/Glade/Tutorials>”
- [18] L<sup>A</sup>T<sub>E</sub>X – A document preparation system “<http://www.latex-project.org/>”
- [19] L<sup>A</sup>X – The Document Processor “<http://www.lyx.org/>”

## 7 Acknowledgement

We are grateful to Ms. Sakshi Surve Computer Engineering Department (TSEC), for guiding us in our BE Project titled, “Grid Computing Framework”. It is her knowledge that has been the guiding light through all our work so far. Her constructive suggestions and insight helped us in exploring our project from all possible perspectives. We also express our sincere gratitude to the Mr. Jayant Gadge, Head of the Computer Engineering Department (TSEC), for all the encouragement and analytical views on topic of our interest.

We would like to thank the Project Lab assistants, Mr. Haresh Jethani and Mr. Ramkuber Shukla, without whom, it was impossible to work on a project of this scale.



## 8 Summary

The Grid Computing Framework was conceptualised with the idea of providing the user with an easy-to-use and easy-to-deploy grid computing infrastructure on a smaller scale.

The design as such has a lot of flexibilities. The PSS format allows the user to use any language / script for working with the tasks. The Execution Monitor Program (EMP) allows the user to dynamically control the tasks' execution. This can be used for re-execution of tasks which have returned erroneous results, or stopping the execution of tasks whose results are no more required after a particular time. Finally, the Result Compilation Program, provided by the user can be used to arrange the results of the tasks, and present it to the user in his desired format.

The framework was tested with two problems, the first was a distributed sort of about  $10^8$  32-bit integers. The second problem was breaking a simple 32-bit encryption algorithm. Both the tests were successful. In general, the framework can solve any problem which can be modelled using the Google Map-Reduce framework [16].