ALGORITHMS FOR LOAD BALANCING IN DISTRIBUTED NETWORK

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology

in

Computer Science and Engineering

By

JAGNYASHINI DEBADARSHINI

Roll no:107CS041

Under the Guidance of

PROF. P.M.KHILAR

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela-769008, Orissa, India

National Institute of Technology

Rourkela

CERTIFICATE

This is to certify that the thesis entitled ***"Algorithms for Load Balancing in Distributed Network "*** submitted by **Jagnyashini Debadarshini (Rollno-107CS041)** in the partial fulfillment of the requirement for the degree of Bachelor of Technology in Computer Science Engineering, National Institute of Technology, Rourkela , is being carried out under my supervision.

To the best of my knowledge the matter embodied in the thesis has not been submitted to any other university/institute for the award of any degree or diploma.

Prof. P.M.Khilar

Date : Department of Computer Science and Engineering

National Institute of Technology

ACKNOWLEDGEMENT

I avail this opportunity to extend my hearty indebtedness to my guide ***Prof.P.M.Khilar***, Computer Science Engineering Department, for his valuable guidance, constant encouragement and kind help at different stages for the execution of this dissertation work.

I also express our sincere gratitude to ***Prof. A.K.Turuk***, Head of the Department, Computer Science Engineering, for providing valuable departmental facilities.

Submitted by:

Jagnyashini Debadarshini

Roll no-107CS041

Computer Science and Engineering

National Institute of Technology

Rourkela

ABSTRACT

A network in which data to be worked on is present in more that one computer is said to be distributed network. The technique of distributing load among processors in order to avoid overloading on a particular processor is called load balancing. If the load on a particular processor is more, then the under loaded processor is scheduled some processes that was assigned to the overloaded processor. Conventional load balancers have effectively increased the CPU utilization, memory utilization, and disk I/O resources in a network. In this thesis a load balancing algorithm is proposed and the performance of the algorithm is studied under various condition.The performance analysis is done and the results are shown graphically.

Keywords: Load balancing, master process, slave process, Message passing interface (MPI)

List of Tables

List of Figures

Contents

Chapter 1

Introduction

1.1 Distributed Network

A network is said to be distributed when the data to be worked on is spread over the network. This means that the data to work on is present in more than one computer over the network. In the distributed network instead of centralized processing, the data is operated efficiently over a number of nodes leading to faster computation.

1.2 Load Balancing

The technique of distributing load among processors in order to avoid overloading on a particular processor is called load balancing. If the load on a particular processor is more, then the under loaded processor is scheduled some processes assigned to overloaded processor. Load balancing can be **static load balancing** and **dynamic load balancing**

1.2.1 Static Load balancing

In this type of load balancing the amount of work load is known from the beginning of the execution of the task. Here the work is distributed equally among processors.Thus there is no extra cost involved for balancing the load during the execution.

1.2.2 Dynamic load balancing:

In this type of load balancing the amount of work load is not know at the beginning of the execution of the task.Thus as a result of which there is transfer of load during the execution of the processes.

Load Balancing can also be of ***centralized load balancing*** and ***distributed load balancing***. Centralized load balancing typically requires a head node that is responsible for handling the load distribution. As the no of processors increases, the head node quickly becomes a bottleneck, causing significant performance degradation. To solve this problem, the workload of the load balancer can be distributed to more than one nodes; hence the concept of distributed dynamic load balancing comes into. In addition, a centralized scheme has the problem of poor reliability because permanent failures of the central load balancer can result in a complete failure of the load-balancing mechanism[1].

1.3 Thesis Objective

1. The project ***"Algorithms for load Balancing in Distributed network"*** is based on designing an algorithm for scheduling the load among the processor in such a way that none of the processor is overloaded. It performs load balancing along with performing the task in parallel.

2. Comparison of the time taken to perform the task using the algorithm for different number of processors.

1.4 Thesis Organization

This thesis is divided into 6 chapters. Each chapter focuses on a specific topic in the field of load balancing.

Chapter 1 gives an overview of the load balancing. It also introduces the concept of

Distributed Network.

Chapter 2 deals with the literature survey of various topics related to the topic of this thesis. It discusses the Message passing interface and what are the basic functions used in it. It also discusses the communication aware load balancing technique and the works related to the load balancing.

Chapter 3 discusses the algorithm of the load balancing technique which is proposed and gives the pseudo code of the algorithm proposed. It gives the details about the assumptions made and the parameters which are considered while doing the analysis.

Chapter 4 shows the experimental results of the proposed algorithm and the timing analysis of the results.

Chapter 5 concludes showing conclusion drawn from the various simulation and experimental results

1.5 Conclusion

This chapter gives a basic overview about the distributed network and the various load balancing techniques.It also presents the objectives of the thesis and the chapter organization of the thesis.

Chapter 2

Literature review

2.1 Message Passing application programmer Interface (MPI)

Message Passing Interface (MPI) is an application program interface specification that allows processes to communicate with one another by sending and receiving messages. It is typically used for parallel programs running on computer clusters and supercomputers, where the cost of accessing non-local memory is high.[7].The processes have separate address spaces. Processes communicate by sending and receiving messages. Each process would be running on a separate node. MPI is also supporting shared memory programming model. This means that multiple processes can read or write to the same memory location [6]. Message passing can be of two types[6]:

1. **Point-to-point message passing operations**

   MPI_Send(start, count, datatype, dest, tag, comm);

   MPI _Recv(start, count, datatype, source, tag, comm, status);

   This is simple but inefficient. Here most of the work is done by process 0. The required data is brought by process 0 and send it to other processes which are idle. Then after the required computation is made the output is collected from the processes

2. **Collective operations to from all**

   MPI_Bcast(start, count, datatype, root, comm);

   MPI_Reduce(start, result, count, datatype, operation, root, comm);

   Here all processes are equally involved in the computation. This is simple, compact, more efficient Here the size for *"count"*and *"datatype"* must be same.

   The library that provides the basic MPI definitions and types is :

   # include "mpi.h"

   Some of the basic functions used in MPI are as follows:

   1. **Start MPI**

      MPI_Init(&argc, &argv);

   2. **exit MPI**

      MPI_Finalize();

   3. **Determining the size of the communication world that is how many processes are there in the communication world**

      MPI_Comm_size (MPI_COMM_WORLD, &no_processor)

   4. **Determining the rank of the of the process in the communication world**

      MPI_Comm_rank(MPI_COMM_WORLD,&myrank)

   5. **Sending message from one process to another process.**

      MPI_Send or MPI_Bcast

   6. **Receiving message from one process**

      MPI_Recv MPI_Reduce

3. **Compiling MPI programs** From the command line the following command is written to compile the MPI program

   mpicc -o prog prog.c

4. **Running MPI program**

   For running the MPI program the following command is written from the command line

   mpirun -np no_of_process prog

2.2 Communication Aware Load Balancing

The dynamic, communication-aware load-balancing scheme for the non dedicated clusters[1] is given below. Each node in the cluster serves multiple processes in time-sharing fashion so that the processes can share the cluster resources dynamically. For this a load-balancing technique is proposed where we need to measure the communication load imposed by these processes.

Let a parallel job formed by p processes is represented by t0, t1 tp-1.

Here ni is the node to which ti is assigned. It is assumed that t0 is a master process, and tj (0 ¡ j ¡ p) are the slave processes. Let Lj,p,COM denote the communication load induced by tj, which can be computed with the following formula, Here Ti,j,COM is execution time of process j in the ith phase taking the communication as the resource[1].

$$L_{j,p,COM} = \begin{cases} \sum_{i=0}^{N} T_{j,COM}^{i} & \text{if } j \neq 0, n_j \neq n_0, \\ 0 & \text{if } j \neq 0, n_j = 0, \end{cases}$$

The first term in the right-hand side of the equation corresponds to the communication load of a slave process tj when the process tj and its master process are allocated to different nodes. The second term represents a case where the slave and the master process are running on the same node, and therefore, the slave process exhibits no communication load.

The communication load on node i, Li,COM , is calculated as the cumulative load of all the processes currently running on the node. Thus, Li,COM is estimated as follows:

$$L_{i,COM} = \sum_{\vee j : n_j = i} L_{j,p,COM}.$$

Thus given a parallel job arriving at a node, the load-balancing scheme attempts to balance the communication load by allocating the jobs processes to a group of nodes with a lower utilization of network resources.

Apart from this number of distributed load-balancing schemes for clusters is proposed, primarily considering a variety of resources, including the CPU [3], disk I/O [4], or a combination of CPU and memory resources [5]. These approaches have proven effective in increasing the utilization of resources in clusters, assuming that network interconnects is not potential bottlenecks in clusters[1].

2.3 Conclusion

This chapter gives a basic overview about the Message Passing Interface(MPI),how the mpi programs are compiled and how to run the MPI programs.It also gives a brief overview about the communication aware load balancing and the performance parameters that are considered in various load balancing algorithms.

Chapter 3

Proposed Algorithms

3.1 Introduction

In the distributed network,a number of processors would be responsible for perfoming the task assigned.The task would be divided into large number of subtasks. In the network there would be a master node and a group of slave nodes. The master would be responsible for dividing the task into subtasks and assigning the subtasks to the slave nodes. The slave node would be performing the required task and sending the result back to the master. The slave processors perform the task in parallal as a result of which the computation becomes fast.

Figure 3.1 gives the basic model how the master divides the tasks into subtasks and assigns each of the subtasks to each slave node in a network.Each node computes the result and sends it back to the master node. Thus the master is involved in communicating to and from the slave nodes.

3.2 Basic assumptions

1. The master process is assumed to be totally dedicated for sending the subtasks to the slave and receiving the results from the slave process.

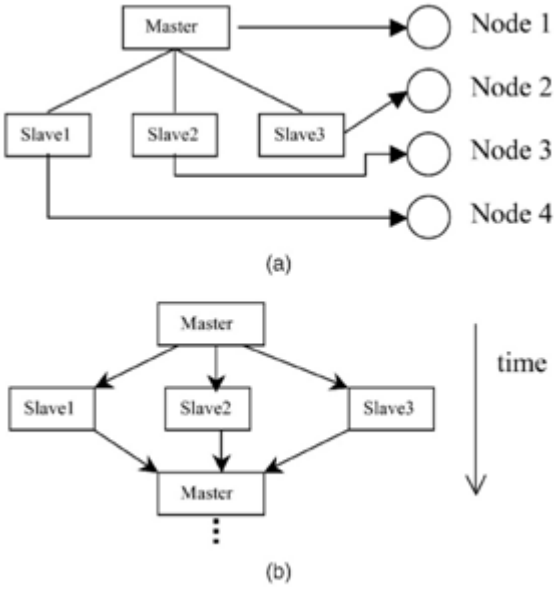

Figure 3.1: *a) four processes are allocated to four nodes. (b) The master communicates to and from slaves.*

2. The master is not responsible for performing any of the subtasks and the slaves would only be performing the subtasks.

3. The algorithm is implemented by five, eight and ten processes and the subtasks are being performed in parallel.

3.3 Performance parameter

The performance metric considered in this experiment is the ***job turn-around time*** for each slave process. The turn-around time of a job is the time elapsed between the jobs submission and its completion. The turn-around time is a natural metric for the job performance due to its ability to reflect a users view of how long a job takes to complete[1].

3.4 Proposed Algorithm for Load balancing

This section discusses the approach for load balancing .The following is the proposed algorithm for load balancing. It further consists of seven algorithms. These are discussed as follows:

1) **Algorithm** load_balancing(WORK,TAG,NO_WORK)

This algorithm initializes the MPI and calls the master(), slave(myrank), stampstartt(), stampstopp(startt,myrank).After the four functions complete there task this algorithm exits the MPI and ends the program. The algorithm for load_balancing is given below.

```
1  Algorithm load_balancing(WORK,TAG,NO_WORK)
2  //This algorithm balances the load among the process available in the communication world
3  //WORK=1,TAG=2,NO_WORK=50 where WORK is the 3
4  //no.of work to be send or received between the processes,
5  //NO_WORK is the total work available in the communication world
6  {
7  MPI_Init(&argc,&argv); //Initializing MPI.
8  MPI_Comm_rank(MPI_COMM_WORLD,&myrank); // determining rank of process
9  if rank=0 then
10 master();
11 else
12 startt=stampstartt();
13 slavee(myrank);
```

```
14 stopp = stampstopp(startt,myrank);
15 MPI_Finalize(); //Exiting MPI
16 return 0;
17 }
```

2) **Algorithm** master(void)

This algorithm is for the master process.Whenever the rank of the process is '0' this algorithm is executed.This basically divides the tasks into large number of subtasks by calling the get_next_work_item() and sends these subtasks to the slave process.After each of the slave process is assigned one subtask it waits for the result from any of the slave process.After it recieves the result from any one of the slave process it sends the next subtask to the same slave process.Thus it continues this process till all the subtasks are assigned to the slave process. The algorithm master(void) is given below:

```
1 Algorithm master(void)
2 //This algorithm describes the work of master process is to send the load to
  various process
3 //and receive the result from various slave processes
4 //no_tasks variable containing the total number of process available, work con-
  tains the current
5 //work, status contains the status of the particular process.
6 {
7 MPI_Comm_size(MPI_COMM_WORLD, &no_tasks); //determining the num-
  ber of processes
8 for rank :=1 to rank := (no_tasks-1) do
9 {
```

```
10 work:=get_next_work_item()
11 MPI_Send(&work,1,MPI_INT,rank,WORK, MPI_COMM_WORLD);
12 }
13 work1:=get_next_work_item();
14 while(work1!=0)
15
16 //Receive result from the slave
17 MPI_Recv(&result,1,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,
   MPI_COMM_WORLD,&status);
18 MPI_Send(&work1,1,MPI_INT,status.MPI_SOURCE,WORK,MPI_COMM_WORLD);
19 work1:=get_next_work_item();
20
21 //Receive the results from the slave
22 MPI_Recv(&result,1,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,
   MPI_COMM_WORLD,&status);
23 //Tell all the slaves to exit by sending the tag
24 for rank:=1 to rank:=(no_task-1)
25 {
26 MPI_Send(0, 0, MPI_INT, rank, TAG, MPI_COMM_WORLD);
27 }
28 }
```

3) **Algorithm** slavee(rank)

This algorithm is basically for the slave process. When the rank of the process is non

zero this algorithm is executed.This algorithm recieves the data from the master process and perform the required computation on the data using the work_to_do(work) function. After performing the required computation it sends the result back to the master process.This continues till the slave process recieves the TAG=2 from the master which will deactivate the slave process. The algorithm for slave process is given below

```
1  Algorithm slavee(rank)
2  // rank is the unique number assigned to the process in the communication
   world
3  // variable work will hold the current load which is send by the master process
4  // variable results the result as computed by the slave process, variable status
   holds the current
5  //the current status of the process.
6  {
7  while(1)
8  {
9  //receive the load from the master
10 MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &sta-
   tus);
11 If (status.MPI_TAG = TAG) then
12 return;
13 result :=work_to_do(work);
14 //send the result back to the master
15 MPI_Send(&results, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
16 }
```

```
17 }
```

4) **Algorithm** get_next_work_item()

This algorithm basically returns the subtasks to the calling function till there is any subtasks left.It returns 0 whenever there is no work to be performed. The algorithm for get_next_work_item() is given below:

```
1  Algorithm get_next_work_item()
2  //This algorithm gets the next work which would be send to the slave processes
3  //NO_WORK contains the total number work work to be balance
4  {
5  NO_WORK–;
6  if NO_WORK! = 0 then
7  return(NO_WORK);
8  else
9  return(0);
10 }
```

5) **Algorithm** work_to_do(work)

This algorithm performs the required computation on the data passed to it and returns the result to tha callin function.The algorithm for work_to_do(work) is given below.

```
1  Algorithm work_to_do(work)
2  // work will hold the integer on which the computation will be performed
3  //This algorithm computes on the work and sends the results back.
4  //variable temp holds the computed result.
5  {
```

```
6 temp=work*work;
7 return (temp);
8 }
```

6) **Algorithm** stampstartt()

This algorithm the determines start time of the process in milli second and returns it to the calling function.The gettimeofday(&tv,&tz) is a predefined function which gets the time of the day in second and microsecond and also the time zone.The localtime(&tv.tv_sec) uses the time returned by the gettimeofday(&tv,&tz) to get the time of the day in hour,minutes and seconds. These values are used inorder to compute the start time in millisecond and this value is returned to the calling function. The algorithm for stampstartt() is given below:

```
1 Algorithm stampstartt()
2 //This algorithm determines the start time of each the slave process in millisecond
3 {
4 gettimeofday(&tv,&tz);
5 tm = localtime(&tv.tv_sec);
6 //This is a predefined function which determines the current system time value
7 startt = tm-¿tm_hour * 3600 * 1000 + tm-¿tm_min * 60 * 1000 + tm-¿tm_sec
  * 1000 + tv.tv_usec / 1000;
8 //The time is converted into millisecond and stored in the start variable
9 return(start);
10 }
```

7) **Algorithm** stampstopp(startt,rankk)

This algorithm takes the start time and the slave process rank as the input .The

gettimeofday(&tv,&tz) is a predefined function which gets the time of the day in second and microsecond and also the time zone.The localtime(&tv.tv_sec) uses the time returned by the gettimeofday(&tv,&tz) to get the time of the day in hour,minutes and seconds. These values are used inorder to compute the stop time in millisecond It then prints the total time elasped.

```
1  Algorithm stampstopp(startt,rankk)
2  //This algorithm determines the stop time of each the slave process in millisec-
   ond and determines
3  //the time elapsed
4  //start variable contains the start time and the rankk variable contains the rank
   of the processor
5  {
6  gettimeofday(&tv,&tz);
7  tm = localtime(&tv.tv_sec);
8  stopp = tm¿tm_hour * 3600 * 1000 + tm¿tm_min * 60 * 1000 +tm¿tm_sec *
   1000 + tv.tv_usec / 1000;
9  // The time is converted into millisecond and stored in the stop variable
10 printf(rrank, stopp - startt);
11 //prints the time taken by the slave to perform the task
12 return(stop);
13 }
```

3.5 Conclusion

This chapter discussses the basic model based on which the algorithm is proposed.It also presents an overview about the basic assumptions made and the parameter that is considered for the analysis of the performance of the proposed algorithm.It also presents the algorithms which are proposed for load balancing.

Chapter 4

Experimental Results

4.1 Platform

The algorithm is carried out under the linux platform.The algorithm is written using the MPI in the C language.For a particular number of task the algorithm can be implemented for different number of processes.In this observation the number of task is assumed to be ten and the number of processes are five,eight and ten . The result of each obsevation are noted down and are analized graphically.

4.2 Experimental Observation

A) **OBSERVATION 1**

These observations are taken considering the number of processes to be five and the number of task to be performed to be ten.The screenshot in Figure 4.1 gives the obsevation under these condition.

RESULT DESCRIPTION:

The first line of the sceenshot gives the command for the creating five processes and executing the algorithm.The second line gives the time at which the algorithm is compiled.The TIMESTAMP-START gives the start time of each of the slave process. The result of the work assigned to each of the slave process is given along

```
jani@jani-laptop:~/Desktop/modified prog$ mpirun -np 5 combine

  Compiled on May  7 2011 at 01:14:18.

TIMESTAMP-START   1:16:26:358747 (~4586358 ms) of processor 4
TIMESTAMP-START   1:16:26:358931 (~4586358 ms) of processor 2
process id:2 the result is:64

new work send to process_id:2 and the work is:5

process id:4 the result is:36

new work send to process_id:4 and the work is:4

process id:2 the result is:25

new work send to process_id:2 and the work is:3

process id:4 the result is:16

new work send to process_id:4 and the work is:2

process id:2 the result is:9

new work send to process_id:2 and the work is:1

process id:4 the result is:4

new work send to process_id:4 and the work is:0

process id:2 the result is:1

process id:4 the result is:0

TIMESTAMP-START   1:16:26:366154 (~4586366 ms) of processor 1
TIMESTAMP-START   1:16:26:366167 (~4586366 ms) of processor 3
process id:3 the result is:49

process id:1 the result is:81

TIMESTAMP-END     1:16:26:366491 (~4586366 ms) of processor 1
```

```
TIMESTAMP-END     1:16:26:366501 (~4586366 ms) of processor 4
ELAPSED   8 ms of processor 4
ELAPSED   0 ms of processor 1
TIMESTAMP-END     1:16:26:366573 (~4586366 ms) of processor 3
ELAPSED   0 ms of processor 3
TIMESTAMP-END     1:16:26:366641 (~4586366 ms) of processor 2
ELAPSED   8 ms of processor 2
jani@jani-laptop:~/Desktop/modified prog$
```

Figure 4.1: *Snapshot for observaton 1*

with the process id of the process which performs the task.After completion of the assigned task what new task is assigned to which process is also displayed. After completion of the assigned tasks the TIMESTAMP-END of each of the slave process is displayed.Finally the times elasped for each of the slave processor is given. Thus as the five process are computing the task in parallel hence the time taken by five processes to perform the task is 8ms.

B) **OBSERVATION 2**

These observations are taken considering the number of processes to be eight and the number of task to be performed to be ten.The screenshot in Figure 4.2 gives the obsevation under these condition.

RESULT DESCRIPTION:

The first line of the sceenshot gives the command for the creating eight processes and executing the algorithm.The second line gives the time at which the algorithm is compiled.The TIMESTAMP-START gives the start time of each of the slave process. The result of the work assigned to each of the slave process is given along with the process id of the process which performs the task.After completion of the assigned task what new task is assigned to which process is also displayed. After completion of the assigned tasks the TIMESTAMP-END of each of the slave process is displayed.Finally the times elasped for each of the slave processor is given. Thus as the eight process are computing the task parallely hence the time taken by five

```
jani@jani-laptop:~/Desktop/modified prog$ mpirun -np 8 combine

  Compiled on May  7 2011 at 01:14:18.
TIMESTAMP-START   1:15:33:951658 (~4533951 ms) of processor 4

TIMESTAMP-START   1:15:33:951864 (~4533951 ms) of processor 1
TIMESTAMP-START   1:15:33:952024 (~4533952 ms) of processor 5
TIMESTAMP-START   1:15:33:952180 (~4533952 ms) of processor 2
TIMESTAMP-START   1:15:33:952223 (~4533952 ms) of processor 6
TIMESTAMP-START   1:15:33:952332 (~4533952 ms) of processor 3
TIMESTAMP-START   1:15:33:952427 (~4533952 ms) of processor 7
process id:1 the result is:81

new work send to process_id:1 and the work is:2

process id:2 the result is:64

new work send to process_id:2 and the work is:1

process id:3 the result is:49

new work send to process_id:3 and the work is:0

process id:4 the result is:36

process id:5 the result is:25

process id:6 the result is:16

process id:1 the result is:4

process id:2 the result is:1

process id:3 the result is:0

process id:7 the result is:9

TIMESTAMP-END     1:15:33:953834 (~4533953 ms) of processor 4
TIMESTAMP-END     1:15:33:953846 (~4533953 ms) of processor 2
ELAPSED   1 ms of processor 2
```

```
ELAPSED   1 ms of processor 2
ELAPSED   2 ms of processor 4
TIMESTAMP-END     1:15:33:953932 (~4533953 ms) of processor 1
ELAPSED   2 ms of processor 1
TIMESTAMP-END     1:15:33:954006 (~4533954 ms) of processor 3
ELAPSED   2 ms of processor 3
TIMESTAMP-END     1:15:33:954026 (~4533954 ms) of processor 5
ELAPSED   2 ms of processor 5
TIMESTAMP-END     1:15:33:954170 (~4533954 ms) of processor 6
ELAPSED   2 ms of processor 6
TIMESTAMP-END     1:15:33:954326 (~4533954 ms) of processor 7
ELAPSED   2 ms of processor 7
jani@jani-laptop:~/Desktop/modified prog$
```

Figure 4.2: *Snapshot for observaton 2*

processes to perform the task is 2ms.

C) **OBSERVATION 3**

These observations are taken considering the number of processes to be ten and the number of task to be performed to be ten.The screenshot in Figure 4.3 gives the obsevation under these condition.

RESULT DESCRIPTION:

The first line of the sceenshot gives the command for the creating eight processes and executing the algorithm.The second line gives the time at which the algorithm is compiled.The TIMESTAMP-START gives the start time of each of the slave process. The result of the work assigned to each of the slave process is given along with the process id of the process which performs the task.After completion of the assigned task what new task is assigned to which process is also displayed. After completion of the assigned tasks the TIMESTAMP-END of each of the slave process is displayed.Finally the times elasped for each of the slave processor is given. Thus as the eight process are computing the task parallely hence the time taken by five processes to perform the task is 1ms.

```
jani@jani-laptop:~/Desktop/modified prog$ mpirun -np 10 combine

  Compiled on May  7 2011 at 01:14:18.

TIMESTAMP-START   1:14:50:579193 (~4490579 ms) of processor 2
TIMESTAMP-START   1:14:50:579339 (~4490579 ms) of processor 4
TIMESTAMP-START   1:14:50:579452 (~4490579 ms) of processor 8
TIMESTAMP-START   1:14:50:579563 (~4490579 ms) of processor 6
TIMESTAMP-START   1:14:50:579688 (~4490579 ms) of processor 1
TIMESTAMP-START   1:14:50:579806 (~4490579 ms) of processor 3
TIMESTAMP-START   1:14:50:579920 (~4490579 ms) of processor 5
TIMESTAMP-START   1:14:50:580032 (~4490580 ms) of processor 7
process id:3 the result is:49

new work send to process_id:3 and the work is:0

process id:4 the result is:36

process id:5 the result is:25

process id:6 the result is:16

process id:7 the result is:9

process id:8 the result is:4

TIMESTAMP-START   1:14:50:580375 (~4490580 ms) of processor 9
process id:1 the result is:81

process id:2 the result is:64

process id:3 the result is:0

process id:9 the result is:1

TIMESTAMP-END     1:14:50:580539 (~4490580 ms) of processor 9
ELAPSED   0 ms of processor 9
TIMESTAMP-END     1:14:50:580577 (~4490580 ms) of processor 8
ELAPSED   1 ms of processor 8
TIMESTAMP-END     1:14:50:580615 (~4490580 ms) of processor 7
ELAPSED   0 ms of processor 7
```

```
TIMESTAMP-END     1:14:50:580653 (~4490580 ms) of processor 4
ELAPSED   1 ms of processor 4
TIMESTAMP-END     1:14:50:580694 (~4490580 ms) of processor 5
ELAPSED   1 ms of processor 5
TIMESTAMP-END     1:14:50:580733 (~4490580 ms) of processor 6
ELAPSED   1 ms of processor 6
TIMESTAMP-END     1:14:50:580771 (~4490580 ms) of processor 3
ELAPSED   1 ms of processor 3
TIMESTAMP-END     1:14:50:580811 (~4490580 ms) of processor 2
ELAPSED   1 ms of processor 2
TIMESTAMP-END     1:14:50:580850 (~4490580 ms) of processor 1
ELAPSED   1 ms of processor 1
jani@jani-laptop:~/Desktop/modified prog$
```

Figure 4.3: *Snapshot for observaton 3*

4.3 TIME ANALYSIS:

A) **OBSERVATION 1**

These observations are taken considering the number of processes to be five and the number of task to be performed to be fifty.The Table 4.1 below gives the time taken by five process to complete the task.Figure 4.4 below represents the time taken by five processes to perform the task.

Table 4.1: Table representing the time taken by five process to complete the task

Rank of process	Time taken to perform the subtask(in millisecond)
1	5
2	8
3	0
4	9

RESULT DESCRIPTION

Figure 4.4 below represents the time taken by five processes to perform the task. The x-axis represents the rank of the process and the y-axis represents the time taken in millisecond. As each of the processes is executing the subtasks in parallel hence

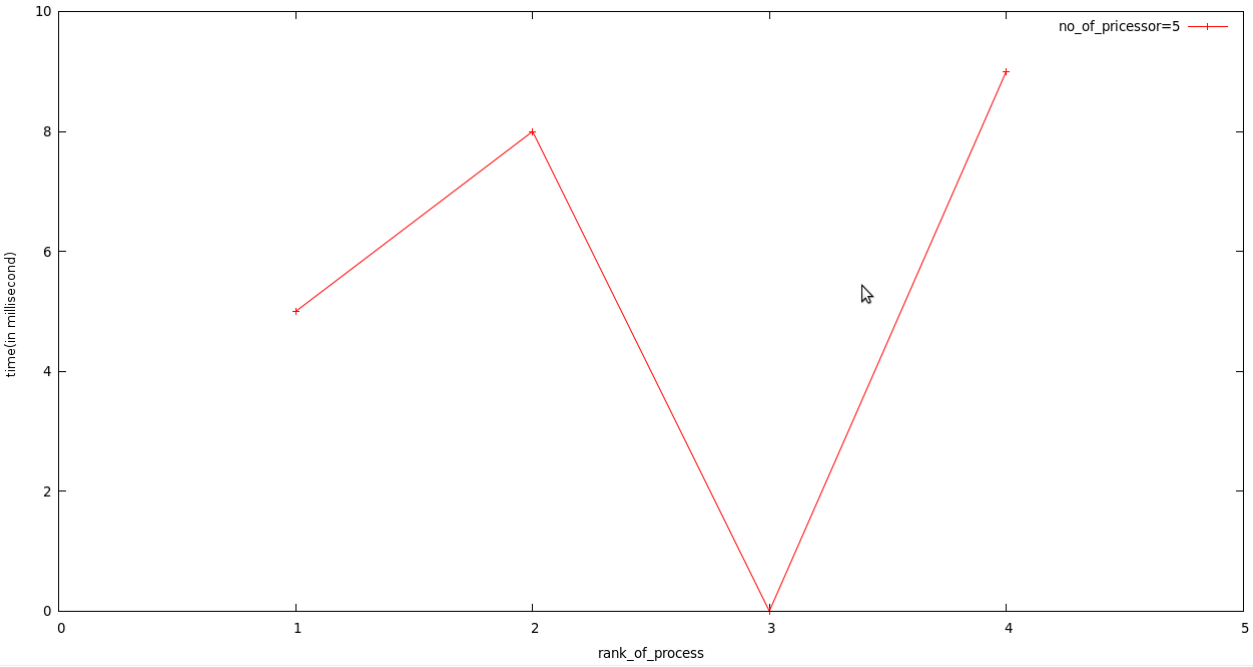

Figure 4.4: *Curve representing the time taken by five processes to complete the task parallel.*

maximum time taken by a process to execute the task will be considered as the time taken for the work to be done. Thus for five number of process the time taken is 9ms.

B) **OBSERVATION 2**

These observations are taken considering the number of processes to be eight and the number of task to be performed to be fifty.The Table 4.2 below gives the time taken by eight process to complete the task.Figure 4.5 below represents the time taken by eight processes to perform the task.

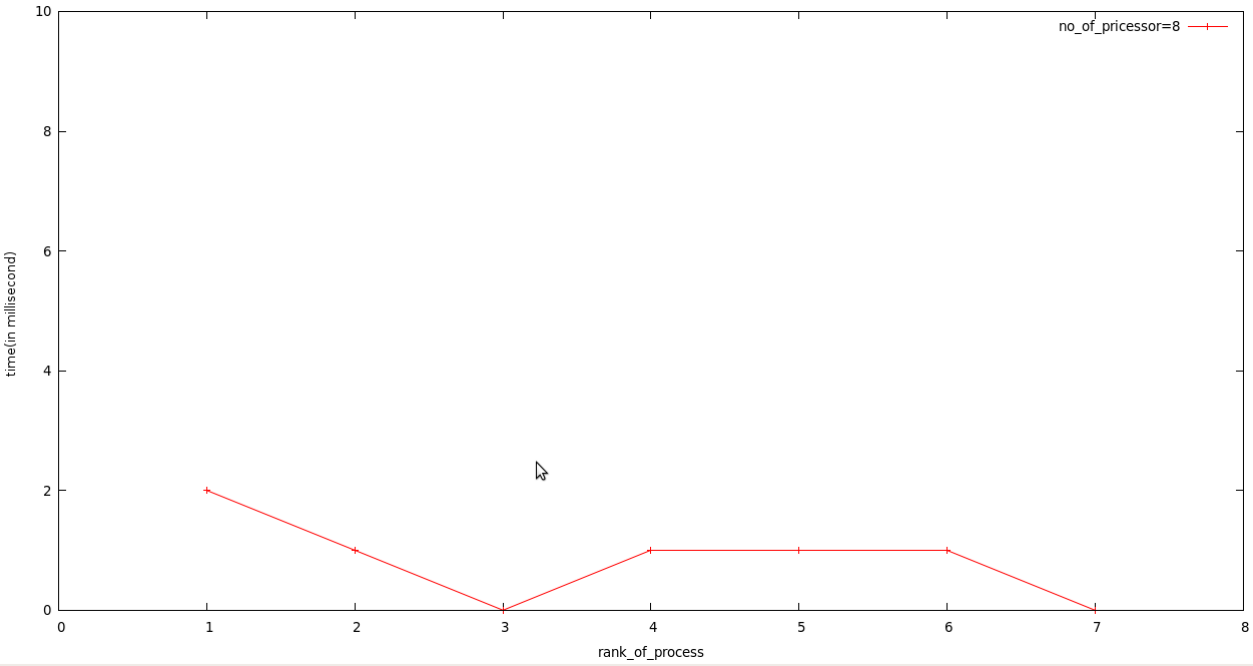

Figure 4.5: *Curve representing the time taken by eight processes to complete the task parallel.*

RESULT DESCRIPTION

Figure 4.5 below represents the time taken by eight processes to perform the task. The x-axis represents the rank of the process and the y-axis represents the time taken in millisecond. As each of the processes is executing the subtasks in parallel hence maximum time taken by a process to execute the task will be considered as the time taken for the work to be done. Thus for eight number of process the time taken is 2ms.

Table 4.2: Table representing the time taken by eight process to complete the task

Rank of process	Time taken to perform the subtask(in millisecond)
1	2
2	1
3	0
4	1
5	1
6	1
7	0

C) **OBSERVATION 3**

These observations are taken considering the number of processes to be ten and the number of task to be performed to be fifty.The Table 4.3 gives the time taken by ten process to complete the task.Figure 4.6 below represents the time taken by ten processes to perform the task.

Table 4.3: Table representing the time taken by ten process to complete the task

Rank of process	Time taken to perform the subtask(in millisecond)
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1

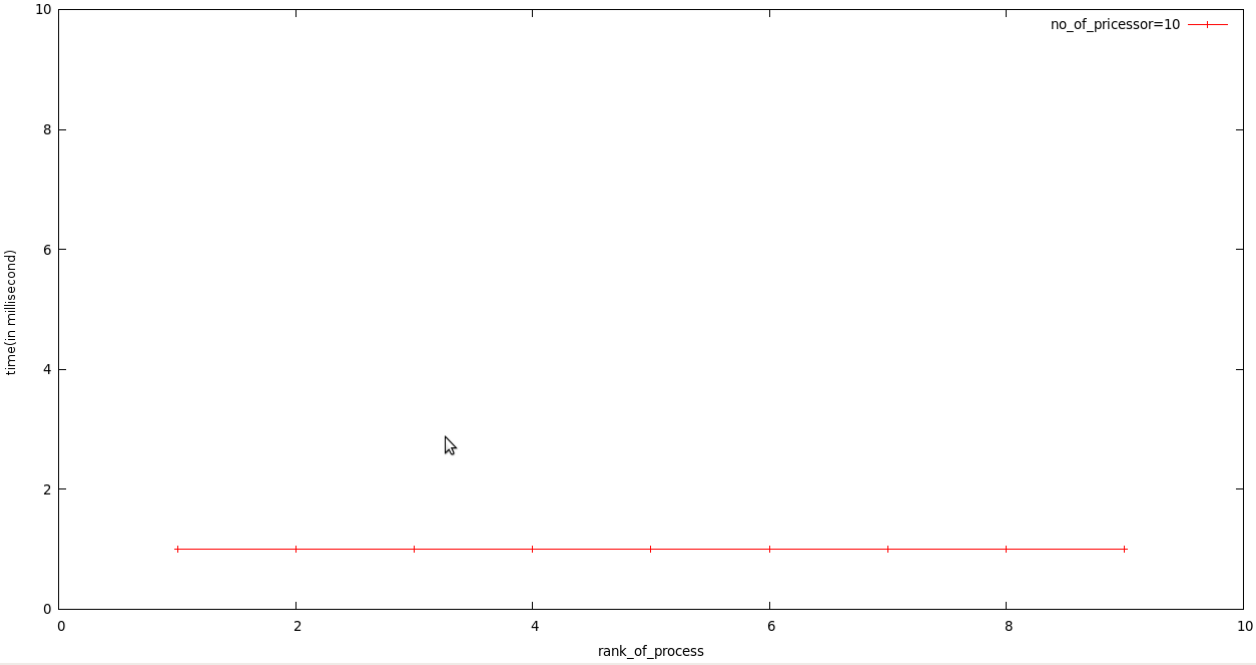

Figure 4.6: *Curve representing the time taken by ten processes to complete the task parallel.*

RESULT DESCRIPTION

Figure 4.6 below represents the time taken by ten processes to perform the task. The x-axis represents the rank of the process and the y-axis represents the time taken in millisecond. As each of the processes is executing the subtasks in parallel hence maximum time taken by a process to execute the task will be considered as the time taken for the work to be done. Thus for ten number of process the time taken is 1ms.

D) **COMBINING ABOVE THREE RESULTS:**

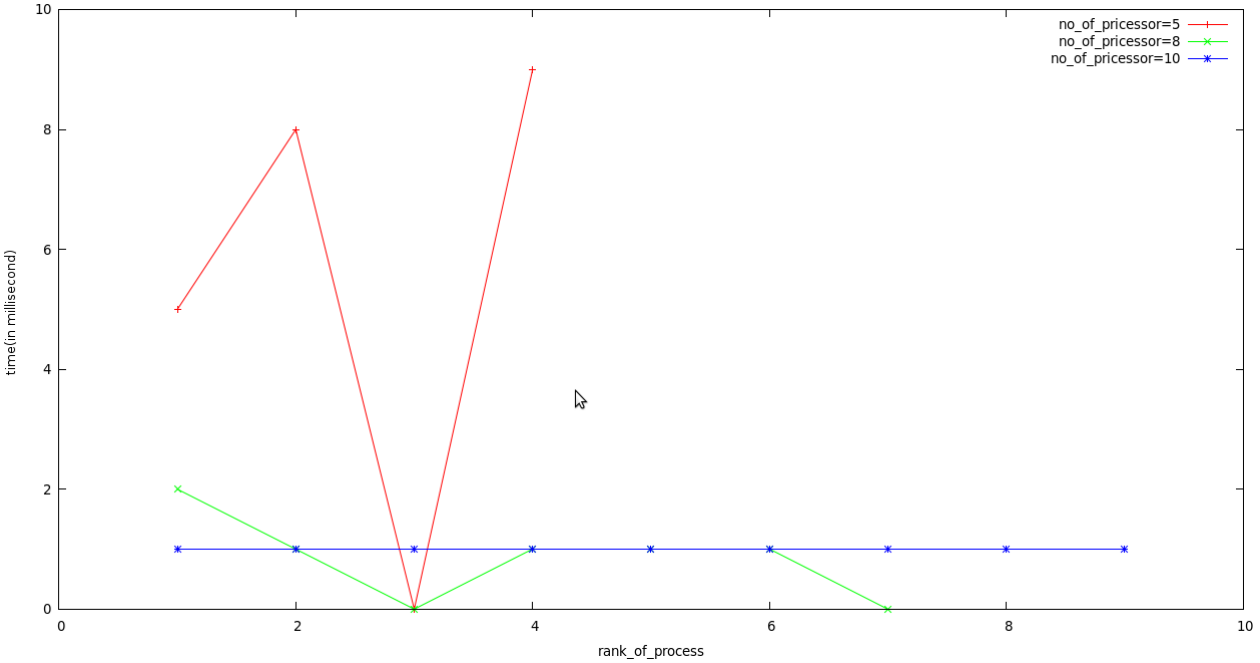

Figure 4.7: *Curves representing the time taken by five, eight, ten processes to complete the task parallel*

RESULT DESCRIPTION

This graph is plotted by taking the number of task as ten. The computation here is to find the square of ten integers.This is calculated at the slave process and the result is collected by the master process. This graph shows the comparison of time taken by the process to execute the subtasks when the number of process is five, eight and ten. As the master process is only responsible for assigning the subtasks and receiving results its timing is not considered. The red graph represents the time taken by each of the slave process when the number of process is five .As each of the processes are executing the subtasks in parallel hence the process taking maximum time to execute the task will be considered as the time taken for the whole task to

be performed. Thus for five number of process the time taken is 9ms. The green graph represents the time taken by each of the slave process when the number of process is eight. As each of the processes are executing the subtasks in parallel hence the process taking maximum time to execute the task will be considered as the time taken for the whole task to be performed. Thus for eight number of process the time taken is 2ms. The red graph represents the time taken by each of the slave process when the number of process is ten. As each of the processes are executing the subtasks in parallel hence the process taking maximum time to execute the task will be considered as the time taken for the whole task to be performed. Thus for ten number of process the time taken is 1ms.Thus it can be seen that when the number of process increases the amount of time taken to do the task decreases.

4.4 Conclusion

This chapter gives a brief overview about the platform which is used for execution of the algorithm. It also gives the experimental results obtained and a brief description about the result.

Chapter 5

Conclusion and Future Work

In this project the main objective was to study load balancing algorithms and to propose a load balancing algorithm.The proposed algorithm was implemented using MPI in C language. From the previous results it can be concluded that the time taken by five processes to do the task is 9ms whereas the time taken by eight processes to complete the same task is 2ms and the time taken by ten processes is 1ms. It can thus be observed that the load is well balanced and the result is computed in minimum time when the number of processor increases. Thus the time taken decreases with the increase in the number of processes.

These results has been observed taking into consideration that the master is involved only in assigning the task and receiving the results from the slave processes. The master while assigning the subtasks to the slave processor takes into account the load on that particular slave process and assigns the task only to those processes which have completed the task. In other words it can be said that master process is involved in scheduling the subtasks and at the same time performing load balancing. However if the master process along with assigning the task and receiving the results will also be involved in doing some subtasks then the result will differ.

Our future work will include intorduction of efficient load balancing algorithms for different class of distributed system.

Chapter 6

References

[1]Xiao Qin, Senior Member, IEEE, Hong Jiang, Member, IEEE, Adam Manzanares, Student Member, IEEE, Xiaojun Ruan, Student Member, IEEE, and Shu Yin, Student Member, IEEE, Communication-Aware Load Balancing for Parallel Applications on Clusters", pg-43-45, january 2010.

[2] E. Altman1, U. Ayesta2,3 , B.J. Prabhu2 INRIA Sophia Antipolis, France 2 LAAS-CNRS, Universite de Toulouse, France, "Load Balancing in Processor Sharing Systems," Ph.D. Thesis, University of Nice Sophia Antipolis, France, October 2005.

[3] A. Acharya and S. Setia, "Availability and Utility of Idle Memory in Workstation Clusters,"Proc. ACM SIGMETRICS, 1999.

[4] X. Qin, "Design and Analysis of a Load Balancing Strategy in Data Grids," Future Generation Computer Systems, vol. 23, no. 1, 2007.

[5] X.-D. Zhang, L. Xiao, and Y.-X. Qu, "Improving Distributed Workload Performance by Sharing Both CPU and Memory Resources," Proc. 20th Intl Conf. Distributed Computing Systems (ICDCS 00), 2000.

[6] Introduction to MPI Programming,Rocks-A-Palooza II

[7] http://en.wikipedia.org/