COMPUTER SCIENCE AND ENGINEERING

May 2021

CONTENTS

INTRODUCTION

1.1. Abstract

Task scheduling in parallel processing is a technique in which processes are assigned to different processors. Task scheduling in parallel processing uses different types of algorithms and techniques which are used to reduce the number of delayed jobs. Nowadays there is a different kind of scheduling algorithms and techniques used to reduce the execution time of tasks scheduling of jobs in parallel is becoming the subject of much research. The problem of job scheduling is to determine how resources should be shared in order to maximize the system's utility. This problem has been extensively studied for well over a decade. In this paper, we will first discuss the execution time of scheduling algorithm (LJF, SJF, PRIORITY SCHEDULING) in series and parallel, we discuss how deployed scheduling policies can be improved to meet existing requirements, specific research challenges and future scope.

1.2. Aim

Our aim is to create two programs, to perform job scheduling. The first program will do so sequentially while the second program will make use of OPENMP to distribute loops to different processors. Upon this, an analysis will be performed which will differentiate the two algorithms based on several factors such as execution time, memory etc.

1.3. CPU Scheduling Algorithms

The Central Processing Unit (CPU) is the brain of the computer system so it should be utilized efficiently. CPU Scheduling is one of the most important concepts of Operating System. Sharing of computer resources among multiple processes is called scheduling.

The various CPU scheduling algorithms are: -

1. SJF (Shortest Job First) CPU scheduling: - In this scheduling the process with the shortest CPU burst time is allocated to CPU first. It is the best way to reduce time. Here the actual time taken by the process to execute is already known by the processor

2. Priority scheduling: - In this scheduling the process with high priority is allocated to CPU first, and process with the same priority are executed in FCFS manner.

Priority can be decided on the basis of the memory allocation, or on the basis of the time or on the basis of some resource requirements.

3. LJF (Longest Job First) scheduling: - The LJF, which stands for Longest Job Scheduling Algorithm keeps track of the Burst time of all the available processes at the arrival time itself and then assigns the processor to that process which has the longest burst time. It a type of non-pre-emptive scheduling algorithm

The various scheduling parameters for the selection of the scheduling algorithm are:

1. Context Switch: A context switch is process of storing and restoring context (state) of a pre-empted process, so that execution can be resumed from same point at a later time. Context switching is wastage of time and memory that leads to the increase in the overhead of scheduler, so the goal of CPU scheduling algorithms is to optimize only these switches.

2. Throughput: Throughput is defined as number of processes completed in a period of time. Throughput is less in round robin scheduling. Throughput and context switching are inversely proportional to each other.

3. CPU Utilization: It is defined as the fraction of time CPU is in use. Usually, the maximize the CPU utilization is the goal of the CPU schedule

4. Turnaround Time: Turnaround time is defined as the total time which is spend to complete the how long it takes the time to execute that process.

5. Waiting Time: Waiting time is defined as the total time a process has been waiting in ready queue.

The various characteristics of good scheduling algorithm are:

1. Minimum context switches.
2. Maximum CPU utilization.
3. Maximum throughput.
4. Minimum turnaround time.
5. Minimum waiting time

LITERATURE SURVEY

We reviewed some past researches for our work. In the past, many researches have been made on how to improve the performance of the job scheduling in run-time to yield faster results.

[1] This paper is about scheduling jobs on distributed memory massively parallel processors. It talks about various scheduling models used in study of scheduling algorithms which can be classified according to the following criteria:

Partition Specification,Job Flexibility,Level of Pre-emption supported,Amount of job and workload knowledge available ,Memory Allocation

[2] This paper compares various scheduling algorithms such as-

1. FCFS
2. SJF pre-emptive
3. SJF, Non pre-emptive
4. LJF
5. Priority Scheduling

And mentions the advantages and disadvantages with respect to various parameters like implementation complexity, starvation, pre-emption .

[3] This paper discusses majorly two topics:

- To present some of the major technological changes and to discuss the additional dimensions they add to the set of JSSPP (JSSPP stands for Job Scheduling strategies for Parallel Processing) challenges.

- To promote and suggest research topics inspired by these dimensions in the JSSPP community.

Some researches were also made that compared different job scheduling algorithms for better optimization in terms of faster runtime output. Paper [4] and [5] mainly emphasizes on the comparison criteria of different CPU Scheduling algorithms. The algorithm is experimentally evaluated with data from a large computing facility. The treatment of shortest process in SJF scheduling tends to result in increased waiting time for long processes.

In the recent years, researchers have started to think of benefits of paralysing job scheduling algorithms for better memory usage and less runtime than serial processing. Many researches now focus on distributing the processes to enhance the performance of job scheduling algorithms [6] In this paper, various scheduling techniques for parallel computing are discussed like longest processing time, list scheduling.

OVERVIEW

3.1. Problem description

To Implement Different CPU Scheduling Algorithms in Parallel Programming Structure. This helps in reducing the execution time of tasks and increasing utility of the system. The variation in time when the scheduling algorithms are executed in parallel and series will be portrayed. Which becomes efficient in real time scenario and avoids delay for complex data structures. Also how deployed scheduling policies can be improved to meet existing requirements, specific research challenges and future scope will be discussed in this project.

3.2. Software Requirements

- Windows 7, Windows 8 or Windows 10
- Mac OSX 10.8, 10.9, 10.10 or 10.11
- Linux Operating system - Terminal

3.3. Hardware Requirements

Processor (CPU) with 2 gigahertz (GHz) frequency or above
A minimum of 2 GB of RAM

Monitor Resolution 1024 X 768 or higher

SYSTEM DESIGN

4.1. Description of methodology

There are two ways to benefit from parallelism: you can run the same program in a shorter time span, or a larger program in the same amount of time. For example if the serial implementation of job scheduling algorithms involves processing multiple processes for each of the iterations on a single processor. Thus, the loop executes once for each process. In the parallel implementation, each task performs this process for a given number of iterations, and reports the sequence of the processes to the master task, which calculates the cumulative average for all the iterations. The more process that participate, the faster the calculation completes. The number of iterations will be equally divided among the tasks.

This is what we have attempted to elucidate in our modules/programs. Through the comparison of numerous methods the findings from the data can be illustrated conclusively.

IMPLEMENTATION

5.1. Description of Modules/Programs

We have implemented job scheduling algorithms in serial and parallel. In serial programming structure, a single processor accounts for running loop for every process. In parallel programming structure, the loop execute parallel for every process. In SJF scheduling, MIN_VALUE is used for each process to compare it to burst time and running the processes for each loop. In LJF scheduling, MAX_VALUE is used for each process to compare it to burst time and running the processes for each loop. In

priority scheduling, the OPENMP is used for the quicksort method to run the processes according to their arrival and burst time.

5.2. Source Code and Execution snapshots

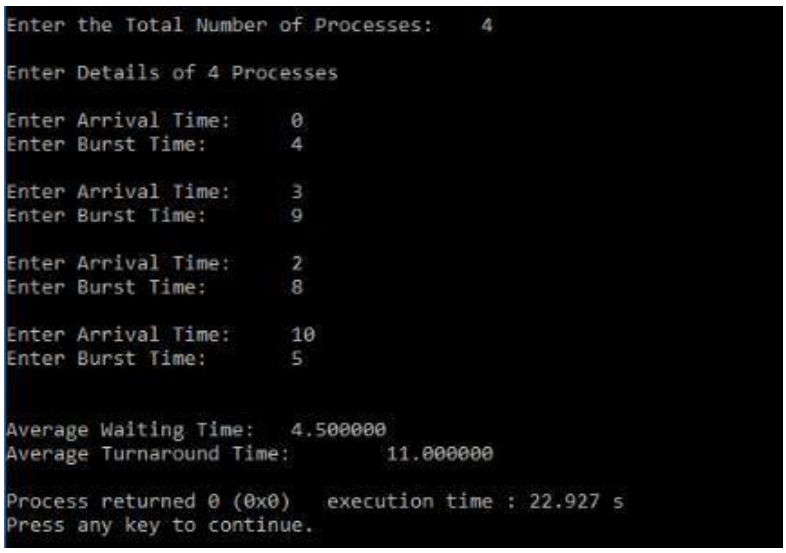
5.2.1 CODES FOR EXECUTION OF SCHEDULING ALGORITHMS IN SERIAL:

1. SJF serial

```
#include <stdio.h>
int main()
{
int arrival_time[10], burst_time[10], temp[10];
int i, smallest, count = 0, time, limit;
double wait_time = 0, turnaround_time = 0, end;

float average_waiting_time,
average_turnaround_time; printf("\nEnter the Total
Number of Processes:\t"); scanf("%d", &limit);
printf("\nEnter Details of %d Processes\n", limit);

for(i = 0; i < limit; i++)
{ printf("\nEnter Arrival Time:\t");
scanf("%d", &arrival_time[i]);
printf("Enter Burst Time:\t");
scanf("%d", &burst_time[i]);
temp[i] = burst_time[i];

}
burst_time[9] = 9999;
for(time = 0; count != limit; time++)

{
smallest = 9;
for(i = 0; i < limit; i++)

{

if(arrival_time[i] <= time && burst_time[i] < burst_time[smallest]
```

```
        && burst_time[i] > 0)

                    {
smallest = i;

                    }

                }
burst_time[smallest]--;
if(burst_time[smallest] == 0)

                {
count++;
end = time + 1;

                    wait_time = wait_time + end - arrival_time[smallest] - temp[smallest];
                    turnaround_time = turnaround_time + end - arrival_time[smallest];
                }

            }
average_waiting_time = wait_time / limit;

            average_turnaround_time = turnaround_time / limit;
            printf("\n\nAverage Waiting Time:\t%lf\n", average_waiting_time);
            printf("Average Turnaround Time:\t%lf\n",
            average_turnaround_time); return 0;
        }
```

OUTPUT:

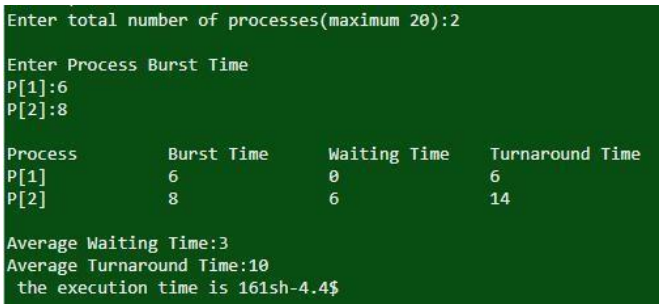


2. LJF serial

```
#include <stdio.h>
Int main()
{
Int arrival_time[10], burst_time[10], temp[10];
Int i, biggest, count = 0, time, limit;

double wait_time = 0, turnaround_time = 0, end;

float average_waiting_time, average_turnaround_time;
printf("\nEnter the Total
Number of Processes:\t"); scanf("%d", &limit);

printf("\nEnter Details of %d Processes\n", limit);

for(i = 0; i < limit; i++)

{

    printf("\nEnter Arrival Time:\t");

    scanf("%d", &arrival_time[i]);

    printf("Enter Burst Time:\t");

    scanf("%d", &burst_time[i]);

    temp[i] = burst_time[i];
} burst_time[9] = 9999; for(time =
0; count != limit; time++) { biggest
= 9;
    for(i = 0; i < limit; i++)

    {

        if(arrival_time[i] <= time && burst_time[i] > burst_time[biggest] &&
burst_time[i] > 0)

        {
biggest = i;

        }

    }
```

```
        burst_time[biggest]--;

        if(burst_time[biggest] == 0)

        {
count++;
end = time + 1;

            wait_time = wait_time + end - arrival_time[biggest] - temp[biggest];
            turnaround_time = turnaround_time + end - arrival_time[biggest];
        }

    }

    average_waiting_time = wait_time / limit;

    average_turnaround_time = turnaround_time / limit;
    printf("\n\nAverage Waiting Time:\t%lf\n", average_waiting_time);
    printf("Average Turnaround Time:\t%lf\n",
    average_turnaround_time); return 0;

}
```

OUTPUT:



3. Priority scheduling algorithm

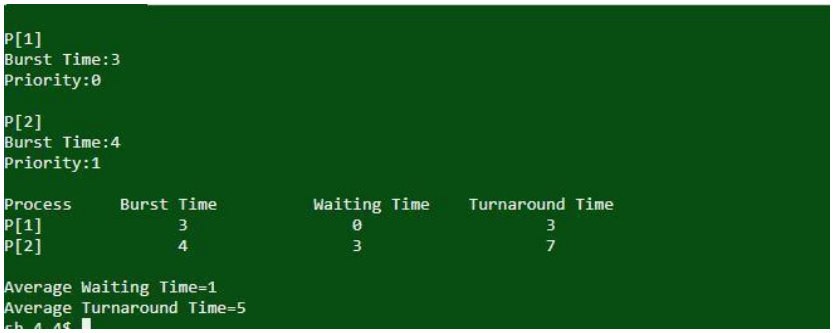
```
#include<stdio.h>

int main()

{
```

```
int
    bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat; printf("Enter
    Total Number of Process:"); scanf("%d",&n);

    printf("\nEnter Burst Time and Priority\n");

    for(i=0;i<n;i++)

    {

       printf("\nP[%d]\n",i+1);

       printf("Burst Time:");

       scanf("%d",&bt[i]);

       printf("Priority:");

       scanf("%d",&pr[i]);

       p[i]=i+1;          //contains process number

    }


    for(i=0;i<n;i++)

    {

       pos=i;

       for(j=i+1;j<n;j++)

       {

          if(pr[j]<pr[pos])

             pos=j;

       }

       temp=pr[i];

       pr[i]=pr[pos];
```

```
    pr[pos]=temp;
    temp=bt[i]; bt[i]=bt[pos];
    bt[pos]=temp;
    temp=p[i];
    p[i]=p[pos];
    p[pos]=temp;
}
wt[0]=0; //waiting time for first process is zero
for(i=1;i<n;i++)
{
    wt[i]=0;
    for(j=0;j<i;j++)
        wt[i]+=bt[j];
    total+=wt[i];
}
 avg_wt=total/n;      //average waiting time
total=0;

printf("\nProcess\t    Burst Time     \tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)
{
    tat[i]=bt[i]+wt[i];      //calculate turnaround time
    total+=tat[i];
     printf("\nP[%d]\t\t %d\t\t     %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
```

```
    }
    avg_tat=total/n; //average turnaround time
    printf("\n\nAverage Waiting Time=%d",avg_wt);
    printf("\nAverage Turnaround
    Time=%d\n",avg_tat); return 0;
}
```

OUTPUT:



5.2.2 CODES FOR EXECUTION OF SCHEDULING ALGORITHMS IN PARALLEL:

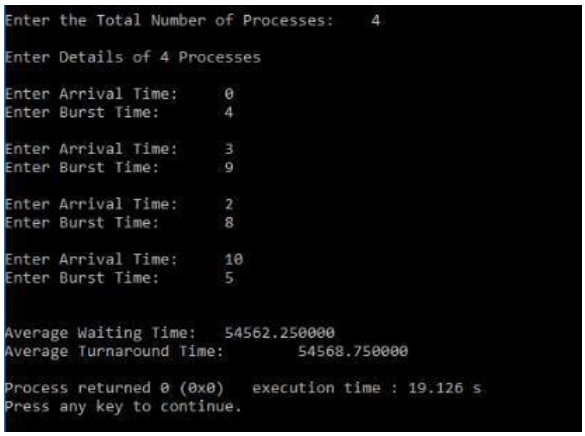
1. Shortest job first (parallel)

```
#include <stdio.h>

#include<omp.h>

int main()

{

    double arr[10];

    omp_set_num_threads(4);

    double min_val=9999;


     int arrival_time[10], burst_time[10], temp[10];

     int i, smallest, count = 0, time, limit;
```

```
  double wait_time = 0, turnaround_time = 0, end;

  float average_waiting_time,
  average_turnaround_time; printf("\nEnter the Total
  Number of Processes:\t"); scanf("%d", &limit);

  printf("\nEnter Details of %d Processes\n", limit);

  for(i = 0; i < limit; i++)

  {

       printf("\nEnter Arrival Time:\t");

       scanf("%d", &arrival_time[i]);

       printf("Enter Burst Time:\t");
       scanf("%d", &burst_time[i]);

       temp[i] = burst_time[i];

  }

  burst_time[9] = 9999;

  for(time = 0; count != limit; time++)

  {

       smallest = 9;


#pragma omp parallel for shared(i) reduction(min: min_val)

for( i=0; i<limit ; i++)

{

    if(burst_time[i] < min_val && arrival_time[i] <= time)

    {


       smallest=i;
```

```
    }

  }


      burst_time[smallest]--;

      if(burst_time[smallest] == 0)

      {
count++;
end = time + 1;

          wait_time = wait_time + end - arrival_time[smallest] - temp[smallest];
          turnaround_time = turnaround_time + end - arrival_time[smallest];

      }

    }

    average_waiting_time = wait_time / limit;

    average_turnaround_time = turnaround_time / limit;
    printf("\n\nAverage Waiting Time:\t%lf\n", average_waiting_time);
    printf("Average Turnaround Time:\t%lf\n",
    average_turnaround_time); return 0;

}
```

OUTPUT:



2.

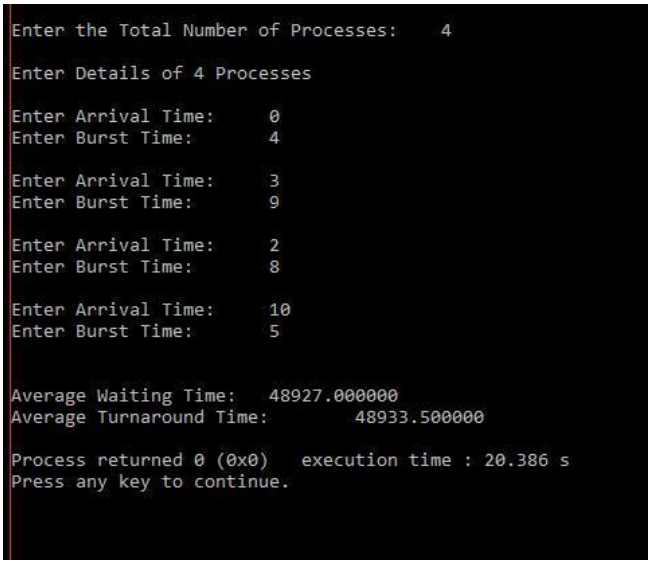
Longest job first (parallel)

```
#include <stdio.h>
#include<omp.h>
int main()
{
    double arr[10];
    omp_set_num_threads(4);
    double min_val=0;

     int arrival_time[10], burst_time[10], temp[10];
     int i, smallest, count = 0, time, limit;
     double wait_time = 0, turnaround_time = 0, end;
     float average_waiting_time, average_turnaround_time;
     printf("\nEnter the Total
     Number of Processes:\t"); scanf("%d", &limit);
     printf("\nEnter Details of %d Processes\n", limit);
     for(i = 0; i < limit; i++)
     {
          printf("\nEnter Arrival Time:\t");
          scanf("%d", &arrival_time[i]);
          printf("Enter Burst Time:\t");
          scanf("%d", &burst_time[i]);
          temp[i] = burst_time[i];
     }
     burst_time[9] = 9999;
     for(time = 0; count != limit; time++)
```

```
    {
        smallest = 9;

   #pragma omp parallel for shared(i) reduction(max: max_val)
   for( i=0; i<limit ; i++)
   {
      if(burst_time[i] > max_val && arrival_time[i] <= time)
      {

         smallest=i;
      }
   }

         burst_time[smallest]--;
         if(burst_time[smallest] == 0)
         {
count++;
end = time + 1;
              wait_time = wait_time + end - arrival_time[smallest] - temp[smallest];
              turnaround_time = turnaround_time + end - arrival_time[smallest];

         }
    }
    average_waiting_time = wait_time / limit;
    average_turnaround_time = turnaround_time / limit;
    printf("\n\nAverage Waiting Time:\t%lf\n", average_waiting_time);
    printf("Average Turnaround Time:\t%lf\n",
    average_turnaround_time); return 0;
```

}

OUTPUT:



3. Priority scheduling (parallel)

```
#include<stdio.h>

#include<omp.h>

int k=0;


int partition(int arr[], int low_index, int high_index)

{

int i, j, temp, key;

key = arr[low_index];

i= low_index + 1;

j= high_index;

while(1)

{
```

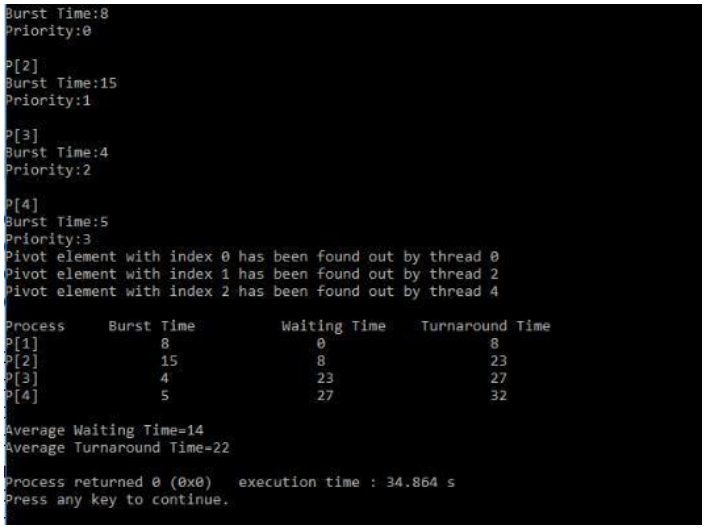
```
while(i < high_index && key >= arr[i])

    i++;

while(key < arr[j])

    j--;

if(i < j)

{

temp = arr[i];

arr[i] = arr[j];
arr[j] = temp;

}

else

{

temp= arr[low_index];

arr[low_index] = arr[j];

arr[j]= temp;

return(j);

}}}

void quicksort(int arr[], int low_index, int high_index)

{

int j;


if(low_index < high_index)

{

j = partition(arr, low_index, high_index);
```

```
printf("Pivot element with index %d has been found out by thread %d\n",j,k);


#pragma omp parallel sections

{

    #pragma omp section

    {

        k=k+1;

        quicksort(arr, low_index, j - 1);

    }


    #pragma omp section

    {

        k=k+1;

        quicksort(arr, j + 1, high_index);

    }}}}

int main()

{

    int bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
    printf("Enter Total Number of Process:");

    scanf("%d",&n);


    printf("\nEnter Burst Time and Priority\n");

    for(i=0;i<n;i++)

    {

        printf("\nP[%d]\n",i+1);
```

```
    printf("Burst Time:");
    scanf("%d",&bt[i]);
    printf("Priority:");
    scanf("%d",&pr[i]);
    p[i]=i+1;           //contains process number
}

//sorting burst time, priority and process number in ascending order using selection sort
quicksort(pr, 0, n - 1);

wt[0]=0; //waiting time for first process is zero

//calculate waiting time
for(i=1;i<n;i++)
{
    wt[i]=0;
    for(j=0;j<i;j++)
        wt[i]+=bt[j];

    total+=wt[i];
}

avg_wt=total/n;       //average waiting time total=0;

 printf("\nProcess\t     Burst Time     \tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)
```

```
    {
        tat[i]=bt[i]+wt[i];      //calculate turnaround time
        total+=tat[i];
        printf("\nP[%d]\t\t %d\t\t     %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=total/n; //average turnaround time printf("\n\nAverage
    Waiting Time=%d",avg_wt); printf("\nAverage Turnaround
    Time=%d\n",avg_tat);

    return 0;
}
```

OUTPUT:



RESULT COMPARISION

	Serial		Parallel	
	Waiting Time (in seconds)	Turnaround Time (in seconds)	Waiting Time (in miliseconds)	Turnaround Time (in miliseconds)
SJF	4	11	545	568
LJB	3	10	480	497
Priority	1	5	14	22

CONCLUSION AND FUTURE SCOPE

In the above table, we can clearly see the vast time improvements when parallel job scheduling algorithms are deployed in comparison to plane old serial job scheduling algorithms.

Task scheduling in parallel processing is a technique in which processes are assigned to different processors. In series, task are assigned to a single processor. Nowadays there is a different kind of scheduling algorithms and techniques used to reduce the execution time of tasks scheduling of jobs in parallel is becoming the subject of much research. The problem of job scheduling is to determine how resources should be shared in order to maximize the system's utility. This problem has been extensively studied for well over a decade. In this paper, we discussed the execution time of scheduling algorithm (LJF, SJF, PRIORITY SCHEDULING) in series and parallel, and we also discussed how deployed scheduling policies can be improved to meet existing requirements, specific research challenges and future scope.

Paralysing job scheduling algorithms brings scope of future researches as in job scheduling, the most important thing need to be considered while executing are throughput, utilization, and response time. For future scope, Round robin is excellent for parallel computing is because it is great in load balancing if the tasks are around same lengths but When scheduling algorithm provides a time slice for each job the ready jobs are placed in queue and new jobs are placed into the end of the queue. In this technique the processor efficiency will worst when time quantum is too short and when it is too long it will generate a poor

response so improving this in future helps to increase task scheduling time and more suitable for complex data structures in future .

REFERNCES

1. By- Uwe Schwiegelshohn & Ramin Yahyapour –"Analysis of First-Come-First-Serve Parallel job Scheduling" JUNE-1998

2. By- Sumit Kumar Nager & Naseeb Singh Gill –"Comparative study of various CPU Scheduling algorithms" MARCH -2001

3. By- Eitan Frachtenberg and Uwe Schwiegelshohn –"New Challenges of Parallel Job Scheduling" JAN-2007

4. By- Arvind Seth, Vishaldeep Singh –"Types of Scheduling Algorithms in Parallel computing" 2007 :

5. By- Yang Rui, Zhang Xinyu –"An Algorithm for assigning tasks in parallel computing" JAN-2010

6. By- Dror G.Feitelson1, Larry Rudolph1, Uwe Schwiegelshohn2, Kenneth C. Sevcik3 and Parkson Wong41- "Theory and Practice in Parallel Job Scheduling"- 12-JULY2015