PARALLEL PROGRAMMING

Parallel Implementation Of Scheduling Algorithms

(Assignment 3)

Prepared by:

NAME	REG NO	ROLL NO
Disha Parwani	150953258	65
Suvigya Agrawal	150953130	31
Aishwarya Yadav	150953076	19
Shubham Rateria	150953084	22

Instructor : Ms. Veena Mayya

Course : Parallel Programming

Date : 03-11-17

Parallel Implementation Of Scheduling Algorithms

Disha Parwani
Computer & Comm. Engg.
Dept. Of Information & Comm. Technology
Manipal Institute Of Technology
disha.parwani9927@gmail.com

Suvigya Agrawal
Computer & Comm. Engg.
Dept. Of Information & Comm. Technology
Manipal Institute Of Technology
suvigya.com@gmail.com

Aishwarya Yadav
Computer & Comm. Engg.
Dept. Of Information & Comm. Technology
Manipal Institute Of Technology
aishwaryayadav91@yahoo.com

Shubham Rateria
Computer & Comm. Engg.
Dept. Of Information & Comm. Technology
Manipal Institute Of Technology
shubham.agarwal982@gmail.com

Abstract **: This paper presents a parallel implementation of various scheduling algorithms. It compares between different types of algorithms and highlights the advantage of using parallel scheduling over serial scheduling of processes in terms of time and space complexity.**

I. INTRODUCTION

The computational speed of CPU has been hugely bounded by that of GPU. GPU is a graphical processing unit which is very efficient at computer graphics manipulation and image processing. The highly parallel structure of GPU makes it more efficient over parallel computation done in general purpose CPUs. The computation of 3D functions is very difficult on CPU, hence GPUs are used for this purpose. GPU was introduced for graphical purposes but it has now been evolved into computing, accuracy and performance. Using the GPU for non-graphical purposes is known as General Purpose GPU or GPGPU, which is used to perform complex mathematical computations in parallel hence achieving low time complexity.

This report proposes the use of parallel platform using CUDA technology to enhance the performance of CPU/process scheduling algorithms. The different types of CPU/job scheduling algorithms include: First Come First Serve (FCFS) , Shortest Job First (SJF) , Round Robin (RR) , Priority Based Scheduling (PBS) . Scheduling is the process of arranging, controlling and optimizing the allocation of system resources to threads,

processes and data flows for maximum utilization. Scheduling is done in order to maintain, balance and share the system resources equally and effectively to achieve a target quality of service. Scheduling is required to perform multitasking and multiplexing. Scheduling is a complex job requiring extensive processing which is better performed on a parallel platform. This report demonstrates the performance of Priority Based Scheduling algorithm checking how fast it could perform on execution of parallel code in CUDA C on GPU implementation in comparison to serial code in C language when executed on single threaded CPU.

II. PARALLEL SCHEDULING

Parallel algorithms for scheduling problems for minimizing the number of unscheduled jobs, considering the problem of scheduling n jobs with different processing times on one machine subject to a common release date and different due dates, in order to maximize the number of jobs that are scheduled in time. This problem has been dealt by Dekel and Sahni , but with an erroneous assumption. The serial code for this problem has to be carefully analyzed ,based on binary tree method as presented by Dekel and Sahni [1983a]. The model used in this is, single instruction multiple data shared memory model debarring only write conflicts and not read conflicts.

Description :

The problem here considered involves, we consider one machine which is continuously available and it processes one job at a time. There are several jobs ranging from 1 to n ,i.e., i=1,2,....n, with a common release date r for all jobs, and processing requirement p_i and due-date d_i for each job. Our main objective here involves the maximization of jobs that can be scheduled in such a fashion that the jobs are processed within the interval of release date and due-date. This accounts for a feasible schedule.

Moore-Hodgson algorithm:

This algorithm is executed keeping in mind the following facts: A subset of the jobs can be scheduled such that no job is tardy ,i.e., no job finishes after its due-date, if and only if the schedule so formed follows the scheme of EDD(earliest due date) is feasible. In other words, a set of jobs with due-dates $d_1,d_2,....\ d_n$ and processing times $p_1,p_2,...,p_i$ is feasible if and only if

```
for j=1,2,...,i:
    r + (p1+p2+...+pi) <= di;
```

The sequential algorithm which is the fastest to solve this, is by Hodgson having 0(n log n) complexity, which follows the greedy-type procedure, and functions as follows:

(i) The jobs are arranged in increasing order of due-dates, done by beginning the scheduling of the jobs at the release date.

(ii) The job taken into consideration is appended to the list of jobs scheduled so far. If its completion time exceeds its due-date, we search for the longest job in the list of jobs scheduled so far (including the job just appended) and remove it from the list. We observe that, all of the jobs now in the list complete in time. We then move forward by taking into account the next job.

(iii) The final list acquired is the set of jobs forming the optimal schedule.

Formally, the algorithms runs as follows :

```
Initially, we assume that
d1<=d2<=...<=dn;
f := r; s := 0;
for i in from 1 to n do:
     f := f+p; s := s U { i };
if f > di then :
     j := argmax {pj | j E S};
     f := f - pj; s := s \ {j};
```

Example:

n=6, r=0**,** the di and pi values are given in the table. The optimal schedule obtained by the algorithm involves the inclusion of the jobs 2, 3, 5, and 6. The start time a_i of the jobs in the schedule is also shown. Job 1 is canceled when job 3 is considered, and job 4 is canceled when it is considered itself.

i	1	2	3	4	5	6
d_i	8	9	10	11	16	17
p_i	6	4	3	5	7	2
a_i	-	0	4	-	7	14

Analysis of the algorithm

THEOREM 1: The optimality of set S constructed by the algorithm can be measured in two ways:

(1) It is a maximum-cardinality feasible set of jobs.

(2) Among the feasible sets with equal cardinality, it has minimum total processing time of the jobs (or equivalently, minimum finishing time).

PROOF: Assume an optimal set of jobs (T) having the jobs that are not in S. We shall prove that T can be converted rather changed into a set T' which :

List Scheduling

In this scheduling, it makes a list of processes according to their priority and assigned the highest priority process to the least loaded processor. Then there are also some Heuristic algorithms of scheduling in parallel processing, which are:

- Opportunistic Load Balancing: In this scheduling, the next task is assigned to the next free processor.

- User Directed Allocation:In it user assigns the next task to the processor with lowest expected time to complete

(ii) the jobs which are scheduled afterwards are finished by pj-pk earlier than in T, and therefore in time.

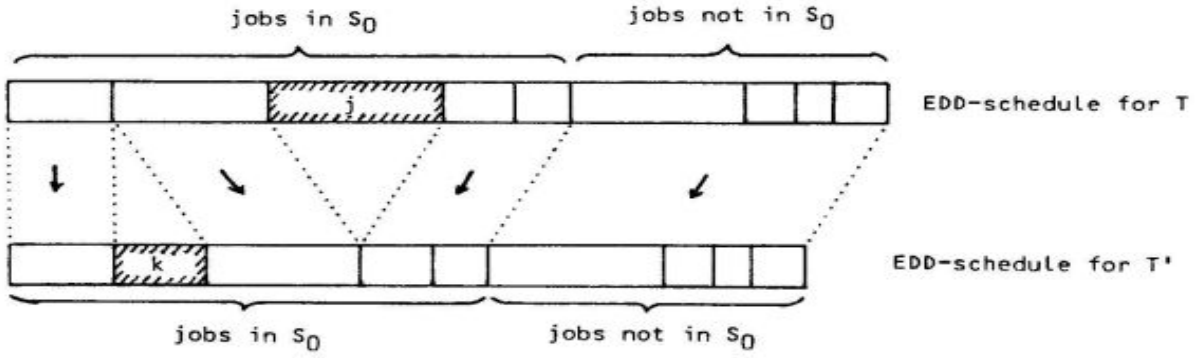


FIGURE 1
Construction of the EDD-schedule T' from the schedule T.

(a) is still feasible,

(b) contains equal number of jobs as before,

(c) has total processing time same as before,

(d) is therefore optimal, and

(e) has one more job in common with S.

Let's consider j to be the first job in T\S (with smallest due-date). We denote the set of jobs from which the job j was canceled in the algorithm, by S_o. All jobs in S_0 are not included in the feasible set of job. Therefore, there must be some job **k** present in S_o but not present in T. By the construction of j, $p_k <= p_j$,we formulate the set T' by substituting j by k in T. Therefore we conclude, that the EDD-schedule for the set **T** and subsequently T' is also feasible on the following basis:

(i) the jobs in S_o, which are scheduled first, are completed on time because they are a subset of a feasible set – $S_0 \setminus j$;

The rest of the properties of T', (b), (c), (d), and (e), are thereby trivial.

By reiterating the above transformation n number of times (depending on the necessity) , we can change T into an optimal set which is a subset of the feasible set S and which is therefore equal to S.

Refinement of the algorithm

One undesirable property of the algorithm, is that when job j is considered, the algorithm does not conclude whether this job will be included in the final schedule. This is partly the reason for the sequential character of the algorithm. We perform some preprocessing to make a final decision on whether a job should be included in the final schedule or not. This is done in a sequential manner. Assume that the jobs are sorted by due-dates, where ties are broken in an arbitrary fashion, and when choosing the job with largest processing time from the set of jobs, ties are broken in favor of the job with the smallest

number. If a job i is added into the list at some time, then it will not finally be present in the list unless all subsequent jobs with shorter (or equal) processing time are already included in the final list, because when the algorithm subsequently searches for a job to be excluded, it will prefer excluding i than any of the subsequent jobs that are shorter. Now let's assume that we are aware of the jobs before i that belong to the optimal schedule, and these jobs form the current list. When we choose job i in the algorithm, we check whether the schedule formed by adding i and all subsequent shorter jobs to the current list is feasible. If not, we pass by i. If it is, then we can hold onto i in the final solution, since subsequent jobs which have larger processing time would be eliminated rather than i, and the remaining jobs can be scheduled feasibly.Thus the algorithm can be changed into:

The subscripts of the variable f have been added to show the values of the steps of the algorithm.

We assume that d1 <= d2 <= ... <= dn.

A) Preprocessing:

```
for each i = 1 ... n do:
     Si := min{ dj - { pk | i<=k<=j ,pk <= pj } | i<=j<=
     Si is the latest possibly start time for job i if a
      pj<=pi are to be scheduled in time. (Actually, the
      be taken over all j with pj <= pi.)
```

```
     if fi-1<=si then fi-1,i(fi-1) = f
i.e., the graph of foi is obtained from
everything which is below the horizonta
```

B) Main loop:

```
f0 := r; s := ;
for i from 1 to n do:
     if   fi - 1 <= s i  then fi := fi - 1 + pi ; s := s U { i };
                    (Job i will be scheduled.)
     else fi := f i - 1;
          (Job i will not be scheduled)
```

For the given same example, the value of s_i and f_i are computed. The jobs that are form the part of optimal schedule are starred (Fig1).

Example:

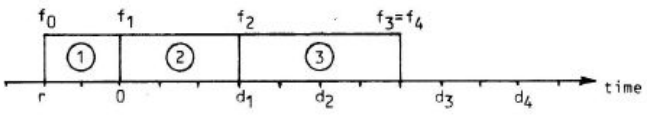
i	0	1	2	3	4	5	6
d_i		8	9	10	11	16	17
p_i		6	4	3	5	7	2
s_i		-7	3	7	6	8	15
f_i	0	0	*4	*7	7	*14	*16

Fig 1

i	1	2	3	4
p_i	2	3	4	5
d_i	3	5	8	10
s_i	1	2	4	5

Fig 2

A similar example is shown in figure 2, where n=4,r=-2. The most optimal schedule is drawn as shown in the Fig3.



Parallelization:

The conversion of the serial code to parallel can be done directly for the preprocessing part, but the main loop is inherently sequential. The value of f_i is obtained from f_{i-1}. The only thing that can be stated is the abstract functional relationship between f_{i-1} and f_i, generalizing it to f_i anf f_j, for i<j, shown as F_{ij}. The function F_{ij} allots the value of j for all possible values of f_i, thereby it can be dealt with without knowing a specific value of f_i. This is how decomposing the problem for solving it for f_n starting from f_0 into independent parts can be executed in parallel. The drawback is that one has to represent F_{ij} for the entire domain of values of f_i, rather for a single value.

It is quite observable, that F_{ik} is composed of F_{ij} and F_{jk} and F_{on} $(f_0)=F_{on}(r)=f_n$ is the time of completion for the optimal schedule. Fig 4 portrays the functions F_{oi}, i=1,2,3,4 for the example shown in Fig 3. The corresponding set S of jobs in the optimal solution is also given for all values of f_o=r. F_{oi} can be obtained by mathematical induction of $F_{O,i-1}$ by applying $F_{i-1,1}$ is shown as the following function above :

THEOREM 2: Let O<=i<j<=n and j—i=m. Then the function F_{ij} has the following properties:

(1) It is linear piecewise with constant slope equal to 1.

(2) The linear pieces are defined on half—open intervals ,i.e., open on the left side and closed on the right side.

(3) There are at most (2^{m+1}) discontinuities called "breakpoints".

(4) At the discontinuities, there are in total at most m different ordinate values which occur as left limits (i. e. at the right endpoints of the linear pieces), and at most m different ordinate values which occur as right limits.

(5) The function consists of at most m+1 strictly increasing portions, and
the cardinality of the solution set S, restricted to {i+1,i+2,...,j}, is constant on each portion

(6) Each of these portions consists of at most m different linear pieces.

PROOF: Properties (1) and (2) are immediate. Now we show (5):

It is distinct that the cardinality of $S_n\{i+1,1+2,...,j\}$ decreases from m to 0 monotonically as we move f_i from left to right. The domain of F_{ij} can be split into m+1 intervals thereby keeping the cardinality constant. Let's consider an interval, where the cardinality of the interval is t. Since $F_{ij}(fi)$ is the sum of f_i and the shortest possible total processing time of a schedule for a subset of the jobs {i+1,i+2,...,j} with cardinality t starting at the release date fi, and since any feasible schedule with release date f_i is also feasible for all release dates $f_{i'}<f_i$, it follows that F_{ij} is strictly increasing in the interval. Properties (3) and (4) will be demonstrated by induction on m: We conclude from $F_{i,i+m-1}$ to.

For m=1 the assertion is obvious.

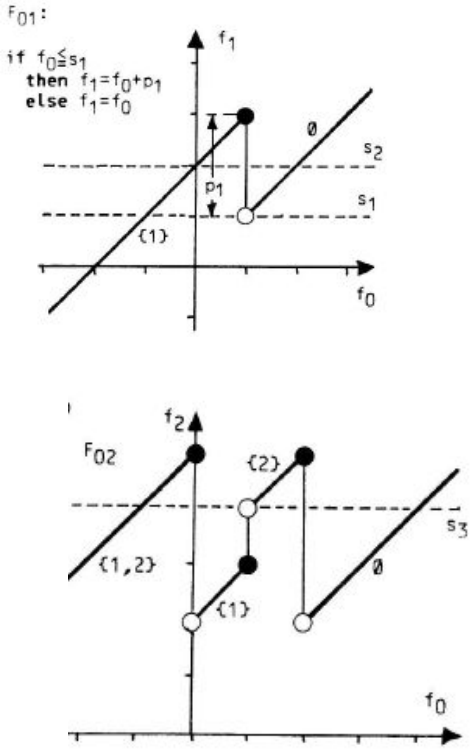
For m>1, F_i is obtained from $F_{i,i+m-1}$ by applying the function $F_{i+m_i,i+m}$, i. e. by adding d_{i+m} to all function values <=s_{i+m}. New breakpoints can be observed when the function $F_{i,i+m_1}$ crosses the horizontal line at s_{i+m} continuously. As there are at most m monotonically increasing portions, this can be observed at most m times, leading to the formation of at most m new breakpoints. This implies (3).

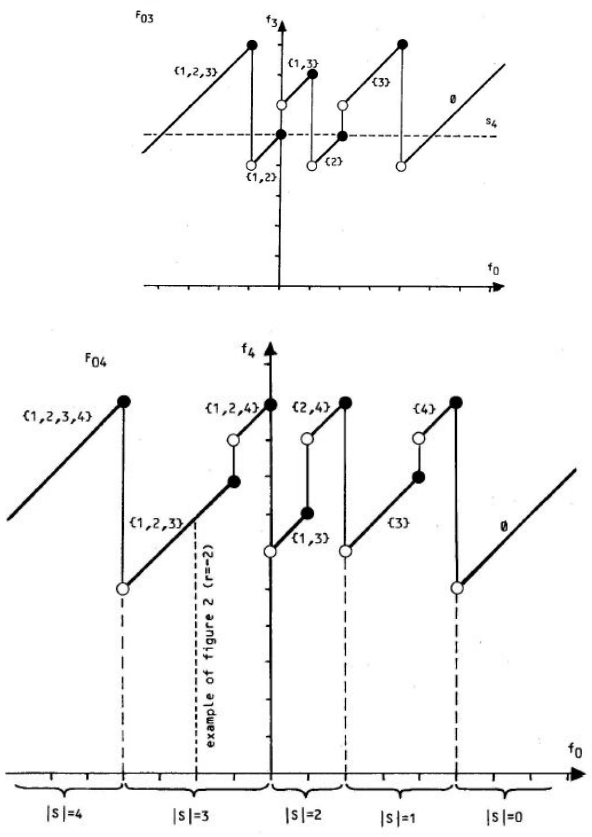
Property (6) follows from (5) and (4).

Remark 1: For the argument leading to (5) to be true, this theorem requires that the values s_i and p_i have been preprocessed.

Remark 2: We drop some elements from the solution set S and exchanged against others as r proceeds from right to left, leading to the monotonic growth of S. This is the false assumption on which the algorithm formulated by Dekel and Sahni was based.

The following four figures represent Fig 4.





Describing Parallel Algorithm:

The method suggested by Dekel and Sahni involved the use of binary tree method to solve the algorithm. The procedure here is to calculate F_{On} by splitting it into two "equal-sized" parts F_0 and F_{in}, computing each parallely followed by the computation of these functions. F_{oj} and F_{in} are further solved by simultaneous decomposition carrying out the process further recursively. In the binary tree, the leaves are associated with the functions F_{i-1} i=1,2,...,n, in that order. It is also seen that the common predecessor of the nodes which are associated with F_{ij} and F_{jk}, respectively, is associated with F_{ik}. The root is associated with F_{On}.

We proceed from the leaves to the root after preprocessing the data. Then we evaluate the function F_{On} at the point r and proceed from the root to the leaves, computing the values fi, i=0,...,n. This is done as follows:

> Assume that we are in node having function Fik, having two child nodes associated with Fij and Fjk, and we have already computed fi. We then compute $f_j:=F_{ij}(f_i)$. Now that we know the values of f_i and f_j, we can go down by one level in the tree.
>
> Finally, we gather this information and formulate the start times for all the scheduled job.

Implementation of parallel code:

LEMMA: Given a sequence $(a_1,a_2,...,a_m)$ of m numbers, the partial sums sequence $(S_1,...,S_m)$ with $S_i=a_1+a_2+...+a_i$ can be computed in 0(log m) time with m processors.

We represent each function F_{ij} by an array of at most (j-i+1)(j-i)/2+1 triples (l,u,c) sorted by the l component. A triple (l,u,c) has the following meaning: "If $l<f_i<=u$, then $Fij(f_i)=f_i+c$."

Overview of the parallel algorithm:

The preprocessing step involves jobs to be ordered by due-dates and computing the s_i values. Sorting is done by the simple scheme of enumeration proposed by Muller and Preparata with a total of n^2 processors having a complexity of 0(log n). The process of computing s_i by computing partial sums and one parallel subtraction and minimum-finding using n processors can be done in 0(log n) time. Thus the computation of all s_i values takes 0(log n) time on n^2 processors.

The storage required for the function F_{ij} is $0((j-i)^2)$, concluding that a total storage of $0(n^2)$ stores all functions.

At each tree node, the determination of $f_j=F_{ij}(f_i)$ happens in constant time having $(j-i)^2$ processors on the function Fij as a list of sorted triples. Concluding the total processor requirement to at most n^2 at each level of tree while traversing from the root to the leaves taking total time of 0(log n). The start times for each job computation can be done by performing the computation of partial in 0(log n) time, considering n processors.

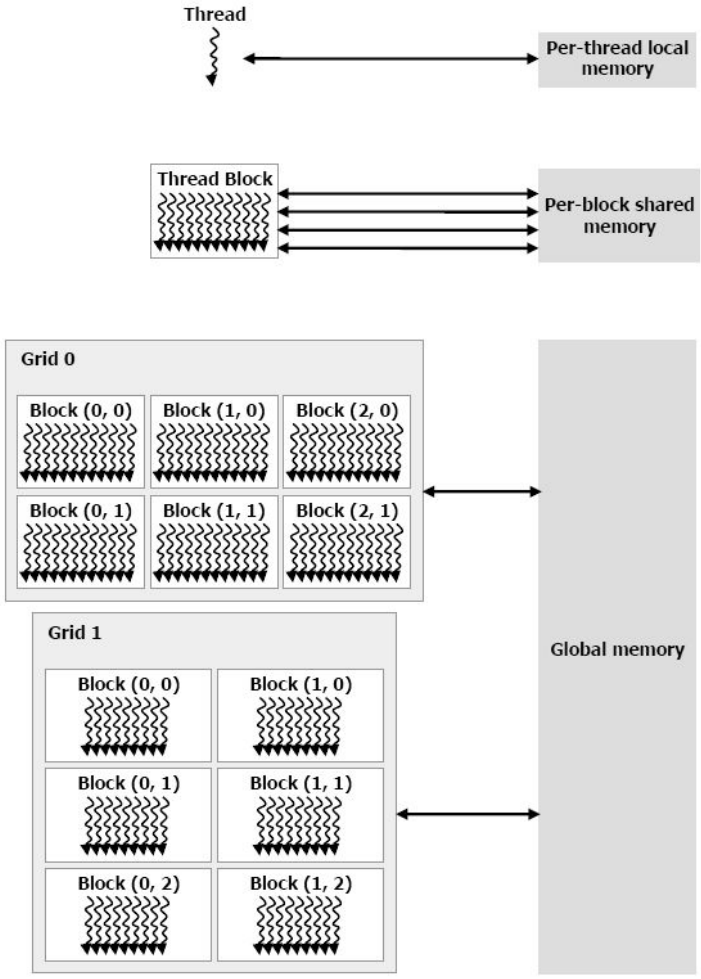
Thus the algorithm, can be implemented in 0(log n) steps provided we have n^2 processors.

III. GPU COMPUTATION

NVIDIA's manufacturing GPU architecture featured in the graphic cards is CUDA (Compute Unified Device Architecture). For general purpose computing CUDA C or C++ programming languages are used. For highly parallel algorithms CUDA is used. GPU has cores that can communicate and exchange information with each other. To increase the performance of the algorithms while running on GPU there is a need to have many threads. Number of threads is proportional to the performance of the algorithm. CUDA is used primarily to execute thousands of threads in parallel. During execution all the threads execute the same function or code known as a kernel. GPU executes these threads using same instructions and different data policy. Each thread has its own ID, based on which it determines the data it has to operate on depending on the code.

Every program written in CUDA is unified and has two parts, one is host code which is executed on CPU and other is device code which is executed on GPU. Only in some cases data parallelism is carried out in host code else high amount of data parallelism is carried out in device code. The host code, as an ordinary CPU process, is written in C language and is compiled with the help of standard C compiler. The device code is written using CUDA keywords for data-parallel functions, called kernels, and their associated data structures. GPU device is split into grids, blocks and threads in a hierarchical structure by the CUDA Architecture. The parallelism achieved using such a hierarchical architecture is very huge since there a multiple threads in a block, multiple blocks in a grid and multiple grids in a single GPU.A grid is a group of many threads which run the same kernel, same code. Each call to device code is made through a single grid. GPUs use many grids for maximum efficiency, hence it is not possible to share grids between different GPUs on a multi-GPU system. Grids are composed of multiple blocks. Each block contains a number of threads and a certain amount of shared memory. Blocks too cannot be shared between multiprocessors. To uniquely identify the current block a built in variable "blockIdx" is used. Blocks, themselves composed of many threads run on the individual cores of the multiprocessors, but unlike grids and blocks, they are not restricted to a single core.

Like blocks, each thread can be uniquely identified by the built in variable "threadIdx". Thread Ids can be 1D, 2D or 3D based on the block dimensions. So can be the block Ids. The thread ID is relative to the block that contains the thread. Each block Id is unique in a grid whereas each thread Id is unique only inside a block.



Scheduling Algorithms

The process manager handles the process scheduling where it removes the running or terminated process from the CPU and selects another process for execution based on the applied algorithm. Using such multiprogramming operating systems, more than one process can be loaded into the executable memory at the same time. These processes which are loaded shares the CPU among them. Factors associated with any scheduling algorithm are:

Turnaround time: time elapsed between the time the job enters and the time that it terminates, including the delay of waiting for the scheduler to start or resume the process.

- *Turnaround Time=Waiting Time + ServiceTime*
- *Start time:* time when the task which is scheduled to run actually starts to run on the CPU. CPU bursts starts with this.
- *Response time:* delay between a task being ready to run and when it actually starts running.
- *Throughput:* number of processes that are completed within a particular amount of time. Throughput can be used to compare which scheduling algorithm processes more number of processes.

Types of scheduling algorithms:

- ***First-come First-served Scheduling (FCFS):***

  The criteria of this algorithm are the process that requests first are holds the CPU first or which process enter the ready queue first is served first. Once a process has been submitted to the CPU, it runs into completion without being interrupted.

- ***Shortest Job Next Scheduling (SJN):***

  Shortest job next (SJN), also known as Shortest Job First (SJF) or Shortest Process Next (SPN), is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm. Shortest

remaining time is a preemptive variant of SJN.

- ***Priority Based Scheduling (PBS):***

  The processes are executed according to priority, such that the process having highest priority will execute first. On the basis of execution of each process, the waiting time and turnaround time is calculated but in the case of similar priority FCFS i.e. first Come First Serve is used.

- ***Round-Robin Scheduling (RR):***

  It is basically designed for time sharing systems. Round Robin Scheduling also called as time-slicing scheduling is a preemptive version of FCFS algorithm with a clock interrupt generated at periodic intervals. When the interrupt occurs, the process currently running is placed in the ready queue and the next ready process is selected on FCFS basis. Since each process is given a slice of time to execute before being preempted, this operation is called as time-slicing.

- **Gang scheduling:**

  In gang scheduling time sharing is used among gangs and schedules the related processes to run concurrently on different processors [. This is used for Inter process communication so that the two processes communicate with each other at the same time. When the gang scheduling is not used over a group of processes then the overhead of context switch occurs

In priority Scheduling algorithm, a priority is assigned to each process, and priority is allowed to run. Each process have their different priorities, according to the priority the processes are executed. Higher priority process will executed first and then the lower priority and so on. Generally, we use the lowest integer as highest priority. If the two or more process has equal priorities then they executed in FCFS manner such that the process which enters first will executed first. The decision of priority can be internally or externally. Internal priority uses some factors these are available to calculate the priority of a process. External priorities are set by criteria outside the operating system.

IV. IMPLEMENTATION

The main objective of the report is to analyse the performance of Priority Based Scheduling Algorithm. The waiting time and burst time of the processes that are under same set of conditions is the foremost criterion to evaluate CPU scheduling. The report has implemented the PBS algorithm on single threaded CPU environment and calculated the execution time of each algorithm, then, the same algorithm is implemented with NVIDIA's GPU programming environment, CUDA v6.0. Thereafter comparison of execution time of the algorithm on both platform is performed and calculation of the speed up achieved in execution of the algorithm on GPU over CPU is done.

The basics of CUDA code implementation are:

a) Allocation of memory on the CPU.
b) Allocation of the same amount of memory on GPU using library function "CudaMalloc".
c) Input the data in memory allocated in CPU.
d) Copying data from CPU to GPU memory using another library function "CudaMemCpy" having parameter (CudaMemcpyHostToDevice) to give the direction of the copy.
e) Processing is performed in GPU memory using kernel calls. Kernel calls are used to transfer the control from CPU to GPU and also to specify the number of grids, blocks and threads required for the program. Hence defining parallelism.
f) Copying the final data from GPU to CPU memory using library function "CudaMemCpy" having parameter (CudaMemcpyHostToDevice).
g) Release the GPU memory or other threads using the library function "CudaFree".

In the kernel calls the programmer determines the parallelism required by the program. The division of given data into appropriate number of threads majorly defines a successful code.

The algorithm implemented, is ***Non-Preemptive Priority Scheduling***. A large number of processes are taken which includes parameters such as burst time, arrival time and priority. The processes are sorted based on the priority first and then the tie is broken between the processes having same priority depending upon their burst time. The scheduling is performed placing the processes with the highest priority and lowest burst time first. (The highest priority corresponds to lowest number)

Sorting can be done using any of the sorting algorithms. But for a large number of processes and having the advantage of sorting it parallely, we use **Bitonic Sort**. The parallel execution of sorting is 10 times faster than the serial execution.

```
Time taken of scheduling 8192 process using CPU: nearly 0.06sec


Time taken of scheduling 8192 process using GPU: nearly 0.00sec
```

V. CONCLUSION

This report introduces a GPU implementation of the Priority Based Scheduling algorithm. The algorithm was designed for the NVIDIA CUDA platform to work in parallel with many threads in execution. It implements the operations of calculating the waiting time and turnaround time on the GPU. With the help of the GPU the execution time of the algorithm was found to be faster on the CPU using C code. This is a significant enhancement in the performance of the algorithm. A further performance increase in GPU is expected with better optimization and more advanced GPUs.

Thus, the scheduling problem of maximizing the number of jobs scheduled in time if n jobs with different processing times and due-dates are given can be solved on n^2 processors with time and space complexities as $0(\log^2 n)$ and $0(n^2)$. In contrast to the sequential complexity of 0(n log n) , which nevertheless involves a thorough analysis of the problem, being considered as a first step towards an optimal solution.

VI. REFERENCES

[1] Types of Scheduling Algorithms in Parallel Computinghttps://www.irjet.net/archives/V3/i5/IRJET-V3I5110.pdf

[2] Analysis of Priority Scheduling Algorithm on the Basis of FCFS & SJF for Similar Priority Jobs http://ijcsmc.com/docs/papers/September2015/V4I9201563.pdf

[3] Efficient Round Robin CPU Scheduling Algorithm http://www.ijerd.com/paper/vol4-issue9/F04093642.pdf

[4] Task Scheduling Algorithms Introduction www.waprogramming.com/download.php?download=50af8515a31a51.07465346.pdf

[5] Parallel Implementation of Scheduling Algorithms on GPU using CUDA https://pdfs.semanticscholar.org/94fe/f215c03b0b8918514b400ca5728590bcb150.pdf

[6] A Parallel Scheduling Algorithm For Minimizing The Number Of Unscheduled Jobs https://page.mi.fu-berlin.de/rote/Papers/pdf/A+parallel+scheduling+algorithm+for+minimizing+the+number+of+unscheduled+jobs.pdf

[7] Task Scheduling in Parallel Processing: Analysis https://www.omicsonline.org/open-access/task-scheduling-in-parallel-processing-analysis-2155-6180-1000257.pdf