Concordia University
Gina Cody School of Engineering and Computer Science
Parallel Programming (COMP 428)
Name: Nellybett Irahola
ID: 40079991

Assignment 1

Programming Questions (50 marks):

In the programming part of this assignment, you will be solving an "embarrassingly" parallel problem, with minimal communication overhead and minimal I/O requirement. The main objectives are to make yourself familiar with MPI programming on the cluster, and also make you familiar with performance measurement.

Q.1. *PI calculation using the Master-Worker paradigm*: The value of π(PI) can be calculated in different ways. An approximate algorithm for calculating PI and its parallel version are provided towards the end of the tutorial that we discussed in the first class. Here is a link to the tutorial:

https://computing.llnl.gov/tutorials/parallel_comp/

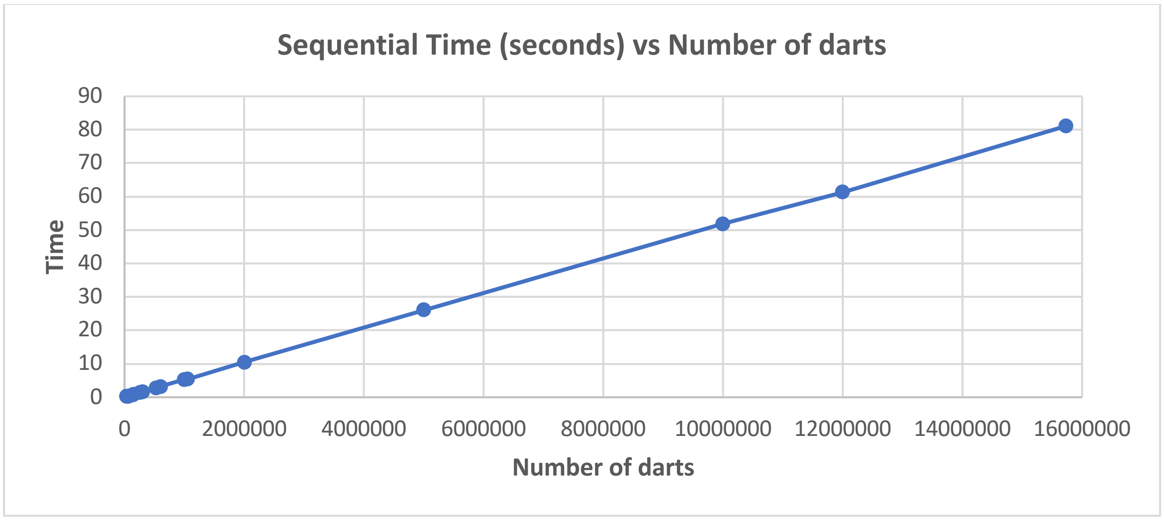
The pseudo-code and C code that uses MPI are also provided in the tutorial. Here is what you are required to do:

a) Write a sequential version of the algorithm and execute it on a single node of the cluster. Measure the execution time. (Note: For fairness of performance comparison, a sequential and parallel version must do equal amount of "total computational work", ignoring any other overheads.)

The number of tasks in the sequential algorithm was counted to simulate an equal amount of total computational work (this sequential solution was developed based on the parallel solution provided to guarantee equal amount of computational work). The sequential algorithm is presented in the sequential_pi.c file specifying all the parameters. It was run using `mpirun -np 1 -pernode sequential_pi.o`. It is a sequential implementation and if the mpi calls are eliminated can be compiled using `gcc -o sequential_pi.o sequential_pi.c`; however, it is compiled using `mpicc -o sequential_pi.o sequential_pi.c` (to illustrate that the program was run in one node) and the default value for the darts is 15728640.

Number of darts	Rounds	Sequential Time (seconds)
32768	100	0.17
40000	100	0.21
60000	100	0.31
65536	100	0.34

131072	100	0.69
150000	100	0.79
262144	100	1.35
300000	100	1.58
524288	100	2.71
600000	100	3.17
1000000	100	5.28
1048576	100	5.4
2000000	100	10.49
5000000	100	26.1
10000000	100	51.81
12000000	100	61.33
15728640	100	81.09

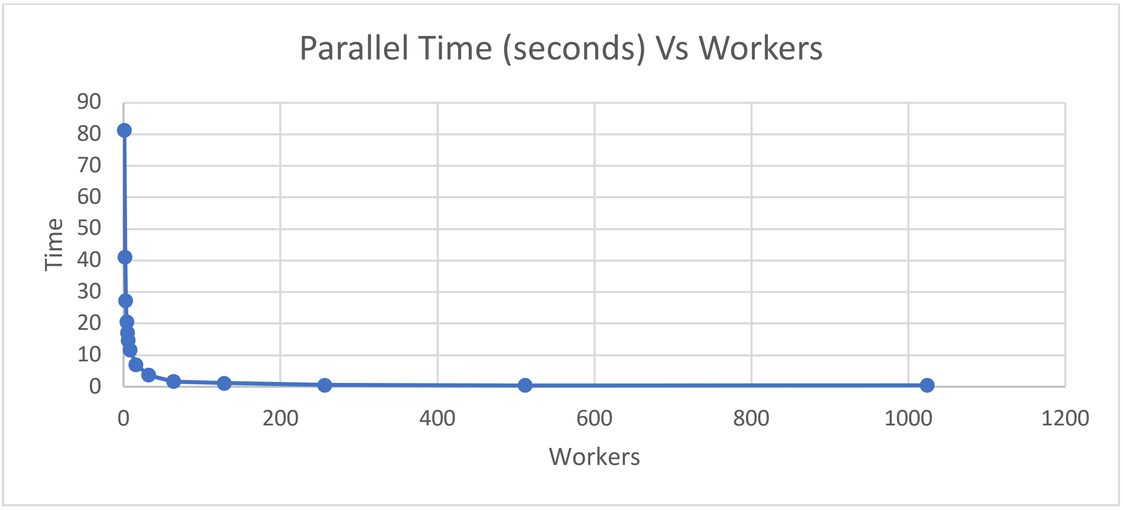


b) Execute the parallel version of the given program with varying number of workers (e.g. 2, 4, 6, etc) and measure the parallel execution time in each case. Ideally, workers should be mapped to distinct nodes of the cluster.

The parallel algorithm provided by the tutorial is presented in the parallel_pi.c. To guarantee that the tasks are mapped to distinct nodes of the cluster the command `mpirun –np 5 –pernode parallel_pi.o` was ran for the execution with less than 6 processes (since the cluster only has 5 nodes).The darts and round were a constant and the task were divided between the workers.

Number of darts	Rounds	Workers	Parallel Time (seconds)
15728640	100	1	81.4
		2	41.03
		3	27.3

		4	20.56
		5	17.15
		6	14.66
		8	11.7
		16	7.02
		32	3.7
		64	1.61
		128	1.16
		256	0.49
		512	0.46
		1024	0.44



c) Referring to b), identify the tasks that are mapped to processes (Note: in this case, the master and each worker is a process). What determines the granularity of a task here? Referring to the above tutorial, which of the following categories best suit the given parallel program (chose all that apply): data-parallel, hybrid, SPMD, MPMD? Explain.

The tasks are the defined by the number of darts, a higher number of darts will give a better approximation of pi. However, the number of darts is fixed. It was divided equally between the multiple processes. The granularity of the task could be as fine grained as one dart or as coarse grained as half of the darts. The granularity of the task was determined by the number of processes since the darts were divided between them (a parallelization with two processes will process two task each one with half of the darts).

According to the tutorial two categories are applicable to the given parallel program:

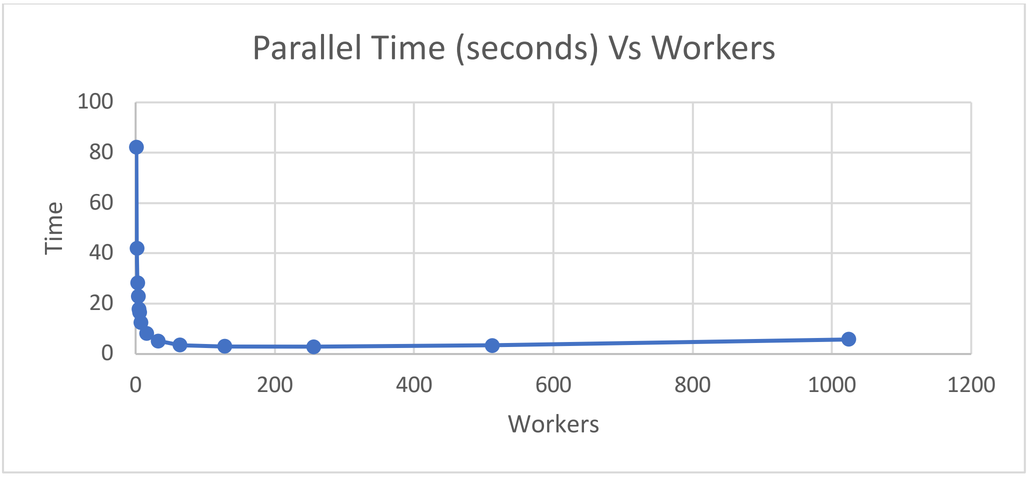
- SPMD is actually a "high level" programming model where all tasks execute their copy of the same program simultaneously and may use different data. This program can be threads, message passing, data parallel or hybrid (the most common implementations are using message passing and hybrid). It is applicable to this parallel program since all the programs run the same code by preserving their own data (passing their result to the master task).

- Message Passing Model that is based in a set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines. Tasks exchange data through communications by sending and receiving messages. Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation. A clear implementation of this model is MPI.

In my opinion a data-parallel model and a hybrid model are not applicable to this program since for the data parallel the address space is treated globally and the parallel work focuses on performing operations in a data set typically organized in a common structure (we divide the tasks between processes but don't manage them in a single structure) and the hybrid requires the combination of models.

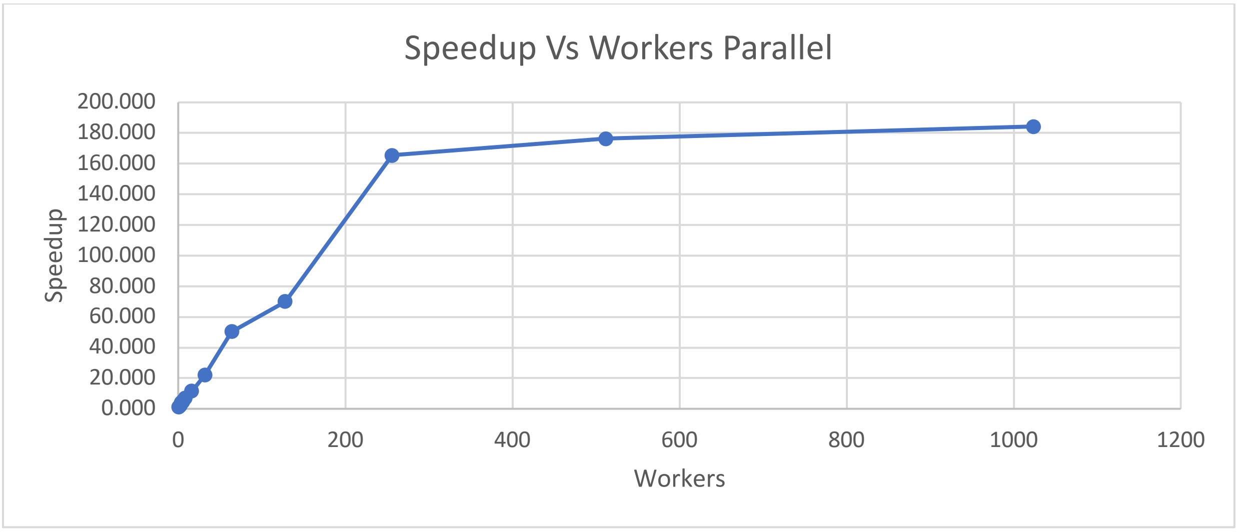
d) In the given program b), the master and the worker processes are "spawned" simultaneously. Another way to implement the program is to first create the Master process, which subsequently spawns the workers through explicitly calling *MPI_Comm_Spawn*. Modify the given program accordingly to use this dynamic spawning mechanism available in the current versions of MPI (MPI 2.0 onwards). Repeat the same experiments as in (b) above.

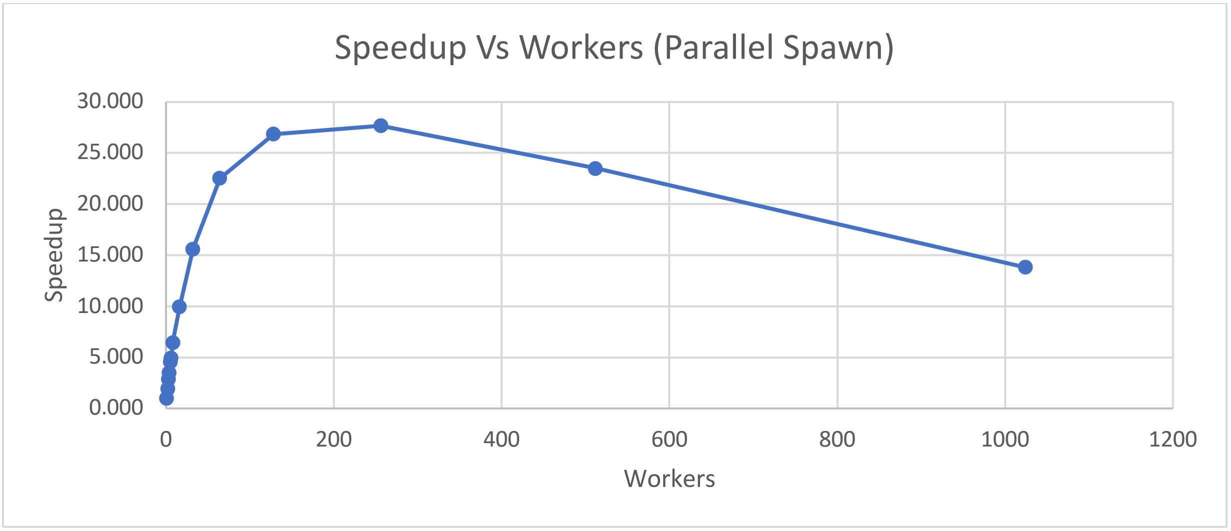
Number of darts	Rounds	Workers	Parallel Time spawn (seconds)
15728640	100	1	82.08
		2	42.08
		3	28.26
		4	22.94
		5	17.76
		6	16.47
		8	12.63
		16	8.16
		32	5.19
		64	3.6
		128	3.02
		256	2.93
		512	3.45
		1024	5.87



e) Plot a *speedup* versus *number of workers* curve based on your experiments in (a) and (d) above. Explain any unusual behavior, e.g., slow down, sub-linear speedup, etc.

Number of darts	Rounds	Workers	Parallel Time (seconds)	Parallel Time spawn (seconds)	Sequential Time	Parallel Speedup	Parallel Spawn Speedup
15728640	100	1	81.4	82.08	81.09	0.996	0.988
		2	41.03	42.08		1.976	1.927
		3	27.3	28.26		2.970	2.869
		4	20.56	22.94		3.944	3.535
		5	17.15	17.76		4.728	4.566
		6	14.66	16.47		5.531	4.923
		8	11.7	12.63		6.931	6.420
		16	7.02	8.16		11.551	9.938
		32	3.7	5.19		21.916	15.624
		64	1.61	3.6		50.366	22.525
		128	1.16	3.02		69.905	26.851
		256	0.49	2.93		165.490	27.676
		512	0.46	3.45		176.283	23.504
		1024	0.44	5.87		184.295	13.814





A sub-linear speedup is observed for both of the graph (speedup less than P). These results can be explained by Amdahl's law (since the problem size is fixed), this law stablishes a maximum for speedup. It says that the speedup is always limited by the sequential part of an algorithm that cannot be parallelized (data setup, reading/writing, etc.) and not the number of processors. Even with infinite number of processors, maximum speedup limited to 1/f where f represents the sequential part of the algorithm. This explains why the speedup decreases after the certain number of workers.

These results can also be explained by what is known as parallel slowdown (parallelization beyond a certain point causes the program to run slower) due to a communications bottleneck (communications overhead created by adding another processing node surpasses the increased processing power that node provides).

Another fact that is important to mention is the difference in execution time between the spawn parallel version and the normal parallel version. The time differences can be explained by the fact that the generation of workers is considered as part of the time in the spawned version.

<u>Written Questions (50 marks):</u>

Q.2. [10 marks] Referring to the programming question 1 above, both solutions (b) and (d) are based on static decompositions of the problem where number of tasks is known a priori. There are also ways to dynamically solve the problem where the workload of a worker is only fixed dynamically (i.e. at run time). Dynamic solutions are sometimes better because they can facilitate better load balancing. Provide the pseudo-code for such a dynamic solution to the previous problem of PI calculation. <u>Hint:</u> You can think of an algorithm that keeps on computing □ _until it converges. It can be based on the "pool of tasks" paradigm, where the Master process holds a pool of tasks for workers to perform. The master repeatedly does the following: sends a worker a task when requested, collects results from workers, generate more tasks if needed, repeat the previous steps until no more tasks. A worker repeatedly does the following: requests task from Master, performs task, sends results back to Master, and repeats the previous steps until no more task.

```
p = list of tasks

find out if I am MASTER or WORKER
while pi not converge
  if I am MASTER
    p= generate tasks
    while(p!=null)
      if task requested
          send a task to WORKERS
          collect WORKER calculations
          calculate result from collected data
          generate more tasks
  else if I am WORKER
      request task from MASTER
      perform task given by MASTER
      send result to MASTER
  endif
```

A generated task will do the next calculations:

```
npoints = 10000
circle_count = 0

num = npoints

 do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
 end do
```

Q.3. [15 marks] The task graph shown in the following figure represents an image-processing application. Each bubble represents an inherently sequential task. There are altogether 17 tasks, out of which there is 1 input task (stage 0), 1 output task (stage 3) and 15 computational tasks (stages 1 and 2). When executed on the same processor, each of the input and output tasks takes 1 unit of time; each of the computational tasks in stage 1 takes 2 units of time, and each of the

computational tasks in stage 2 takes 5 units of time. The arrows show the control dependency relations among tasks, i.e., a task can start if and only if all its predecessors have completed.

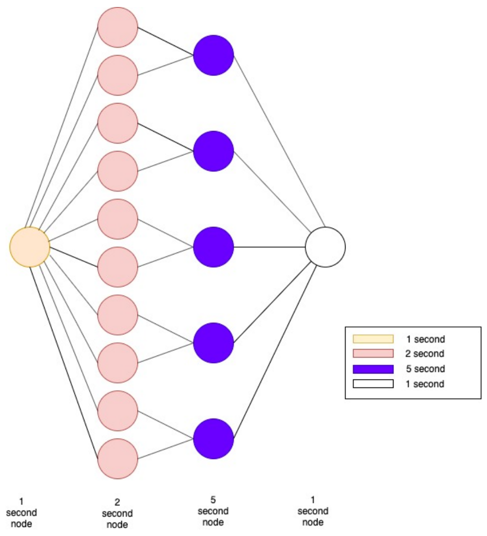
a) Assuming all identical processors, what is the minimum number of processors required to obtain the lowest execution time of the above task graph? What is the corresponding efficiency of the parallel system?

The maximum number of tasks that can be executed simultaneously in a parallel program at any given time is known as its maximum degree of concurrency. In the particular exercise the maximum number of tasks is in stage 2 (10 tasks). The critical path length in the task dependency graph represents a sequence of tasks that must be processed one after the other. The longest such path determines the shortest time in which the program can be executed in parallel. In the particular exercise the critical path length is 9.

To accomplish this time it is necessary to have 10 processors, since the tasks in stage 2 cannot be executed if those in stage 1 have not finished (because of the dependency of tasks) and the only way of ending all the tasks in stage 2 at the same time assuming equal performance between the processors is if the number of tasks is equal to the number of processors (maximum degree of concurrency).

Knowing that:

*Time of sequential execution= 1 + 10*2 +5*5+1 = 47*
Time of parallel execution= critical path = 9
Number of processors = 10



$$E = \frac{Time\ of\ Sequential\ Execution}{Time\ of\ Parallel\ Execution * Number\ of\ processors}$$

$$E = \frac{47}{9 * 10} = 0.52$$

b) Which one(s) of the following is (are) guaranteed to increase the efficiency of the above parallel system: (i) increasing the number of processors, (ii) decreasing the number of processors, (iii) changing to faster processors? Explain your answer.

(i) Increasing the number of processors will not increase the efficiency of the system since the maximum level of parallelization have been reach. It means that any additional processor will be idle and create communication overhead decreasing efficiency.

$E = \frac{47}{9 * 11} = 0.47$

(ii) Decreasing the number of processors will not necessarily guarantee an increase in the efficiency of the above parallel system, since the tasks of stage 3 are depend on the tasks of stage 2 and a difference between the number of processors and the maximum degree of concurrency will generate additional execution time that is not necessarily compensated by the decrease in the number of processors. However, there are several cases that increase the parallel execution time but increase the efficiency since the number of processors compensate the time addition.

9 processors:

$E = \frac{47}{11 * 9} = 0.47$

4 processors:

$$E = \frac{47}{16 \ * \ 4} = 0.734$$

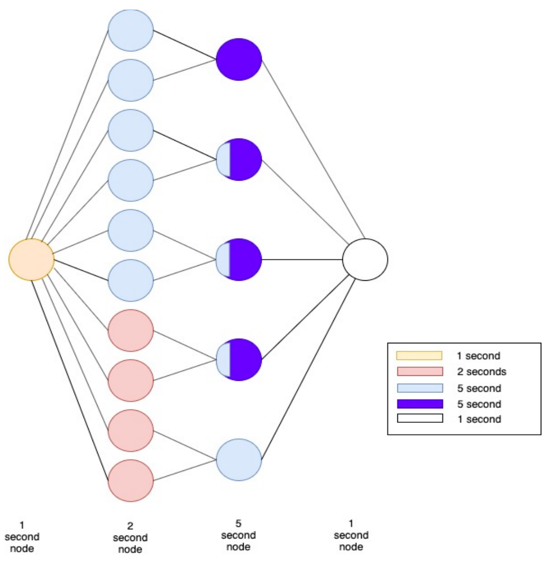
(iii) Changing to faster processors can generate greater efficiency since the required number of processors can be decreased without affecting the execution time. Assuming that the new processors finish the tasks in half of the time only 5 processors will be necessary in stage 2 to maintain the same execution time.

$$E = \frac{47}{9 \ * \ 5} = 1.04$$

c) What is the maximum speedup of the above parallel system when it is solved using 4 identical processors? Show your calculations.

Since the maximum degree of concurrency is 10 and the problem is solved using only 4 processors:

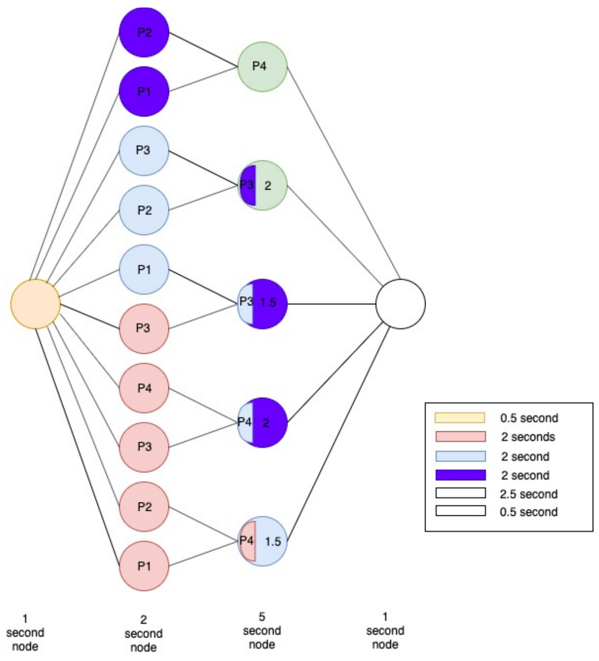
*Time of sequential execution= 1 + 10*2 +5*5+1 = 47*
$Parallel\ execution\ time = 1 + 2 + 5 + \ 5 + 1 = 14$



$$Speedup = \frac{Time\ of\ sequential\ execution}{Parallel\ execution\ time} = 3.36$$

d) What is the maximum speedup of the above parallel system when it is solved using 4 processors of varying speeds as follows: processors 1 and 2 have identical speed; processors 3 and 4 have identical speed; processor 1 (processor 2) is twice as fast as processor 3 (processor 4). Assume that, for speedup calculation, sequential time is computed on the fastest processor(s).

*Time of sequential execution= (1/2) + (10*2/2) +(5*5/2)+(1/2) = 23.5*
Time of parallel execution= =9.5



$$Speedup = \frac{Time\ of\ sequential\ execution}{Parallel\ execution\ time} = 2.47$$

The execution time is limited by the slowest processors and the dependency between the tasks.

Q.4. [15 marks] Consider a perfectly balanced quick-sort tree (i.e. the pivot chosen is always at the middle). Recall that quick-sort employs the divide-and-conquer strategy. Answer the following questions:

a) What is the maximum possible speed-up when a perfectly balanced quick-sort tree that sorts N numbers is naively parallelized using an unlimited number of identical processors? Show all your calculations.

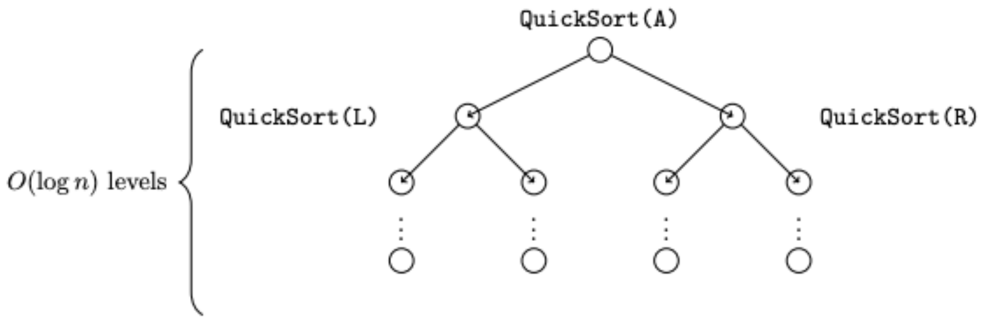
For the sequential execution of the algorithm if the elements are split into two roughly equal-size subsequences of and elements, the runtime is given by T(n) = 2T(n/2) + Q(n), whose solution is T(n) = Q(n log n).

The naïve parallelization involves recursive decomposition. During the algorithm the elements are partitioned into two parts and each part is solved recursively (it uses the divide-and-conquer strategy). Sorting the smaller parts represents two independent subproblems that can be solved in parallel.

The algorithm is usually parallelized by executing the partitioning in a single process and then when it performs its recursive calls assign one of the subproblems to another process (each of these processes sorts its array by using quicksort and assigns one of its subproblems to other processes). The algorithm terminates when the arrays cannot be further partitioned. Upon termination, each process holds an element of the array, and the sorted order can be recovered by traversing the processes.

If the partitioning step is performed in time Q(1), using Q(n) processes, it is possible to obtain an overall parallel run time of Q(log n), which leads to a cost optimal formulation (O(1) partition O(logn) levels). However, the parallelization of the partition is hard to achieve (the communication overhead) there are only certain methods that can be used.

Sequential execution time= O(nlogn)
Parallel execution time (cost optimal) = O(log n)



$$Speedup = \frac{Time\ of\ sequential\ execution}{Parallel\ execution\ time\ (optimal)} = O(n)$$

b) What is the efficiency in a) if the quick-sort tree is parallelized using N identical processors (N is also the input size)? Show your calculations

$$E = \frac{Time\ of\ Sequential\ Execution}{Time\ of\ Parallel\ Execution\ (optimal) * Number\ of\ processors} = \frac{O(nlogn)}{O(logn)*n} = O(1)$$

c) Referring to b), is the parallel system cost optimal? Explain.

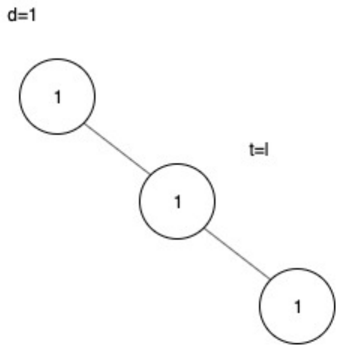
It is possible that the system is cost optimal if the partitioning step is done in O(1) using n processes and not in a sequence (since one processor will have to transverse all the elements). It is cost optimal since the efficiency is O(1) which means that the cost of solving a problem on a parallel computer is asymptotically identical to serial cost. In the sequential case the system

Q.5. [10 marks] Let *d* be the maximum degree of concurrency in a task dependency graph with *t* tasks and critical path length *l*. Prove that $ceil\left(\frac{t}{l}\right) \leq d \leq t - l + 1$. Assume that each task has a weight of 1 unit.

There are two cases the first when d=1 which means that the graph only has one node in each level and when d!=1.

1) If d=1 and t= l

t=3 l=3
t=4 l=4
t=5 l=5 …



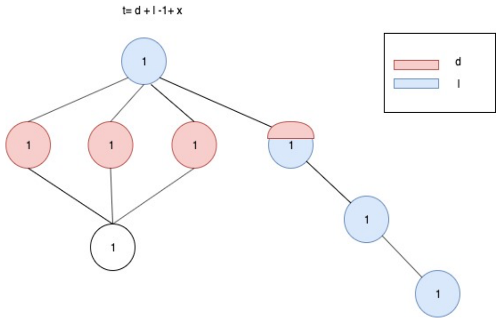
By definition if d=1 then t= l since the number of elements is equal to the largest path to an end node.

t/ l =1
t- l =0
1$\leq 1 \leq 1$ *is True*

2) If d!=1

l=4, d=4, x=1
t= (d+l -1)+x

-1 represents the intersection between l and one of the nodes of d, x could be 0 but in this case is 1, 0 is the base scenario since a bigger t just leads to a more equal distribution in t/l and a higher value in t-l+1). l is the number of levels and when divides t distributes all the nodes equally between the levels, the level with more nodes represents d.



By definition if d!=1 we have to cases, since t contains all the nodes (tasks) of the graph if it was smaller than d+l-1 it would be a contradiction because if a task is in the critical path (it is in levels of the graph vertically) it is included in t, and the degree of concurrency represent tasks in the horizontal level which means that this tasks are also include in t.

2.1) t= d+ l -1 with $l > 0$

Replacing t in both sides of the inequality:

2.1.1) (d+ l -1)/ l = ((d-1)/ l) + 1

If l=1 and d>1
$d \leq d$ *is true*
If l > 1 and d >1
$((d-1)/l)$ + 1 $\leq d$ *is true*

2.1.2) d+l-1-l+1=d
$d \leq d$ *is true*

2.2) If t > d+ l -1 then the previous prove is also valid for this case.

Submit all your answers, including well documented source code for the parallel program, in pdf and/or text formats only. All files should be archived into a single file (e.g., a single .zip file), and submitted through EAS.