DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING
CONCORDIA UNIVERSITY
COMP 428/6281: Parallel Programming
Fall 2014
ASSIGNMENT 1
Due date: Sunday, September 28th before midnight
Total marks: 100

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:

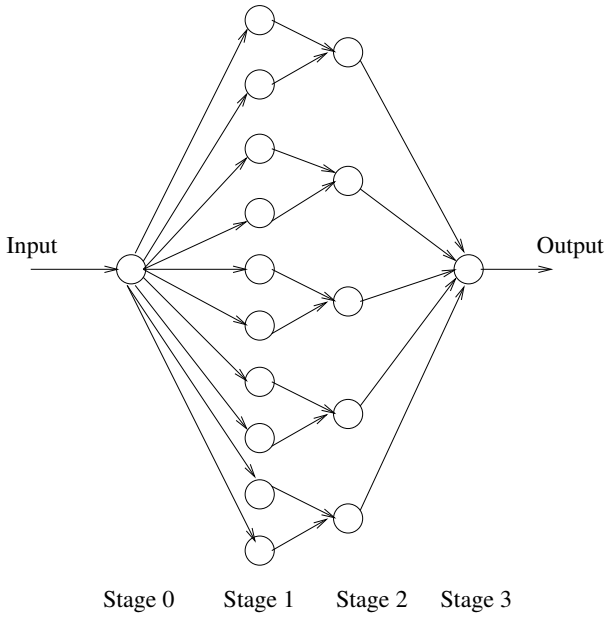
a) Write a sequential version of the algorithm and execute it on a single node of the cluster. Measure the execution time. (Note: For fairness of performance comparison, a sequential and parallel version must do equal amount of "total computational work", ignoring any other overheads.)
b) Execute the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

Written Questions (50 marks):

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. Hint: You can think of an algorithm that keeps on computing π 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 do. It 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.

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

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

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 $\lceil t/l \rceil \leq d \leq t - l + 1$. Assume that each task has a weight of 1 unit.

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.