EECS 587 Homework Problem

Due at start of class, 12 November 2015

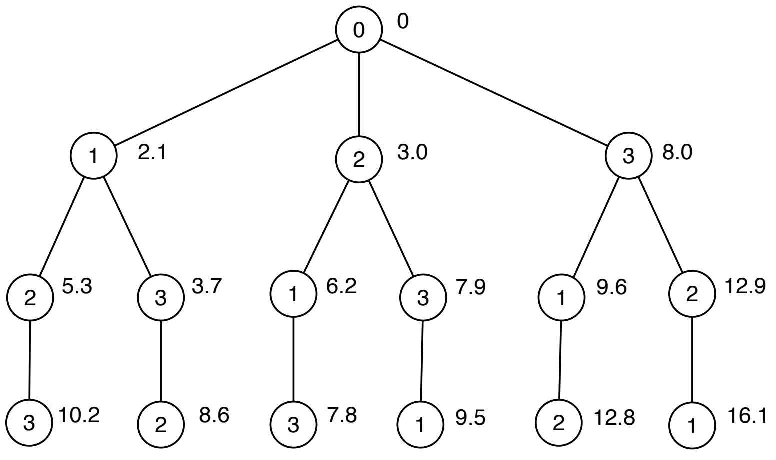
As for all homework, you must do this problem on your own. However, see the notes near the end.

For this assignment you will use OpenMP on the greenfield computer at the Pittsburgh Supercomputing Center to solve the Forgetful Student Problem (FSP) using branch-and-bound. For FSP there are a set of classrooms [0, …, $c-1$]. It's the first week of classes and the forgetful student wants to take the parallel computing class, but after arriving at classroom 0 and seeing that the class isn't there, she realizes she can't remember where the class is. She wants a path to visit all of the other classrooms until she finds the right one. She wants to optimize the worst-case time to visit all of them, and she will never revisit a classroom.

She decides to use branch-and-bound to find an optimal path, that is, she will consider all possible paths, and the bound should help eliminate some. Let $t(p, q)$ denote the time to go from room p to room q. For the simple example below $c = 4$ and the times are symmetric, i.e., $t(p, q) = t(q, p)$. The times are:

		q			
		0	1	2	3
	0	0	2.1	3.0	8.0
p	1	2.1	0	3.2	1.6
	2	3.0	3.2	0	4.9
	3	8.0	1.6	4.9	0

All possible paths are represented in the following tree, where a path corresponds to starting at the root and going down to a leaf. The nodes represent the classrooms, and the number to their right is the time along the path so far.



The standard serial approach for this is to use a depth-first search on the tree, keeping track of the best path found so far and its total time (which will be the bound). Initially `bound` $= \infty$. The search starts at 0 and tries the first (leftmost) option (1), then the first from it (i.e., the next one that isn't already on the path). This is 2, and then the first option from it is 3, finishing the path. The total time is 10.2, so $\texttt{bound} = \min\{\texttt{bound}, 10.2\} = 10.2$ Then the search backtracks one step, to 2. There are no more options at 2 so it backtracks another step, to 1, where it now tries the 2nd option, 3, and then goes to 2. This finishes another path and $\texttt{bound} = \min\{\texttt{bound}, 8.6\} = 8.6$ It backtracks to 3, then back to 2, then back to 0. The next option from 0 is 2. When the path 0, 2, 1, 3 is finished `bound` is set to 7.8. Backtracking from 3 to 1 and then to 2, the 2nd option at 2 is tried. It gives a time of 7.9, even though the path isn't completed, which is

$\geq$ `bound`. Continuing on this path cannot find a better one and therefore the search backtracks to 0, without finishing the path. Then the 3rd option (3) from 0 is tried. It already takes time longer than the bound, so the entire subtree is eliminated (pruned) and the search is over.

Depth-first search is inherently serial, so a different approach is required for parallel. Breadth-first is the natural candidate. That is, one looks at the partial paths in the order

0, 1   0, 2   0, 3   then
0, 1, 2   0, 1, 3   0, 2, 1   0, 2, 3   0, 3, 1   0, 3, 2   then
0, 1, 2, 3   0, 1, 3, 2   0, 2, 1, 3   0, 2, 3, 1   0, 3, 1, 2   0, 3, 2, 1

which is the top-down, left-to-right ordering. Here pruning using the bound isn't helpful because the bound doesn't change until the lowest level of nodes is reached. However, there is much potential parallelism here. For example, if there are 3 threads then once partial paths 0,1 0,2 0,3 are evaluated thread 0 can start working on the subtree rooted at 0,1 thread 1 can work on the subtree rooted at 0,2 and thread 2 can work on the subtree rooted at 0,3. These subtrees might be evaluated depth-first or breadth-first. Occasionally the threads should check the global variable `bound` to see if they can eliminate parts of their search.

If there are even more threads then perhaps 2 levels of the breadth-first search should be created before threads start working on subtrees. Here there isn't much work in the very small subtrees, but if, say, $c = 10$ and there are 20 threads then the first level has only 9 subtrees, but the second has 9*8=72. A thread may work on a subtree and when finished go back to the collection of subtrees to get another to work on.

However, if a very good bound has been found, then it may be that one thread is stuck with a very large subtree that where it can't eliminate very much of it, while other threads have run out of work to do. In this case the thread with a large subtree might put parts of it back in the collection of subtrees that haven't been finished, so that other threads can work on them.

This basic approach of keeping a collection of tasks that need to be done, and having threads get more tasks when they are done with the one they were working on, is the *manager-worker*, or *master-slave* paradigm. In a distributed memory context the manager might be one processor keeping track of all of the tasks that need to be done, with workers sending messages to request tasks. The manager may keep the bound, or perhaps workers occasionally use a broadcast to inform all of the others when they have found a better bound. There are many variations on distributed memory manager-worker implemenations.

In a shared memory system the collection of tasks is usually a global data structure and the bound is a global variable. Usually all of the threads are workers and all are managers in that they can add or remove tasks from the data structure. However, they need to be careful to not interrupt each other when updating the data structure.

In the FSP the tasks that need to be finished are subtrees where there are still paths that haven't been explored nor pruned. The global data structure may just keep the subtrees in an arbitrary order, it may have them in a stack, in a queue, in a priority queue, or some other special order. A stack corresponds to depth-first search, that is, the first thread starts going down its first path, and at each node it encounters it continues with its first option and puts the others in the collection of unfinished trees. As it goes down it adds the subtrees to the beginning of the structure, so that other threads work on larger problems first. These threads may themselves put subtrees back in the collection of unsolved ones, again putting them at the beginning.

Another option is that whenever a subtree is added to the collection of unfinished tasks it is put at the end. This is a queue, which corresponds to breadth-first search. Yet another option is that whenever a task is added to the collection it is given a priority, which is a guess about how important it is to evaluate the subtree quickly, and the tasks are maintained in order so the next subtree to be worked on is one with the highest priority. The priority might be the total time of the path so far, in which case partial paths taking a long time are worked on earlier, perhaps in the hope that they can be eliminated with only a few more steps. Another option is that the priority is the negative of the time so far, so faster paths are evaluated earlier, in the hope that they soon complete a path which lowers the bound. However, a priority queue is more complicated to maintain.

Returning to the FSP: the classrooms have 2-dimensional coordinates, and she can go from any room to any other room directly. However, there is so much construction on campus that the time to go from classroom p, at location (p_1, p_2), to classroom q, at (q_1, q_2), is $(|p_1 - q_1|^{1.5} + |p_2 - q_2|^{1.5})^{2/3}$. Let $t(p, q)$ denote this time. The total time of a path going from $r_0 = 0$ to r_1, r_2, …, r_{c-1} is $\sum_{i=1}^{c-1} t(r_{i-1}, r_i)$. Remember that she cannot go back to a class already visited.

Report: All timing should be done by thread 0. It calls the clock routine at the beginning, before any parallelism is started, then calls it after the calculations and all parallelism is finished. This determines the elapsed time. Remember to turn on optimization flags.

Whenever you submit a program, put a time limit of 5 minutes or less on it. Failure to follow this rule may result in a grade of F in the class. If you cannot run the program for, say, 1 core, within 5 minutes then just include that fact in the report and try using more cores.

Print out the elapsed time, the total time of the fastest path found, and the path in the order that the classrooms are visited. Do this for 1, 4, 16, and 32 cores. Use $c = 15$ classes, where class i, $i = 0, 14$, is at location $(100.0 * \sin(i), 101.0 * \cos(i^2))$

Your report should analyze how the time changed as a function of the number of cores, and explain why it has that behavior. You should explain how you parallelized the problem, including a discussion of whatever ordering of tasks was used by the manager. If you tried several parallelization variations then explain what you tried and what the effects were. Your report will be graded on the program's performance and your analysis of the performance if it is not perfect speedup. The more precise and insightful the analysis the better the grade. Poor English can reduce the grade.

Notes:

1. While timings must be done on the Pittsburgh computer, most compilers (such as gcc) include OpenMP and you can debug on any computer you want.
2. OpenMP has a `task` command which can be used to implement manager-worker. However, you are not allowed to use it.
3. There are various optimizations possible, but I want you to concentrate on the parallelism, not the serial optimizations. Therefore you cannot change the ordering of the classrooms.
4. Similarly, you cannot use lower bound heuristics to eliminate paths early. That is, if you had a function $f(u)$ which for a partial path u gave a lower bound on the time to complete the path, then you could prune the subtree whenever the time on u so far, plus $f(u)$, is $\geq$ `bound`. You cannot use this approach.
5. A simple speedup you are allowed use is to compute the distances and put them in a table before the search starts. This must be included in the elapsed time, even if it is done serially.
6. Since the emphasis in this class is on parallel, and since the problem is a bit more complicated than the first programming project, you *can* look on the web or anywhere else to find *serial* algorithms or code for the problem. You're more likely to find such code looking for the Traveling Salesman Problem (or Traveling Salesperson Problem) instead of FSP. I think students are more interesting than salesmen, but apparently others don't. However, you *cannot* look for, nor use, code for parallel versions.
7. Unfortunately your efforts to help her find an efficient path are not useful because after she has visited every room she remembers that the class started yesterday.