Parallel Programming

Peyman Shobeiri

ID: 40327586

Fall 2024

Table of Contents

Question 1

Consider a perfectly balanced quicksort 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.**

Quicksort is an algorithm that follows the divide-and-conquer paradigm. The first step for quicksort would be to select a pivot element "p". After that, an array A of size n would be partitioned into two subsequences: A_0, which consists of all elements smaller than the pivot (p); and A_1, consisting of all the elements greater or equal to the pivot (p). Next, each of the subsequences A_0 and A_1 is recursively sorted by the quicksort call. Each call to quicksort results in partitioning the sequences. This process continues until all subarrays are sorted, resulting in a fully sorted array.

```
4 function QuickSort(A, p, r){
5         if p < r then{
6             q ← Partition(A, p, r)
7             Quicksort(A, p, q – 1)
8             Quicksort(A, q + 1, r)
9 }}
```

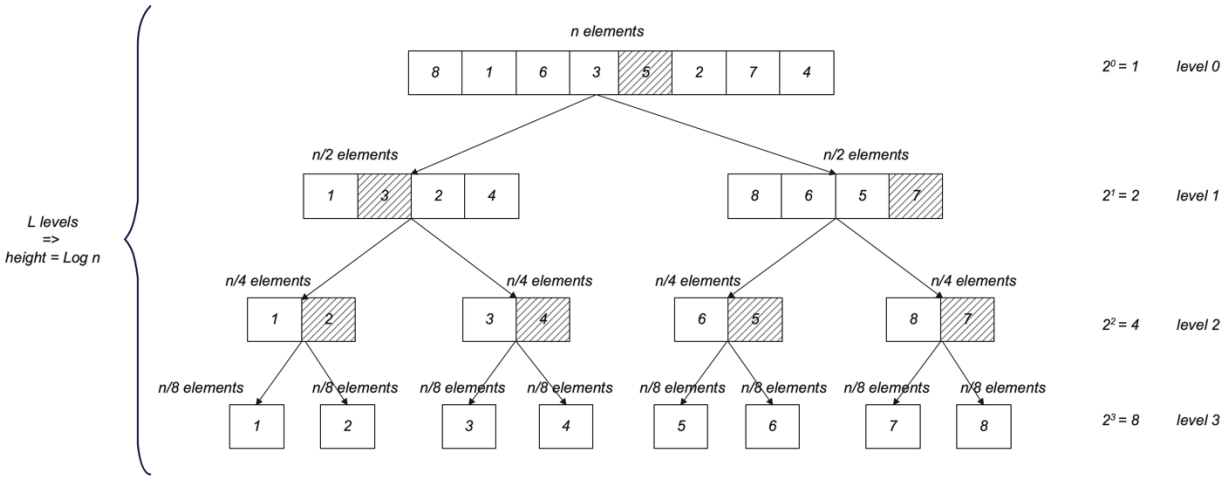
```
15 function Partition(A, p, r){
16          x = Random element (or in this example is the middle element)
17          i <- p–1
18          for j = p to r – 1 do {
19                if A[j] < x then {
20                      i←i+1
21                      A[i] ↔ A[j]
22                }
23          A[i + 1] ↔ A[r]
24         }
25 return i + 1
```

Here is a pseudocode of the Quicksort algorithm: In the code above, QUICKSORT (A, p, r) is a function call that recursively sorts the array. It starts by checking if there is more than one element to sort, then it calls itself recursively in a partitioning around the pivot by calling PARTITION (A, p, r). In PARTITION, the pivot can be any random element x, all elements smaller than the pivot are brought to the left of the pivot, and all elements larger than or equal to the pivot are brought to the right. The pivot is then placed in its correct sorted position. Quicksort then recursively sorts the subarrays to the left and right of the pivot. This process continues until all the sub-arrays are sorted, hence giving a complete sorted array.

In this problem, we want to solve the problem using parallelization with an unlimited number of processors. That means that the subsequence generated on each recursive call of Quicksort is assigned to a new processor. Therefore, each processor is independent of other processors. This is called Recursive Decomposition, whereby a problem is broken down into a set of independent

sub-problems, where each independent subproblem is then subdivided into smaller ones and the results are combined in the final step to form the final result [1].

It can be observed that for this sequential version, the algorithm splits up the problem into two pieces which are each further subdivided into four, and then subdivided into eight, and so on. If the original array contains (n) elements, the subproblem at the next level will be dealing with two arrays of size (n/2), then four arrays of size (n/4), and so on. In the example below you can see that we have 8 elements in our array. In the first call to Quicksort, the array is split into two subarrays, each of size 4. Again, the two subarrays are further divided into two small subarrays each of size 2, and so on. The number of subarrays at each level of the recursive tree doubles, which can be represented by 2^L, where L is the level of the tree. So, in order to find the height of our recursive tree, we need to calculate log n, where n is the number of elements in our array. In the example, you can see that the height of the tree is 3 which is equal to the $\log_2 8$.



In each step of the Quicksort algorithm, the array is divided into two subarrays, and after sorting, the final result is obtained by combining these subarrays. The time complexity can be calculated using the following:

$$T(n) = 2T(n/2) + f(n)$$

Simplifie as :

$$T(n) = 2T(n/2) + n$$

We can now expand this relation step by step:

$$T(n) = 2(2T(n/4) + n/2) + n = 4T(n/4) + n + n = 4T(n/4) + 2n$$
$$T(n) = 4(2T(n/8) + n/4) + 2n = 8T(n/8) + n + 2n = 8T(n/8) + 3n$$
$$T(n) = 8(2T(n/16) + n/8) + 3n = 16T(n/16) + 4n$$
…

Continuing this pattern, we observe that the final relation becomes:

$$T(n) = n\,T(1) + n\log(n)$$

Since we can ignore the constant T (1) the overall time complexity of the serial Quicksort algorithm is:

$$T(n) = O(n \log n)$$

In the parallel version of this algorithm, even though there are an unlimited number of processors, the partitioning of the array and selection of the pivot still occur sequentially. In the first level, partitioning the n elements takes n time, in the second level it takes n/2 time, and in the third level it takes n/4 time, and so on.

Thus, the total time is calculated as:

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \ldots + 1 = 2n + 1$$

This is a geometric series that sums to approximately 2n - 1, so the time complexity of the parallel algorithm becomes:

$$T(n) = O(2n + 1) \rightarrow O(n)$$

Thus, while the serial algorithm has a time complexity of O (n log n), the parallel algorithm reduces this to O(n), due to parallel processing, but is still constrained by the sequential partitioning step.

$$Speedup = \frac{\text{time required for sequential version}}{\text{time required for parallel version}}$$

$$= \frac{O(n \log n)}{O(n)} = O(\log n)$$

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**

The efficiency of the quick-sort tree can be determined as follows if N identical processors are used for parallelization:

$$efficiency = \frac{\text{speedup}}{\text{number of processors}} = O\left(\frac{\log n}{n}\right)$$

So, the efficiency if we use N identical processors is $O\left(\frac{\log n}{n}\right)$.

c) **Referring to b), is the parallel system cost optimal? Explain your answer. If your answer is "No", explain intuitively what can be done to increase cost optimality of the system.**

Now we have a naive parallelization of Quicksort: it parallelizes the recursive subproblems but keeps the partitioning step serial. It has a total work cost of $O(n^2)$. The reason this is such a high cost, basically, is that while recursion for sorting may be parallelized, the partitioning step acts like a bottleneck to achieve optimal efficiency. This is because the parallel time complexity is O(n) since the n elements are partitioned at each level of recursion. The fact that the algorithm distributes the sorting tasks over various processors does not outweigh the serial partitioning process, which in this case is dominant in this system. Therefore, the cost-time product is $O(n^2)$, meaning this algorithm is not cost-optimal.

However, if the partitioning step could be parallelized which would drop its time complexity to O (1) then the entire algorithm would scale better. Since Quicksort's recursive tree is log n levels deep, a parallel partitioning would drop its overall parallel time complexity to O (log n), at the total work cost of O(n) processors. At this point, it would be a cost-optimal algorithm, as the product of parallel cost and time would equal the sequential work of O (n log n). This parallel algorithm will only remain cost-optimal if it can make use of the full extent of available processors. The most important factor in such algorithms is maintaining balanced divisions of subproblems within the recursion tree. Therefore, full parallelization for the sorting and partitioning stages will ensure that we obtain an optimal parallel system with a time complexity of O (n log n).

Question 2

A task can be divided into *m* subtasks which can be processed in sequence. This is known as a computational pipeline. One way to speed-up the executions of multiple identical tasks is to implement an *m*-stage pipeline using *m* identical processors. Pipeline stage *i*, $1 \leq i \leq m$, executes subtask *i*, passes the result to the next stage (except the last stage), and then executes the subtask *i* of the next task. Given that each pipeline stage takes 1 unit of time to execute a subtask, what is the maximum possible speedup in executing *n* identical tasks? Show your calculations.

Given a pipeline of m stages and having n identical tasks, we need to find the maximum speedup. This is defined as the ratio between time spent by a sequential execution, versus time taken for its parallel pipeline execution.

During sequential execution, each task will be composed of m number of subtasks and all these shall be executed back-to-back. The total time taken by sequential execution is just the time to execute all subtasks for all tasks, which is:

$$T_{sequential} = n * m$$

this means for n tasks, each having m subtasks, sequential execution takes n * m units.

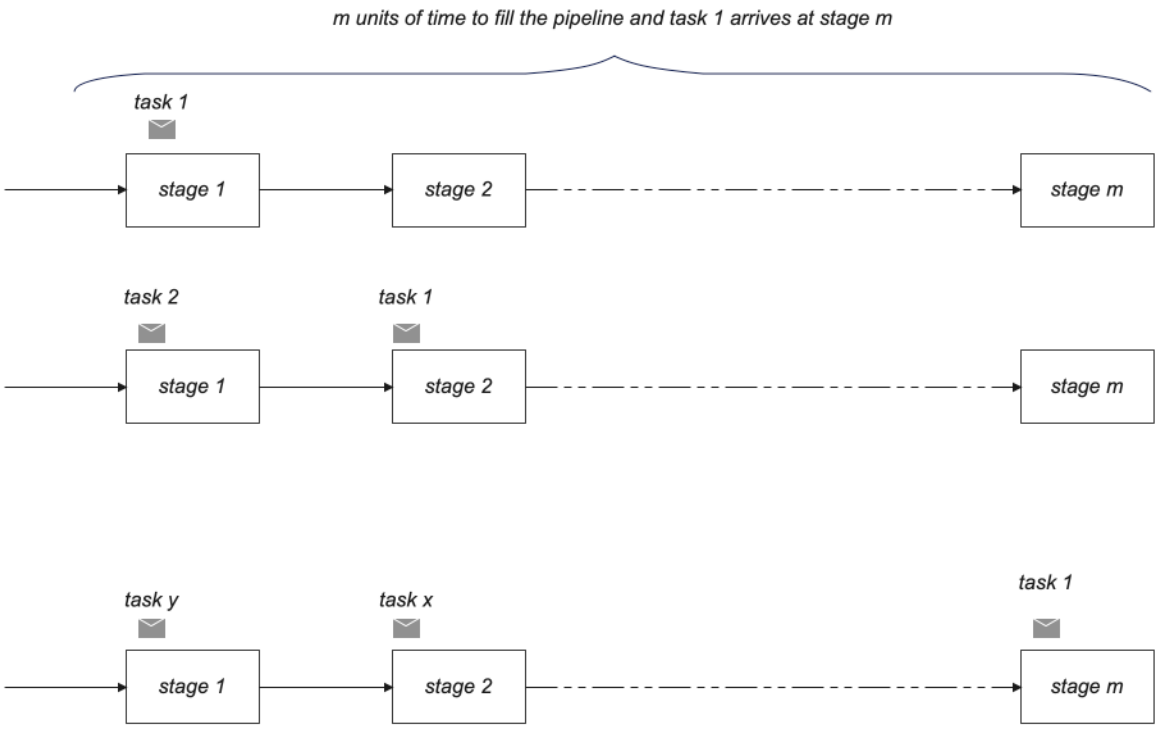
In pipeline execution, the stages get filled up one after another as each task is in process. The first task starts at time zero and stays in each stage for 1 unit of time. When the first subtask is completed in stage 1, it forwards its result to the next stage, stage 2, and immediately starts processing the next task that arrives in stage 1. After m time units, the pipeline is full, and from then on, one new task is finished at the end of every time unit. The total execution time of the pipeline can be divided into two parts: (1) filling of the pipeline needs m units of time, because the first job to pass through all stages, and (2) the remaining tasks n - 1 need n - 1 units of time to processes. Therefore, the pipeline execution time is given by

$$T_{pipeline} = m + (n - 1)$$

The ratio of sequential execution time to pipeline execution time yields the following for speedup:

$$\text{Speedup} = \frac{T \text{ sequential}}{T \text{ pipeline}} = \frac{n * m}{m + (n-1)}$$

where for very large n ($n \to \infty$), the term m in the denominator will become unimportant compared to n - 1, and the speedup will approach m. Hence, the maximum speedup possible is m, for an infinite number of tasks ($n \to \infty$), being processed by the pipeline. Thus, the maximum obtainable speedup in executing n identical tasks with an m stage pipeline is m, with a large enough number of tasks to keep the pipeline fully busy.



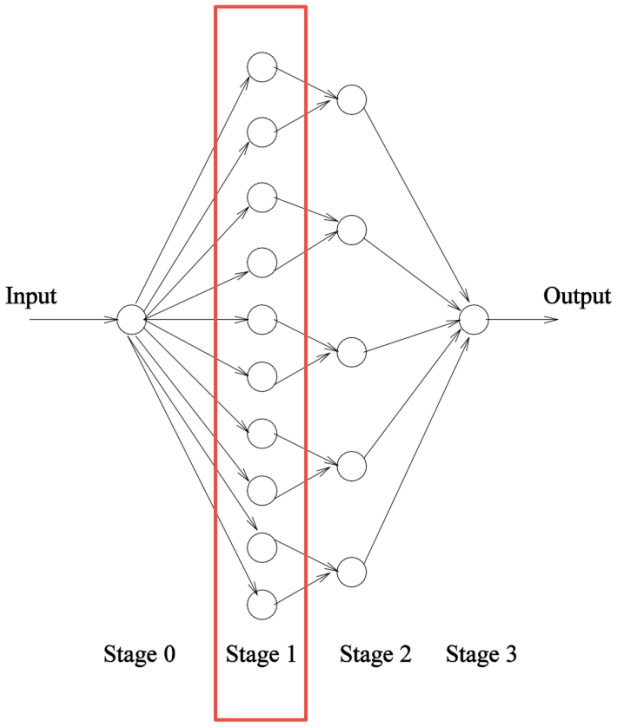
Question 3

The task graph shown in the following figure (next page) 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 2 units of time; each of the computational tasks in stage 1 takes 1 unit of time, and each of the computational tasks in stage 2 takes 8 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 number of processors should be just enough to execute the maximum number of tasks parallelly for the lowest execution time. A parallel program's maximum degree of concurrency is the maximum number of jobs that can run simultaneously at any instant of time. In this graph, the maximum degree of concurrency is 10 since the largest number of tasks that can run in parallel is 10 which has been highlighted in red color in the following figure.

$$\boldsymbol{Minimum\ number\ of\ processors\ for\ lowest\ runtime} = \mathbf{10}$$



For the second part of the question, we need to have the parallel execution time and sequential execution time to be able to calculate the efficiency. To find the sequential version, a single processor has to go through all the tasks in each stage to compute the final result. Therefore, the execution time in this case will be:

$$\boldsymbol{T_{s\ (sequential)}} = \mathbf{2} + (\mathbf{10} * \mathbf{1}) + (\mathbf{5} * \mathbf{8}) + \mathbf{2} = \mathbf{54}$$

In the parallel version, the critical path Which is the longest directed path from the start node to the finish node is the parallel execution time. This path is calculated as follows:

$$\boldsymbol{critical\ path\ length} = \mathbf{2} + \mathbf{1} + \mathbf{8} + \mathbf{2} = \mathbf{13}$$

$$\boldsymbol{T_{p\ (parallel\ time)}} = \boldsymbol{critical\ path\ length}$$

Now, we can find the efficiency with this formula:

$$\text{efficiency} = \frac{\text{sequential execution time}}{\text{parallel execution time} * \text{number of processors}}$$

$$= \frac{54}{13 * 10} = 0.415385$$

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.**

For (i) increasing the number of processors will not improve the efficiency of our parallel system. This is because we have already calculated the maximum degree of concurrency, which is 10. Any increase in the number of processors beyond 10 will result in extra processors being idle and will increase overhead. As a result, efficiency will decrease as we increase the number of processors.

$$\text{efficiency} = \frac{\text{sequential execution time}}{\text{parallel execution time} * \text{number of processors}}$$

number of processors = 11 → $\frac{54}{13*11}$ = 0.3776

number of processors = 12 → $\frac{54}{13*12}$ = 0.3462

We can see that by increasing the number of processors the efficiency will decrease.

In part (ii), we observe that a reduction in the number of processors could result in increased efficiency because of higher utilizations that arise from fewer processors. Of course, this is under various assumptions: when the task dependency graph is such, when the execution times for each task are unity, and etc. One important thing is the delay introduced by tasks needing to wait for available processors. Due to the reduction in processor count, these nodes begin to wait longer to get executed, delaying overall performance.

Number of processors = 9
Parallel runtime in this test case = 2 + 1 + 1 + 8 + 2 = 14
Efficiency = $\frac{54}{14 * 9}$ = 0.4286

Number of processors = 7
Parallel runtime in this test case = 2 + 1 + 1 + 8 + 2 = 14
Efficiency = $\frac{54}{14 * 7}$ = 0.5510

Number of processors = 5

Parallel runtime in this test case = 2 + 1 + 1 + 8 + 2 = 14

Efficiency = $\frac{54}{14 * 5}$ = 0.7714

Number of processors = 3

Parallel runtime in this test case = 2 + 1 + 1 + 1 + 8 + 8 + 2 = 23

Efficiency = $\frac{54}{23 * 3}$ = 0.7826

Number of processors = 2

Parallel runtime in this test case = 2 + 1 + 1 + 1 + 1 + 1 + 8 + 8 + 8 + 1 + 2 = 33

Efficiency = $\frac{54}{33 * 2}$ = 0.8181

Please note that the delay calculations have been performed as needed. For example, in the first test case, the extra 1 represents the delay that a node will wait before being assigned to a process. It can be seen that by reducing the number of processors the efficiency will always increase.

For (iii) upgrading the current processors to faster models won't ensure efficiency because doing so will result in shorter execution times for each CPU. In order to calculate the efficiency for the same 10 processors, assuming the new processors are twice as fast as the old ones is:
sequential execution time will be half and will be 27 and the parallel execution time will be also half and will become 6.5, so:

$$\text{efficiency} = \frac{27}{6.5 * 10} = 0.41538$$

Also, if the processers are three times faster the sequential execution time will become 18 and the parallel execution time will be 4.33, so the efficiency will be:

$$\text{efficiency} = \frac{18}{4.33 * 10} = 0.\ 41570$$

Since both parallel and sequential execution times will decrease in the same proportion, the efficiency came out to be approximately the same.

c) **What is the maximum speedup of the above parallel system when it is solved using 3 identical processors? Show your calculations.**

In this case, we have 3 processors, and in order to calculate the speedup, we need to find the parallel runtime which is calculated below:

Parallel runtime = 2 + 1 + 1 + 1 + 8 + 8 + 2 = 23

Now to calculate the speedup we have:

$$Speedup = \frac{\text{time required for sequential version}}{\text{time required for parallel version}}$$

$$= \frac{54}{23} = 2.35$$