Algorithms: Design and Analysis - CS 412

Weekly Challenge 03: Parallel Computing

Ali Muhammad Asad - aa07190

1. (1 point) The worst-case complexity for any sequential comparison-based sorting algorithm is $\Omega(nlgn)$. However, parallel computing can further reduce the running time of the algorithms. In its most basic sense, parallel computing refers to the simultaneous execution of tasks using multiple processors and cores. The algorithms can be parallelized when some of their steps can be computed in parallel. Lack of dependencies implies potential for parallel execution. For example, divide-and-conquer approach-based algorithms are often well-suited for parallelization. The division of the problem into sub-problems that can be solved independently facilitates parallel execution.

Selection Sort is a well-known sorting technique that scans an array to find the maximum item, puts it at the last location in the array, and then scans the array $(1...n-1)$ for the second maximum item, places it before the last location, then third maximum $(1...n-2)$ and so forth, until reaches the smallest item to be put at the first location of the array. It has $O(n^2)$ complexity.

Is it possible to parallelize the classic selection sort without making any changes and gain performance improvement? If yes, explain the aspects of the algorithm that can be parallelized. Identify an approach with the help of which parallelization capabilities of selection sort can be improved. Explain your proposed approach in 4 to 5 lines.

Note: Performance improvement in parallel computing comes from many different factors like the nature of the algorithm, available computing architecture, number of processors or cores, size of data, etc. Note that for this particular assignment, we are only interested in exploring the nature of the algorithm. Consider that the number of available processors or cores is n, where $n > 1$.

Useful Resources

- Example (Bubble Sort): 220- L20 (usfca.edu) (Reference: CS 220: Introduction to Parallel Computing, University of San Francisco)
- Chapter 26 of Introduction to Algorithms, fourth edition. Authors: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

Solution: The classic selection sort algorithm is inherently sequential, as also described above in the question, wherein, the maximum element from the array is selected and placed in its correct position, and is repeated for the rest of the array sequentially. Hence, each

step of the algorithm is dependant upon the previous step. So the classic selection sort algorithm cannot be parallelized without making any changes.

However, we can implement a variant on the selection sort, and call it **Parallel Selection Sort**. Instead of finding a single minimum element in each pass (which is sequential), we find multiple minimium elements in parallel. A high-level description can be as follows:

1. Divide the array into p partitions, where p is the number of available processors.
2. Each processor finds the minimum in its partition.
3. A global comparison finds the overall minimum.
4. Swap the overall minimum with the first element of the array.
5. Repeat the above steps for the remaining elements of the array.

In this Parallel Selection Sort, the array is divided into p partitions, with each processor finding the minimum in its partition, which happens in $O(n/p)$ time (since each processor is responsible for n/p elements). The global comparison to find the overall minimum takes $O(p)$ time, since there are p processors. Hence, the total time taken for each iteration is $O(n/p+p)$. Since there are n elements in the array, the total time taken is $O(n(n/p+p)) = O(n^2/p + np)$.

However, this still doesn't improve our performance over the classic selection sort. This is because of the inherent sequential nature of the algorithm. So even if we divide our array into p partitions, we still need to wait for the previous iteration to complete before we can start the next iteration. Thus, we still need to perform n iterations over n elements. Even with $p = n$ processors, we still get $O(n^2)$ complexity; although each processor is in charge of only one element, there are still n comparisons between n elements, which means n iterations, therefore, is still in $O(n^2)$. It is even evident through our time complexity equation; $O(n^2/p + np)$. If we increase p, the first term decreases, but the second term increases by the same proportion, and if we decrease p, the second term decreases but the first term increases then. Therefore, the time complexity is still $O(n^2)$, and we don't get any performance improvement.