Assignment-4 Parallelization of Quicksort Algorithm (Part-1)

Distributed Computing

Report by Elham Ghiasi

Introduction:

In this assignment, we first learned and understood the quicksort algorithm and how it works. Briefly, the quicksort algorithm recursively partitions the data or array into two halves based on a pivot element. After familiarizing ourselves with the Quick Sort algorithm, we followed and understood the code provided in the handout. We executed the provided code and then elaborated on the algorithm's time complexity using examples. Executing the provided code allowed us to gain a deeper understanding, and we then expanded on the algorithm's time complexity, with some examples.

Quick Sort Algorithm:

As we know, the Quick Sort Algorithm is a Divide and Conquer Algorithm. In divide and conquer, we first divide the problem into subproblems, then we conquer and combine solutions. In the Quick Sort Algorithm, there are three key stages: choosing the pivot, partitioning the array, and employing recursion. On average, the standard Quicksort algorithm operates with a time complexity of O(nlogn), while in the worst-case scenario, it exhibits a time complexity of O(n^2). The method of selecting the pivot and partitioning the array can vary, and this choice affects the performance of the algorithm.

parallelization of the Quicksort algorithm In C#:

Parallelization in Quicksort Algorithm offers significant performance enhancements, particularly for large datasets or multi-core processors. By employing multiple threads or processes to partition and sorting different segments of the array concurrently, it can achieve considerable speed improvements compared to single-threaded implementations. This approach optimally utilizes modern multi-core architectures, ensuring efficient resource utilization and scalability as dataset sizes increase. Moreover, parallelization reduces latency by overlapping computations and improves throughput, making it particularly advantageous for tasks where sorting is a critical bottleneck, such as

databases or data processing pipelines. Overall, parallel Quicksort efficiently harnesses available resources, resulting in faster sorting times and superior performance, especially for handling extensive data volumes where sequential algorithms might struggle.

Efficiently managing large datasets is crucial, and various methods exist to parallelize sorting algorithms. **One simple strategy involves generating two parallel tasks for each partitioned list.**

Time Complexity of Parallelized Quicksort algorithm:

The time complexity of the Parallelized Quicksort algorithm is O(nlogn). The parallelized version can reduce the constant factor, especially for large datasets, by distributing the workload across multiple processors or cores.

Implementation and Steps of Execution:

There are various methods to parallelize the sorting algorithm, and one straightforward approach is to create two parallel tasks for each of the partitioned lists. In this way by doing two separate tasks for each group of items being sorted. This helps to get things done faster by working on different parts of the list all at once. It is especially useful when you have a lot of stuff to sort out. To implement the code provided in the handout, start by creating a Windows Forms Application and name the project "QuickSort." Then, add a class to the project and name it "QuicksortAlgorithms."

The **sequential Quicksort implementation** in the code follows the divide-and-conquer strategy: it recursively divides the array into smaller subarrays, partitions them around a pivot element, and sorts them individually until each subarray contains only one element. By gradually combining the sorted subarrays, the entire array becomes sorted. This approach, while efficient for smaller arrays, may encounter performance limitations with large datasets due to its single-threaded nature. To address this, parallel Quicksort implementations are utilized, leveraging parallel processing to improve efficiency, particularly for larger datasets where the computational load can be distributed across multiple threads.

After creating our project and naming it QuickSort., we defined a class called QuicksortAlgorithms. The aim of this class is to provide implementations of the Quicksort algorithm. The following class can be used for both sequential and parallel versions of

Quicksort, allowing users to choose between a single-threaded or multi-threaded approach depending on their needs and the dataset being sorted. By encapsulating these sorting algorithms within a class, it promotes code organization, reusability, and maintainability, making it easier to integrate Quicksort functionality into their applications without needing to reimplement the algorithm from scratch.

By this line of code: "public static void Quicksort<T> (T [] data, int left, int right) where T:" we defined a method named Quicksort that accepts an array of elements of type T, along with indices defining the range of the array to be sorted. The method utilizes the IComparable<T> interface to compare elements during the sorting process. The Quicksort method uses the IComparable<T> implementation to sort the list entries.

Next, the method involves sorting the array recursively by dividing it based on a chosen pivot element and then sorting the subarrays on either side of the pivot. It begins by verifying if the right index is larger than the left, serving as the recursive termination condition. If true, it invokes the Partition function to determine the pivot index. Subsequently, it applies Quicksort recursively to the subarray on the left of the pivot (ranging from left to pivot - 1) and the subarray on the right of the pivot (ranging from pivot + 1 to right).

After that, by this line of code “public static void QuickSortParallel<T> (T [] data, int left, int right) where T: IComparable<T>,” we defined another method. This is a method named QuickSortParallel<T>, resembling the Quicksort<T> method in functionality. It operates on an array along with left and right indices, maintaining the same generic constraint. The method incorporates a constant, SEQUENTIAL_THRESHOLD, which determines whether to employ parallel or sequential sorting based on the size of the array segment. If the difference between the right and left indices falls below SEQUENTIAL_THRESHOLD, the method resorts to sequential sorting using Quicksort. Otherwise, it partitions the array around a pivot and conducts parallel sorting of the subarrays using Parallel.Invoke.

Another method present in this code is the Partition Method. It divides the array into two sections based on a chosen pivot, positioning elements less than the pivot to its left and greater elements to its right. It iteratively adjusts pointers to swap elements until the pivot is correctly positioned, returning its index afterward. Here is the line of code and the method implemented:

private static int Partition<T> (T [] data, int low, int high) where T: IComparable<T>

The last method employed is the Swap Method, which exchanges two elements within the array:

private static void Swap<T> (T [] data, int i, int j) {

T temp = data [i]; data [i] = data [j];

data [j] = temp;}

```
private static void Swap<T>(T[] data, int i,int j)
{
    T temp = data[i];
    data[i] = data[j];
    data[j] = temp;
}
```

To test our code, we added three buttons to Form1 and named them: "Sort Sequential," "Sort Sequential Large," and "Sort Parallel Large." Each button handler has specific duties, and we coded them accordingly.

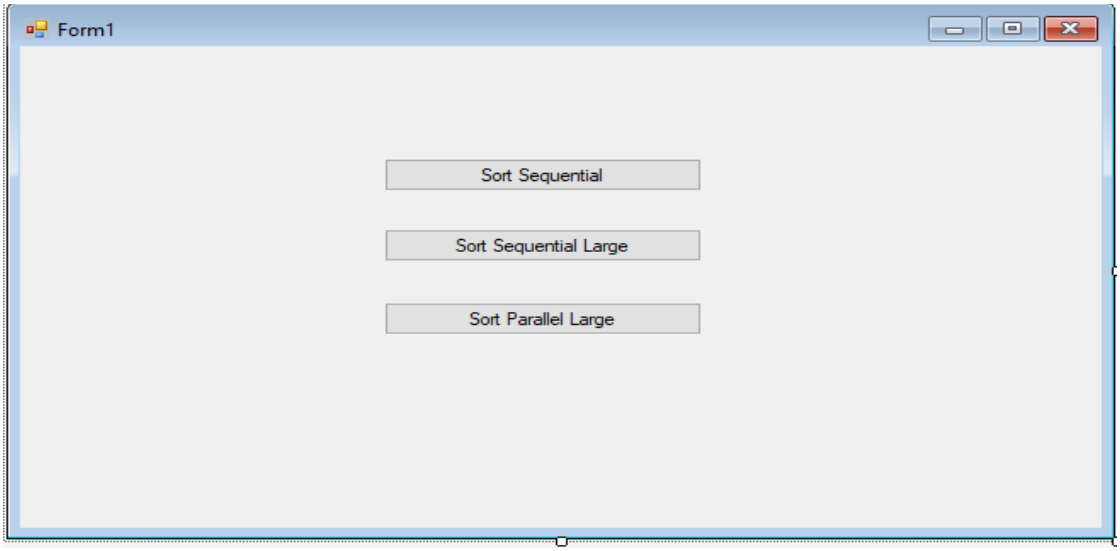
Briefly, the btnSortSequential_Click method sorts a small array sequentially and displays the sorted elements in a message box, while btnSortSequentialLarge_Click sorts a much larger array sequentially and displays the time taken to sort it. Similarly, btnSortParallelLarge_Click sorts the large array in parallel and displays the sorting time. The code measures the execution time of each sorting operation using a Stopwatch and showcases how parallelizing the quicksort algorithm can significantly reduce sorting time for large datasets by leveraging multiple CPU cores simultaneously.

One important part of the code is the implementation of the Quicksort algorithm in both sequential and parallel versions.

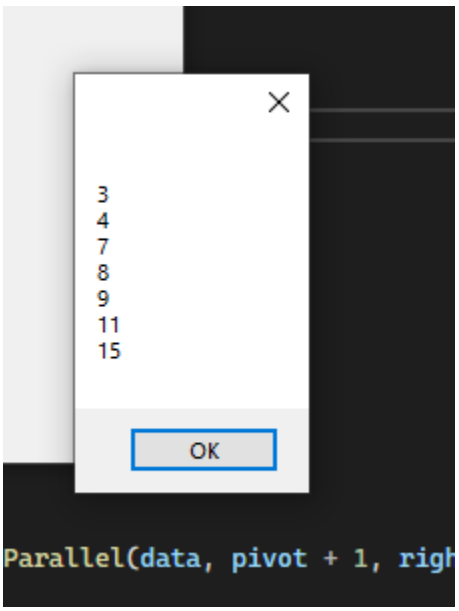
In the sequential version, the Quicksort algorithm is called using the QuicksortAlgorithms.Quicksort<double> (data, 0, data. Length - 1) method. This method takes the array data, the starting index (0), and the ending index (data. Length - 1) as parameters. The Quicksort algorithm recursively divides the array into smaller sub-arrays and sorts them using a pivot element to partition the array.

In the parallel version, the QuicksortAlgorithms.QuickSortParallel<double> (data, 0, data. Length - 1) method is used instead. This parallel implementation utilizes multiple threads

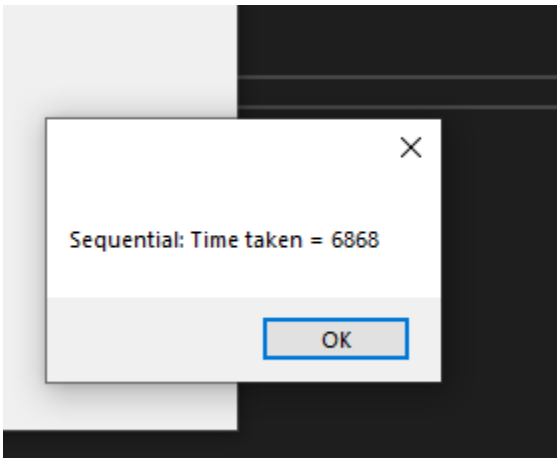
to sort different parts of the array concurrently, thus potentially speeding up the sorting process for large datasets.



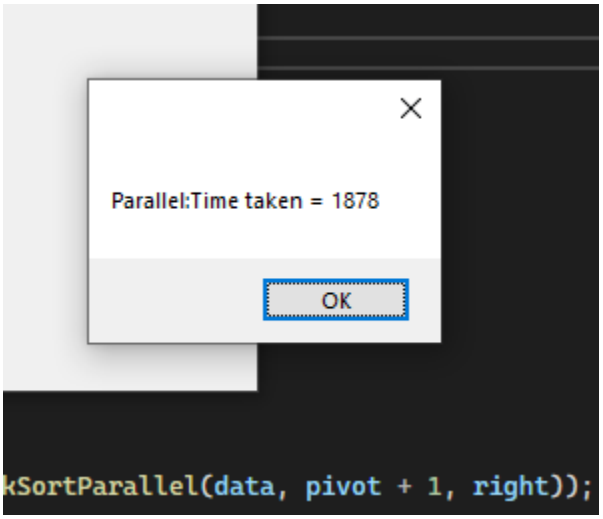
When the "Sort Sequential" button is clicked, this becomes visible:



When we click the "Sort Sequential Large," button, this appears:



And, at the end Upon clicking the " Sort Parallel Large " button, this is displayed:



Observation:

As can be seen, the sequential process takes about 6.8 seconds, while the parallel process takes 1.8 seconds, indicating that parallel execution is significantly faster than sequential execution.

The sequential version operates in a single thread, while the parallel version leverages multiple threads to distribute the sorting workload, potentially leading to faster execution times on multi-core processors.

Modified the SEQUENTIAL_THRESHOLD from 1000 to 10000:

Increasing the SEQUENTIAL_THRESHOLD from 1000 to 10000 in the QuickSortParallel method would reduce the level of parallelism in the algorithm, potentially leading to fewer parallel tasks being initiated and underutilization of available resources. While this might

decrease the overhead associated with parallel execution, it could also result in diminishing returns or even performance degradation, particularly if the overhead outweighs the benefits of parallelism. The worst-case time complexity of the algorithm remains O(n^2), but the actual runtime behavior may vary, with the efficiency of the algorithm being influenced by factors such as input distribution and pivot selection.

Alternative methods to enhance the parallelization of Quicksort:

To enhance the parallelization of Quicksort, several methods can be employed. These include parallel pivot selection and partitioning, which distribute the workload among threads more evenly by performing these critical steps concurrently.

Hybrid parallelization techniques combine parallel and sequential approaches, leveraging the benefits of each based on the size of the data. Task-based parallelism frameworks provide higher-level abstractions for managing threads and task scheduling, improving resource utilization and simplifying implementation. Load balancing strategies ensure that work is evenly distributed among threads, preventing bottlenecks, and maximizing efficiency. By carefully considering factors such as data distribution and hardware

architecture, and by measuring performance, the most effective parallelization method can be identified for a given problem and computing environment.

Results:

The variable **SEQUENTIAL_THRESHOLD** in the **QuickSortParallel** method serves as a threshold to determine whether the sorting should be done using the sequential quicksort algorithm (Quicksort) or the parallel version (QuickSortParallel). When the size of the portion of the array to be sorted (determined by the difference between right and left indices) is smaller than the SEQUENTIAL_THRESHOLD, the sequential version of the quicksort algorithm is used. This is because for small datasets, the overhead of parallelization (such as thread management and synchronization) might not be justified, and a sequential sort can be more efficient.

In this case, the SEQUENTIAL_THRESHOLD is set to **10,000**, meaning that if the number of elements to sort is less than 10,000, the algorithm will resort to a sequential quicksort instead of continuing with the parallel approach. This threshold is used to optimize performance by combining the benefits of parallel processing for large datasets with the efficiency of sequential algorithms for smaller datasets.

Conclusion:

In conclusion, this assignment provided a comprehensive exploration of the Quick Sort algorithm, from understanding its basic principles to implementing and analyzing both sequential and parallel versions. We began by grasping the fundamentals of Quick Sort, recognizing its divide-and-conquer nature and its pivotal stages of pivot selection, partitioning, and recursion. Delving into the parallelization of Quick Sort, we uncovered its potential for significant performance enhancements, especially for handling large datasets and leveraging multi-core processors effectively.

Through the implementation steps outlined and the examination of execution times using provided code, we witnessed firsthand the efficiency gains achieved through parallelization. Notably, the parallelized version demonstrated a substantial reduction in sorting time compared to its sequential counterpart, highlighting the benefits of concurrent processing in speeding up computational tasks.

Moreover, the adjustment of the SEQUENTIAL_THRESHOLD parameter shed light on the trade-offs involved in tuning parallel algorithms. While increasing the threshold can reduce parallel overhead, it may also impact performance depending on a range of factors such as input distribution and system resources.

This assignment not only deepened our understanding of Quick Sort but also underscored the significance of parallel computing techniques in optimizing algorithmic performance for real-world applications, particularly in scenarios where sorting constitutes a critical bottleneck. By embracing parallelism, we unlock new avenues for tackling computational challenges efficiently and harnessing the full potential of modern computing architectures.