Operating System - 2

Assignment 1

Parallel Sorting using Multi-Threading

Objective

The objective of this assignment is to check and compare the efficiency of the merge function using multiple threads.
To explore and understand we implemented these operations in three different methods.
We are using the merge sort algorithm for the best results.

- **Sequential Algorithm**
  In this method we are diving the complete array into 2^p segments and then we are sorting each of the segments using the merge sort algorithm. After completion of the sorting of each segment, we are calling the merge function in the main function only to merge all the segments.
  This the basic approach without using any concepts of threading.

- **Method 1**
  In this method we are using the concept of threading to sort each segment out of p segments. To compute this task we created 2^p threads whose task is to sort the assigned segment. Once all the segments are done, these slave threads will be merged and exited.
  After the sorting of each segment, we are using the merge function which merges each segment with the right neighbor after each step of merging. We are doing this job directly in the main thread. We are not using the threading concepts in doing this merge task.
  This method will sort each segment very quickly because of multiple threads that are used to do this task. But while merging or the bigger task, it is just computing it directly through the main thread which reduces the performance of this method when the number of segments is larger.

  The sudo code can be written as

  t0 = clock()

```
// Storing the current time in a time data time.h library
pthread_t thread[p]
// Creating p threads variable.
for( i =0 to p-1)
        pthread_create( … )
        // Calling the merge sort function to sort the segment i
for( i =0 to p-1 )
        pthread_merge( … )
        // Merging the threads created earlier
for( i = 1 to p )
        merge( i and i +1 segment )
        // Also note after each iteration we are adding the right segment to the
previous main segment.

t1 = clock();
time required = t1 - t0
```

- **Method 2**

  This is using is multiple threads to compute the merge task which is done by the main thread in Method 1. It is creating new threads to perform the task at a very speed which in turn reduces the time required to compute the sorting when the number of segments is large too. This is the drawback of Method 1 which is overcome by this method.

  More interestingly we are merging two segments in iteration using a new thread. This means in each round of merging we are using a different which improves the performance by a great amount when the number of segments is too large.

  The sudo code for the merge function used in this method

  p: number of segments

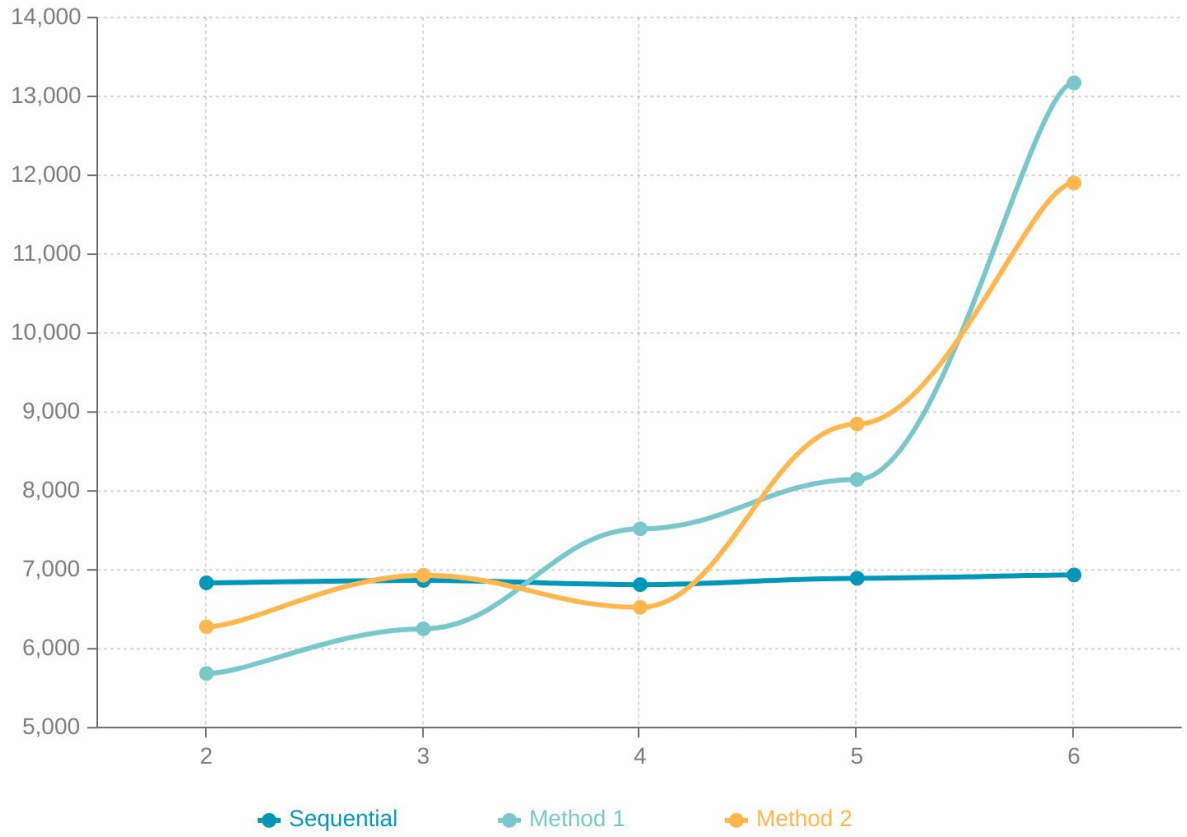
```
        while (p!=1)
        // The loop will run until there is only one segment is left
           {
              pthread_t th[n];
              for( int i=0; i < p/2; i++)
```

```
        {
            pthread_create ( &th[i], NULL, merge_2, &i);
             // Calling the merge function using the new thread.
        }
        for( int i=0; i < p/2; i++)
        {
            pthread_join(th[i], NULL);
        }
        p = p/2;
         // Since we merged the segments, therefore, the number
of segments reduced to half
    }
```

The Graphs

Graph 1

The parameter n = 13
X-axis: parameter p
Y-axis: time in microseconds



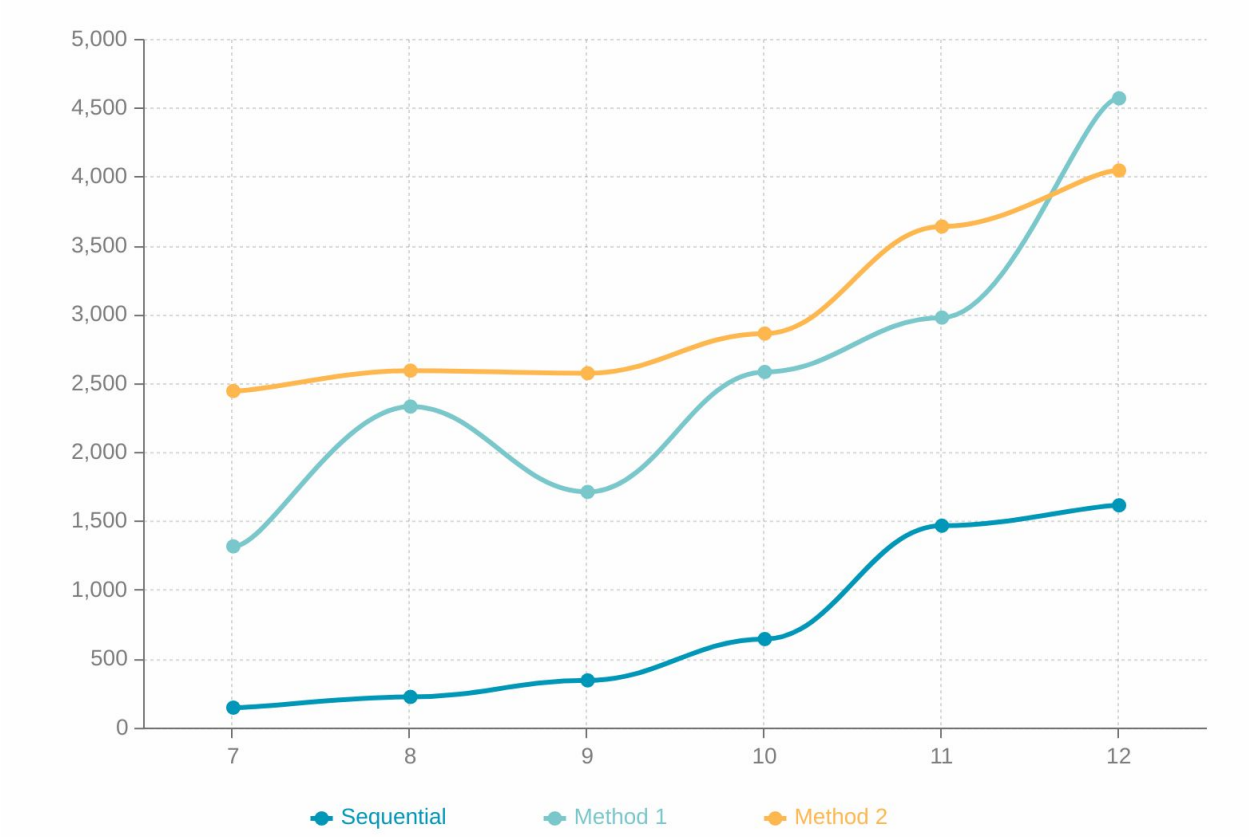
Analysis:

- The sequential method line is almost parallel to the x-axis as the number of input numbers are always equal.
- The difference between method 1 and method 2 is not very great for the initial values of p
- Method 2 is taking less time for higher values of p
- Multithreading is more beneficial for higher values of the input, as it is observable from the graph of the sequential line.
- The curve for method 1 and method 2 also crosses each other as the input values for p are not very high.

Graph 2

X-axis: parameter n
Y-axis: time in microseconds

Parameter p = 4



Analysis:

- The sequential method is good for lower values of parameter n but it is not ideal for higher values.
- Method 1 is good whenever the number of segments is less. But even for higher values of n the method 2 will be best for all the cases.
- After a certain limit, the sequential graph will beat the graph for both method 1 and method 2.

Some more analysis

- Method 1 and Method 2 are highly efficient for sorting a high number of inputs. The threading increases the performance by an appreciable amount.

- Method 2 is more efficient when the number of segments increases. This is due to the implementation of the merge function using multiple threads. In method 1 we are using the main thread to complete the merge task which in turn uses very high CPU usage when the number of segments increases.
  Some examples:
  (All the times are in microseconds)
  - n = 15 & p = 5
    - Method 1: 30549
    - Method 2: 24112
      (The difference is not high the number of the segment are less)
  - n = 15 & p = 8
    - Method 1: 66633
    - Method 2: 42270
  - n = 15 & p = 12
    - Method 1: 724425
    - Method 2: 483503
      (The difference is very high as the total number of the element as well as the total number of the segment is high)
  - n = 16 & p = 4
    - Method 1: 42217
    - Method 2: 41430
      (Here again the number of segments are normal and thus not a major difference in performance)
  - n = 18 & p = 12
    - Method 1: 4788235
    - Method 2: 440317
      (The difference is a very high number, and using only method 2 will be advisable)

Conclusion

The threading improves the performance of the sorting algorithm. More effectively the merge using the threading method is more efficient than the merge in the sequential method.
Method 2 is the most efficient as compared to method 1 for a higher value of inputs.