REPORT

SRI HARI. M
CS19BTECH11039

Aim : Parallel sorting using multithreading

Sort algorithm used: **Quicksort**(randomised)
Merge algorithm used: **Merge two sorted arrays** O(m+n)
Both the algorithms are attached at the end of this document.

Code Design:

First and the foremost thing is including the header files to work on and utility. I included all the useful Header files. Next, to proceed into the methods, we must have read input by then. As the input file is mentioned with naming convention as “inp.txt”, it is hardcoded in the C source file to read the inputs N and P from the input file. After this, there are the sorting operations performed to sort the array using threads and finally, we get the output of the program execution in “Output.txt” file. It consists of the array before sorting in its first line, the very same array after sorting in the second line and the time taken to complete the sorting is printed in the third line. This is the work generally done by the program. Let’s dive into the design of method1 and method2.

As mentioned above, after including the header files and reading the input from the input file, we first calculate the size of the array, number of threads to be used and the size of the segment. I used the in-built power function which is present in math.h

library for computing the above properties. I've created a structure called properties which has all the properties related to array and sorting. **We didn't use global variables and global arrays as it becomes increasingly hard to figure out which functions actually read and write those data.** So an array within the properties is used to implement this program. The memory is allocated to the properties structure followed by allocating the memory to the array. Now we need to populate the array with random values. For that, I used a function and populated the array with random values. The program is seeded with NULL to populate random values every single time(but graph's observations are taken with single input). Next to that, I stored all the information using two worker arrays on how to perform sorting segment-wise and phase-wise. Using the pthread_create function, I created segment threads and called the sort function. The count in properties is increased whenever it goes in the thread function and the count recorded at each time is responsible for sorting the segments correctly. After that, I used the pthread_join function to join all the worker threads involved in segment sorting to ensure that none of the threads should alter the values after the merge process starts.
Till here, both the methods carry out the same process.

Method1:
Now in method1, after all the segments are sorted, the merge process is called iteratively. As it is to be carried out with the main thread, the helper array contains all the details which follow the merge function deal. It is carried out in the manner that the first two segments get to merge and then it proceeds till all the segments are merged and it eventually takes the (number of threads - 1) steps to carry out the merging process. After that, the array is printed in the output file. The timestamps are recorded at the start of the sorting process and at the end of the merging

process using the structure timeval variables. After that, the time taken to execute the sorting and merging process is calculated and is printed in the output file.

Execution:

- Reading the input from the input file
- Populating the array
- Printing the array before sorting
- Recording the time stamp as start
- Thread creation and calling quicksort function
- Waits till all the segments are sorted,
- Merging process(array will be sorted by the end of this)
- Recording the time stamp as end
- Printing the array after sorting
- Calculating the time taken and printing the time taken.
- Exiting the thread notion.

Method2:

Now in method2, after all the segments are sorted, the merge process is called until the phases are complete. It's generally designed in a while loop as for every iteration, half the number of previously existing threads are created and sent to the merge process. Again there will be a count increment whenever the merge process is called. The count recorded is responsible for the merging process. After the segments are sorted, a total number of (number of threads - 1) are created in total for the merging process. As the number of threads before the start of the merging process are equal to the number of threads which are involved in segment sorting, with keeping track of that number, creating the number of new threads, and then decreasing the number to half of itself and the loop continues, till the array is sorted. After that, the array is printed in the output file. The timestamps are recorded at the start of the sorting process and at the end of the merging process using

the structure timeval variables. After that, the time taken to execute the sorting and merging process is calculated and is printed in the output file.

Execution:

- Reading the input from the input file
- Populating the array
- Printing the array before sorting
- Recording the time stamp as start
- Thread creation and calling quicksort function
- Waits till all the segments are sorted,
- Merging process such that half the number of previously existing threads are used (array will be sorted by the end of this)
- Recording the time stamp as end
- Printing the array after sorting
- Calculating the time taken and printing the time taken.
- Exiting the thread notion.

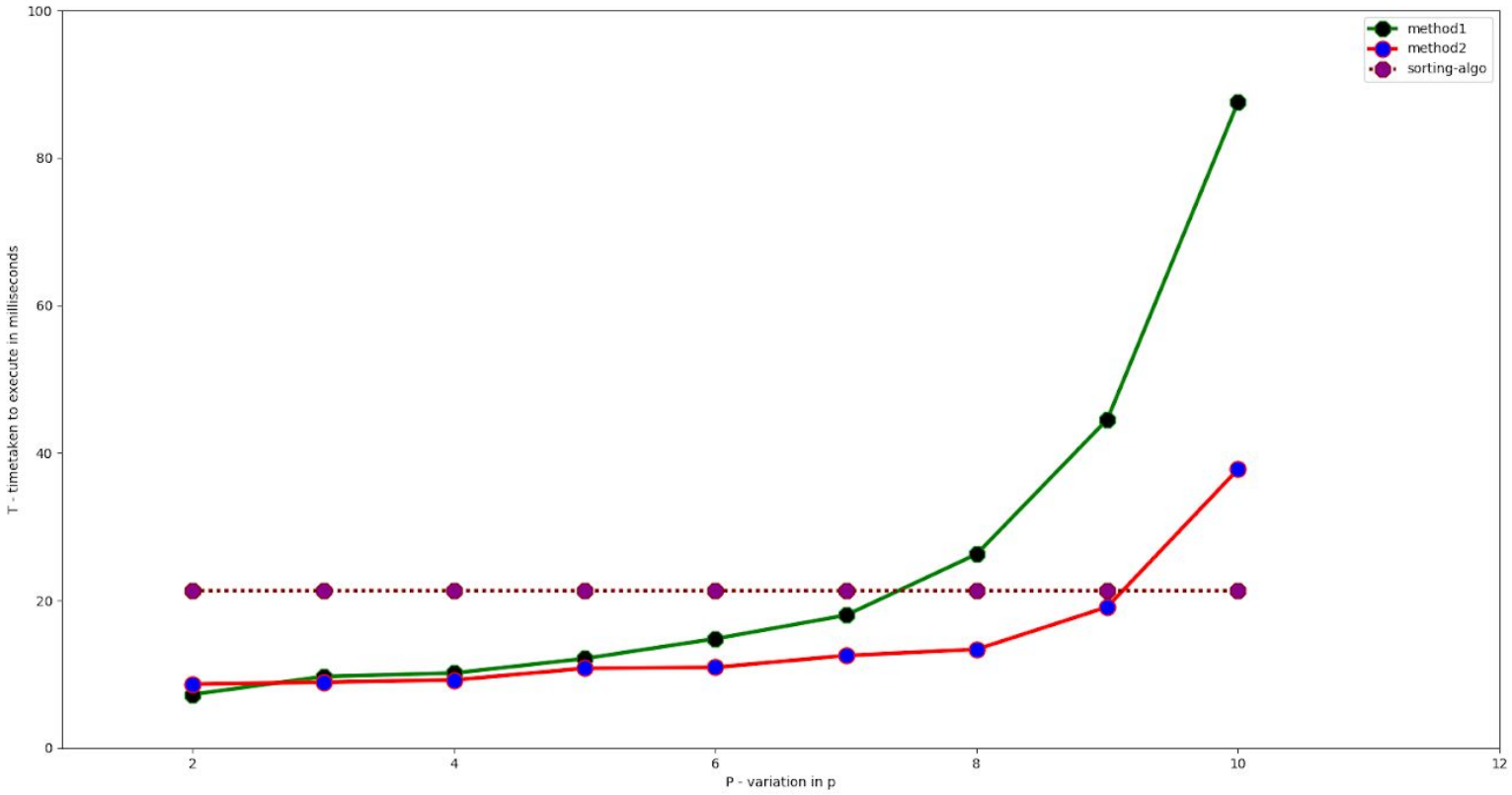
Using timeval structure and its properties, I calculated the time taken for execution.
This is the clear picture of the code design for both the methods. Comments are written for every part of the code.

Graphs and analysing the performance:

***Each value of time recorded for the particular value of N and P for method1, method2 and the sorting algorithm is the average value of 100 recorded observations. This is done to get the minimal error in recording observations and plot the graphs.**

Graph1 :

The following graph is plotted with the given constraint N = 15, i.e fixing the size of the array to 2 raised to the power 15. The graph shows the variation in times taken by the sorting algorithm, method1 and method2 respectively.

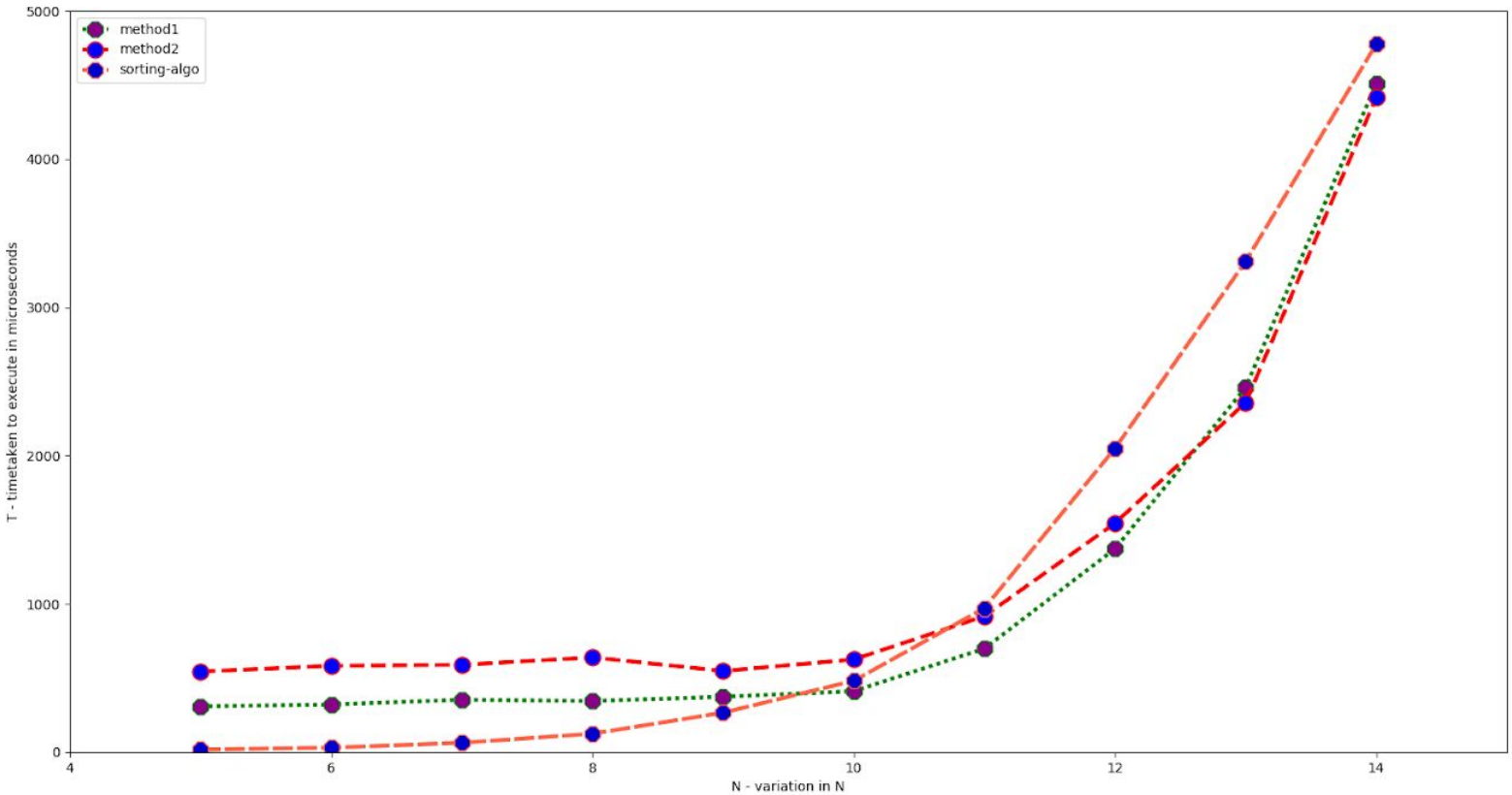


Clearly, when N is fixed to 15, we can see that the method1 is taking too long to execute than the method2. Initially, both the methods are performing reasonably well and in fact performing better than the sorting algorithm. But as threads increase the overhead time is adding to the execution time and making them inefficient. But one of these methods is to be preferred over sorting algorithm and method2 is to be preferred over method1. If we see at P = 2, method2 is taking more time than method1 because of the unnecessary creation of threads, where they are not really required.

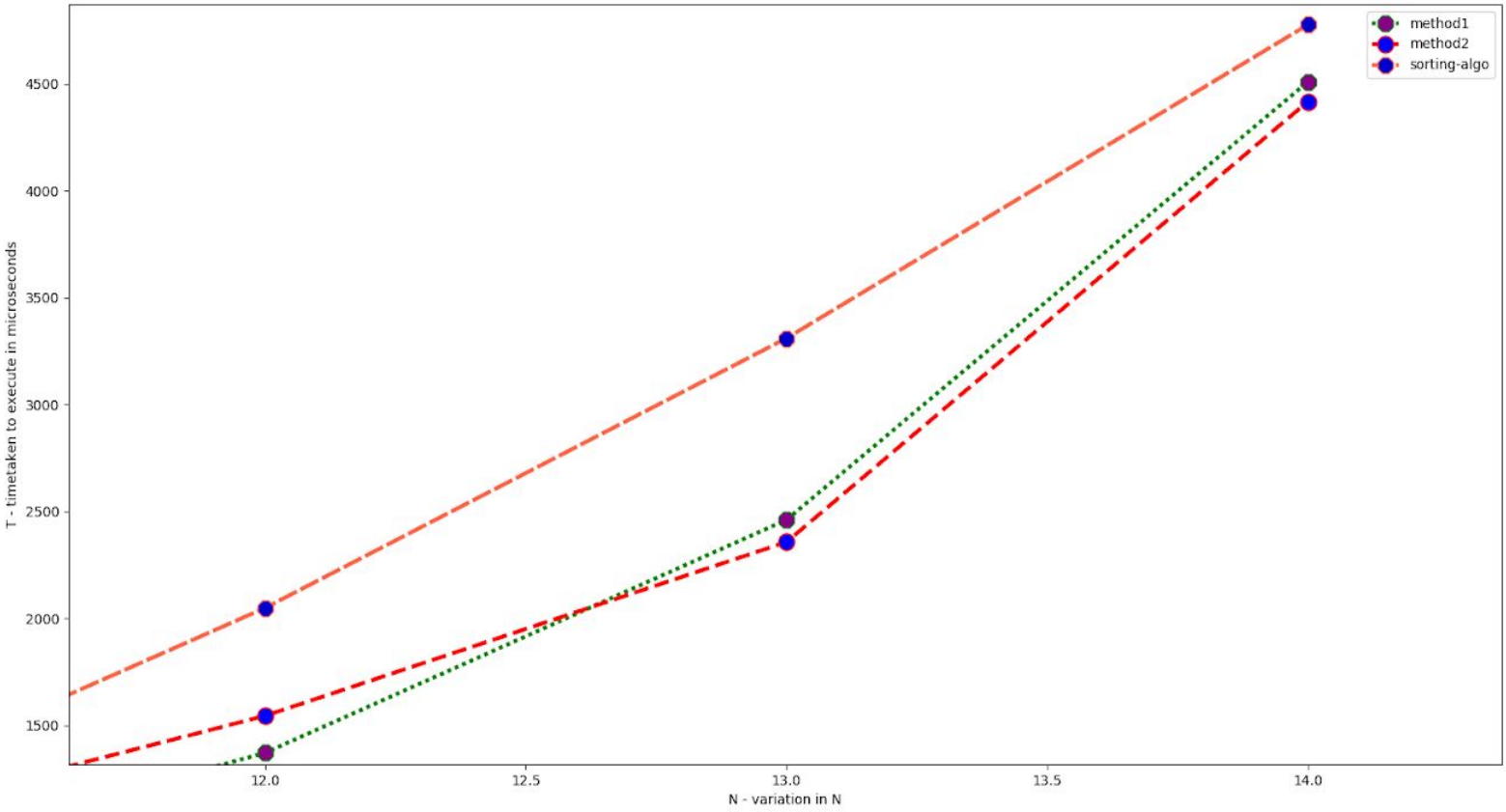
But at a later point, method2 is the clear winner over method1. At the end, where the P-value is 10-12, method2 is performing twice more efficiently than method1. Also as the sorting algorithm is independent of threads, it remains horizontal. Let's not conclude here, and see the following graphs for drawing a conclusion.

Graph2:

The following graph is plotted with the constraint fixed with p = 4 i.e the number of threads is fixed to 16. The graph shows the variations in time taken for method1, method2 and the sorting algorithm.



From this graph, one can clearly see that the time taken for sorting by sorting algorithm quicksort is increasing exponentially compared to method1 and method2. For every value of N, the time taken by quicksort to execute sorting is approximate equals 1.9* times of that time taken to execute the algorithm with value N-1. While coming to method1 and method2 we can see that for given constraints initially, method1 is performing well but later on, as N increases we can see that method2 is performing well. As the picture, in the end, is not that clear, a zoomed pic is attached here.



In the end, we can see that the methods are performing better than the original sorting algorithm and method2 is performing better than method1 when it is in the high value of N, suggesting that as the size of array increases method2 performs well. Because the

overhead for threads decreases reasonably well with an increase in the size of an array. Where in method1 we are performing the merge operation again and again over the array, but in method2 even though the same is being done, it is done in a stipulated way such that it is logically better than method1. If the value of N is increased a bit more, we can see that method2 performs best than method1 and the sorting algorithm.

The lines in the graph are approaching the same time because of the choice of the value of P. **If P changes reasonably, then the lines are also diverging with a better margin.**

Final conclusion:

From the given constraints, one can come to the conclusion that both the methods are efficient than the original sorting algorithm and method2 is efficient than method1. For dealing with real-life problems which involve the size of the array is as large as 2 raised to the power of 15 or more, method2 is to be preferred over both the other algorithms. From graph1, we shouldn't conclude that the sorting algorithm is a way better than method2 and method1 as we can see in graph2 and graph3, with a change in both N and P, method2 performs well.

Quicksort function:

```
//utility function for swapping integers in array
void swap(int* a, int* b)
{
   int t = *a;
   *a = *b;
   *b = t;
}
```

```
//partition function for quick sort
int partition(int arr[], int low, int high)
{
   int i = low - 1;
   int j = high + 1;
   int pivot = arr[low];

   for( ; ; )
   {
       i++,j--;

       while (arr[i] < pivot)
       i++;

       while (arr[j] > pivot)
       j--;
       if (i >= j)
       return j;

       swap(&arr[i], &arr[j]);
   }
}

//quicksort function for sorting the array
void quicksort(int arr[], int low, int high)
{
   if (low < high)
   {
       int random = low + rand() % (high - low);
       swap(&arr[random], &arr[low]);
       int pi = partition(arr, low, high);
       quicksort(arr, low, pi);
       quicksort(arr, pi + 1, high);
   }
}
```

Merge function:

```
//function to merge two sorted arrays in time O(p+q) ,p and q are
the size of arrays
void merge(int* array,int first,int second)
{
   int* temp_array = (int*)malloc(second*sizeof(int)+1024);
   int i = 0,j = first+1,k = 0;

   while(i <= first && j <= second)
   {
       if(array[i] < array[j])
       temp_array[k++] = array[i++];
       else
       temp_array[k++] = array[j++];
   }

   while(i <= first)
   temp_array[k++] = array[i++];
   while(j <=  second)
   temp_array[k++] = array[j++];

   for(int i = 0 ; i < k ; i++)
   array[i] = temp_array[i];

   free(temp_array);
}
```

THE END!