Merge Sort using Parallel Programming

Mohd Mohtashim Nawaz* , Harsh Aryan†, Mayank Taksande‡ and Sneha Mishra§
Indian Institute of Information Technology, Allahabad
Allahabad-211012
Email: *itm2017005@iiita.ac.in, †itm2017003@iiita.ac.in, ‡ itm2017008@iiita.ac.in, § itm2017004@iiita.ac.in

Abstract—This Paper introduces an algorithm to sort a given array using merge sort which makes use of parallel programming. Parallel programming is a technique of dividing the work among multiple threads so that multiple tasks can be done at the same time. This reduces the total time of execution because the tasks which were earlier executed in a queue are now being executed parallelly. The solution simply breaks the problem in parts and each thread performs its own job independently and at the end, all threads are joined. The idea is to find out an efficient algorithm such that the time complexity, as well as space complexity, is minimized.

Keywords-Parallel Programming, Merge Sort, Parallelism, Time Complexity, Algorithm

I. INTRODUCTION

Merge sort is a divide and conquer approach of sorting an array in which an array in progressively divided into subarrays and the subarrays are merged such that the merged array remains sorted. This is a classic sorting approach which performs the task of sorting in nlogn time.
However, the work can be done more efficiently by making use of parallel programming.
Parallel programming is the technique of parallel computing in which the work is divided into sub-parts and each part performs its work independently.
Parallel programming can be realized by using threads which can be thought of as a subprocess running within the parent or independently (depending on the type of thread).
In the algorithm used to solve the given problem maximum of four threads are being used to solve the problem.
The array is randomly generated and the user is prompted to input the number of threads to be used to solve the problem. Then the problem is divided into those many parts as the number of threads and each thread runs to sort their part of the array independently.
The threads are joined so all threads wait till the last thread completes the execution. The output of all threads is a sorted subarray which is then simply merged to other subarrays such that the resulting array remains sorted.
Thus the problem of sorting the array is solved.

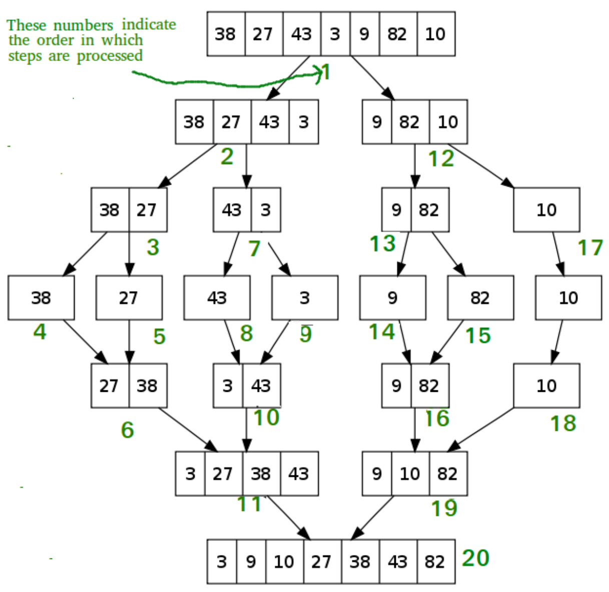

Fig 1.1: An illustration of merge sort

Here we will try to develop an efficient algorithm to solve the problem using the branch and bound.

This algorithm is tested multiple times for time complexity, space complexity and accuracy. In our case, we tend to focus on establishing a rule or a relationship between the time and input data.

This report further contains -

II. ALGORITHM DESIGN AND ANALYSIS

An efficient algorithm can only be created by considering every possibility. So we have to consider every possibility while designing the algorithm.

Since the algorithm involves parallel computing, care should be taken to avoid any kind of error. Threads should be created and joined properly which might otherwise lead to an error.

Steps for designing the algorithm include:

1). Checking for the sequence of array taken as the input if it is already sorted or not.

2). Creating the number of threads taken as input from the user.

3). Passing the mergeSort function to each of the newly created threads.

4). Finally, we will have sorted sequences in all the threads which are to be merged again using the merge function.

5). At the last, we have a sorted sequence which will be printed.

Array and structure used

```
MAX ← 1000000
int:ar[MAX]
struct task{
    int:task_no;
    int:task_l;
    int:task_r;
};
```

Function to merge the arrays

```
def merge(l, m, r)
{
        int:n1 ← m-l+1
        int:n2 ← r-m
        int:L[n1],R[n2]
        int:i,j,k
        for i=0 to n1 do
                L[i] ← ar[l+i]
        for j=0 to n2 do
                R[j] ← ar[m+1+j]
        i ← 0
        j ← 0
        k ← 1
        while i < n1 and  j<n2 repeat
                if L[i] <= R[j] then
                        ar[k] ←  L[i]
                        i ← i+1
                else
                        ar[k] ← R[j]
                        j ←  j+1
                k ←  k+1

        while i<n1 repeat
                ar[k] ← L[i]
                i ← i+1
                k ← k+1

        while j<n2 repeat
                ar[k] ← R[j]
                j ←  j+1
                k ←  k+1
end
```

Time Analysis:
15+27(r-l) units
Worst Case: O(n)
Best Case: Ω(n)
Average Case: Θ(n)

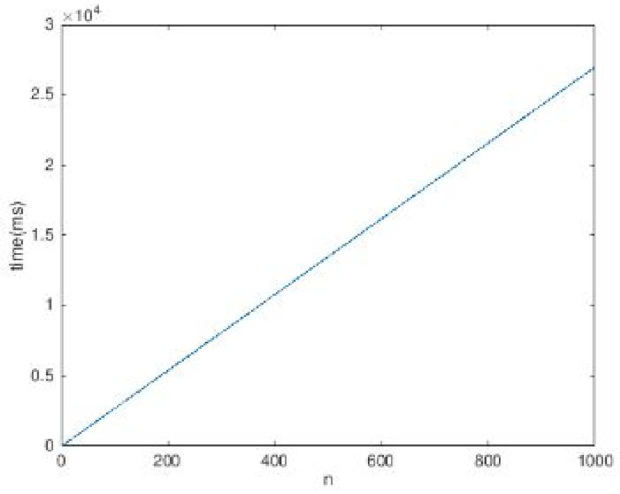

Fig 2.1: Graph depicting time complexity of merge function

Function to initiate the merge sort

```
def mergeSort(l, r)
   if l<r then
          int:m ← l+(r-l)/2
          mergeSort(l,m)
          mergeSort(m+1,r)
          merge(l,m,r)
end
```

Time Analysis:
= 6+2 T(n/2)+27(l-r)
Worst Case: O(nlogn)
Best Case: Ω(1)
Average Case: Θ(n log n)

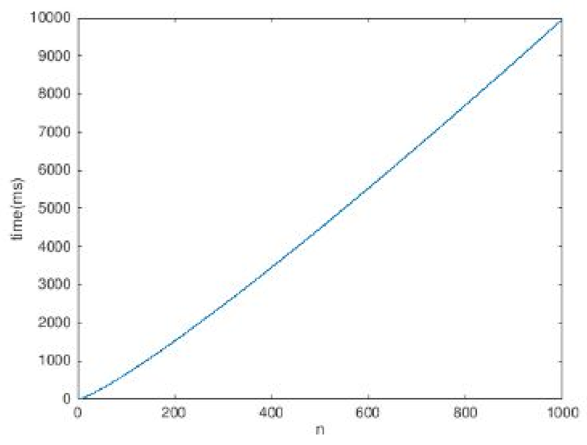

Fig 2.2: Graph depicting O(nlogn) complexity

Function to be passed in each thread

```
def *merge_Sort( *arg)
    struct task: *tsk ← arg
    int:l
    int:r
    l ← tsk → task_l
    r ← tsk → task_r
    int:m ← l+(r-l)/2
    if l<r then
        mergeSort(l,m)
        mergeSort(m+1,r)

        merge(l,m,r);
    }
    //free(tsk);
    return 0;
}
```

Time Complexity:
13+2 T(n/2)
Worst Case: O(n log n)
Best Case: Ω(n log n)

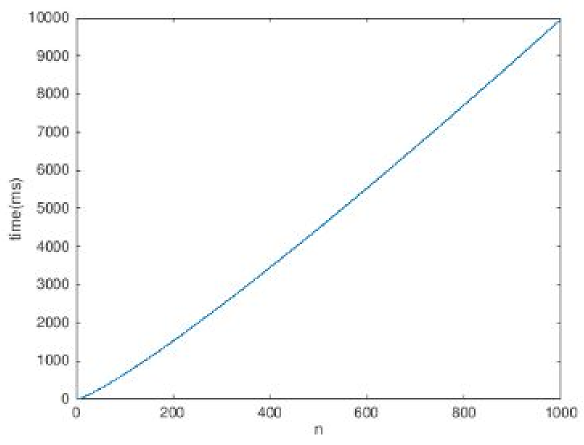

Fig 2.3: Graph depicting O(nlogn) complexity

Main function (code body)

```
def main()
    clock_t:start, end
    double:cpu_time;
    start:clock();
    int:n,i,k
    int:flag1 ← 0,flag2 ← 0
    int:thread_count
    input ← n
    for i=0 to n do
        input ← arr
    for i=0 to n do
        if ar[i] < ar[i+1] then
            flag1 ← flag1+1
    for i=0 to n-1 do
        if ar[i]>ar[i+1] then
            flag2 ← flag2+1
    if flag1=n-1 or flag2=n-1 then
        exit(1)
    else
        input ← thread_count
        pthread_t:threads[thread_count]
        struct task:tasklist[thread_count]
        struct task:*tsk;
        int:l ← 0
        int:len ← n/thread_count
        for i=0 to thread_count do
            tsk ← &tasklist[i]
            tsk → task_no ← i
            tsk → task_l ← l
            tsk → task_r ← l+len-1
            if i=thread_count-1 then
                tsk → task_r ← n-1
        for i=0 to thread_count do
            tsk ← &tasklist[i]
        pthread_create(&threads[i],NULL,merge_Sort,tsk)
        for i=0 to threa_count do
            pthread_join(threads[i],NULL)
        struct task:*tskm ←  &tasklist[0]
        for i=1 to thread_count do
            struct task:*tsk ← &tasklist[i]
```

```
                    merge(tskm → task_l,tsk → task_l - 1,tsk
→ task_r)
          //Display Sorted sequence
    end ← clock()
    cpu_time ← ((double)(end-start))/CLOCKS_PER_SEC
    //Display CPU Time
    return 0
end
```

Time Complexity:

Worst Case: O((nlogn)/t +28t+c)
The nlogn term arises due to merge sort, it is divided by t, the number of threads, as it all happens in parallel. 30+28 is the overhead due to implementing parallelism and c is the extra unknown time taken to ensure proper scheduling of threads, that is the communication cost.

Best Case: Ω(nlogn+11n)

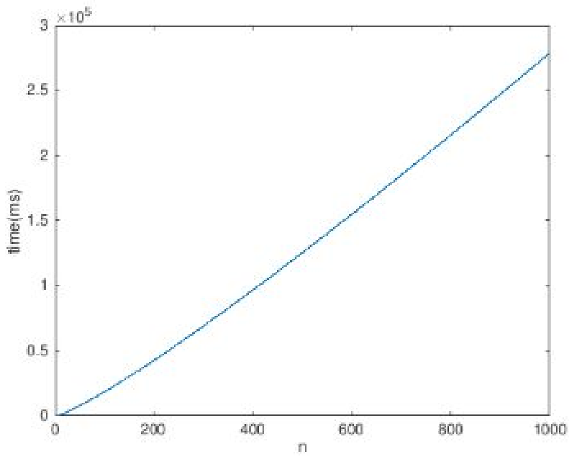

Fig 2.4: Graph depicting the complexity of the main function

III. Illustration

We start with taking inputs from the user which are: the size of the array, the option to input array or use a randomly generated array and finally the number of threads to be used.

Let us take an example array as shown in figure 3.1.
And let the number of threads be two. Following the procedure of code we will first calculate the length of the array to be passed in each thread by the formula: size/thread_count which here evaluates to two, thus we pass array [6,2] and [9,5] shown in the figure to the two threads created.

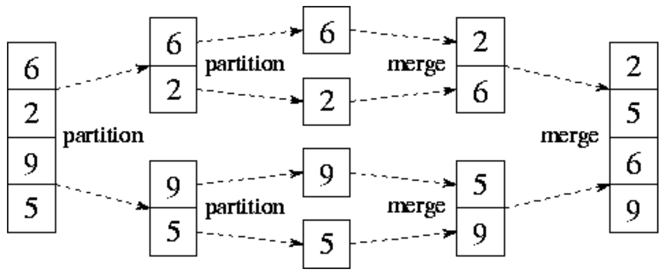

Fig 3.1: Figure illustrating Merge Sort

Now, these two threads will recursively call the merge sort function on the arrays they have to sort each array. When each array is sorted in both the threads we will have [2,4] and [5,9] as shown in the figure. Next, we finally merge the two arrays to form the final sorted array.

IV. Comparison with Sequential Approach

Merge sort can simply be implemented without any parallel programming. This implementation is quite similar to the algorithm implemented here.
Instead of dividing the work to different threads, only a single parent process performs all tasks. This approach is quite simple and easy to implement as there is no headache of managing the threads and dividing the work.
On one hand, the sequential approach is quite simple and readable, on the other hand, parallel programming can be more efficient if used in conjunction with the sequential approach wisely.
For small input sizes, parallel implementation and sequential implementation perform nearly equal or sequential may perform better because of the overhead of context switches.
A pure parallel approach is not the best approach for the merge sort implementation because a lot of threads are formed and at the last steps there are too many threads which may decrease computation cost but increases the overhead of communication among threads and context switch overhead in increased heavily.

Mixed use of the parallel and sequential approach is best which optimizes the costs of computation and communication effectively.

V. Result

Overall Time Complexity

Time Complexity of the algorithm is not equal to the actual the time required to execute particular code but the number of times a statement executes.
We have here used the Parallel approach to solve this Merge sort problem.
The sequential time complexity is O(nlogn). In the case of the parallel algorithm, the complexity involves both communication cost and computational cost. So, on one hand, the time reduces because computation occurs parallelly but the communication overhead is added.

Sometimes this communication overhead may exceed the time saved and parallel may take more time than sequential one.
The algorithm is dividing the array each time into two halves. If the array is composed of 1024 elements, it means the first execution handles 1024 elements, then the second one 512, then 256 etc. Until the last executions which have to perform a task with only 2 elements.

It means that we will have a bunch of tasks consisting of simply sorting two integers. Those tasks are going to be available for work-stealing which results in causing a huge performance penalty.

Indeed, it is way faster to have one thread performing two times a simple job rather than having two threads synchronizing each other to split and share this job. Hence this anomaly.

Space Complexity :O(n)

In merge sorting when we are merging the two sorted arrays we create two temporary arrays. L[ ]=Arr[left,mid](left array)to temporarily store the old array from left to mid(sorted left half) and R[ ]=Arr[mid+1,right](right array)to temporarily store the old array from mid+1 to right(sorted right half) ,then we merge the two temporary array into the original one .
The fact that we create two temporary arrays to store the numbers of the original since the original array has n elements the temporary arrays are of size n respectively and hence the extra space of n and an O(n) space complexity.

VI. Conclusion

The algorithm for the problem to sort the array using merge sort implemented through parallel programming is given above. On the basis of test cases and complexities, it can be argued that the given solution is the best possible solution.

Here, the number of threads used is finite and is almost equal to the number of processors. However, on the basis of capability of machine and processor, the number of threads in the solution can be increased. This may or may not increase the benefits of multiprogramming or parallel programming.

At first, it may look like the greater extent of parallel programming is always for the betterment but it may not be the case. Too much threading can be harmful. This is because of the limited cache memory and scheduling of multiple processes.

Therefore, the number of threads used in the algorithm is just perfect such that it does not burden the system and serves the purpose.
So the algorithm proposed is efficient and reliable than all nearby approaches.

VII. References

[1] ”Parallel computing”, Wikipedia [Online]
Available:- https://en.wikipedia.org/wiki/Parallel_computing
[Accessed: April 20, 2019]

[2] ”Merge Sort”, GeeksforGeeks [Online]
Available:-https://www.geeksforgeeks.org/merge-sort/
[Accessed: April 20, 2019]

[3] “Multithreading in C”, GeeksforGeeks [Online]
Available:- https://www.geeksforgeeks.org/multithreading-c-2/
[Accessed: April 20, 2019]

[4] Thomas H. Corman, Introduction to Algorithms, 3rd Edition 2003

[5] Ellis Horowitz,Sartaj Sahni,Sanguthevar Rajasekaran,Fundamentals of Computer Algorithms, 2nd Edition