SCTR's Pune Institute of Computer Technology
Dhankawadi, Pune

A PROJECT REPORT ON “Merge Sort and Multithreaded Merge Sort”

SUBMITTED BY
41141 Kalme Akash Namdev
41124 Dhawale Harsh Vijay

Under the guidance of
Prof. S. W. Jadhav

DEPARTMENT OF COMPUTER ENGINEERING
Academic Year 2023-24

DEPARTMENT OF COMPUTER ENGINEERING

SCTR's Pune Institute of Computer Technology
Dhankawadi, Pune, Maharashtra 411043

CERTIFICATE

This is to certify that the SPPU Curriculum-based Mini Project titled 'Merge Sort and Multithreaded Merge Sort'

Submitted by

41141	Akash Kalme
41124	Harsh Dhawale

has satisfactorily completed the curriculum-based Mini Project under the guidance of Prof. S. W. Jadhav towards the partial fulfillment of the final year of Computer Engineering Semester VII, Academic Year 2023-24 of Savitribai Phule Pune University.

Date:

Place: PUNE

Name & Sign of Project Guide:

Acknowledgment

It gives me great pleasure to present the mini project on - Build a machine learning model that predicts the type of people who survived the Titanic shipwreck using passenger data (i.e., name, age, gender, socio-economic class, etc.).

First, I would like to take this opportunity to thank my guide Prof. S. W. Jadhav for giving me all the help and guidance needed. I am grateful for his kind support and valuable suggestions that proved to be beneficial in the overall completion of this project.

I am thankful to our Head of the Computer Engineering Department, Dr. G. V. Kale, for her indispensable support and suggestions throughout the internship work. I would also genuinely like to express my gratitude to the CC, Prof. Samadhan Jadhav, for his constant guidance.

Finally, I would again like to thank my mentor, Prof. S. W. Jadhav, for his constant help and support during the overall process.

❖ Title

Merge Sort and Multithreaded Merge Sort.

❖ Problem Statement

Implement merge sort and multithreaded merge sort. Compare time required by both the algorithms. Also analyse the performance of each algorithm for the best case and the worst case.

❖ Objectives

The primary objective of this report is to compare and analyze the Merge Sort and Multithreaded Merge Sort techniques:

- Implement merge sort using multi-threading.
- To analyze the performance of the multi-threading approach on merge sort.
- To analyze the complexity performance of the multi-threading approach on merge sort.

❖ Theory and Algorithms

The **Merge Sort** algorithm is a sorting algorithm that is based on the **Divide and Conquer** paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

- **Merge Sort Working Process:**

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

- **Merge Function:**

In the merging function, we use three *while* loops. The first one is to iterate over the two parts together. In each step, we take the smaller value from both parts and store it inside the *temp* array that will hold the final answer.

Once we add the value to the resulting *temp*, we move the *index* one step forward. The variable *index* points to the index that should hold the next value to be added to temp.

In the second *while* loop, we iterate over the remaining elements from the first part. We store each value inside *temp*. In the third *while* loop, we perform a similar operation to the second *while* loop. However, here we iterate over the remaining elements from the second part.

The second and third *while* loops are because after the first *while* loop ends, we might have remaining elements in one of the parts. Since all of these values are larger than the added ones, we should add them to the resulting answer.

The complexity of the merge function is $O(len1 + len2)$, where *len1* is the length of the first part, and *len2* is the length of the second one.

Note that the complexity of this function is linear in terms of the length of the passed parts. However, it’s not linear compared to the full array A because we might call the function to handle a small part of it.

```
Algorithm 1: Merge Function
  Data: A: The array to be sorted
        L1: The start of the first part
        R1: The end of the first part
        L2: The start of the second part
        R2: The end of the second part
  Result: Return the merged sorted array

  Function $merge(A, L1, R1, L2, R2)$:
      $temp \leftarrow \{\}$;
      $index \leftarrow 0$;
      while $L1 \leq R1$ AND $L2 \leq R2$ do
          if $A[L1] \leq A[L2]$ then
              $temp[index] \leftarrow A[L1]$;
              $index \leftarrow index + 1$;
              $L1 \leftarrow L1 + 1$;
          else
              $temp[index] \leftarrow A[L2]$;
              $index \leftarrow index + 1$;
              $L2 \leftarrow L2 + 1$;
          end
      end
      while $L1 \leq R1$ do
          $temp[index] \leftarrow A[L1]$;
          $index \leftarrow index + 1$;
          $L1 \leftarrow L1 + 1$;
      end
      while $L2 \leq R2$ do
          $temp[index] \leftarrow A[L2]$;
          $index \leftarrow index + 1$;
          $L2 \leftarrow L2 + 1$;
      end
      return $temp$;
  end
```

- **Merge Sort:**

Firstly, we start *len* from *1* which indicates the size of each part the algorithm handles at this step.

In each step, we iterate over all parts of size *len* and calculated the beginning and end of each two adjacent parts. Once we determined both parts, we merged them using the *merge* function defined in algorithm 1.

Note that we handled two special cases. The first one is if *L2* reaches the outside of the array, while the second one is when *R2* reaches the outside. The reason for these cases is that the last part may contain fewer than *len* elements. Therefore, we adjust its size so that it doesn't exceed *n*.

After the merging ends, we copy the elements from *temp* into their respective places in A.

Note that in each step, we doubled the length of a single part *len*. The reason is that we merged two parts of length *len*. So, for the next step, we know that all parts of the size $2 \times len$ are now sorted.

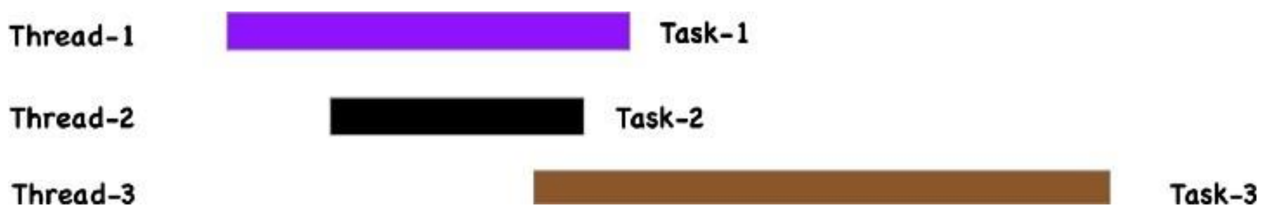
Finally, we return the sorted *A*.

The complexity of the iterative approach is $O(n \times log(n))$, where *n* is the length of the array.

The reason is that, in the first while loop, we double the value *len* in each step. So, this is $O(log(n))$. Also, in each step, we iterate over each element inside the array twice and call the *merge* function for the complete array total. Thus, this is $O(n)$
.

```
Algorithm 2: Iterative Merge Sort
  Data: A: The array
        n : The size of the array
  Result: Returns the sorted array
  $len \leftarrow 1$;
  while $len < n$ do
      $i \leftarrow 0$;
      while $i < n$ do
          $L1 \leftarrow i$;
          $R1 \leftarrow i + len - 1$;
          $L2 \leftarrow i + len$;
          $R2 \leftarrow i + 2 \times len - 1$;
          if $L2 \geq n$ then
              break;
          end
          if $R2 \geq n$ then
              $R2 \leftarrow n - 1$;
          end
          $temp \leftarrow merge(A, L1, R1, L2, R2)$;
          for $j \leftarrow 0$ to $R2 - L1 + 1$ do
              $A[i + j] \leftarrow temp[j]$;
          end
          $i \leftarrow i + 2 \times len$;
      end
      $len \leftarrow 2 \times len$;
  end
  return $A$;
```

- **Multi-threaded Merge sort:**



We employ divide-and-conquer algorithms for parallelism because they divide the problem into independent subproblems that can be addressed individually. Let's take a look at merge sort:

```
Algorithm 6: Merge Sort (parallel)
Merge-Sort(a, p, r)
if $p < r$ then
    $q = (p + r)_{\bar{2}}$
    spawn Merge-Sort(a, p, q)
    Merge-Sort(a, q+1, r)
    sync
    Merge(a, p, q, r)
```

As we can see, the dividing is in the main procedure *Merge-Sort*, then we parallelize it by using spawn on the first recursive call. *Merge* remains a serial algorithm, so its work and span are $\theta(n)$as before.

- **Complexity Analysis**

The dividing is in the main procedure *Merge-Sort*, then we parallelize it by using spawn on the first recursive call. *Merge* remains a serial algorithm, so its work and span are $\theta(n)$ as before.

Here's the recurrence for the work $T_1(n)$ of *Merge-Sort* (it's the same as the serial version):

$$T_1(n) = 2 \times T_1(\frac{n}{2}) + \theta(n) = \theta(n \times \log(n))$$

The recurrence for the span $T_\infty(n)$ of *Merge-Sort* is based on the fact that the recursive calls run in parallel:

$$T_\infty(n) = T_\infty(\frac{n}{2}) + \theta(n) = \theta(n)$$

Here's the parallelism:

$$\frac{T_1(n)}{T_\infty(n)} = \theta(\frac{n \times \log(n)}{n}) = \theta(\log(n))$$

As we can see, this is low parallelism, which means that even with massive input, having hundreds of processors would not be beneficial. So, to increase the parallelism, we can speed up the serial *Merge*.

For Example-:

1. **Input** −int arr[] = {3, 2, 1, 10, 8, 5, 7, 9, 4}

   Output −Sorted array is: 1, 2, 3, 4, 5, 7, 8, 9, 10

   Explanation −We are given an unsorted array with integer values. Now we will sort the array using merge sort with multithreading.

2. **Input** −int arr[] = {5, 3, 1, 45, 32, 21, 50}

   Output −Sorted array is: 1, 3, 5, 21, 32, 45, 50

   Explanation −We are given an unsorted array with integer values. Now we will sort the array using merge sort with multithreading.

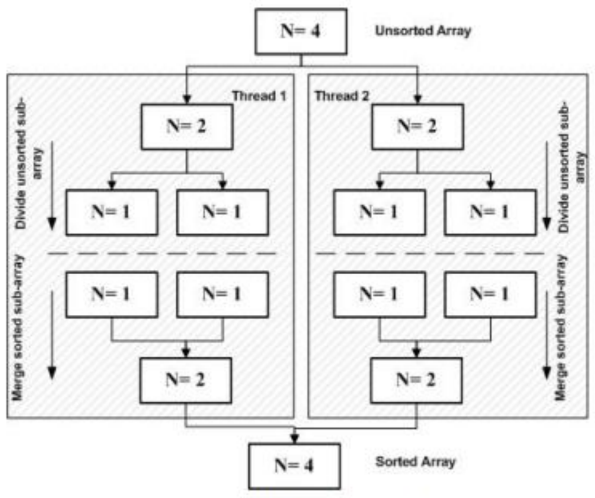


Fig. . Multithread Merge Sort

❖ **Results**

It was observed that the multi-threaded approach always had a better performance time than the Single Threaded approach. The difference in their performance started becoming visible as the input sizes began growing.

The following data has been calculated by varying the length of the array while keeping the other parameters constant with the values: Number of Multiprocessing cores = 4

Array Length	Single-Threaded	Multi-Threaded
1	2.934e-06	0.0285377
10	2.2908e-05	0.0246235
100	0.000198492	0.0248281

1000	0.00220744	0.0421945
10000	0.0243417	0.0359442
100000	0.297823	0.166326
1000000	3.68507	1.23563
10000000	45.5687	11.9969

Table 1. Result of the program on different parameters of Input in seconds

❖ Conclusion

In this project we have successfully implemented merge sort by using a multithreading approach. We have also analyzed the performance and complexity of merge sort using a multithreading approach. It was observed that the multi-threaded approach performs better than the naive approach by a great margin.