OpenMP + MPI based implementation of Merge Sort Algorithm

Hiten Padaliya
DA-IICT
Gandhinagar, India
201901401@daiict.ac.in

Piyush Suthar
DA-IICT
Gandhinagar, India
201901404@daiict.ac.in

Satyam Bhut
DA-IICT
Gandhinagar, India
201901408@daiict.ac.in

Jinesh Kanjiya
DA-IICT
Gandhinagar, India
201901412@daiict.ac.in

Harshil Soni
DA-IICT
Gandhinagar, India
201901416@daiict.ac.in

Jenish Rathod
DA-IICT
Gandhinagar, India
201901435@daiict.ac.in

***Abstract*—Merge-Sort is the efficient way of sorting any kind of data and any variation of data like random, sorted, or reverse sorted data in the deterministic amount of time. This report consists of the complete analysis of serial merge sort. The focus of this study lies in the assertion that parallelizing the sorting process is faster and more useful for multi-core systems. The result demonstrates how the different serial approaches have an impact on sorting performance.**

***Index Terms*—linear programming, data sorting, parallel algorithm, large-scale problem, merge sort, in-place, 3 way merge sort, OpenMP**

I. Introduction

In computer science, merge sort is an efficient, general-purpose, and comparison-based sorting algorithm [15]. Most implementations produce a stable sort, which means that the order of equal elements is the same in the input and output. Merge sort was invented by John von Neumann in 1945 [13]. A detailed description and analysis of bottom-up merge sort appeared in a report by Goldstine and von Neumann as early as 1948 [14].

Merge Sort Algorithm belong to Divide And Conquer Algorithm like Quick Sort. The merge sort's overall concept is to split down an input array into numerous sub-arrays until each subarray has only one element, then merge those subarrays into a sorted list. Here an array is a collection of similar data elements stored at contiguous memory locations. It's time complexity is $O(nlog(n))$ is better than insertion sort, selection sort and bubble sort. It's important because for an input "n large enough it will grow slower than other algorithms. However, conventional merge sort does not work inplace, which implies that each recursive call must make complete copies of the whole subarray. Insertion sort, on the other hand, just requires one array entry instead of copies of all array entries. In addition, the merge sort method is an excellent starting point for using the divide and conquer theory. It breaks the input array into two segments, calls the two halves recursively, and then merges the two sorted segments. Merging two segments is accomplished with the $merge()$ method. The $merge(arr, l, m, r)$ is a key process that assumes that for the given unsorted array $arr[l..r]$, the sub-segments $arr[l..m]$ and $arr[m+1..r]$ are already sorted and merges the two sorted sub-arrays into one. [2]

Merge sort is the very standard example of a divide and conquer algorithm but it has vast application domains. The primary use of merge sort is in the NO SQL databases where large-scale data needs to be computed with less computational complexity with the possibility of dividing tasks into multiprocessors because the ability to quickly arrange information, which is required for the analysis of acquired data and the production of relevant reports, is the primary concern with the NoSQL databases. Another application of the merge sort is in the field of eCommerce. It would imply the concept of inversions. The inversions in the array of items are defined as the number of pairs where the larger element comes before the smaller one. So this analogy is applied in eCommerce to recommend the product to users for a future buy. They generally kept an array of all the user accounts, and whichever one has the fewest inversions with the available options of product, they start proposing that product to the user. [8]

Background Reading:

1) Analysis of Parallel Mergesort [7]
The research presents a performance model for a fine-grained, parallel mergesort using O(logN) processors to sort N elements in O(N) time. Both the communication time required to merge the elements and the decomposition time required to activate and terminate the binary tree of processes are predicted by the model. The parallel algorithm is written in Joyce and executed on a Multimax Encore. Fine-grained parallelism is illustrated by the parallel mergesort. Its performance is constrained not only by the speed with which

processes communicate, but also by the overhead of creating new processes.

The approach activates a binary tree of parallel processes in a recursive manner. A finite, unordered sequence of elements is fed into the leaf, which splits it into sequences of length one. The branch processes gradually combine smaller, ordered sequences into larger, ordered sequences until the root process outputs a single, ordered sequence. The process tree is gradually terminating from the top down while this is going on.

2) Parallel Merge Sort with Load Balancing [10]
This paper starts informing the uses of the parallel merge sort algorithm and problems with the conventional parallel merge sort algorithm. Then the solution for the problem with the conventional merge sort is also described in this paper. The parallel merge sort is useful for progressively sorting a huge amount of data. The merge sort should be carefully parallelized, as the traditional technique has poor performance due to the gradual reduction of the number of participating processors by half, and finally to one in the final merging stage. All processors are used during the calculation in the proposed load-balanced merge sort. It distributes data to all processors in each stage in a consistent manner. As a result, each processor must work in all phases. Up to a speedup of $(P-1)/logP$, where P is the number of processors, significant performance improvements have been made.

3) Shared memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs [11]
While the theory of parallel merge sort is well established, little is known about how to implement parallel merge sort with standard parallel programming platforms like OpenMP and MPI and run it on mainstream SMP-based systems like multi-core computers and multi-core clusters. In this paper three parallel merge sorts are discussed: shared memory merge sort on SMP systems with OpenMP; message-passing merge sort on computer clusters with MPI; and combined hybrid merge sort on clustered SMPs with both OpenMP and MPI. On a dedicated Rocks SMP cluster and a virtual SMP lustre in the Amazon Elastic Compute Cloud, the parallel merge sort algorithms were tested. In the tests, the shared memory merge sort with OpenMP yielded the best speedup.

4) Fully Flexible Parallel Merge Sort for Multicore Architectures [8]
The advancement of multicore architectures has resulted in a new generation of processors that can divide tasks amongst their logical cores in a flexible manner. These require flexible, quick, and stable models of efficient algorithms.A new generation of efficient sorting algorithms may be able to help as these systems are designed to make efficient use of all available resources. Processes and calculations must be distributed across cores in a flexible manner to make the performance as high as possible. In this paper a fully flexible sorting method designed for parallel processing is presented. The concept described in this article is based on modified merge sort, which is developed for multicore architectures in parallel form. The uniqueness of the concept lies in the manner it is processed. A fully flexible approach that can be used with a variety of processors is created. To improve sorting efficiency, the workloads are flexibly distributed among logical cores.

5) Parallel Merge Sort Based Performance Evaluation and Comparison of MPI and PVM [12]
Parallel computing is based on the idea that complex problems may frequently be broken down into smaller ones that can then be tackled concurrently to save time (wall clock time) by utilising non-local resources and overcoming memory restrictions. Merge sort is a popular and easy-to-understand representation of the rich class of divide-and-conquer algorithms, therefore a better grasp of merge sort parallelization can help with divide-and-conquer parallelization in general. The major goal of this paper is to provide a standard single node model for MPI and PVM that shows how parallel merge sort performance is dependent on the RAM of the nodes (desktop PCs) utilised in parallel computing. A user, a master capable of processing requests from the user, and a slave capable of accepting problems from the master and returning the answer make up the single node model are included in the single node.The goal is to assess and compare these statistics in order to determine which of MPI and PVM performs better. In this paper comparison has been made with the serial execution in order to demonstarte the communication overhead involved in parallel computation. Also comparison and analysis for time statistics acquired for different RAM sizes under parallel execution utilising MPI and PVM in a single node with only two cores, one of which is the master and the other is the slave has been made in this paper.

Why Parallelization is required:
While tracking the flow of merge sort, we conclude that the data is divided into two parts, and both the parts are sorted independently in a single processor one after the other. During this time, the remaining part of the data is unutilized However, we can make this more efficient by allowing another processor to work upon the rest of the data and then combine the results of both processors. This approach saves much of our time. Further, we can think of many processors sorting the chunks of data simultaneously and then combining the result of all the processors. This motivates the parallelizing of the conventional merge sort algorithm using multicore processors.

Many coworking logical processors are supported by multicore architectures, which receive tasks allocated for parallel processing. Perfomance oriented advanced programming techniques are required in present day computing. We can improve the performance of algorithms

by parallelizing them. It must be implemented in such a way that it maintains an appropriate dissociation of concerns, which will help to prevent cross actions and interferences between involved processors. These aspects are crucial for the efficiency of database systems, information processing and data overwrites, where several processors are used in a flexible manner to enhance the efficiency [1]

Many researchers have contributed in the field of parallelization of conventional sorting algorithms. Aho, Hopcrof, and Knuth [1] presented classic versions of numerous sorting algorithms, three of which have had a significant impact on information processing development: heap sort, quick sort, and merge sort. These mechanisms are updated on a regular basis to ensure that they work as effectively as possible on modern architectures. The quick sort is enhanced by developing a new pivot method and avoiding deadlocks. Merge sort was also enhanced by some fresh concepts of sublinear methods and numerous parallelization approaches. Carlsson et al. [4] described a new sublinear approach for composing sorted substrings. Cole provided an overview of the theoretical foundations for the first parallel version approach [5]. Knuth ascribes the sequential mergesort to John von Neumann [6]. Many researchers have examined the parallel mergesort's running time, assuming that the cost of process generation is nominal. [7].

II. Serial Algorithm Development

The flowchart of conventional merge sort is shown in Fig.(1) and Fig.(2).

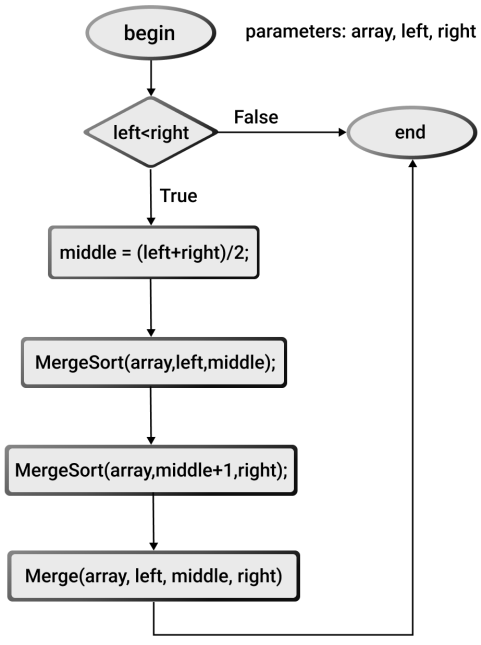


Fig. 1. mergeSort() Flowchart

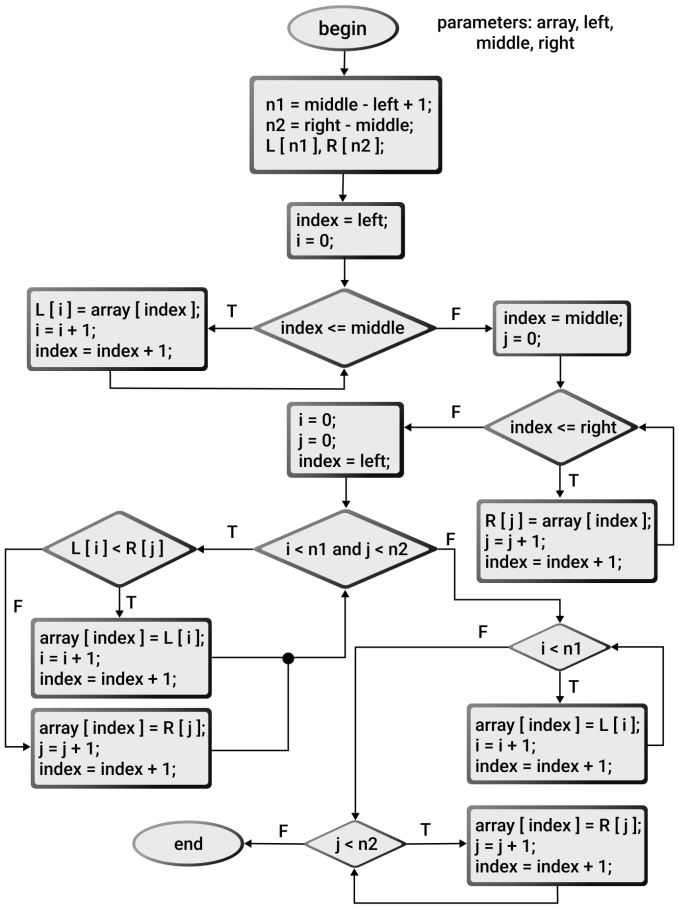


Fig. 2. Merge() Flowchart

Fig.(1) is the flowchart of the merge sort dividing and conquering technique. It takes the argument as complete input array and left and right variables denotes the portion in which the sorting takes place. (For the 1^{st} iteration $[left, right] = [0, n-1]$, where n is input size.) In the next step, if left if greater than or equal to right, the program returns to parent call. Else it will further divide the array in two halfs from the middle ($[left, middle]$ *and* $[middle+1, right]$) and recursively call the $mergeSort()$ on both the sub-arrays one after the other. When both the sub-array get sorted and returned, we will call $Merge()$ function to merge two sorted sub-arrays.

Fig.(2) is the flowchart of the $Merge()$ function. It takes the argument as complete input array and left, middle and right variables denotes the two consecutive sorted portion($[left, middle]$ *and* $[middle+1, right]$) which is to be merged. First we declared two arrays $L[n1]$ and $R[n2]$ of size $n1 = middle - left + 1$ and $n2 = right - middle$ respectively. Then we separate two sub-arrays as $L[0, n1-1] = array[left, middle]$ *and* $R[0, n2-1] = array[middle+1, right]$. Then we lake a loop on two separated arrays simultaneously until one of the array goes

out of bound. Inside the loop we check one-one elements of both the separated arrays and put the smaller element in the original array. After coming out of this loop, it is possible that, some elements from any one array will left out to get place in the original array. So, we will check which sub-array is left out by comparing the index of two sub-arrays with the corresponding length and place those elements linearly inside the original array. After completing this step, function terminates. This is how the two sorted consecutive sub-arrays are merged in a single sorted array.

The following three are the key process in the merge sort algorithm.

1) **Divide :-** First we divide the array in two parts at midpoint . So, we have to find the midpoint using first and last index .It takes constant time indicated by $\mathcal{O}(1)$.
2) **Conquer :-** Here, we recursive calls for $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ elements . If we use the symbolic representation T(n) for running time of merge sort on n element , then recursive calls will cost $T(\lfloor \frac{n}{2} \rfloor)$ and $T(\lceil \frac{n}{2} \rceil)$. So , approximate it is $2 * T(\frac{n}{2})$.
3) **Merge :-** This step combines all n elements . So, it will take $\mathcal{O}(n)$ time.

So, overall total complexity is,

$$T(n) = \mathcal{O}(1) + 2 \cdot T\left(\frac{n}{2}\right) + \mathcal{O}(n) \tag{1}$$

Now, we convert it to recurrence relation and then solve it.

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \tag{2}$$

Now, put the value of $T(\frac{n}{2})$ in Eq.(2)

$$T(n) = 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot n \tag{3}$$

Now, continuing this process to k steps such that $n = 2^k$, we got the following result,

$$\begin{aligned} T(n) &= 2^k \cdot T\left(\frac{n}{2^k}\right) + k \cdot n \\ T(n) &= n \cdot T(1) + n \cdot \log_2 n \\ &= n + n \cdot \log_2 n \\ \therefore T(n) &= n \cdot \log_2 n \end{aligned} \tag{4}$$

Hence, Time Complexity of Merge sort algorithm is $\mathcal{O}(n \log n)$

Pseudocode for mergeSort() and merge()

Algorithm 1 mergeSort(A,l,r)

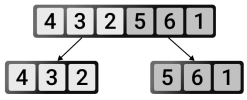
```
if l < r then
    mid ← (l + r)/2
    mergeSort(A, l, mid)
    mergeSort(A, mid + 1, r)
    Merge(A, l, mid, r)
end if
```

Algorithm 2 Merge(A,l,mid,r)

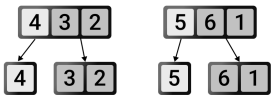
```
n1 ← mid − l + 1
n2 ← r − mid
Initiate L[n1] and R[n2]
for i = 1 to n1 do
    L[i] ← A[l + i − 1]
end for
for j = 1 to n2 do
    R[j] ← A[mid + j]
end for
i ← 1, j ← 1, k ← l
while i ≤ n1 and j ≤ n2 do
    if L[i] < R[j] then
        A[k] ← L[i]
        k ← k + 1, i ← i + 1
    else
        A[k] ← R[j]
        k ← k + 1, j ← j + 1
    end if
end while
while i ≤ n1 do
    A[k] ← L[i]
    k ← k + 1, i ← i + 1
end while
while j ≤ n2 do
    A[k] ← R[j]
    k ← k + 1, j ← j + 1
end while
```

III. Flowchart With Running Example

Now let's look at the working example of the merge sort. Initially we have taken the array of size 6 as A = [4, 3, 2, 5, 6, 1]. First of all we will cal the function $mergeSort(A, 0, 5)$.



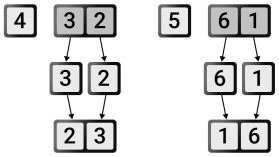
Now as the length of array is more than 1, it is divided from the middle and again call merge sort two time one after the other $[mergeSort(A, 0, 2)\ and\ mergeSort(A, 3, 5)]$



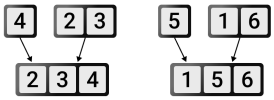
Again the length of both the subarrays are more than 1, so they will recursively call the merge sort dividing from the middle.

1) $A(0, 2) \rightarrow mergeSort(A, 0, 0)\ and\ mergeSort(A, 1, 2)$
2) $A(3, 5) \rightarrow mergeSort(A, 3, 3)\ and\ mergeSort(A, 4, 5)$

Now the subarray with length 1 will return as it is because it's sorted. But the subarrays with length 2 again call mergeSort dividing into two parts with length 1. And both the parts will return as it is and call the Merge function.

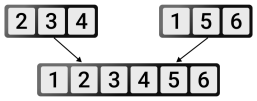


1) $A(1,2) \rightarrow mergeSort(A,1,1)\ and\ mergeSort(A,2,2)$
   a) $Merge(A,1,1,2)$
2) $A(4,5) \rightarrow mergeSort(A,4,4)\ and\ mergeSort(A,5,5)$
   a) $Merge(A,4,4,5)$



Now, we have sorted subarrays $A(1,2)$ and $A(4,5)$. These both will return at the place where they divided and get merged with whom it got divided.

1) $A(0,0)\ and\ A(1,2) \rightarrow Merge(A,0,0,2)$
2) $A(3,3)\ and\ A(4,5) \rightarrow Merge(A,3,3,5)$



Now, we have sorted subarrays $A(0,2)$ and $A(3,5)$. These both will return at the first place where we divide the original array into two parts. So, at last these both parts will get merged and we will get our final sorted array.

1) $A(0,2)\ and\ A(3,5) \rightarrow Merge(A,0,2,5)$

IV. Serial Code Development and Analysis

A. Hardware Specifications

The implementation of the code in done in Visual Studio. And the specification of the environment we have used to run and analyse the serial code is shown in Table.(I).

Properties	HPC Cluster	Remote PC
Model Name	Intel(R) Xeon(R) CPU E5-2640 v3 @2.60GHz	Intel® Core™ i5-4590H CPU @ 3.30GHz
Architecture	x86 64	x86 64
CPU(s)	16	4
Socket(s)	2	1
Core(s) / socket	6	4
L1 cache	32KB(L1d), 32KB(L1i)	32KB(L1d), 32KB(L1i)
L2 cache	256KB	256KB
L3 cache	15MB	6MB

TABLE I
Hardware Specifications of HPC Cluster and Remote PC.

B. Code Development

We have developed three variations of serial algorithms. The code for all the three approaches are available at *github*.

1) **Conventional Merge Sort Algorithm:** As developed in the Pseudocode.
   - Time Complexity : $\Theta(n \log_2 n)$
   - Space Complexity : $\Theta(n)$
2) **Three-Way Merge Sort Algorithm:** Instead of partitioning the data into two halves, we divide it into three parts.
   - Time Complexity : $\Theta(n \log_3 n)$
   - Space Complexity : $\Theta(n)$
3) **In-Place Merge Sort Algorithm:** Same as conventional algorithm without using any extra space.
   - Time Complexity : $\Theta(n \log_2 n)$
   - Space Complexity : $\Theta(1)$

C. Results And Analysis

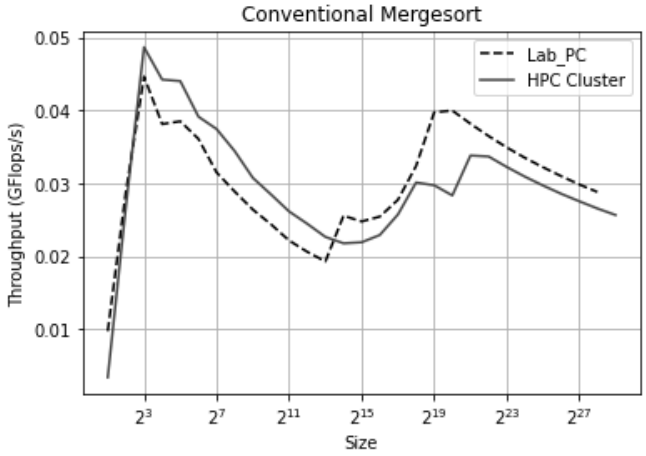


Fig. 3. Compute Throughput for conventional merge sort

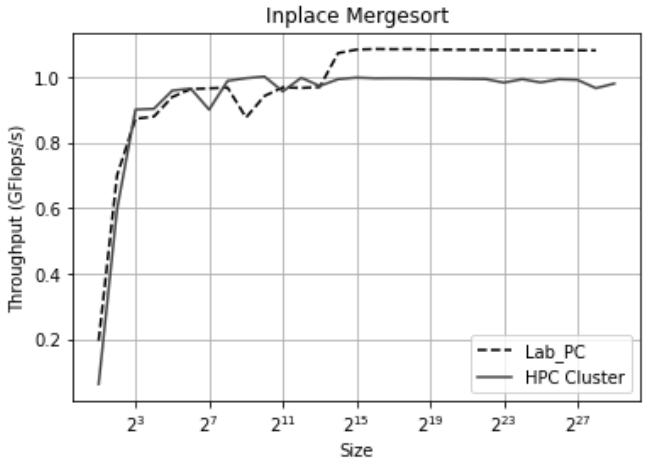


Fig. 4. Compute Throughput for conventional merge sort

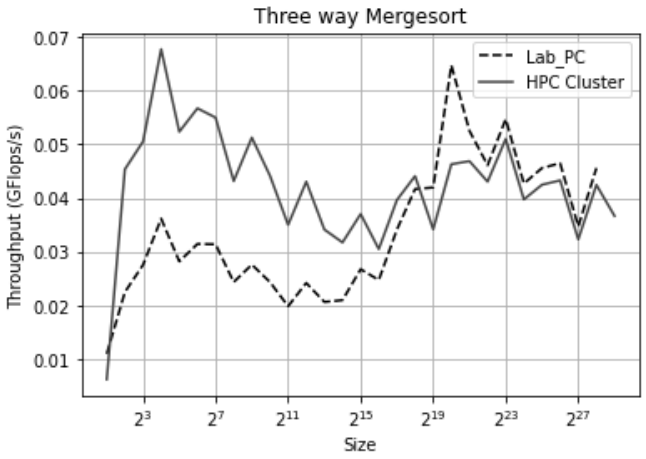


Fig. 5. Compute Throughput for conventional merge sort

The above three figures i.e. Fig.(3),Fig.(4), Fig.(5) represents the comparison of the compute throughput for the three approaches. Let's consider the throughput for the problem size of 2^{28} because considering smaller problem size does not hold significance in real life scenario. Hence it is clear that the throughput for the in-place merge sort is highest which is nearly around 1, then after comes the throughput for three way merge sort which is around 0.035 and lastly, for conventional merge sort the throughput is lowest which is around 0.025 for the problem size of 2^{28} in all the three cases in HPC Cluster. This is because of the following reasons:

In conventional merge sort and three way merge sort there is an extra array used for sorting the array where as in inplace merge sort there is no extra array used for sorting the array. Now definition of throughput is number of flops performed per second. In conventional merge sort and three way merge sort, there will be some time spent to bring the elements of the extra array into cache whenever there will be cache misses whereas in inplace merge sort as extra array is not present so, no additional time is spent for accessing the elements. As a result of this in inplace merge sort less time is consumed to sort the array compared to the other two methods. Hence less time consumed means that for same amount of floating point operations inplace merge sort takes less amount of time compared to other two methods and thus it has highest throughput.

Now comparing the value of throughput for the conventional merge sort and three way merge sort the value of throughput for three way merge sort is slightly higher than that of the conventional merge sort. This is because in three way merge sort the array gets divided into three equal parts whereas in conventional merge sort the array gets divided into two equal parts. Thus in three way merge sort the array gets divided into smaller chunk size as a result of which it can be brought entirely in cache and thus more flops will be performed per second compared to the conventional merge sort. Hence three way merge sort has slightly higher throughput compared to that of conventional merge sort.

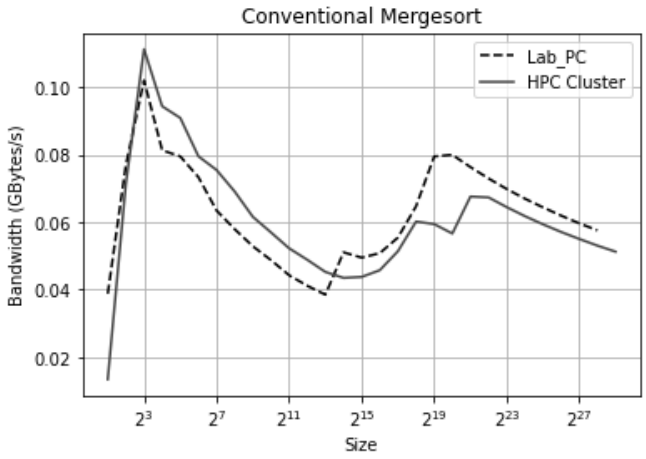


Fig. 6. Memory Throughput

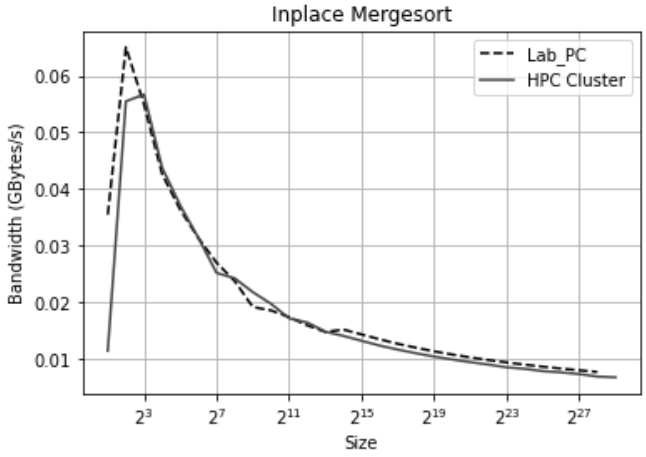


Fig. 7. Memory Throughput

From Table.(II), we can infer that the run time of all three approaches increases gradually. Also, the run-time of the 3-way and conventional approach are almost equal and much less than the in place approach. In 3 way merge, the number of recursive levels will be $log_3 n$, whereas it will be $log_2 n$ in the conventional approach. As the run time depends upon system architecture and the algorithm's asymptotic behavior, although the three-way merge sort uses more comparisons than the two-way merge sort, the 3-way merge sort has less run-time than the conventional merge sort. In general, the run-time of the 3-way merge sort and conventional approach are same, but the in-place merge sort have worst performance compared to the 3-way merge sort and conventional approach. Because the in-place approach performs more expensive operations like multiplication and modulo for merging the two arrays where the other two just

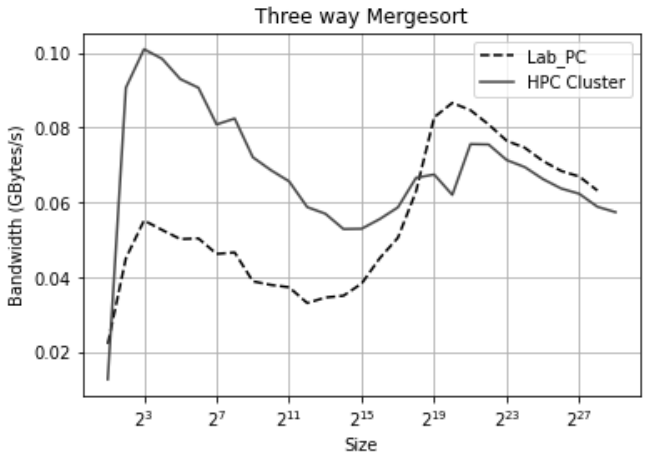


Fig. 8. Memory Throughput

Sample Size	Convectional MS	Three Way MS	In-place MS	SOTA MS
2^2	0.000001	0.000001	0.000001	0.000001
2^8	0.000055	0.000046	0.000079	0.000044
2^{12}	0.001249	0.001039	0.001859	0.000996
2^{16}	0.021334	0.017581	0.039594	0.017091
2^{18}	0.064910	0.058650	0.177953	0.056334
2^{20}	0.275902	0.251978	0.791002	0.230893
2^{22}	0.928370	0.827712	3.483545	0.785178
2^{24}	4.046232	3.600156	15.198929	3.451049
2^{26}	17.514639	15.692607	65.841499	14.946443
2^{28}	75.395624	67.990455	291.486625	67.604083

TABLE II
RUN TIME (IN SEC) ANALYSIS ON CLUSTER FOR SERIAL ALGORITHM

executes swap operations to merge the array. So generalizing for the k way merge sort, when we see at the mathematics then there isn't any asymptotic difference in the run time for different values of k and so the run time depends on the architecture of the system. As a result of this the conventional approach of merge sort is used on most of the cases. The last column refers to the results of state of the art. Here, the mergeSort function provided by the standard library template(STL) is considered state of the art. It is regarded as the most efficient implementation of the serial mergeSort now. The results are pretty close to the SOTA algorithm, as shown in the table.

D. Profiling:

The problem is compute-bound, where the number of elementary computation steps is the deciding factor, and the problem is said to be memory-bound, where the time to complete the given computational problem is decided primarily by the amount of memory required to hold the working data hence the problem is memory bound. Because the program executes no computation operations except finding the middle element, which takes 1 floating-point operation, other than that, the whole algorithm performs only swapping operations.

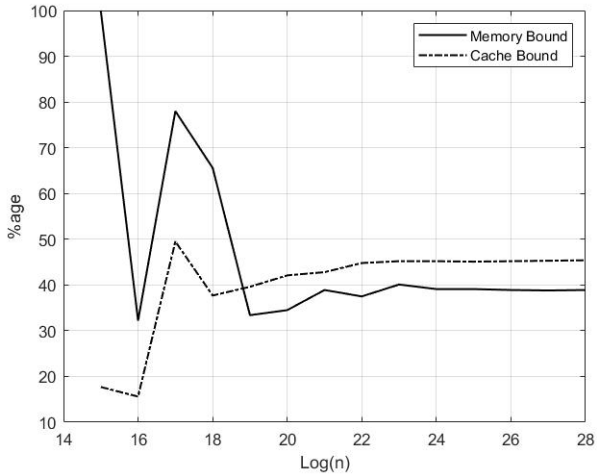


Fig. 9. Percentage vs Problem size (log(N))

To collect the profiling information about the algorithm, whether the algorithm is memory access bound or cache access bound, we used the vtune performance snapshot. Memory bound metric shows how memory subsystem issues affect performance. Memory bound measures the fraction of execution pipeline slots that could be stalled due to demand memory load and stores, whereas Cache bound shows how often the machine was stalled on L1, L2, and L3 caches. Fig(9) shows the percentage for memory and cache bound by varying the problem size to a significant amount. The performance is limited by the cache as the problem size increases; hence the problem is cache access bound.

Firstly we had done the recursive approach for the merge sort. Now in the recursive approach the recursion stack is used. As the stacks are located in RAM it takes more time for accessing the elements of the arrays and hence more time for sorting the array. So we had moved to the iterative approach of the merge sort. In the iterative approach of the merge sort we had used for loops to divide the arrays into sub arrays and then we used the standard merge procedure same as parallel merge procedure for merging the sub arrays. Thus in the iterative approach there is no use of the recursion stack because of which we got better performance compared to the iterative merge sort.

V. IMPLEMENTING PARALLEL ALGORITHM

After we performed profiling on the conventional approach we came across the fact that the merge function consumes around 28% of CPU time while running the serial code so it needs to be improved. But, we cannot parallelize the merge

function, since it has a loop carrying dependency. However, if this merge function is working on some part of the memory (let say on some subarray), it is completely independent from the other part of the memory(i.e., rest of the array). So, we can design the better algorithm, i.e, the parallel algorithm in which one or more processors are running the merge function simultaneously on the different part of the array. So we are now implementing an algorithm that has better speedup and scalability.

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems, including Solaris, AIX, FreeBSD, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior [16]. OpenMP is an implementation of multithreading, a method of parallelizing whereby a primary thread (a series of instructions executed consecutively) forks a specified number of sub-threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. [A thread is the smallest sequence of programmed instructions that can be managed independently by a scheduler.]

In OpenMP API the memory is shared among different processors. That is, every processor is using the same memory to sort the array. In the first approach we use the dynamic technique of creating threads to solve merge sort. In OPenMP API, the task pragma can be used to explicitly define a task and can be useful for parallelizing irregular algorithms such as pointer chasing or recursive algorithms. We will use the task pragma whenever we want to identify a block of code to be executed in parallel with the code outside the task region.

Let us first develop the pseudocode for the OpenMP algorithm. We are passing the address of both arrays (i.e., arr and dummy) to the function because the compiler uses the same address space to work with arrays rather than to make a new copy every time while calling the function. Dummy is used as a supporting array to merge two sorted arrays. Below, task and shared(arr) worked the same as discussed above. Now we have fixed the value of $TASK\_SIZE$ to some suitable integer(e.g., 100). Because the condition if(n> $TASK\_SIZE$) refers that if we have an array of size more than $TASK\_SIZE$, then it's feasible to generate a parallel region and divide the work among different threads; otherwise, if the array size is not sufficiently larger, then it is better to run the sequential code for that size of the array instead generate parallel region. Because when the input size is smaller parallel overhead limits the performance with creating, distributing tasks among threads, and joining. So if the condition is not satisfied, then the whole line acts like comments, and the code works serially.

bf Pseudocode for Parallel mergeSort() and merge()

Algorithm 3 Parallel_mergeSort(*arr,n,*dummy)

```
1: if n < 2 then
2:     return;
3: end if
4: end ← n/2
   #pragma omp task shared(arr) if( n > TASK_SIZE)
5: Parallel_mergeSort(arr, end, dummy)
   #pragma omp task shared(arr) if( n > TASK_SIZE)
6: Parallel_mergeSort(arr+end, n−end, dummy+end)
   #pragma omp taskwait)
7: Parallel_merge(arr, n, dummy)
```

Algorithm 4 Parallel_Merge(*arr,n,*dummy)

```
1: i ← 0, k ← 0
2: j ← n/2
3: while i < n/2 and j < n do
4:     if arr[i] < arr[j] then
5:         dummy[k] ← arr[i]
6:         k ← k + 1, i ← i + 1
7:     else
8:         dummy[k] ← arr[j]
9:         k ← k + 1, j ← j + 1
10:    end if
11: end while
12: while i < n/2 do
13:    dummy[k] ← arr[i]
14:    k ← k + 1, i ← i + 1
15: end while
16: while j < n do
17:    dummy[k] ← arr[j]
18:    k ← k + 1, j ← j + 1
19: end while
20: for q = 1 to N do
21:    arr[q] = dummy[q]
22: end for
```

VI. Parallel complexity

A. Time Complexity

We will find the complexity of the parallel mergeSort assuming that we have n number of processors.In the parallel mergeSort algorithm if the passed array size to the function. If the problem size is lesser than TASK_SIZE then we are calling the serial mergeSort. Let us denote the complexity of the parallel mergeSort for n elements with n processors as $MS_n(n)$.The complexity is given by the recurrence as

Here we have taken n numbers of processors but in reality that is not possible so the complexity with p processors will be greater than what we are getting with n processors.

B. Space Complexity

The inplace parallel merge sort will not take any extra memory other than the given input array, hence space complexity will be $\Theta(1)$ whereas in the conventional parallel merge sort we are taking another array to store the sorted output array,so the space complexity will be $\Theta(n)$.

C. Results and Analysis of Parallel Implementation

The speedups obtained for the three implementations of parallel merge sort are shown below;

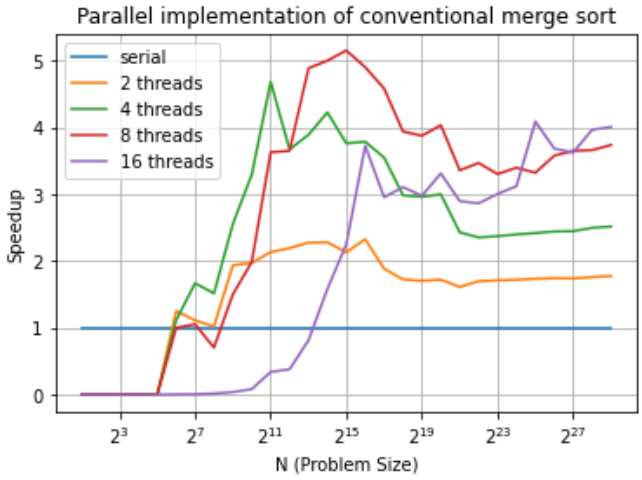


Fig. 10. Speedup vs Problem size

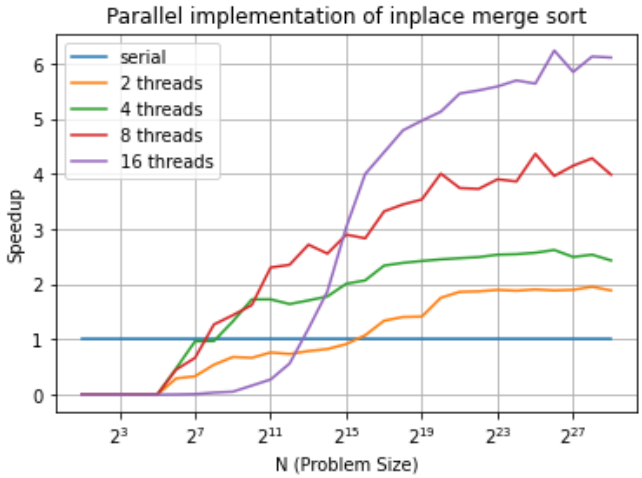


Fig. 11. Speedup vs Problem size

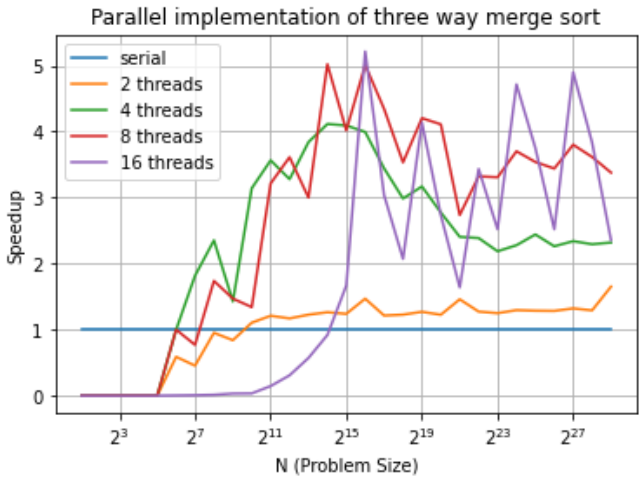


Fig. 12. Speedup vs Problem size

The graphs of efficiencies obtained for the three different implementations are as follows:

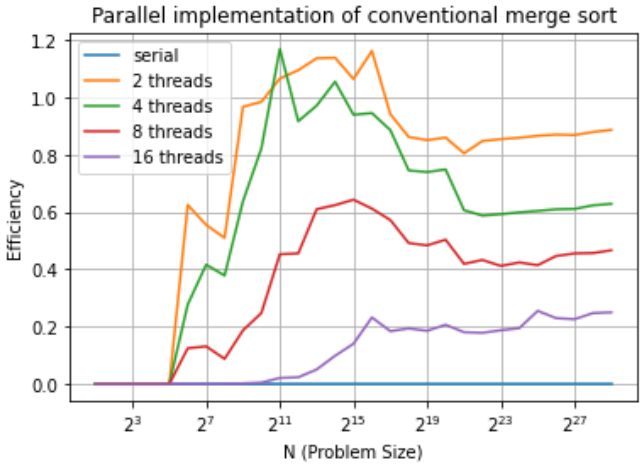


Fig. 13. Efficiency vs Problem size

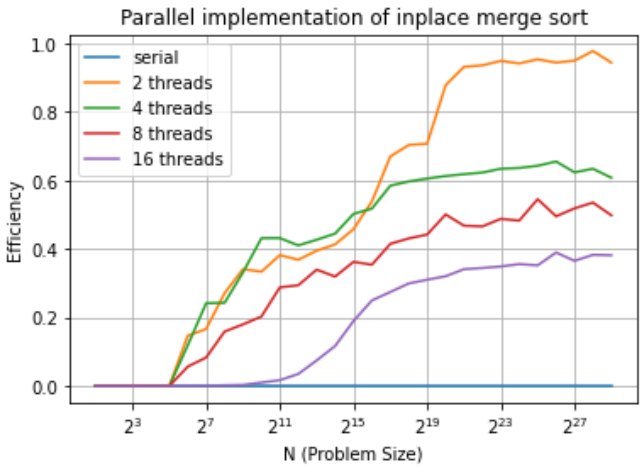


Fig. 14. Efficiency vs Problem size

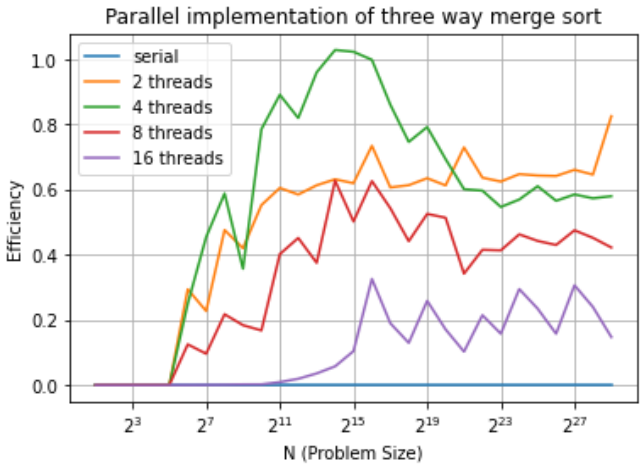


Fig. 15. Efficiency vs Problem size

p	Properties	Conventional	3-way	In-place
2	Efficiency	0.887165	0.825695	0.944437
	Speedup	1.774330	1.651391	1.888874
4	Efficiency	0.628815	0.580061	0.608364
	Speedup	2.515262	2.320246	2.433458
8	Efficiency	0.466832	0.422458	0.498612
	Speedup	3.734656	3.379669	3.988901
16	Efficiency	0.250245	0.147395	0.382011
	Speedup	4.003928	2.358320	6.112184

TABLE III
EFFICIENCY AND SPEEDUP FOR 3 DIFFERENT ALGORITHMS FOR LARGER INPUT LIKE 2^{29}

D. Conclusion

We demonstrated the efficiency and speedup of the merge sort algorithm using the serial implementation as well as openMPI parallel implementation. Here we had implemented the three versions of the merge sort algorithm which are conventional merge sort, 3 way merge sort and inplace merge sort. We had tried to compare our implementation with the state of the art and the results obtained are depicted here. Here the conventional merge sort and three way merge sort uses the space complexity of $O(n)$ for sorting the entire input array whereas the inplace merge sort uses the constant space to sort the input array and thus the speedup obtained for the inplace merge sort is highest for 16 threads compared to other implementations. Thus we observed that compared to all the three versions of the merge sort the inplace version has the best performance in terms of speedup for both the case i.e. serial and parallel.

REFERENCES

[1] Zbigniew Marszalek, Marcin Wozniak, and Dawid Polap, “Fully Flexible Parallel Merge Sort for Multicore Architectures,” Institute of Mathematics, Silesian University of Technology, Kaszubska 23, 44-100 Gliwice, Poland, 2 December 2018.

[2] GeeksForGeeks, “Merge Sort", https://www.geeksforgeeks.org/merge-sort/.

[3] H. Bing-Chao and D. E. Knuth,“A one-way, stackless quicksort algorithm" BIT. Nordisk Tidskrif for Informations behandling (BIT), vol. 26.

[4] S. Carlsson, C. Levcopoulos, and O. Petersson,“Sublinear merging and natural mergesort" Algorithmica. An International Journal in Computer Science, vol. 9, 1993.

[5] R. Cole, “Parallel merge sort" SIAM Journal on Computing, vol.17, no. 4, 1988.

[6] D.E. Knuth, “The Art Of Computer Programming". vol.3, Sorting And Searching. Addison-Wesley, Reading, MA, 1973.

[7] Per Brinch Hansen, “Analysis of a Parallel Mergesort". School of Computer and Information Science Syracuse University, July 1989.

[8] Zbigniew Marszalek, “Parallelization of Modified Merge Sort Algorithm", Institute of Mathematics, Silesian University of Technology, ul. Kaszubska 23, 44-100 Gliwice, Poland, 1 September 2017.

[9] Christopher Zelenka, “Parallel Merge Sort", Computer Science Department, San Jose State University, San Jose, CA 95192, 408-924-1000.

[10] Minsoo Jeon1 and Dongseung Kim1, "Merge Sort with Load Balancing", International Journal of Parallel Programming, Vol. 31, No. 1, February 2003 (2003)

[11] Atanas Radenski, "Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs", School of Computational Sciences, Chapman University, Orange, California, USA, the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, CSREA Press(H. Arbania, Ed.), 2011, pp. 367 - 373.

[12] Nanjesh B R, Tejonidhi M R, Rajesh T H, Ashwin Kumar H V, "Parallel Merge Sort Based Performance Evaluation and Comparison of MPI and PVM", Proceedings of 2013 IEEE Conference on Information and Communication Technologies (ICT 2013)

[13] Knuth, Donald (1998). “Section 5.2.4: Sorting by Merging". Sorting and Searching. The Art of Computer Programming. Vol. 3 (2nd ed.). Addison-Wesley. pp. 158–168.

[14] Katajainen, Jyrki; Träff, Jesper Larsson (March 1997). “A meticulous analysis of mergesort programs". Proceedings of the 3rd Italian Conference on Algorithms and Complexity. Italian Conference on Algorithms and Complexity. Rome. pp. 217–228.

[15] Merge Sort. [Wikipedia] https://en.wikipedia.org/wiki/Merge_sort#cite_note-3

[16] OpenMP. [Wikipedia] https://en.wikipedia.org/wiki/OpenMP