Comparing Performance of Merge Sort in Serial and Parallel Environments

Manish V , Mohammad Rahil K A, Moulidhar B, Manoja U
USN: 1MS18CS067 , 1MS18CS072 , 1MS18CS075 , 1MS18CS068

Abstract

We must utilize all of the CPU's resources in order to get the most out of it. Sorting is the process of arranging a collection of numbers in ascending or descending order. To activate parallel programming, we apply a merging strategy in merge sort. To assess parallel programming's performance, we compare it to sequential merge sort, which does not make efficient use of the CPU. To determine whether an algorithm is more efficient, we evaluate the performance of sequential and parallel sorting algorithms. In a parallel sorting algorithm, we use a message passing interface (MPI).

Introduction

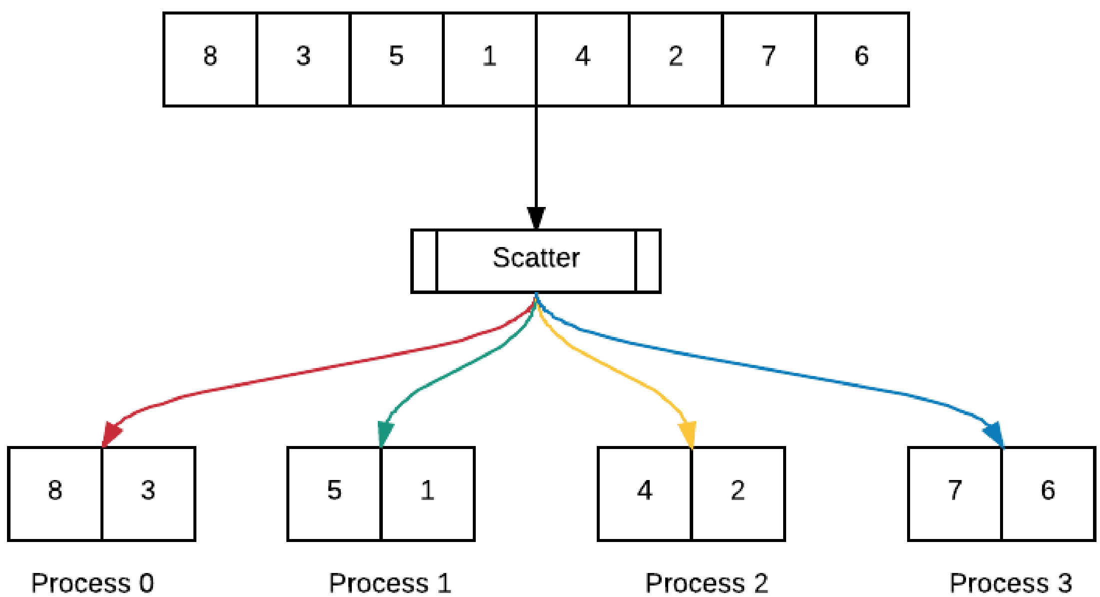
New applications always need faster processors in today's world to operate efficiently. Parallel systems are employed in a lot of business applications nowadays. These systems can handle massive amounts of data in a variety of ways, resulting in excellent efficiency. Parallel systems are all about breaking out distinct pieces of a program's instructions so that they can be executed on many CPUs at the same time.

Parallel systems, also known as tightly connected systems, are designed to reduce the execution time of programmes by dividing them into multiple fragments and processing them all at the same time. By constructing a parallel processing bundle or a combination of both entities, a parallel system can deal with many processors, machines, computers, or CPUs, among other things.

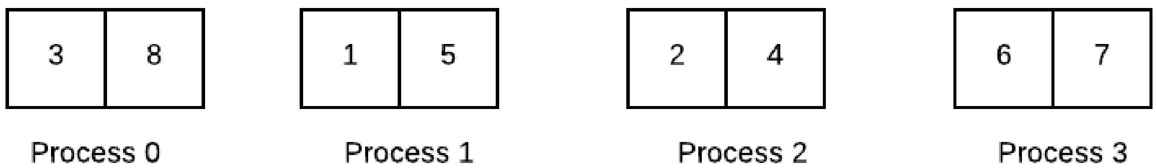
Parallel processing is commonly used to perform complex tasks and computations. Data scientists will commonly make use of parallel processing for compute and data-intensive tasks.

To parallelize this algorithm, we will use a mixed strategy in which the sublists are sorted by a sequential sorting algorithm and the merging of sublists is done in parallel between processes. We chose to stick with cases in which the number of processes is a power of two so that all processes are doing roughly the same amount of work.

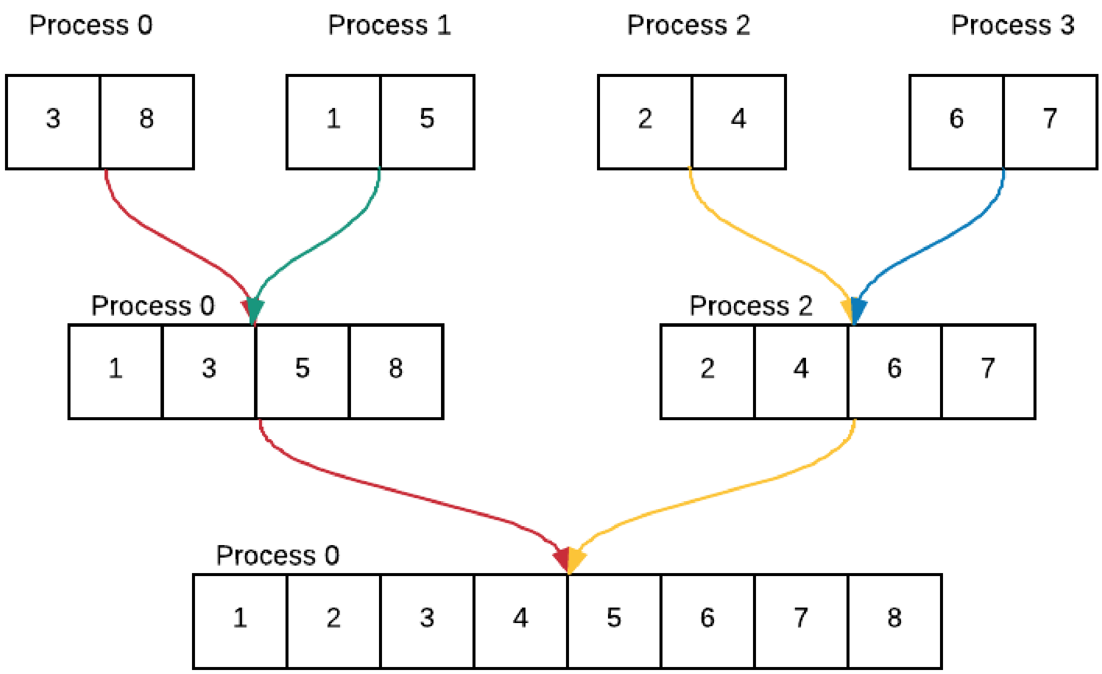
Part I: Divide list into unsorted sublists : For this portion of the problem, we begin with a single unsorted list. This list is scattered to all of the processes such that each process has an equal chunk of the list. Suppose we have 4 processes and a list containing 8 integers. The code is executing as follows:



Part II: Sort sublists : We can sort these sublists by applying a serial sorting algorithm. We use the C library function qsort on each process to sort the local sublist. After sorting the processes have the following sorted sublists:



Part III: Merge sublists : The merging of the sublists to form a single list is done by sending and receiving sublists between processes and merging them together. Each initial sorted sublist (with a size of 2) provides the sorted result to the parent process. That process combines the two sublists to generate a list of size 4, and then sends that result to its parent process. Lastly, the root process merges the two lists to obtain a list of size 8 that is fully sorted.



Code :

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <time.h>
#include <mpi.h>

void parallelMerge(int *, int *, int, int, int);
void parallelMergeSort(int *, int *, int, int);

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    /* create temp arrays */
    int L[n1], R[n2];
    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    /* Copy the remaining elements of L[], if there are any */
    while (i < n1) {
```

```
        arr[k] = L[i];
        i++;
        k++;
    }
    /* Copy the remaining elements of R[], if there are any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;
        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}
```

```
int main(int argc, char** argv) {
	/********** Create and populate the array **********/
	int n = atoi(argv[1]);
	int *original_array = malloc(n * sizeof(int));
	int c;
	srand(time(NULL));
	printf("This is the unsorted array: ");
	for(c = 0; c < n; c++) {
		original_array[c] = rand() % n;
		printf("%d ", original_array[c]);
		}
	printf("\n");
	printf("\n");
	/********** Initialize MPI **********/
	int world_rank;
	int world_size;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
	MPI_Comm_size(MPI_COMM_WORLD, &world_size);

	/********** Divide the array in equal-sized chunks **********/
	int size = n/world_size;
	/********** Send each subarray to each process **********/
	int *sub_array = malloc(size * sizeof(int));
	MPI_Scatter(original_array, size, MPI_INT, sub_array, size, MPI_INT, 0,
MPI_COMM_WORLD);
	/********** Perform the merge sort on each process **********/
	int *tmp_array = malloc(size * sizeof(int));
	parallelMergeSort(sub_array, tmp_array, 0, (size - 1));
	/********** Gather the sorted subarrays into one **********/
	int *sorted = NULL;
	if(world_rank == 0) {
		sorted = malloc(n * sizeof(int));
	}
```

```
	MPI_Gather(sub_array, size, MPI_INT, sorted, size, MPI_INT, 0,
MPI_COMM_WORLD);
	/********** Make the final mergeSort call **********/
	clock_t begining,end;
	if(world_rank == 0) {
		int *other_array = malloc(n * sizeof(int));
		begining = clock();
		parallelMergeSort(sorted, other_array, 0, (n - 1));
		end = clock();
		/********** Display the sorted array **********/
		printf("This is the sorted array: ");
		for(c = 0; c < n; c++) {
			printf("%d ", sorted[c]);
		}
		printf("\n");
		printf("\n");
		/********** Clean up root **********/
		free(sorted);
		free(other_array);
		}
	/********** Finalize MPI **********/
	MPI_Barrier(MPI_COMM_WORLD);
	MPI_Finalize();
	float elapsed = (float)(end - begining) / CLOCKS_PER_SEC * 1000;
	printf("Parallel needed %f miliseconds \n\n", elapsed);

	printf("Given array is \n");
	printArray(original_array, n);
	clock_t s_begining = clock();
	mergeSort(original_array, 0, n - 1);
	clock_t s_end = clock();
	printf("\nSorted array is \n");
	printArray(original_array, n);

	float s_elapsed = (float)(s_end - s_begining) / CLOCKS_PER_SEC * 1000;
```

```
	printf("Serial needed %f miliseconds \n\n", s_elapsed);
	free(original_array);
	free(sub_array);
	free(tmp_array);
}

/********** Merge Function **********/
void parallelMerge(int *a, int *b, int l, int m, int r) {
	int h, i, j, k;
	h = l;
	i = l;
	j = m + 1;

	while((h <= m) && (j <= r)) {

		if(a[h] <= a[j]) {

			b[i] = a[h];
			h++;
		}
		else {
			b[i] = a[j];
			j++;
		}
		i++;
	}
	if(m < h) {
		for(k = j; k <= r; k++) {
			b[i] = a[k];
			i++;
		}
		}
	else {
		for(k = h; k <= m; k++) {
			b[i] = a[k];
			i++;

		}
	}
```

```
        for(k = l; k <= r; k++) {
                a[k] = b[k];
        }
}

/********** Recursive Merge Function **********/
void parallelMergeSort(int *a, int *b, int l, int r) {
        int m;
        if(l < r) {
                m = (l + r)/2;
                parallelMergeSort(a, b, l, m);
                parallelMergeSort(a, b, (m + 1), r);
                parallelMerge(a, b, l, m, r);

        }
}
```

Output:

This is the unsorted array: 42 44 0 81 63 10 60 27 41 40 13 33 32 30 1 55 95 4 59 33 25 28 40 70 39 76 45 97 23 69 58 17 66 58 50 81 68 10 61 61 51 26 47 35 56 48 43 3 4 54 88 30 83 29 52 22 5 98 71 80 19 29 97 85 39 99 19 59 9 80 20 12 6 67 0 14 67 43 17 72 49 58 2 32 87 6 6 92 4 29 24 24 10 73 61 49 72 32 8 81

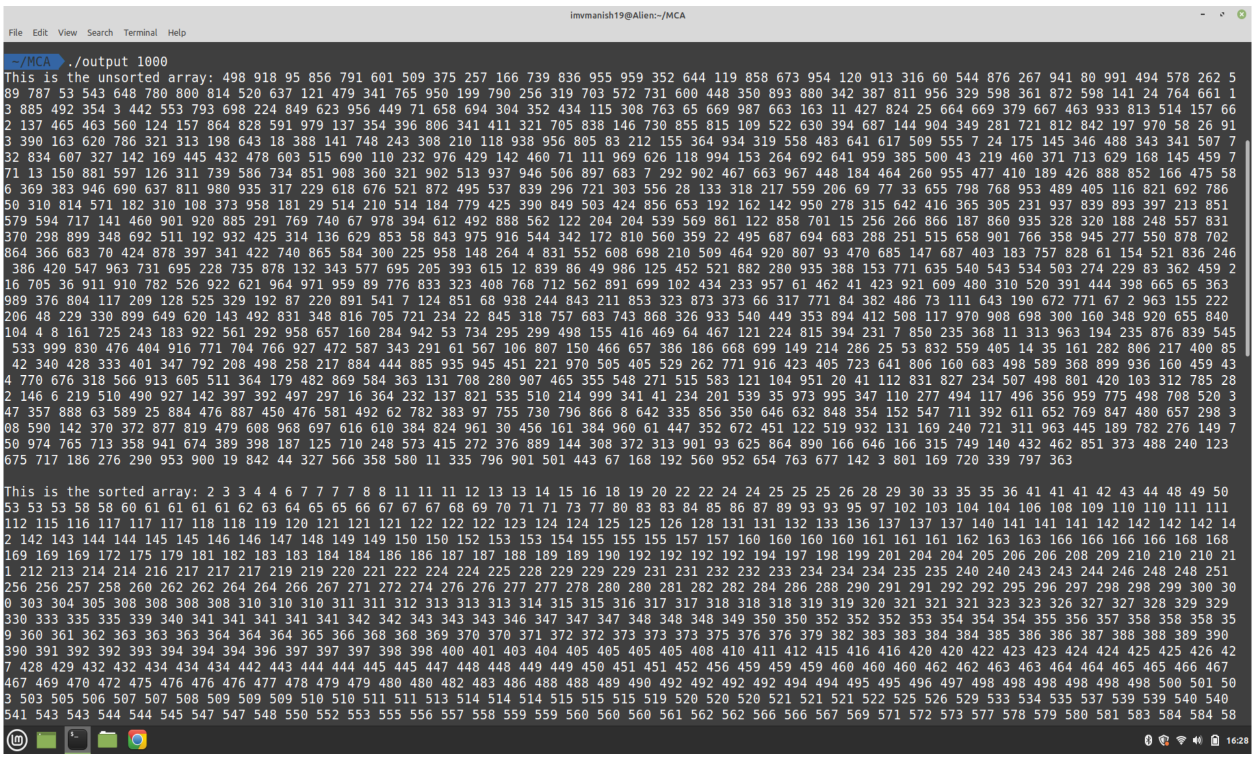
This is the sorted array: 0 0 1 2 3 4 4 4 5 6 6 6 8 9 10 10 10 12 13 14 17 17 19 19 20 22 23 24 24 25 26 27 28 29 29 29 30 30 32 32 32 33 33 35 39 39 40 40 41 42 43 43 44 45 47 48 49 49 50 51 52 54 55 56 58 58 58 59 59 60 61 61 61 63 66 67 67 68 69 70 71 72 72 73 76 80 80 81 81 81 83 85 87 88 92 95 97 97 98 99
Parallel needed 0.038000 milliseconds

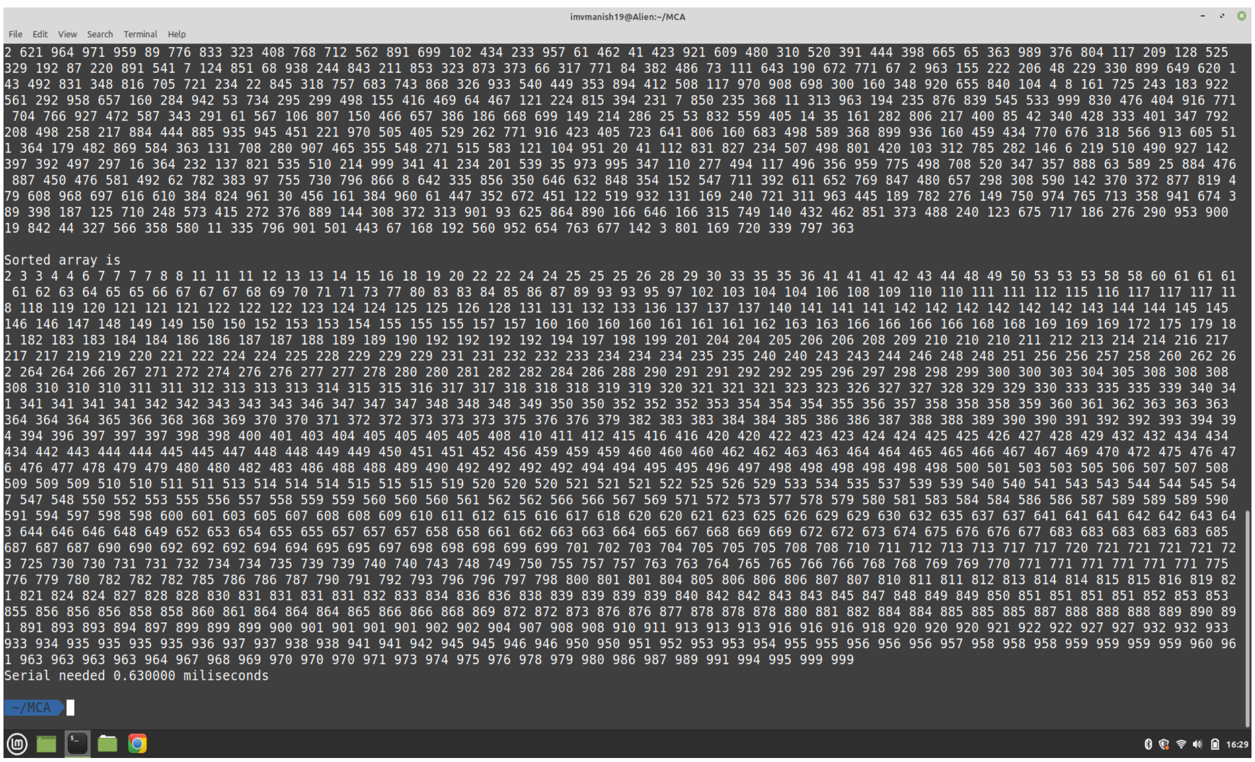
Given array is
42 44 0 81 63 10 60 27 41 40 13 33 32 30 1 55 95 4 59 33 25 28 40 70 39 76 45 97 23 69 58 17 66 58 50 81 68 10 61 61 51 26 47 35 56 48 43 3 4 54 88 30 83 29 52 22 5 98 71 80 19 29 97 85 39 99 19 59 9 80 20 12 6 67 0 14 67 43 17 72 49 58 2 32 87 6 6 92 4 29 24 24 10 73 61 49 72 32 8 81

Sorted array is

0 0 1 2 3 4 4 4 5 6 6 6 8 9 10 10 10 12 13 14 17 17 19 19 20 22 23 24 24 25 26 27 28 29 29 29 30 30 32 32 32 33 33 35 39 39 40 40 41 42 43 43 44 45 47 48 49 49 50 51 52 54 55 56 58 58 58 59 59 60 61 61 61 63 66 67 67 68 69 70 71 72 72 73 76 80 80 81 81 81 83 85 87 88 92 95 97 97 98 99

Serial needed 0.071000 milliseconds

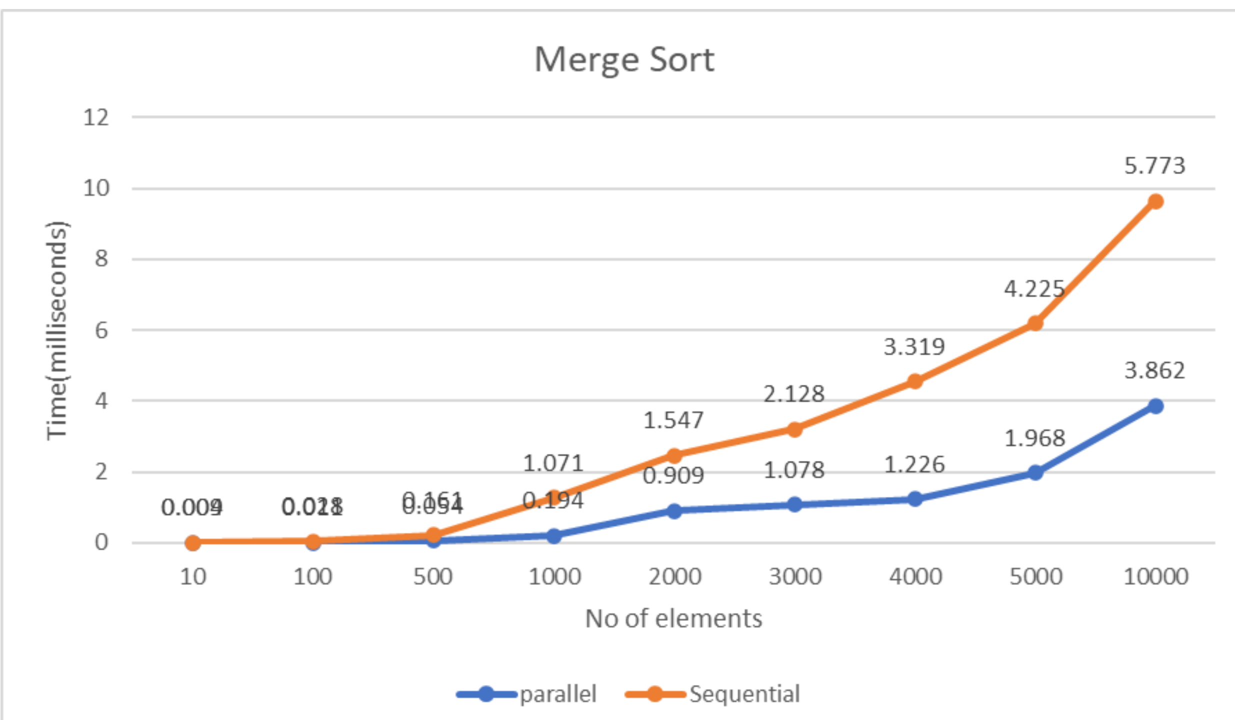




Similarly we ran the algorithm in serial and parallel environments and obtained the table as shown below.

n (Number of elements in the array)	Time taken for the Parallel Algorithm in ms	Time taken for the Serial Algorithm in ms
10	0.009	0.004
100	0.011	0.028
500	0.054	0.161
1000	0.194	1.071
2000	0.909	1.547
3000	1.078	2.128
4000	1.226	3.319
5000	1.968	4.225
10000	3.826	5.773

<u>Comparison of serial and parallel merge sort algorithms</u> :



This is the output graph we obtained after running the merge sort algorithm in serial and parallel environments. We observed as the number of elements of the array increases the sequential run time increases exponentially.But in case of parallel the run time is increasing very slowly which shows that parallel merge sort algorithm is much more efficient than the serial merge sort algorithm. We are using a dual core machine and still we got great results during parallel execution.If we had a system with multiple cores then we would have got more better results. So we can conclude that Merge Sort Algorithm is more efficient in parallel environment.

References

1. Jeon, Minsoo & Kim, Dongseung. (2003). Parallel Merge Sort with Load Balancing. International Journal of Parallel Programming. 31. 21-33. 10.1023/A:1021734202931.
2. https://stanford.edu/~rezab/classes/cme323/S16/notes/Lecture04/cme323_lec4.pdf
3. https://www.tutorialspoint.com/parallel_algorithm/parallel_algorithm_sorting
4. https://pediaa.com/what-is-the-difference-between-serial-and-parallel-processing-in-computer-architecture