LEBANESE AMERICAN UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND MATHEMATICS

CSC447: Parallel Programming and Multi-Cluster Systems

Project – Parallelizing Merge Sort

Hussein Charafeddine
ID#: 202103881

Lara Tawbeh
ID#: 202102927

Date: 23-04-2024

Contents:

Description of the Application/Algorithm:

For this project, we parallelized merge sort using MPI, CUDA C, OpenMP, and OpenACC. Merge sort has an intuitive implementation making it a valuable and fundamental sorting algorithm. It is one of the classic algorithms that uses a divide-and-conquer strategy to efficiently sort arrays or lists. Merge sort has performance measures that are consistent and can be predicted, making it easier for us to parallelize and understand its scalability. As such, along with its divide-and-conquer essence, it lends itself to parallelization where it can significantly enhance its performance, especially in the case of large datasets.

Articles Working On It:

1. Parallel Multi-Deque Partition Dual-Deque Merge sorting algorithm using OpenMP - PMC (nih.gov)
2. (PDF) Parallel Merge Sort with Load Balancing (researchgate.net)
3. Empirical Analysis Measuring the Performance of Multi-threading in Parallel Merge Sort (thesai.org)

Levels of Parallelism:

i. OpenMP

For this implementation, task-level parallelism was done through OpenMP's sections directive. The directive allows sorting the left and right halves of the array to occur as independent tasks that can be executed concurrently by multiple threads.

ii. MPI

For this implementation, data-level parallelism was done where different processes work on separate portions of the data concurrently using scatter and gather operations. The original array is divided into smaller chunks and each process sorts its assigned sub-array using scatter then the sorted arrays are gathered back using gather.

iii. CudaC

For this implementation, data-level parallelism was achieved through splitting the data of the array into blocks. Thread-level parallelism occurs in each block where the threads load elements into the shared memory, perform computations, then copy data to global memory.

iv. OpenACC

Similar to OpenMP this implementation also exploits task-level parallelism.

Techniques of Parallelism:

i. **OpenMP**

- Task parallelism:
    1. Tasks are created for sorting the left and right halves of the array independently.
    2. Each task can be executed concurrently by multiple threads.
- Divide-and-conquer:
    1. The array is recursively split into smaller sub-arrays until they contain one element each.
    2. Sorting independent sub-arrays is performed concurrently and then merged back together.

ii. **MPI**

- Data level parallelism:
    1. The array is divided into equally sized sub-arrays based on the number of processes in the communicator.
    2. Each process receives a local sub-array using MPI Scatter.
    3. The sorted chunks are gathered back using MPI_Gather.
- Divide-and-conquer:
    1. Same idea as above

iii. **CudaC**

- Data level parallelism and Thread Level Parallelism:
    1. The array is divided into blocks, each of 256 threads, which allows for concurrent processing of different parts of the array by multiple threads.
    2. Every thread operates independently on a segment of the array using shared memory.
    3. The sorted blocks are gathered back using MergeSortedBlocks() function.
- Divide-and-conquer:
    1. Same idea as above

iv. **OpenACC**

Task parallelism: Similar to OpenMP dividing the work into tasks that can be executed concurrently.

Pseudo For Parallelizing the Computation:

i. OpenMP

```
if (array size > 1) {
    // Calculate the midpoint of the array
    mid = midpoint of the array;

    // Create two tasks for sorting the left and right halves of the array
    Task left_sort = create_task(mergesort_parallel_omp(arr, start, mid,
temp, threads / 2));
    Task right_sort = create_task(mergesort_parallel_omp(arr, mid + 1, end,
temp, threads - threads / 2));

    // Wait for both tasks to complete
    wait_for_tasks(left_sort, right_sort);

    // Merge the sorted left and right halves
    merge(arr, start, mid, end, temp);
}
```

ii. MPI

```
if (array size > 1):
    local_size = n / size

    if (rank == 0):

        arr = ReadArrayFromFile("array1.txt")

    MPI_Barrier(MPI_COMM_WORLD)

    MPI_Scatter(arr, local_size, MPI_INT, local_arr, local_size, MPI_INT, 0,
MPI_COMM_WORLD)

    mergeSort(local_arr, 0, local_size - 1)

    MPI_Gather(local_arr, local_size, MPI_INT, arr, local_size, MPI_INT, 0,
MPI_COMM_WORLD)

if (rank == 0):
       mergeSortedSubarrays(arr, n, size);
       WriteArrayToFile(arr, "sorted.txt")
```

iii. CudaC

```
int main()
{
    cudaMalloc(&d_arr, n * sizeof(int));

    int num_blocks = (n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

    cudaMemcpy(d_arr, arr, n * sizeof(int), cudaMemcpyHostToDevice);

    mergeSort<<<num_blocks, THREADS_PER_BLOCK, THREADS_PER_BLOCK *
sizeof(int)>>>(d_arr, temp, n);

    cudaDeviceSynchronize();

    cudaMemcpy(arr, d_arr, n * sizeof(int), cudaMemcpyDeviceToHost);
    mergeSortedBlocks(arr, n, THREADS_PER_BLOCK);

    cudaFree(d_arr);
    free(arr);
}

__device__ void merge(int *arr, int *temp, int l, int m, int r)
{
    int i = l, j = m + 1, k = l;

    while (i <= m && j <= r)
    {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }

    while (i <= m)
        temp[k++] = arr[i++];
    while (j <= r)
        temp[k++] = arr[j++];

    for (i = l; i <= r; i++)
```

```
        arr[i] = temp[i];
}
__global__ void mergeSort(int *arr, int *temp, int n)
{
    extern __shared__ int shared_mem[];

    int tid = threadIdx.x;
    int block_size = blockDim.x;
    int block_start = blockIdx.x * block_size;

    if (tid < block_size && block_start + tid < n)
    {
        shared_mem[tid] = arr[block_start + tid];
    }
    __syncthreads();

    for (int curr_size = 1; curr_size < block_size; curr_size *= 2)
    {
        for (int left_start = 0; left_start < block_size; left_start += 2 *
curr_size)
        {
            int mid = left_start + curr_size - 1;
            int right_end = min(left_start + 2 * curr_size - 1, block_size - 1);

            if (mid < right_end)
            {
                merge(shared_mem, temp, left_start, mid, right_end);
            }
        }
        __syncthreads();
    }

    if (tid < block_size && block_start + tid < n)
    {
        arr[block_start + tid] = shared_mem[tid];
    }
}

void mergeSortedBlocks(int *arr, int n, int block_size)
{

    for (int size = block_size; size < n; size *= 2)
    {
        for (int left_start = 0; left_start < n; left_start += size * 2)
        {
            int mid = left_start + size - 1;
            int right_end = min(left_start + size * 2 - 1, n - 1);

            if (mid < right_end)
            {
                mergeSequential(arr, left_start, mid, right_end);
            }
        }
    }
}
```

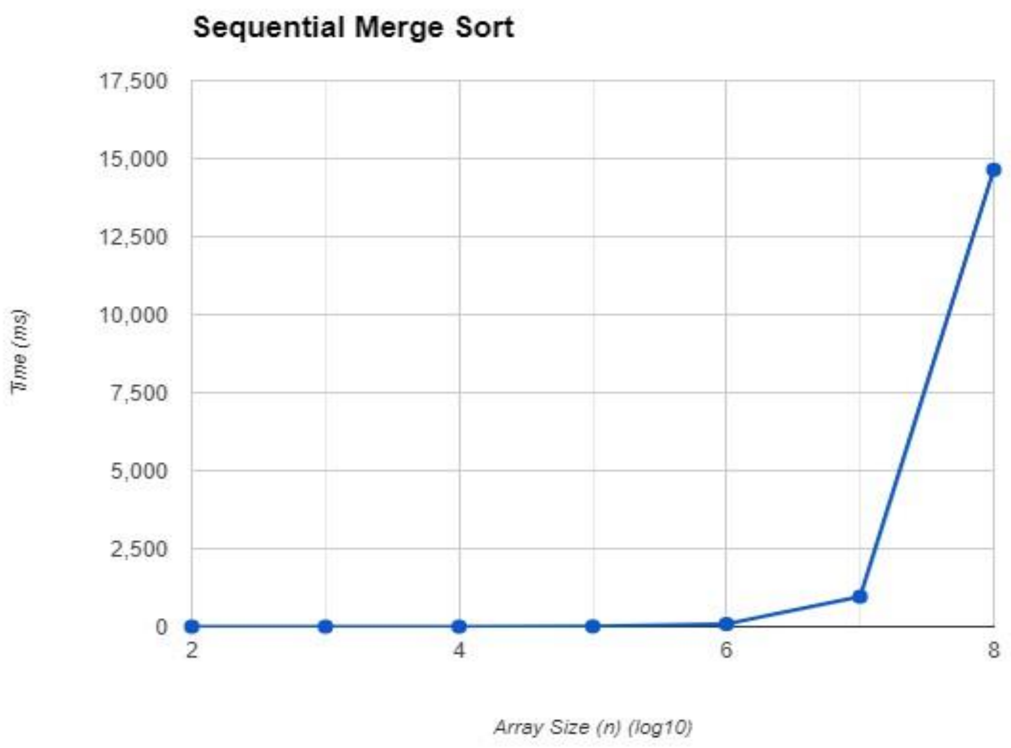
iv. OpenACC

```
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;

        // Apply OpenACC parallelism to the first half of the array
#pragma acc parallel present(arr[l : m + 1])
        mergeSort(arr, l, m);

        // Apply OpenACC parallelism to the second half of the array
#pragma acc parallel present(arr[m + 1 : r + 1])
        mergeSort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

Results:

Sequential

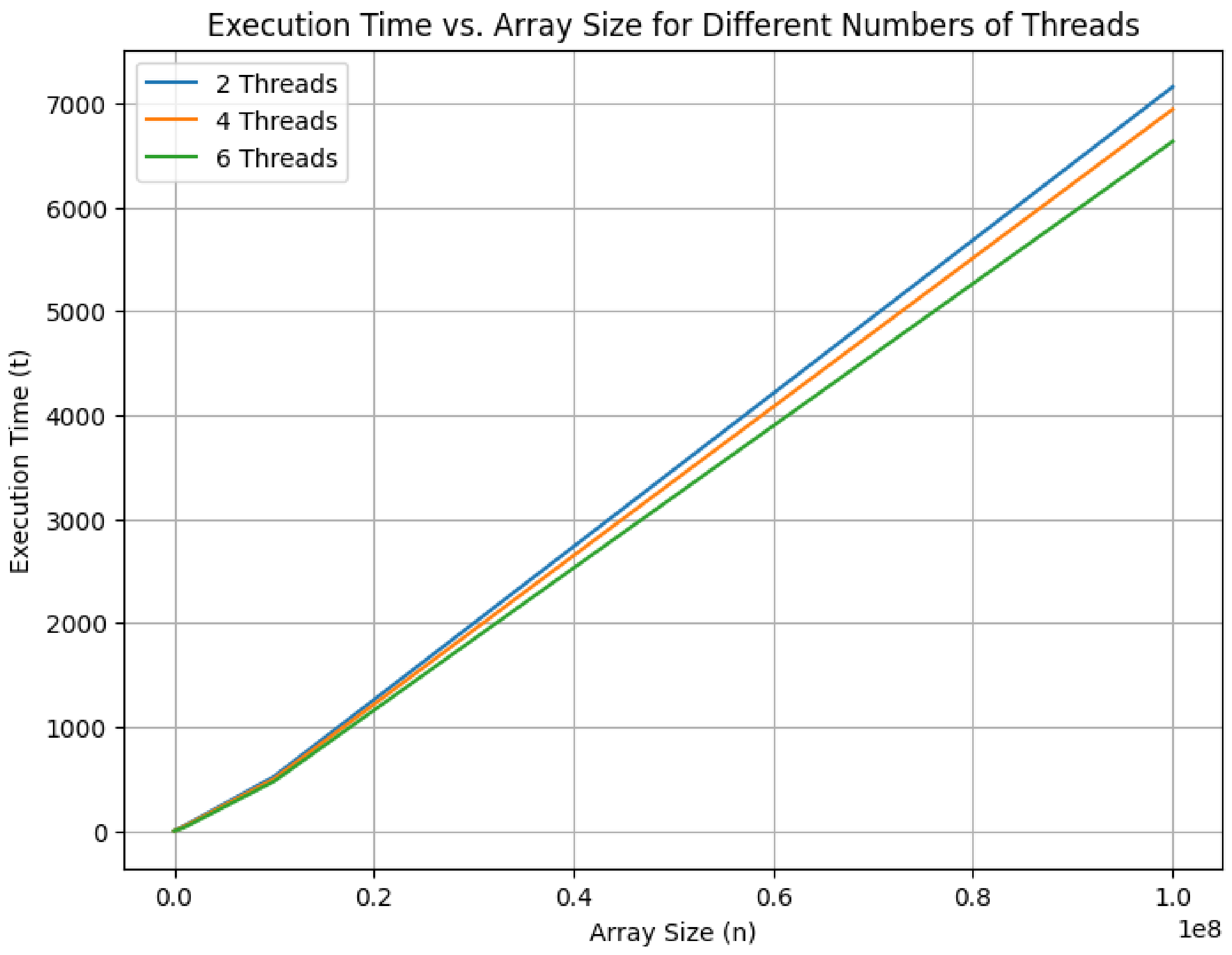
Array Size (n)	Time (ms)
100	0
1,000	0.05
10,000	0.59
100,000	7.24
1,000,000	83.03
10,000,000	961.5
100,000,000	14641.14



OpenMP

Time in ms

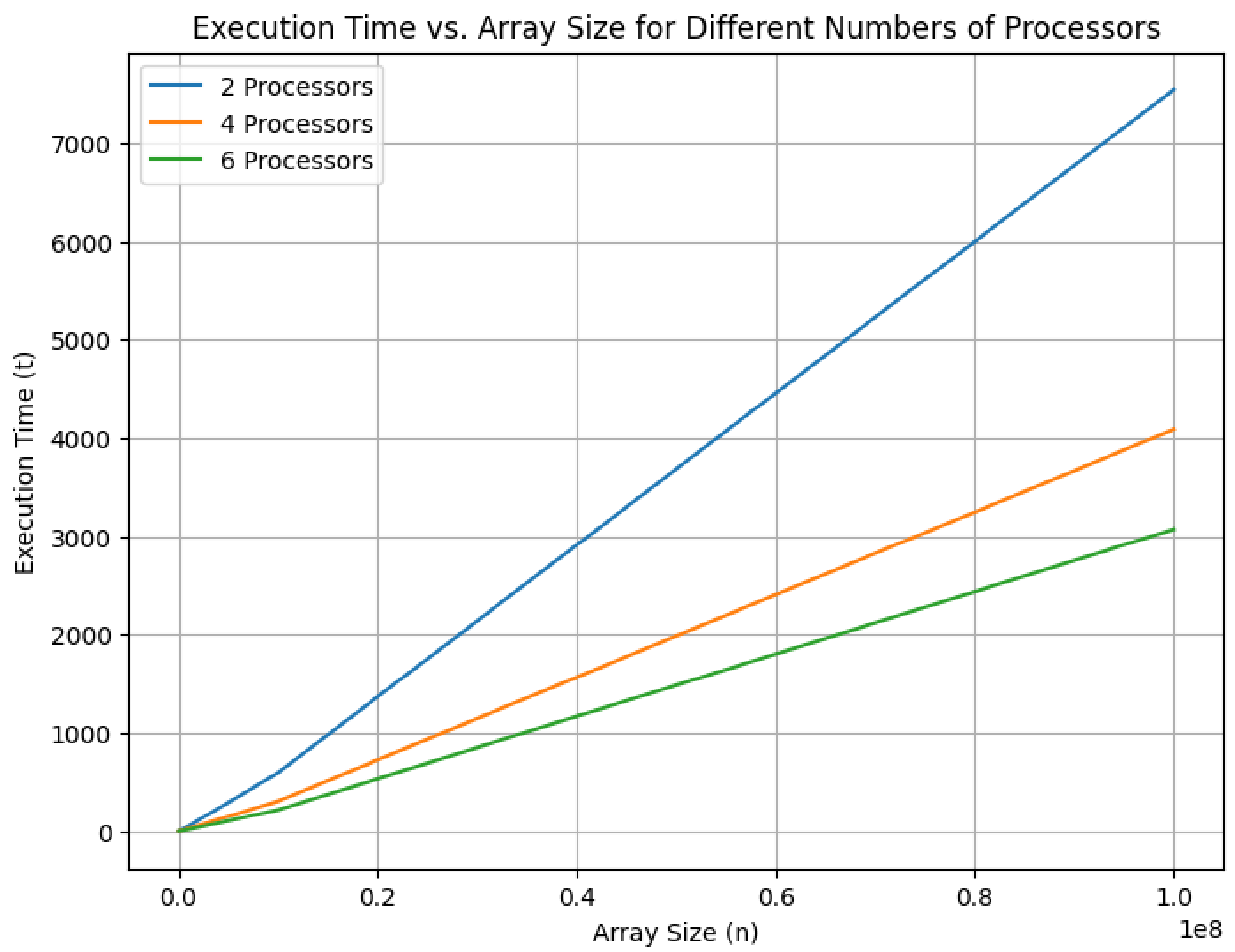
Threads/n	10000	100000	1000000	10000000	100000000
2	0.54	4.97	48.17	523.34	7356.94
4	0.41	4	42.74	501.26	6940.82
6	0.36	3.23	31.92	476.3	6631.8



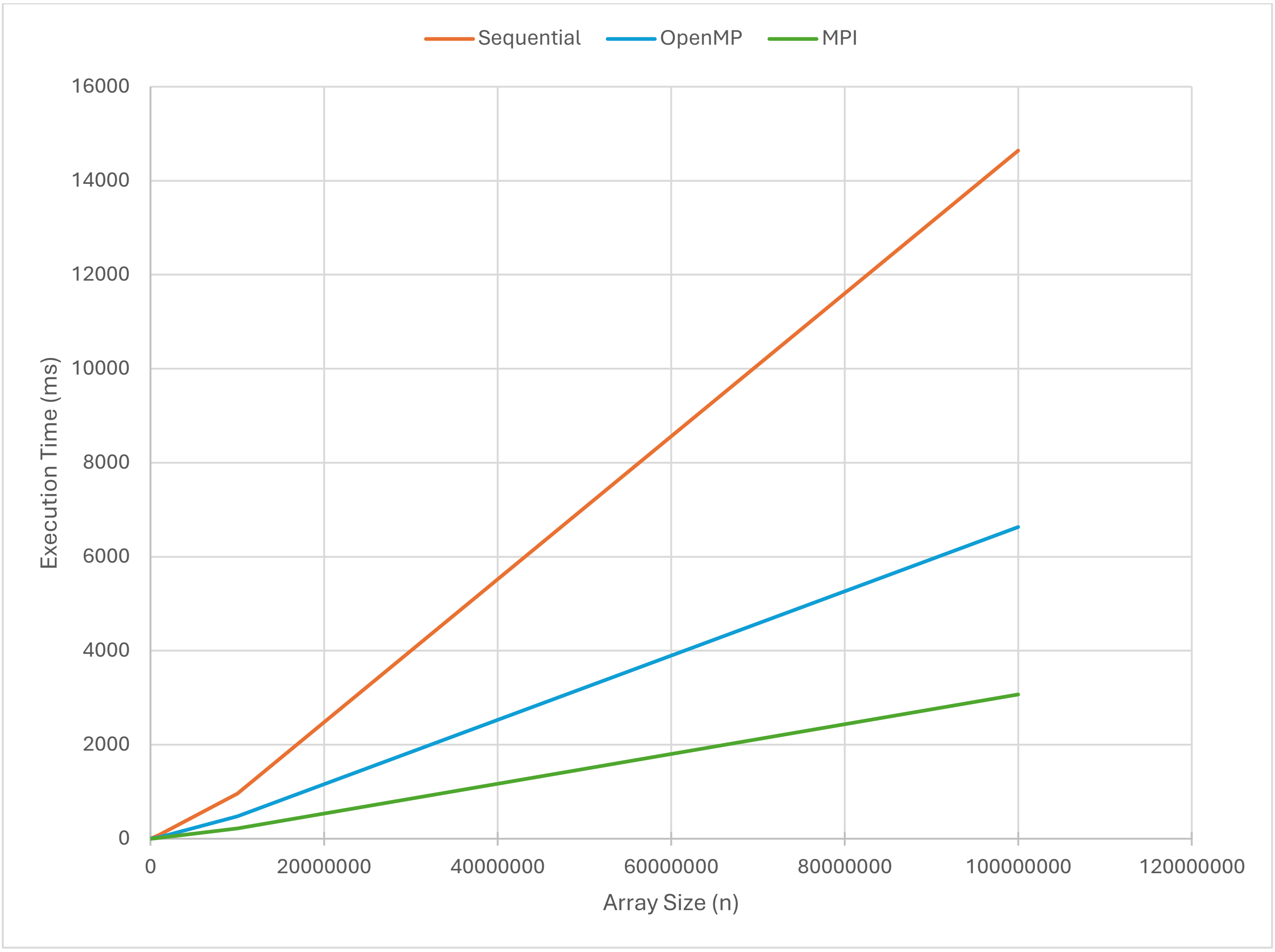
MPI

Time in ms

Processors/n	n=10000	n=100000	n=1000000	n=10000000	n=100000000
2	0.51	4.84	53.77	595.02	7539.88
4	0.22	2.67	28.72	305.95	4084.88
6	0.18	1.80	20.00	217.28	3069.61



Parallel vs Sequential:



Speedup efficiency and cost

Now we fix n to 100,000,000 to plot S(p) vs p:

1. **Speedup Factor:**

$$S(p) = \frac{ts}{tp} \quad \text{Sequential: } 14641.14$$

OMP

- 2 threads: 14641.14/7356.94= 1.99
- 4 threads: 14641.14/6940.82= 2.11
- 6 threads: 14641.14/6631.8= 2.2

MPI

- 2 p: 14641.14/7539.88 =1.94
- 4 p: 14641.14/4084.88 = 3.58
- 6 p: 14641.14/3069.61 = 4.77

2. **Efficiency:**

$$E = \frac{S(p)}{p} * 100\%$$

OMP

- 2 threads: 1.99/2 * 100 = 99.5%
- 4 threads: 2.11/4 * 100 = 52.75%
- 6 threads: 2.2/6 * 100 = 37%

MPI

- 2 threads: 1.94/2 * 100 = 97%
- 4 threads: 3.58/4 * 100 = 89.5%
- 6 threads: 4.77/6 * 100 = 79.5%

3. **Cost:**

OMP

- 2 threads: 7356.94* 2= 14713.88
- 4 threads: 6940.82* 4 = 27763.28
- 6 threads: 6631.8* 6 = 39790.8

MPI

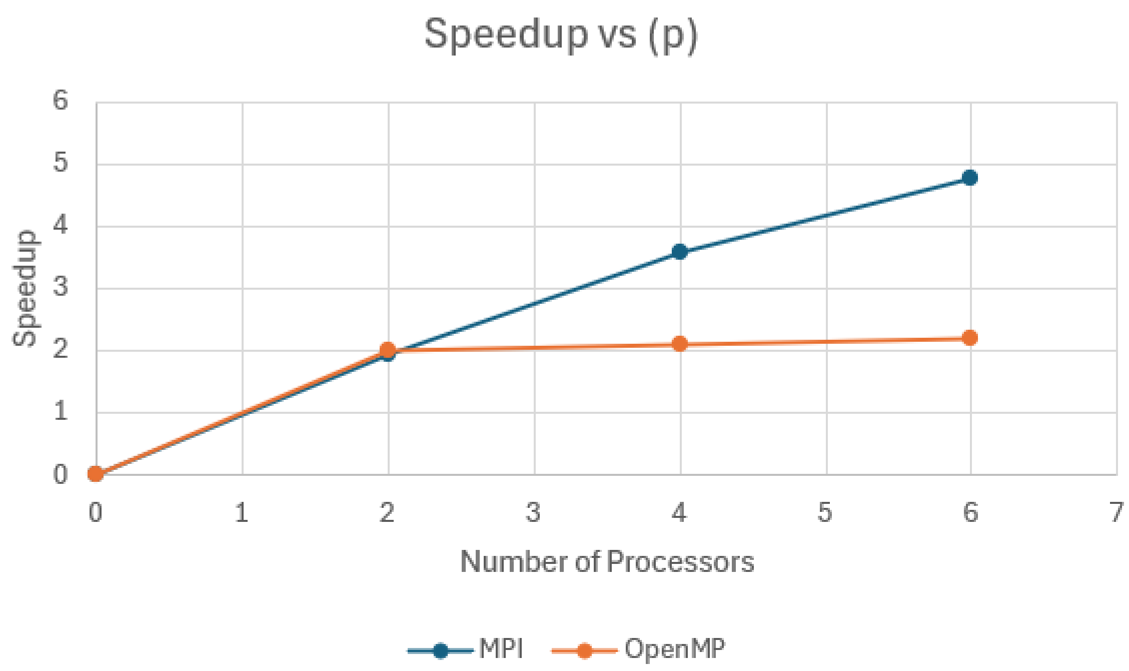
- 2 threads: 7539.88 * 2= 15079.88
- 4 threads: 4084.88 * 4 = 16339.52
- 6 threads: 3069.61 * 6 = 18417.66

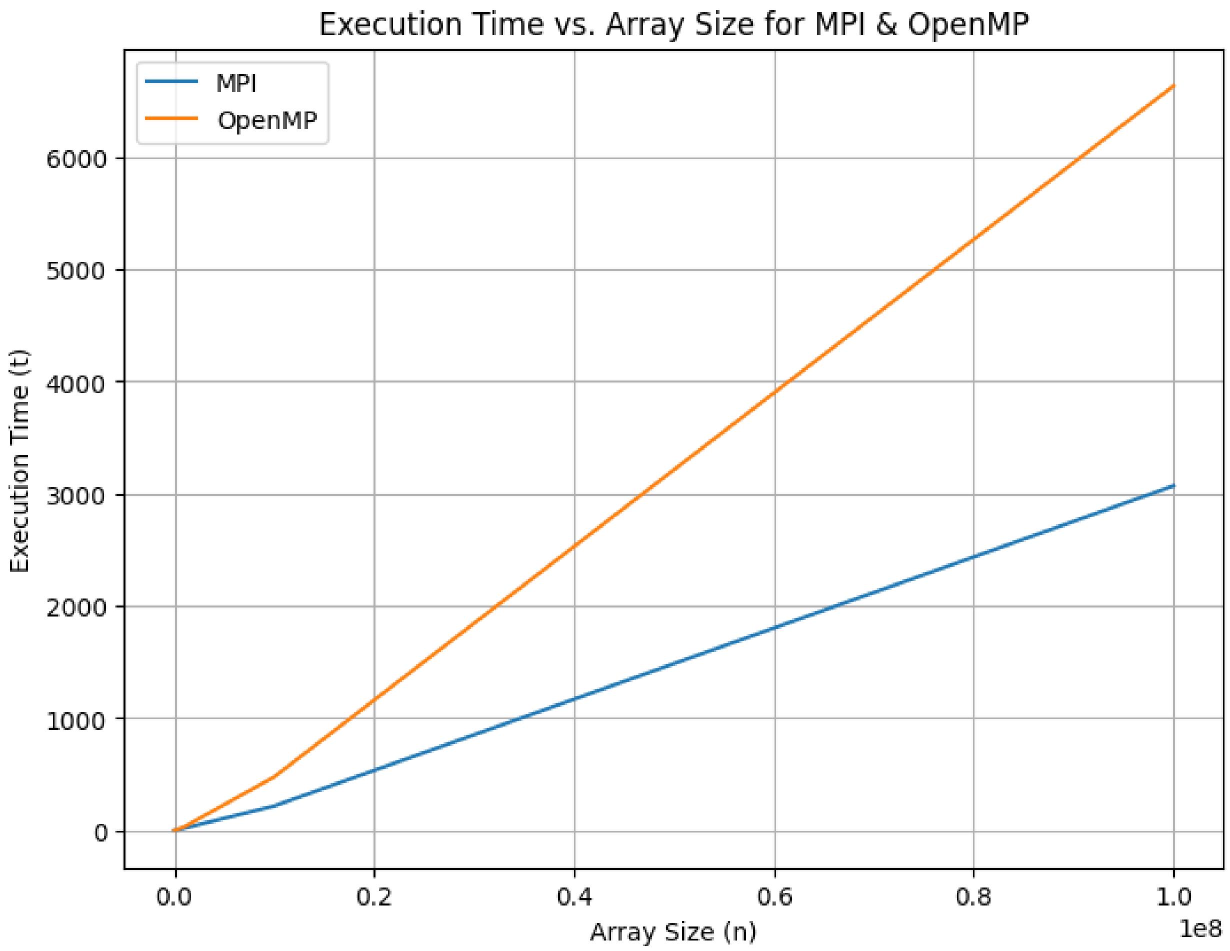
		Speedup	Efficiency	Cost
	MPI	1.94	97%	15079.88
2	OMP	1.99	99.5%	14713.88
	MPI	3.58	89.5%	16339.52
4	OMP	2.11	52.75%	27763.28
	MPI	4.77	79.5%	18417.66
6	OMP	2.2	37%	39790.8

OpenMP vs MPI:

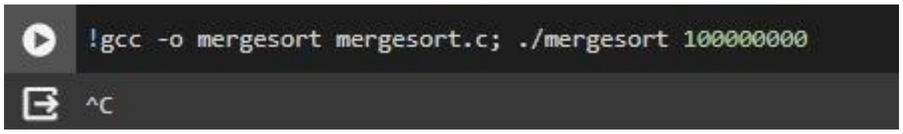
We compare for 6 threads/processors:

Threads or P/n	n=10000	n=100000	n=1000000	n=10000000	n=100000000
MPI	0.18	1.80	20.00	217.28	3069.61
OpenMP	0.36	3.23	31.92	476.3	6631.8

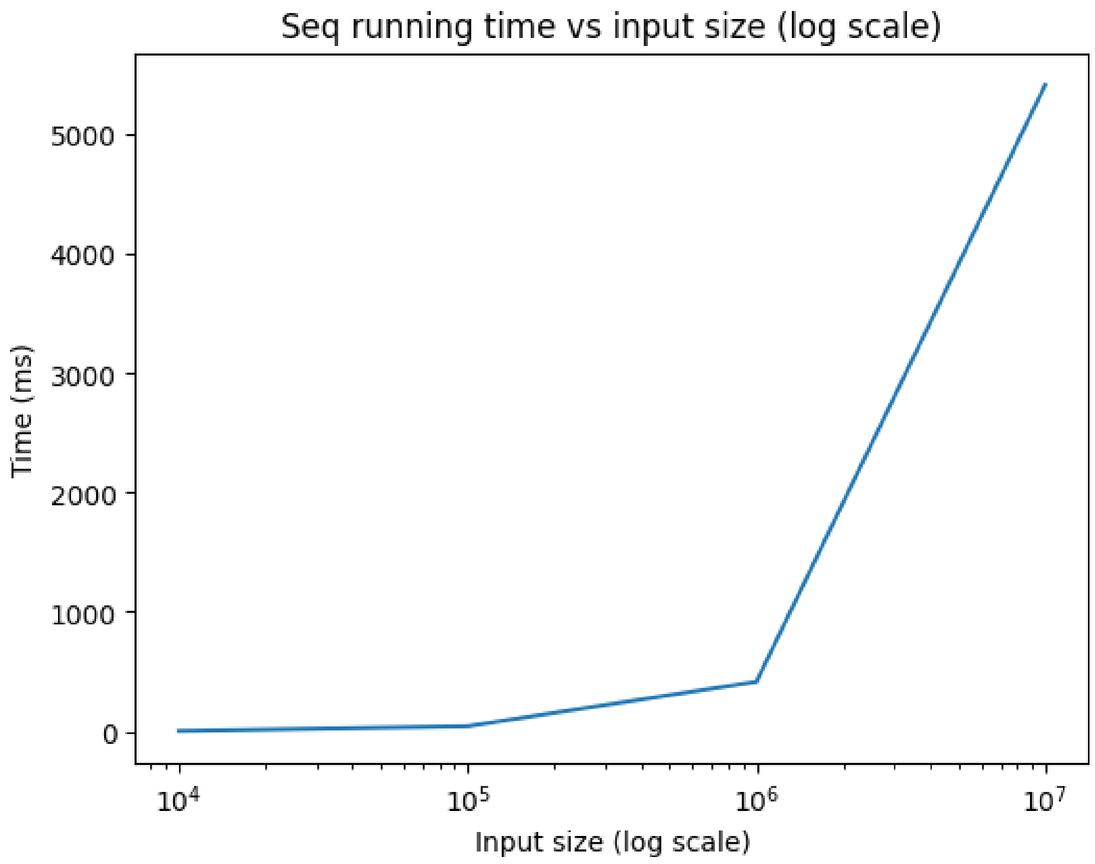

Speedup vs (p)
Speedup
Number of Processors
0
1
2
3
4
5
6
7
MPI
OpenMP


Execution Time vs. Array Size for MPI & OpenMP
MPI
OpenMP
Execution Time (t)
0
1000
2000
3000
4000
5000
6000
0.0
0.2
0.4
0.6
0.8
1.0
1e8
Array Size (n)

Sequential on Google Colab

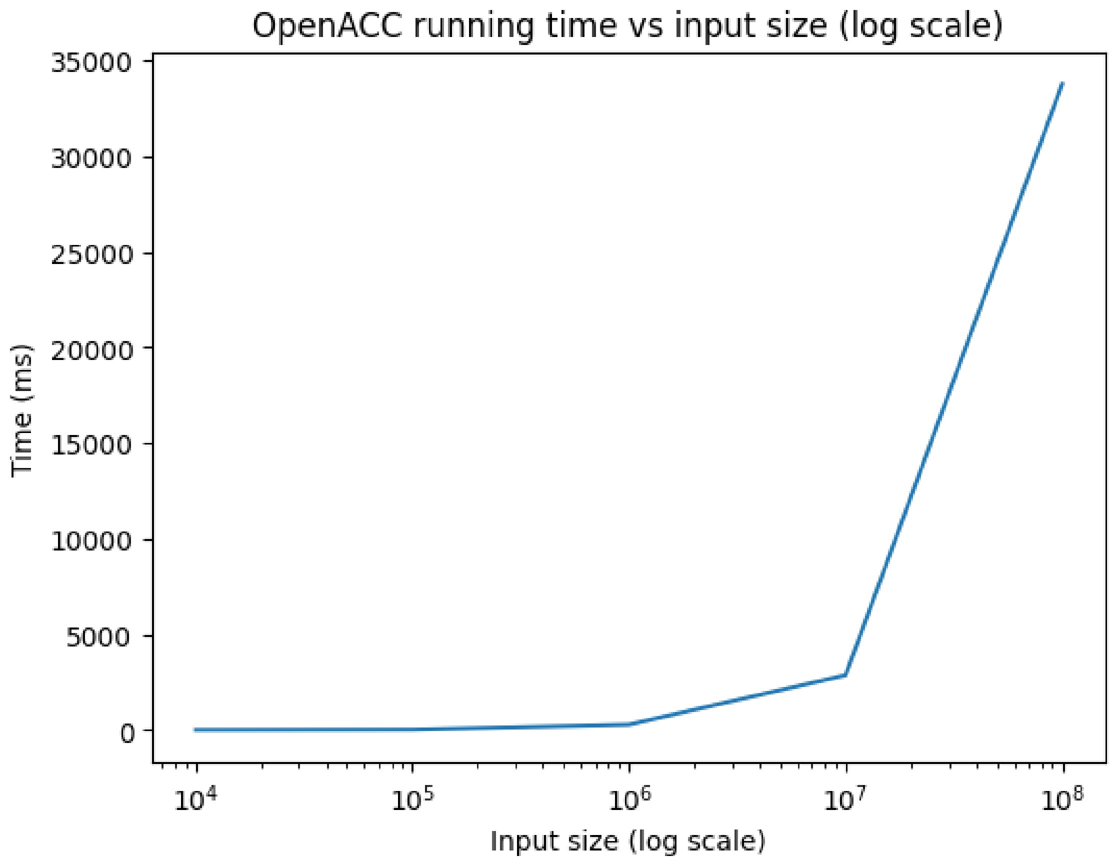
Array Size (n)	Time (ms)
10,000	3.08
100,000	42.48
1,000,000	415
10,000,000	5415
100,000,000	Indeterminate 

```
!gcc -o mergesort mergesort.c; ./mergesort 100000000
^C
```



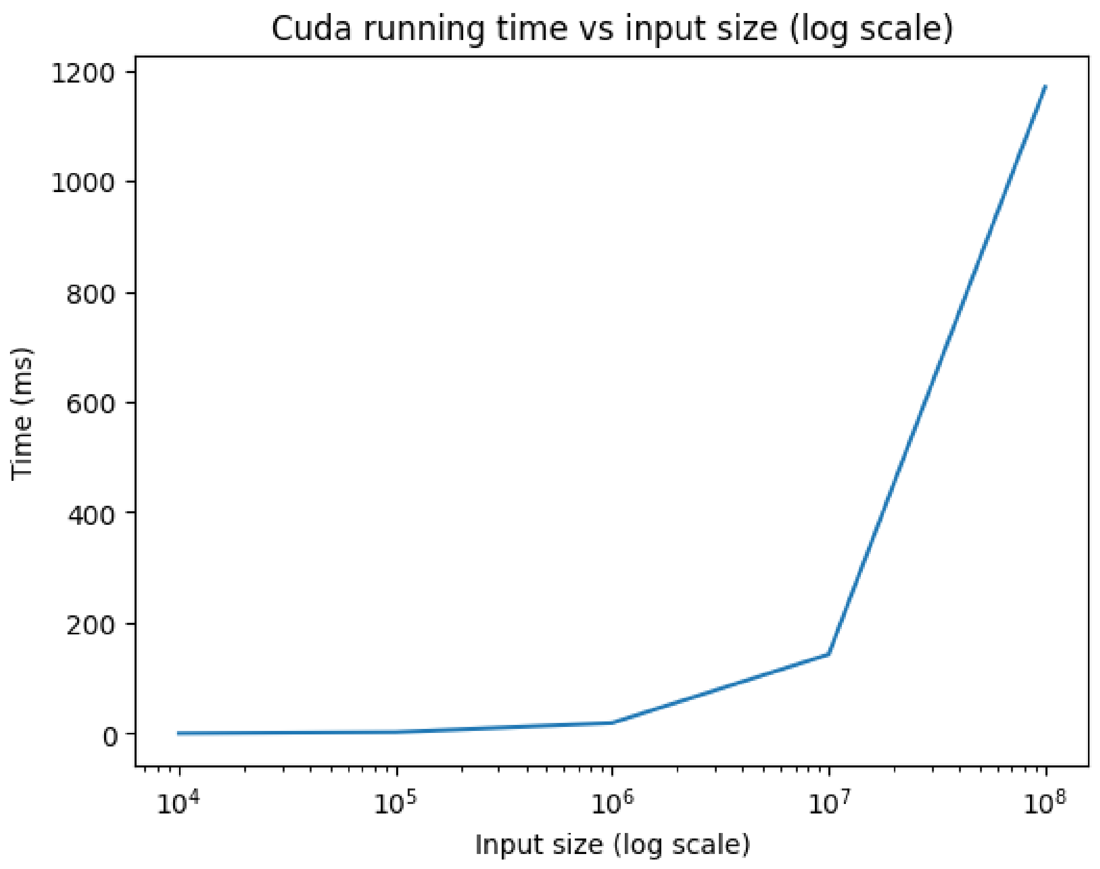
OpenACC

n	Time (ms)
10000	1.04
100000	16.01
1000000	278.67
10000000	2858.76
100000000	33766.56

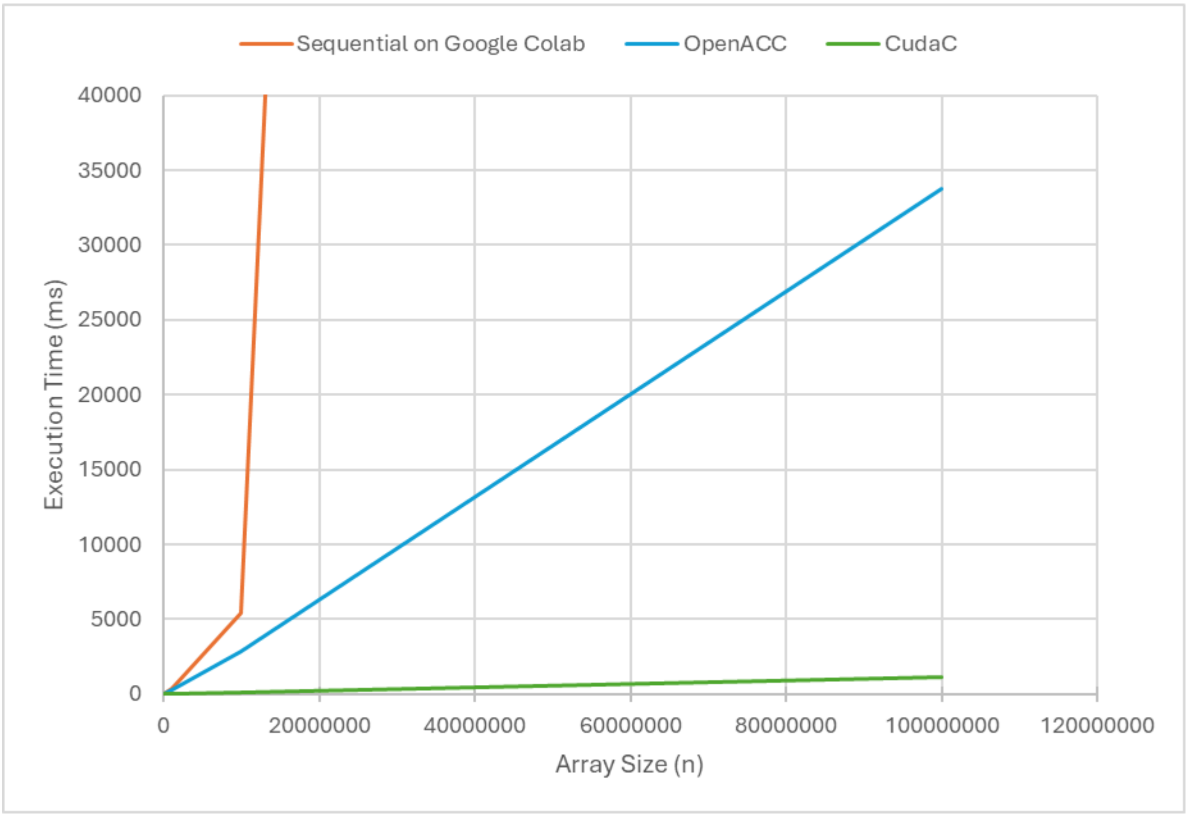


CudaC

n	Time (ms)
10000	0.28
100000	2.41
1000000	19.12
10000000	142.99
100000000	1169.65



Parallel vs Sequential:



Speedup

OpenACC

- Input Size 10^4: 3.08/1.04= 2.96
- Input Size 10^6: 415/278.67= 1.49
- Input Size 10^7: 5415/2858.76= 1.89

CUDA

- Input Size 10^4: 3.08/0.28= 11
- Input Size 10^6: 415/19.12= 21.7
- Input Size 10^7: 5415/142.99= 37.87

Efficiency

OpenACC

- Input Size 10^4: 2.96 / 1024 * 100= 0.28%

- Input Size 10^6: 1.49 /1024 * 100= 0.15%
- Input Size 10^7: 1.89 /1024 * 100 = 0.28%

CUDA

- Input Size 10^4: 11/1024 * 100 = 1%
- Input Size 10^6: 21.7/1024 * 100 = 2.1%
- Input Size 10^7: 37.87 /1024 * 100 = 3.69%

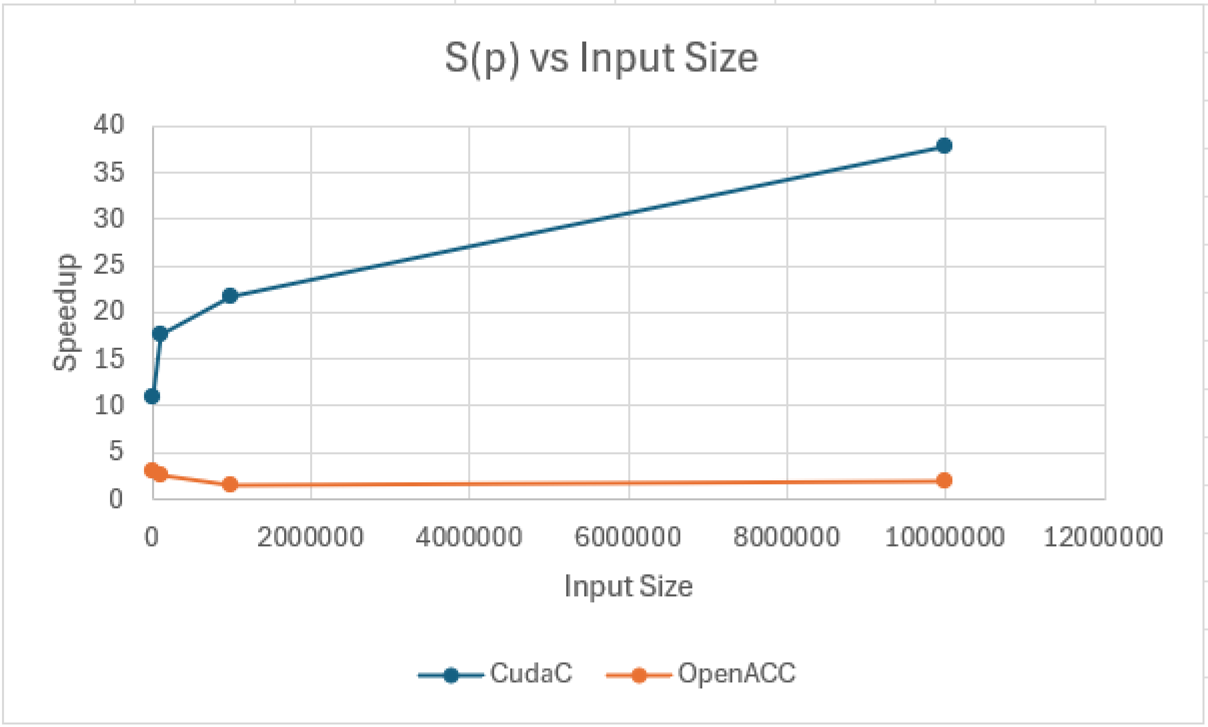
Cost

OpenACC

- Input Size 10^4: 1.04 *1024 = 1064.96
- Input Size 10^6: 278.67 *1024 = 285358.08
- Input Size 10^7: 2858.76 *1024 = 2827370.24

CUDA

- Input Size 10^4: 0.28*1024= 286.72
- Input Size 10^6: 19.12*1024= 19578.88
- Input Size 10^7: 142.99*1024= 146421.76



	CUDA C (10^4)	CUDA C (10^6)	CUDA C (10^7)	OpenACC (10^4)	OpenACC (10^6)	OpenACC (10^7)
Speedup	11	21.7	37.87	2.96	1.49	1.89
Efficiency	1%	2.1%	3.69%	0.28%	0.15%	0.18%
Cost	286.72	19578.88	146421.76	1064.96	285358.08	2827370.24

Conclusion:

- OpenACC and OpenMP rely on the compiler to automatically parallelize code, which may not always result in optimal performance as we have less control over what is happening.
- The results show promising speedup, efficiency, and costs for the MPI implementation.
- As for OpenMP potential bottlenecks lie in thread management and synchronization.
- The integration of OpenMP with MPI might leverage both of their parallelizing powers and achieve a higher speedup than each separate implementation.
- The Cuda C implementation is underperforming because of the dependencies of the threads on each other leading to low efficiency.

Code:

https://github.com/HusseinCharafEddine/ParallelizingMergeSort

Work Distribution:

Discussed designs and concepts together however, the implementation was done separately as follows:

Lara: OpenMP and OpenACC

Hussein: Cuda C and MPI

The report and presentation were done together equally.