Analysis of Merge Sort and its Parallel Implementations

Mauti Enzo

July 18, 2025

Abstract

This report contain a quick analysis of the merge sort algorithm, 2 parallelized implementation of the algorithm, one using OpenMP, the other using MPI. We will then analyse the strong and weak scaling of each of the implementation and finish by presenting possible enhancement of those implementations.

1 Introduction

Merge Sort is a divide-and-conquer algorithm with a worst-case and average-case time complexity of $O(n \log n)$.

2 Sequential Merge Sort

2.1 Algorithm Description

```
void mergeSort(std::vector<int>& arr, int left, int right)
{
  if (left >= right)
    return;

  int mid = left + (right - left) / 2;
  mergeSort(arr, left, mid);
  mergeSort(arr, mid + 1, right);
  merge(arr, left, mid, right);
}
```

2.2 Complexity Analysis

With $T(k)$ as the time taken to sort k element and $M(k)$ the time take to merge k element, we can write:

$$T(N) = 2 \times T(\frac{N}{2}) + M(N) \tag{1}$$

$$T(N) = 2 \times T(\frac{N}{2}) + constant \times N \tag{2}$$

$$T(N) = 2^k \times T(\frac{N}{2^k}) + 2 \times N \times constant \tag{3}$$

$$T(N) = N \times T(1) + N \times log_2 N \times constant \tag{4}$$

$$T(N) = N + N \times log_2 N \tag{5}$$

Therefore the time complexity is $O(N \times log_2 N)$

3 MPI Parallel Merge Sort

3.1 Algorithm Description

```
Initialize MPI
master process:
  create random vector 'arr' of size 'vector_size'
compute 'counts' and 'displacements' for each rank
  scatter chunks of 'arr' to each process → 'local_arr'
each process:
  sort its 'local_arr' with mergeSort
gather all 'local_arr' segments back to master
master process:
   for each segment received:
     merge in-place with previous segments
   check if final array is sorted
   print elapsed time
finalize MPI
```

3.2 Parallel Strategy

In this implementation, we divide the array so that we can apply the sorting in multiple places at the same time. The division is made based on the number of cores available at each time. The transfert of datas is handled by the MPI functions `Scatterv` and `GatherV`. Each subarray are then sorted, and sent back to the master process. we then proceeds to a final merge among all subarrays.

4 OpenMP Parallel Merge Sort

4.1 Algorithm Description

```
create random vector 'arr' of size 'vector_size'
start OpenMP parallel region with 'num_threads':
   single thread:
     call mergeSort(arr, 0, vector_size - 1)
       recursively split into subtasks until size < THRESHOLD
       execute subtasks in parallel and merge in-place
check if final array is sorted
print elapsed time
```

4.2 Parallel Strategy

In this implementation, we divide the workload between multiple threads, until a certain size threshold is achieved. To accomplish that, we use the `#pragma omp task` to create new threads to sort the left and right side of the subarrays. After spawning, `#pragma omp taskwait` ensures that both halves are sorted before merging. By controlling task creation with a threshold, we avoid overhead from creating too many small tasks, which improves performance.

5 Performance Comparison

5.1 Graphical Comparison

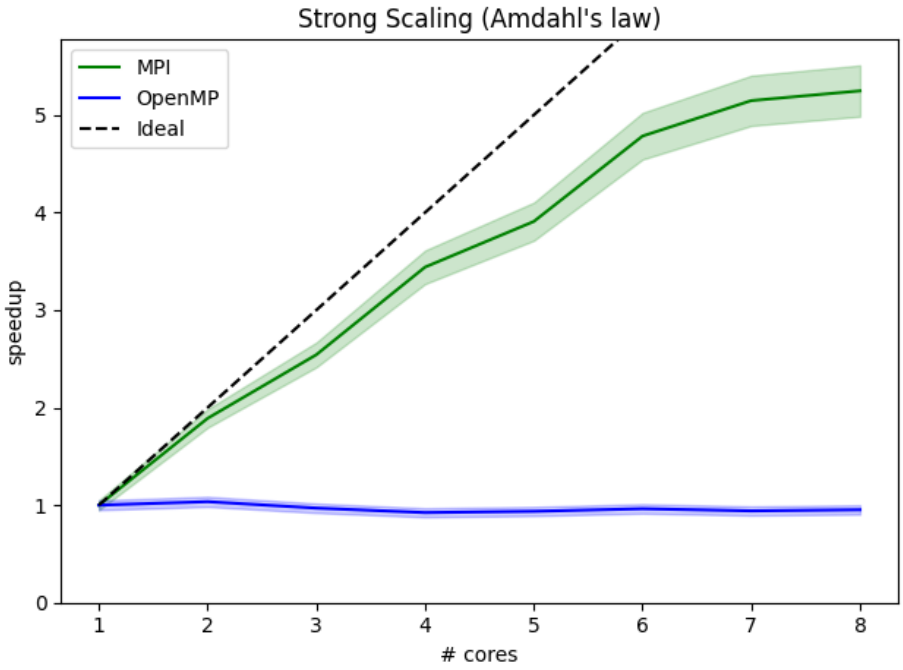

Figure 1: Strong scaling of the parallelized implementations

Analysis of Figure 1: In this graph, we can observe the Strong scaling of the different implementations of the algorithm. As we can see, the results are in favor of the MPI implementation, due to the diminution in size of the sub parts that each cores will have to sort.

Figure 2: Weak scaling of the parallelized implementations

Analysis of Figure 2: In this graph, we can observe the Weak scaling of the different implementations of the algorithm. For each core, we add a total As we can see, the results are in favor of the MPI implementation for a small number of core, but is replaced by the OpenMP implementation for big number of threads, probably due to a better decoupling of concrete work done for each tasks.

6 Conclusion

In this report, we have explored the sequential implementation of Merge Sort, followed by two parallel implementations using MPI and OpenMP. We then analyzed their performance through strong and weak scaling experiments. From the results, we can conclude that the MPI implementation provides better performance when dealing with smaller numbers of processes, thanks to better distribution of workload across nodes. However, as the number of threads increases, OpenMP becomes more efficient, likely due to reduced communication overhead and better local memory usage.

Each approach presents trade-offs between ease of implementation, memory management, and scalability. While MPI is suitable for distributed environments with explicit communication control, OpenMP is better in shared-memory systems with simpler thread parallelism.