Assignment 2b

Final Report submitted by Nooshin Behrooz

ORFEO username nbehrooz

Problem: parallel implementation of the Quick-Sort Algorithm

▪ Introduction of Quick-Sort Algorithm

Quick sort algorithm is widely used because of its efficiency and simplicity. It is a Divide-and-Conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted. This process repeats until the entire array is sorted. The average time complexity of Basic Quick sort is $O(n\,log(n))$, while in the worst-case scenario is $O(n^2)$.

Quick sort function: snippet

```
void quicksort(data_t *arr, int low, int high) {
   if (low < high) {
      int pi = partition(arr, low, high);
      quicksort(arr, low, pi - 1);
      quicksort(arr, pi + 1, high);  }}
```

▪ Implementation of MPI Quick Sort Algorithm

To write a parallel implementation (MPI: Message Passing Interface), of the serial Quick-Sort algorithm, we need to modify the standard serial code using MPI_Routins. The main focus is on parallelizing the quicksort algorithm across multiple processes to efficiently sort large datasets. Additionally, the code aims to ensure that the final sorted array is globally sorted across all elements by employing a parallel merge sort operation.

In the following we explain the code structure briefly:

1. Header Files:

The code includes standard **C header** files such as stdlib.h, stdio.h, math.h and sring.h for basic functionalities, as well as mpi.h for MPI communication and time.h for measuring execution time.

2. Data Structures and Functions:

Data_t Structure: representing a double data type containing an array of floating-point numbers.
Compare Function: it is used within the quicksort algorithm to compare elements of data_t.
Partition Function: Implements the partitioning step of the quicksort algorithm.

Quicksort Function: Implements the quicksort algorithm to sort an array of data_t elements.

3. Implementation of Main Functions:

- **Data Initialization**:

MPI_Init: Initializes MPI
MPI_Com_rank: Obtains the rank of the MPI communicator MPI_COMM_WORLD.
MPI_Com_size: Obtains the size of the MPI communicator MPI_COMM_WORLD.
rand function: Generates random data in the range of $[0, 1)$ on the root process.

- **Data Distribution**:

MPI_Bcast: broadcasts the size of the data to all processes.
MPI_Scatter: Splits the data (generated on the root) equally among all processes.

- **Local Sorting:**

quicksort function: Each process independently sorts its local portion of the data using the quick-sort algorithm that has been defined before.

- **Data Gathering:**

MPI_Gather: Gathers sorted data from all processes to the root process.

- **Parallel Merge Sort**:

parallel_merge function: is a placeholder function for a parallel merge sort operation. It's responsible for merging the sorted arrays obtained from different processes into a single sorted array on the root process (which allocates memory for merged data).

- **Execution Time Measurement:**

MPI_Wtime function: measures the execution time.

- **Memory Management**:

Free function: frees allocated memory for local data and global data.

- **MPI Finalization:**

MPI_Finalize function: finalizes MPI communication.

Overall, to parallelize the quick-sort algorithm, we used the following MPI_Routines , for distributed-memory parallel computing, MPI_Init, MPI_Com_rank, MPI_Com_size and MPI_Finalize to initialization of data and we applied collective operations MPI_Bcast, MPI_Scatter and MPI_Gather, for distributing data among processors and gathering sorted arrays from different processors to a root node. Finally, we performed the last sorting of the sorted arrays into a single array. Here we provide the snippet of code:

MPI Quick Sort Algorithm: snippet

```
// definition of functions
int compare(const void *a, const void *b) {…}
int partition(data_t *arr, int low, int high) {…}
void quicksort(data_t *arr, int low, int high) {…}
void parallel_merge(data_t *merged_data, int size, int N, int DATA_SIZE) {…}
// implementation of functions
int main(int argc, char **argv) {
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   int N = 100000;
   start_time = MPI_Wtime();
   MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Scatter(data, N / size * DATA_SIZE, MPI_DOUBLE, local_data, N / size * DATA_SIZE, 0,
MPI_COMM_WORLD);
   quicksort(local_data, 0, N / size - 1);
   MPI_Gather(local_data, N / size * DATA_SIZE, MPI_DOUBLE, sorted_data, N / size * DATA_SIZE, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
   if (rank == 0) {
      parallel_merge(merged_data, size, N, DATA_SIZE);
      end_time = MPI_Wtime();   }
   MPI_Finalize();
```

In the following sections, we will explain the results of performing strong and weak scaling experiments for our model (MPI quick_sort) with varying numbers of cores. Details on how to compile and run the code are provided in the ReadMe.md and slurm.job files available in the GitHub repository, the address of which is mentioned at the end of this report.

▪ Performance and Results

a. Strong scalability

Definition: According to **Amdahl's law** states that, for a fixed problem, the upper limit of speedup is determined by the serial fraction of the code (s).

$$S(p) = \frac{T_{(serial)}}{T_{(parallel)}} = \frac{T_1}{s+\frac{p}{N}}$$

$$\lim_{N\to\infty} S(p) = \frac{1}{s}$$

Therefore, **Strong scalability** refers to how the **execution time** of a parallel program changes as the **number of nodes** (processors or threads) is **increased** for a **fixed total problem size**. In an ideal strong scaling scenario, doubling the number of processors should ideally halve the execution time, resulting in a speedup that is close to doubling.

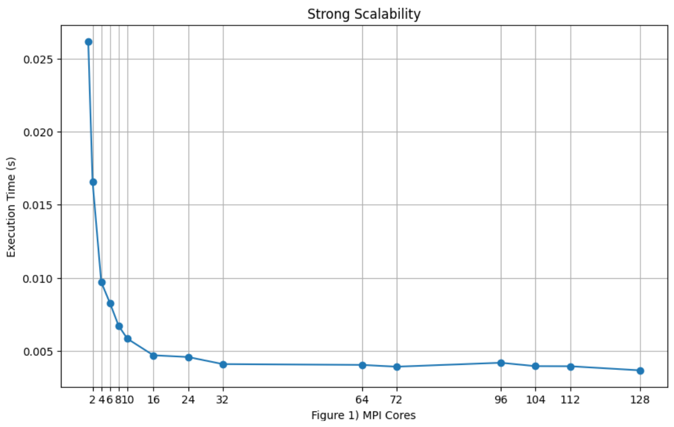
$$E(p) = \frac{S_p}{N}$$

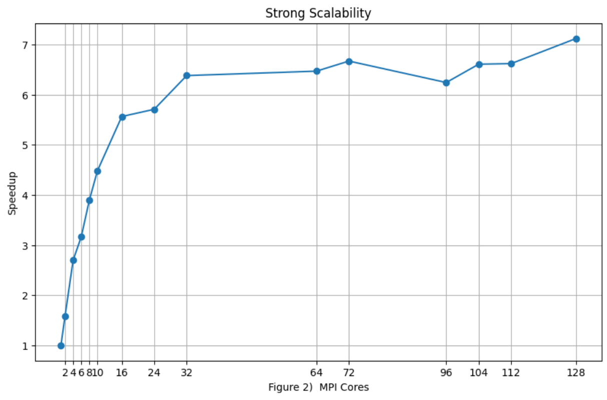
Interpretation of result

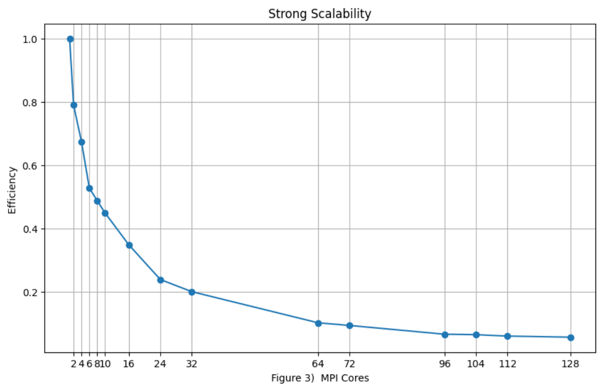
We performed strong scaling for our model (MPI quick_sort) with different numbers of MPI cores on the epyc002 node of the ORFEO HPC center, while keeping the number of data randomly generated on the root node constant at 100,000. The execution time for generating the dataset is tracked for each computation, and the outcomes are displayed in Table 1 and Figure 1. We also calculated speedup and efficiency for different numbers of cores using the formula shown above, and the results are presented in Table 1.

Table 1

#Data	**#node**	**Execution_Time (s)**	$S(p) = \frac{T_1}{T_p}$	$E(p) = \frac{S_p}{N}$
100000	1	0.026169	1	1
100000	**2**	0.016569	1.579395	0.789698
100000	**4**	**0.009702**	**2.697279**	0.674320
100000	6	0.008272	3.163564	0.527261
100000	**8**	**0.006716**	**3.896516**	0.487065
100000	10	0.005837	4.483296	0.448330
100000	**16**	**0.004702**	**5.565504**	0.347844
100000	24	0.004583	5.710015	0.237917
100000	**32**	**0.004100**	**6.382683**	0.199459
100100	**64**	**0.004044**	**6.471068**	0.101110
100000	72	0.003922	6.672361	0.092672
100000	96	0.004191	6.244094	0.065043
100000	104	0.003959	6.610003	0.063558
100000	112	0.003952	6.621711	0.059122
100000	**128**	**0.003674**	**7.122754**	0.055647







As seen in table 1 and figure 1, by increasing the number of cores of 1 node (for a fixed problem size) and splitting the job among them, the total execution time starts to decrease. As can be inferred from the results, by increasing the cores from 1 to 32, the execution time decreases from the serial case of 0.026169 s to 0.004100 s for 32 cores. However, beyond that, by doubling the number of cores to 64, the execution time does not change significantly, with it being 0.004044 s, and the difference being only about 0.0001 s. Even with an increase in using more cores up to 128, the final execution time remains at 0.003674 s, which is almost the same result as with 32 cores. Moreover, figure 2 shows that up to 32 nodes, the speedup grows very fast from 1 in the serial case to 6.382683 (around 7 times more), while in the interval [32 – 128], speedup changes only from 6.382683 to 7.122754, which is not very outstanding. For a comprehensive analysis, the efficiency of the model has been plotted in figure 3. It shows that, in general, increasing the number of cores leads to a decrease in efficiency, and by applying more than 32 cores, it reaches below 0.2, which is very low efficiency.

b. Weak scalability

Definition: According to **Gustafsson's law** in parallel problems, if the **total problem size increases** as the **number of nodes increase**, the **sequential proportion** of the computations will **decrease**. With Gustafson's law the scaled speedup increases linearly with respect to the number of processors (with a slope smaller than one), and there is no upper limit for the scaled speedup.

So, **for scaled speedup** we have the following formula:

$$S(p) = s + Np \quad \text{(s: time of serial part, N: number of nodes, p: time of parallel part)}$$
$$T(p) = s + \frac{Np}{N} = S + P = \quad CONSTANT$$

Execution time remains constant when increasing parallel workload.

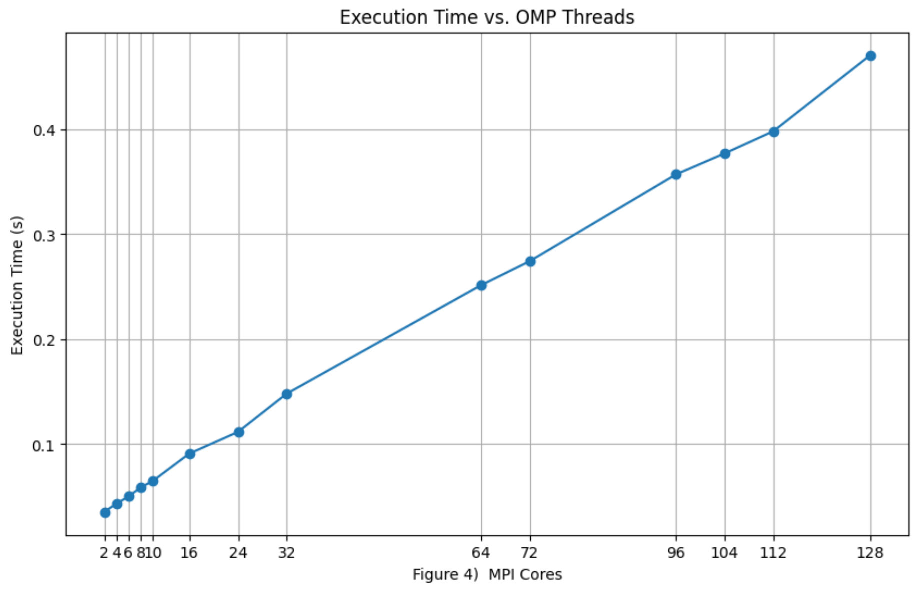
Interpretation of results

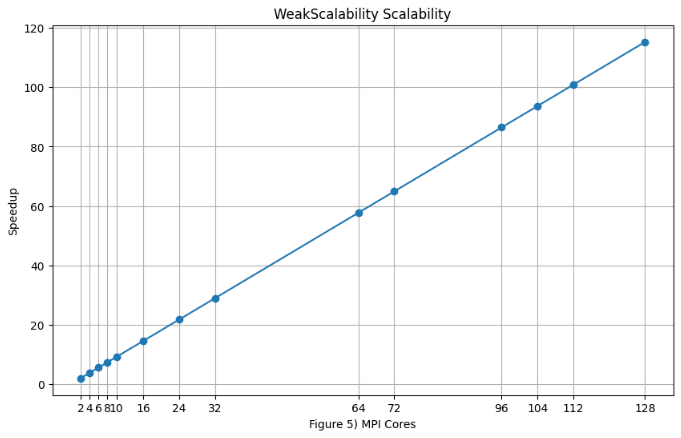
For examining the weak scaling of our model, we ran our MPI model with different numbers of cores while simultaneously increasing the amount of data to expand the problem size. We adjusted the size of the problem to correspond with the increasing number of cores for sake of simplicity. The results of the weak scaling test are presented in Table 2 and Figures 1, 2, and 3.

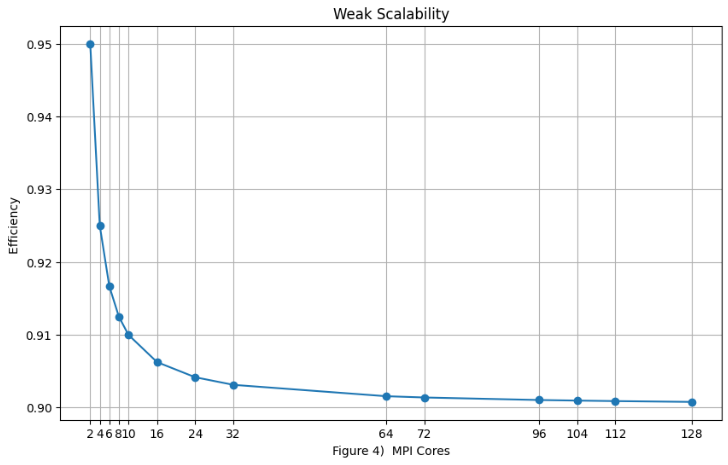
To calculate speedup and subsequent efficiency, we considered s = 0.1 and p = 0.9 as an average of these values, to observe the linear behavior of scaled speedup in terms of the number of cores. As we know, by expanding the size of the problem, the role of serial part of the code shrinks while the parallel workload increases.

Table 2

#Data	#node	Execution_Time (s)	$S(p) = s + Np$	$E(p) = \frac{S_p}{N}$
100000	1	0.026169	1	1
2*100000	2	0.035420	1.90	0.9500
4*100000	4	0.043295	3.70	0.9250
6*100000	6	0.050235	5.50	0.9166
8*100000	8	0.058678	7.30	0.9125
10*100000	10	0.065079	9.10	0.9100
16*100000	**16**	**0.091082**	**14.50**	**0.9062**
24*100000	24	0.111708	21.70	0.9042
32*100000	32	0.148139	28.90	0.9031
64*100000	64	0.251571	57.70	0.9016
72*100000	72	0.274441	64.90	0.9014
96*100000	96	0.356943	86.50	0.9010
104*100000	104	0.376941	93.70	0.9010
112*100000	112	0.398003	100.90	0.9009
128*100000	128	0.470686	115.30	0.9008







According to weak scaling, we expect that as we increase the problem size linearly with the increasing parallel workload, the execution time will remain approximately constant. However, in our performance, we observed that the execution time started to rise as we increased both the problem size and the number of cores (see Figure 1). Nevertheless, the speedup, as shown in Table 1 and Figure 2, exhibits a linear behavior in terms of the number of nodes, without any upper limit. Additionally, the efficiency for [2 – 128] cores only vary between [0.95 – 0.91], indicating a relatively constant efficiency. Hence, we can conclude that depending on the size of our problem, we can determine the number of cores needed to achieve efficient performance.

▪ Conclusion

In the **strong scaling** analysis, as illustrated in Table 1 and Figures 1, 2, and 3, increasing the number of cores leads to a decrease in execution time. We performed the analysis for 128 cores on 1 epyc node and observed that for the **first 32 cores**, the increase in the number of cores significantly decreased the execution time while also increasing their corresponding speedup. However, doubling the number of cores in compared to 32 cores, (64) or even tripling them (128) did not significantly decrease execution time; and it remained approximately constant. Therefore, considering more than 32 cores to run our model appears to be wasteful in terms of resources. If we are limited in resource usage, considering **only 2 or 4 cores** results in halving the execution time and even less for 4 cores, (from 0.026169 to 0.016569 and 0.009702)**,** while maintaining an efficiency above 0.5 (i.e. 0.789698 for 2 cores and 0.674320 for 4 cores).

In weak scaling test of our model, the execution time of problem did not remain constant with increasing the problem size adjusted by increasing the number of the cores. However, we observed a linear behavior for speedup and a consistent trend for the efficiency of the model which is relatively constant. From this, we can conclude that to achieve the best results for our model, we need to adjust the number of cores depending on the problem size. For smaller datasets, fewer cores may be sufficient, whereas for larger problem sizes, more cores are needed.

In conclusion, our **MPI Quick Sort** model exhibited **better behavior in the strong scaling test**, as the execution time decreased with the increasing number of cores from 1 to **32**. However, in the weak scaling test, our model did not meet its expectation.

▪ Possible improvements on the code:

1. **Optimize memory** usage by minimizing unnecessary allocations and deallocations.
2. Add **error handling** to MPI function calls for robustness and fault tolerance.
3. Explore **load balancing** techniques to distribute data more evenly among processes.
4. Consider **optimizing the quicksort algorithm** for better performance on large datasets.

The Codes, Slurms and Readme file are available on **Github** at:
https://github.com//2021Assignment01