Parallel Sort Investigation – Lab Assignment

Student details: Manghiuc Ioana, Informatica, 10LF223

Device configuration:

Laptop: ASUS VivoBook S 15 M3502RA

Processor AMD Ryzen™ 9 6900HX

Base speed: 3.30 GHz

Cores: 8

Logical processors: 16

Integrated GPU: AMD Radeon™ 680M

RAM: 16GB

**Evaluation and discussion of performance results**

* 1. Execution times, communication times, and processing times

The execution and computation time, as well as the communication overhead, were measured as follows:

The Bucket Sort, Odd-Even Sort, and Shell Sort algorithms were tested using two, four, and eight (my device’s maximum) cores. Then, for each core configuration the algorithm was tested three times to ensure data consistency. All these three algorithms were tested on an array of 1.000.000 elements and on an array of 10.000.000 elements, which are stored in two separate files.

The Direct (Selection) Sort was tested only on an array of 1.000.000 elements, using two, four, and eight cores. Each core configuration was tested three times.

The Ranking Sort algorithm was tested on an array of 1.000.000 elements using two, four, and eight cores. Each core configuration was tested once. I made this decision because the execution time with eight cores was already past the thirty minute mark, and in my previous tests where I was only measuring the execution time, the result was similar, and therefore concluded the differences in minutes or seconds would do little to aid in the following analysis.

* + 1. EXECUTION TIME

To measure the total execution time of each parallel sorting algorithm, I used the MPI\_Barrier and MPI\_Wtime functions, in the following manner:

* MPI\_Barrier(MPI\_COMM\_WORLD) was called **before** and **after** the sorting function to ensure that all processes are synchronized. This guarantees that the timing starts and ends consistently across all MPI ranks.
* MPI\_Wtime() was used to record the wall-clock time in seconds. It provides a high-resolution timestamp of the current time, making it suitable for performance measurements in parallel applications.
* The computation\_time and communication\_time variables are updated **inside the sorting function** using additional calls to MPI\_Wtime() to break down the execution time into computation vs. communication components. These values are passed by reference to the sorting function.
* The signature of the sorting function is identical for each algorithm.

The code is structured as:

MPI\_Barrier(MPI\_COMM\_WORLD);

double computation\_time = 0.0, communication\_time = 0.0;

double execution\_time = MPI\_Wtime();

// Sorting function call

MPI\_OddEvenSort(data, rank, size, computation\_time, communication\_time);

MPI\_Barrier(MPI\_COMM\_WORLD);

execution\_time = MPI\_Wtime() - execution\_time;

* + 1. COMPUTATION TIME & COMMUNICATION OVERHEAD

To evaluate the performance of each parallel sorting algorithm, I measured both the **computation time** and the **communication overhead** separately using MPI\_Wtime(). These times are calculated inside the sorting functions and accumulated in computation\_time and communication\_time variables passed by reference.

The computation time includes only the time spent on performing local calculations that typically include sorting local partitions (using std::sort or DirectSort), calculating ranks, maximum values, or bucket assignments, merging sorted chunks, local comparisons or evaluations in loops.

Each computational section is structured as follows:

start\_time = MPI\_Wtime();

// perform computation

computation\_time += MPI\_Wtime() - start\_time;

Communication overhead was measured by wrapping all MPI communication calls with MPI\_Wtime() and summing the durations. These include:

* MPI\_Scatter / MPI\_Gather / MPI\_Gatherv: Distributing or collecting data
* MPI\_Bcast: Broadcasting data to all processes (used in Ranking Sort)
* MPI\_Sendrecv: Peer-to-peer communication (Odd-Even and Shell Sorts)
* MPI\_Allreduce: Global status aggregation or finding max values
* MPI\_Alltoallv / MPI\_Alltoall: Complex many-to-many exchanges (Bucket Sort)

Each communication section is structured as follows:

start\_time = MPI\_Wtime();

MPI\_<function>();

communication\_time += MPI\_Wtime() - start\_time;

For all five sorting algorithms, the communication overhead has a significantly less value in comparison to the computation time, which oftentimes was very close in value, if not identical, to the total execution time, meaning the time spent by MPI processes for communication does not affect the total time greatly. In the following scalability analysis, I will provide real data that I recorded for a better understanding.

* + 1. SCALABILITY ANALYSIS

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sorting method | Data size | Cores | | Execution nr. | Execution time | Computation Time | Communication time |
| Direct Sort | 1.000.000 | 2 | | 1. | 1435.58 | 1435.34 | 0.238449 |
| 2. | 1311.69 | 1311.69 | 0.0012313 |
| 3. | 1342.55 | 1340.84 | 1.7028 |
| ***AVERAGE TIMES*** | | | | ***1363.27*** | ***1362.62*** | ***0.64749*** |
| 1.000.000 | 4 | 1. | | 462.747 | 460.397 | 2.34866 |
| 2. | | 362.982 | 360.679 | 2.31723 |
| 3. | | 521.992 | 519.621 | 2.36876 |
| ***AVERAGE TIMES*** | | | | ***449.240*** | ***446.899*** | ***2.344883*** |
| 1.000.000 | 8 | 1. | | 101.841 | 101.541 | 0.297848 |
| 2. | | 124.074 | 122.728 | 1.34468 |
| 3. | | 138.478 | 137.347 | 1.12929 |
| ***AVERAGE TIMES*** | | | | ***121.4643*** | ***120.5386*** | ***0.9239*** |
| The execution time for Direct Sort increases as the number of MPI processes decreases, due to fewer cores sharing the computational workload. Although the use of MPI parallelism does improve performance to some extent, the algorithm’s intrinsic time complexity of O(n2) severely limits its scalability.  While the initial partitioning and gathering of data are parallelized, the rest of the sorting work remains local to each process.  While testing on a smaller dataset of 1,000 elements, I observed that the collation (merge) operation was the most time-consuming. To optimize this part of the implementation, I used rvaluereferences for the input arrays, allowing them to be safely moved from. Additionally, I applied std::move to transfer ownership of elements rather than copying them, thereby reducing memory overhead and improving efficiency during the merge.  Despite this optimization, the algorithm still exhibits poor scalability, with diminishing returns as the number of processes decreases or the dataset size increases. This makes Direct Sort a weak candidate for scalable parallel execution when compared to other algorithms that distribute both computation and merging more evenly across processes. | | | | | | | |
| Bucket sort | 1.000.000 | 2 | 1. | | 0.374329 | 0.369164 | 0.0025128 |
| 2. | | 0.368792 | 0.364334 | 0.0023298 |
| 3. | | 0.36999 | 0.364954 | 0.0023744 |
| ***AVERAGE TIMES*** | | | | ***0.371037*** | ***0.36615*** | ***0.0024056*** |
| 10.000.000 | 2 | 1. | | 4.44484 | 4.40238 | 0.0249644 |
| 2. | | 4.42754 | 4.38871 | 0.0195519 |
| 3. | | 4.44286 | 4.40592 | 0.0178908 |
| ***AVERAGE TIMES*** | | | | ***4.438413*** | ***4.399003*** | ***0.02080236*** |
| 1.000.000 | 4 | 1. | | 0.369947 | 0.360363 | 0.0072893 |
| 2. | | 0.360507 | 0.354747 | 0.0034577 |
| 3. | | 0.362048 | 0.35565 | 0.004141 |
| ***AVERAGE TIMES*** | | | | ***0.3641673*** | ***0.35692*** | ***0.0049626*** |
| 10.000.000 | 4 | 1. | | 4.1921 | 4.15086 | 0.028771 |
| 2. | | 4.189 | 4.15473 | 0.0214581 |
| 3. | | 4.47079 | 4.43848 | 0.0195564 |
| ***AVERAGE TIMES*** | | | | ***4.283963*** | ***4.248023*** | ***0.02326183*** |
| 1.000.000 | 8 | 1. | | 0.348963 | 0.342014 | 0.0049728 |
| 2. | | 0.362234 | 0.355027 | 0.0050962 |
| 3. | | 0.356924 | 0.344396 | 0.0105488 |
| ***AVERAGE TIMES*** | | | | ***0.3560403*** | ***0.3471456*** | ***0.0068726*** |
| 10.000.000 | 8 | 1. | | 4.17184 | 4.11901 | 0.038731 |
| 2. | | 4.13118 | 4.06617 | 0.0550711 |
| 3. | | 4.13967 | 4.08679 | 0.0413536 |
| ***AVERAGE TIMES*** | | | | ***4.147563*** | ***4.090656*** | ***0.0450519*** |

The execution time for the Bucket Sort algorithm remains relatively consistent across different numbers of MPI processes. For a dataset of 10 million elements, the execution time only varies slightly (e.g., between 4.10 and 4.45 seconds). This stability reflects efficient workload distribution through balanced value-based buckets.

The values are effectively partitioned into nearly balanced buckets, with each process having data within an allocated range (e.g., process 0 has data between 0 and 250 etc.), and local sorting is performed using std::sort, which is much faster than the Selection Sort used in Direct Sort. This combination ensures low computation time and strong scalability regardless of the number of processes.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Odd Even | 1.000.000 | 2 | 1. | 0.203022 | 0.169429 | 0.0284726 |
| 2. | 0.204297 | 0.170137 | 0.0271901 |
| 3. | 0.20222 | 0.168313 | 0.0288998 |
| ***AVERAGE TIMES*** | | | ***0.2031796*** | ***0.169293*** | ***0.0281875*** |
| 10.000.000 | 2 | 1. | 2.29512 | 1.96192 | 0.282614 |
| 2. | 3.48884 | 2.98204 | 0.451242 |
| 3. | 2.37936 | 2.02593 | 0.278151 |
| ***AVERAGE TIMES*** | | | ***2.721106*** | ***2.323296*** | ***0.3373356*** |
| 1.000.000 | 4 | 1. | 0.143749 | 0.0883962 | 0.0313318 |
| 2. | 0.136588 | 0.0816625 | 0.0245181 |
| 3. | 0.141281 | 0.0906397 | 0.0245264 |
| ***AVERAGE TIMES*** | | | ***0.1405393*** | ***0.08689946*** | ***0.0267921*** |
| 10.000.000 | 4 | 1. | 1.61296 | 1.03227 | 0.289638 |
| 2. | 1.91337 | 1.12219 | 0.419559 |
| 3. | 1.61772 | 1.0686 | 0.275457 |
| ***AVERAGE TIMES*** | | | ***1.714683*** | ***1.074353*** | ***0.328218*** |
| 1.000.000 | 8 | 1. | 0.126923 | 0.0533552 | 0.0265541 |
| 2. | 0.11382 | 0.0488368 | 0.0245451 |
| 3. | 0.115237 | 0.0470248 | 0.0296468 |
| ***AVERAGE TIMES*** | | | ***0.11866*** | ***0.04973893*** | ***0.0269153*** |
| 10.000.000 | 8 | 1. | 1.21149 | 0.530941 | 0.27321 |
| 2. | 1.60533 | 0.823481 | 0.321144 |
| 3. | 1.20826 | 0.532383 | 0.291041 |
| ***AVERAGE TIMES*** | | | ***1.341693*** | ***0.628935*** | ***0.2951316*** |

The compare and exchange Odd-Even algorithm has the best overall performance among the five implementations. Although the difference between execution times for different numbers of MPI processes is more noticeable, it consistently outperforms the others in terms of speed. The iterative compareAndExchange pattern allows processes to refine their data ordering through minimal and structured communication. As such, even with communication overhead it remains the fastest solution for both 1 million and 10 million datasets.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Ranking Sort | 1.000.000 | 2 | 1. |  |  |  |
| 4 | 1. | 3966.48 | 3966.47 | 0.003315 |
| 8 | 1. | 2337.14 | 2331.05 | 6.08485 |

The Ranking Sort method very noticeably delivers the worst performance out of all the five implementations. Due to its O(n2) time complexity, it scales poorly and remains inefficient even when parallelized. Although using eight MPI processes improves execution time compared to four or two, the total still reaches a decidedly inefficient 2337 seconds, which is abysmal even compared to the slowest Direct Sort run. This highlights the algorithm’s fundamental inefficiency and poor suitability for large-scale parallel processing.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Shell Sort | 1.000.000 | 2 | 1. | 0.199478 | 0.193433 | 0.0016941 |
| 2. | 0.202363 | 0.189831 | 0.0017396 |
| 3. | 0.198994 | 0.19036 | 0.0016229 |
| ***AVERAGE TIMES*** | | | ***0.2002783*** | ***0.191208*** | ***0.00168553*** |
| 10.000.000 | 2 | 1. | 2.32521 | 2.25006 | 0.015385 |
| 2. | 2.27525 | 2.20085 | 0.0146576 |
| 3. | 2.27381 | 2.20846 | 0.0157982 |
| ***AVERAGE TIMES*** | | | ***2.2914233*** | ***2.21979*** | ***0.01528026*** |
| 1.000.000 | 4 | 1. | 0.133759 | 0.109682 | 0.0020478 |
| 2. | 0.138265 | 0.107101 | 0.002021 |
| 3. | 0.131723 | 0.108722 | 0.0022244 |
| ***AVERAGE TIMES*** | | | ***0.1345823*** | ***0.1085016*** | ***0.00209773*** |
| 10.000.000 | 4 | 1. | 2.25899 | 1.94434 | 0.0245447 |
| 2. | 1.52009 | 1.28654 | 0.0233569 |
| 3. | 1.54834 | 1.28483 | 0.0240759 |
| ***AVERAGE TIMES*** | | | ***1.775806*** | ***1.505236*** | ***0.0239925*** |
| 1.000.000 | 8 | 1. | 0.0885662 | 0.0612442 | 0.0025764 |
| 2. | 0.0922914 | 0.0617859 | 0.003225 |
| 3. | 0.0994363 | 0.0635539 | 0.0025386 |
| ***AVERAGE TIMES*** | | | ***0.0934313*** | ***0.0621946*** | ***0.00278*** |
| 10.000.000 | 8 | 1. | 1.36737 | 1.00849 | 0.0325305 |
| 2. | 1.36252 | 1.09506 | 0.0313459 |
| 3. | 1.36553 | 1.11074 | 0.031535 |
| ***AVERAGE TIMES*** | | | ***1.36514*** | ***1.07143*** | ***0.0318038*** |

Following very closely behind the Odd-Even implementation, the execution times recorded for the Shell Sort algorithm are the second most efficient. Its hybrid structure—using local std::sort followed by structured compareAndMerge operations—enables efficient parallel sorting with minimal communication overhead. While its execution time does increase as the number of MPI processes decreases, Shell Sort scales well overall and remains highly efficient even for larger datasets.

* 1. Discussion and analysis of results
     1. COMPARATIVE ANALYSIS OF SORTING METHODS

Compare the performance of the implemented sorting algorithms in terms of execution time and communication overhead. Analyze which algorithms perform best and what factors influence this performance.

To reiterate, as far as execution time goes from best to worst performance, the algorithm rank as follo

* + 1. DISCUSSION OF COMMUNICATION OVERHEAD

Provide a detailed discussion of how communication time affects algorithm performance, particularly for distributed algorithms. Which algorithm is most sensitive to communication latency, and how could it be optimized?

* + 1. ANALYSIS OF COMPUTATIONAL TIME AND BOTTLENECKS

Detail the process steps for each algorithm and identify potential bottlenecks or inefficiencies in terms of computation. If applicable, analyze whether implementing a more efficient computation technique would reduce processing time

* + 1. SPEEDUP AND EFFICIENCY EVALUATION

Discuss the speedup achieved by using MPI compared to a sequential implementation of the sorting algorithms. Analyze the "theoretical speedup" vs. "actual speedup" and explain why deviations exist.

* + 1. POSSIBLE INEFFICIENCIES AND SUGGESTIONS FOR IMPROVEMENT

Identify and discuss potential inefficiencies and resource contention (e.g., processes not being utilized effectively, where communication time exceeds processing time). Provide clear suggestions for improving the implementation (e.g., optimizing data flow, load balancing strategies).