Project 3 Report for CSC4005 Parallel Computing

Jingquan Li (119010148)

November 17, 2024

Contents

1 Introduction

Sorting algorithms are essential in computer science for organizing data efficiently. This project explores how to implement these algorithms in a distributed and parallel environment to leverage the power of multi-core processors for better performance.

I implemented four parallel sorting algorithms: Process-Level Parallel Bucket Sort, Parallel Quick Sort with K-Way Merge, Parallel Sorting with Regular Sampling (PSRS), and Thread-Level Parallel Merge Sort.

In this report, I provide a summary of the implementation process, the experimental results, and the performance optimizations applied, along with profiling analysis to evaluate their effectiveness.

2 Compilation and Execution

To compile and execute the program on the cluster, I used two bash scripts: `sbatch-uniform.sh` and `sbatch-normal.sh`. These scripts automate the process of running sorting algorithms on different datasets. Below is a summary of the key compilation and execution steps.

2.1 Compilation

The project is built using `CMake`. To compile the code, navigate to the project directory and execute the following commands:

```
cd /path/to/project3
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j4
```

These commands will generate the necessary build files and compile the code with optimization level `-O2` for improved performance. The compiled executables are placed in the `build/src/` directory.

2.2 Execution on the Cluster

Once the compilation is complete, the provided `sbatch` scripts can be used to submit jobs to the cluster. These scripts utilize SLURM workload manager commands to handle the execution of each sorting algorithm.

The commands to submit jobs using SLURM are:

```
sbatch ./src/sbatch-uniform.sh
sbatch ./src/sbatch-normal.sh
```

During execution, the `perf` tool is used to collect performance metrics such as CPU cycles, cache misses, and page faults. This is done to understand the behavior and efficiency of the parallel algorithms in more depth. The `srun` command is used for each task to execute the compiled code, specifying the number of cores, tasks, and other necessary parameters to ensure optimal use of cluster resources.

3 Design and Implementation

3.1 Parallel Bucket Sort with MPI

The Bucket Sort algorithm was implemented using MPI to achieve process-level parallelism, focusing on optimizing performance for large datasets. The approach involved dividing the input data into multiple buckets, with each bucket assigned to a different MPI process for sorting.

Data Distribution and Sorting: The input vector is divided among all available MPI processes. The MASTER process initially splits the data into chunks based on the number of processes and sends these chunks to worker processes using `MPI_Send`. This approach helps to distribute the workload evenly across the available nodes, aiming to minimize load imbalance.

Once the data is received, each process assigns the data to several buckets. The bucket boundaries are determined based on the range of values in the data, ensuring that data values are distributed approximately evenly. Each bucket is then sorted locally using the `insertionSort` function. Insertion Sort is chosen for its efficiency on small datasets, which is typical for each bucket after partitioning.

Communication and Gathering Results: After sorting, each process gathers its locally sorted data, and all sorted results are sent back to the MASTER process using `MPI_Send`. The MASTER process then merges all sorted buckets and performs a final sort to ensure that the overall dataset is in order.

One of the main challenges addressed was the potential load imbalance, especially if data was not evenly distributed among the buckets. This was mitigated by dynamically calculating bucket boundaries and ensuring that each process received a proportional amount of data. Furthermore, careful use of `MPI_Probe` and `MPI_Get_count` allowed efficient handling of varying message sizes between processes.

Key Challenges and Solutions: (1) Load Imbalance: To mitigate load imbalance, I calculated bucket boundaries based on the range of values, ensuring that the data was split as evenly as possible. (2) Efficient Communication: The use of `MPI_Probe` and `MPI_Get_count` enabled dynamic handling of message sizes, which is crucial for efficient data exchange. (3) Error Handling: Throughout the code, MPI error handling was implemented using `MPI_Abort` to gracefully handle communication errors and prevent deadlocks.

3.2 Parallel Quick Sort with K-Way Merge using MPI

The Parallel Quick Sort algorithm was implemented using MPI to achieve parallel sorting across multiple processes. The input dataset was divided into blocks, with each block being sorted independently by a separate MPI process using the standard quick sort algorithm. After sorting, the sorted sublists were merged using a K-Way merge approach to obtain the final sorted output.

Data Distribution and Sorting: The MASTER process initially divides the input data among all available processes. This division is based on the number of processes, ensuring that each process receives an approximately equal portion of the dataset. The MASTER process retains its own portion and sends the remaining segments to worker processes using `MPI_Send`.

Each process then sorts its local segment independently using the C++ `std::sort` function. This local sorting ensures that each partition of the dataset is sorted in parallel, thereby reducing the overall sorting time.

K-Way Merge: Once all the processes have sorted their respective partitions, the MASTER process gathers the sorted segments back using `MPI_Recv`. To merge these sorted segments into a single sorted list, I used a K-Way merge algorithm. The merging was implemented using a min-heap (priority queue) to efficiently merge multiple sorted lists into one. Each sorted partition was treated as a separate list, and the min-heap was used to always extract the smallest element from the available lists, ensuring that the final result was sorted.

Key Challenges and Solutions: (1) Data Partitioning: Ensuring that each process received an equal amount of data was challenging due to potential remainder elements. To address this, I distributed the remainder elements among the first few processes to balance the load. (2) Efficient Merging: Merging multiple sorted segments efficiently was achieved using a priority queue, which allowed me to maintain a sorted order while merging multiple lists. This minimized the time complexity compared to a naive merging approach. (3) Communication Overhead: Data distribution and collection were managed using `MPI_Send` and `MPI_Recv` to ensure that all processes received and returned their data efficiently. By using these functions along with careful offset calculations, I minimized the communication overhead.

3.3 Parallel Sorting with Regular Sampling (PSRS) using MPI

The Parallel Sorting with Regular Sampling (PSRS) algorithm was implemented using MPI to efficiently handle large datasets while ensuring balanced workloads among processes. The PSRS algorithm involves multiple phases, including data distribution, local sorting, regular sampling, pivot selection, data exchange, and final merging.

Data Distribution: In the first phase, the MASTER process generates the input dataset and distributes it among all available MPI processes using `MPI_Scatterv`. Each process receives a portion of the dataset to sort locally. The dataset is divided as evenly as possible to ensure that each process has a similar amount of data, minimizing load imbalance.

Local Sorting and Regular Sampling: After receiving their data, each process sorts its local partition using `std::sort`. Once sorted, each process selects a set of regular samples from its sorted data. These regular samples are used to help determine global pivot values, which are crucial for effectively partitioning the entire dataset across all processes.

Pivot Selection: The local regular samples from all processes are gathered at the MASTER process using `MPI_Gather`. The MASTER process then sorts all the gathered samples and selects a set of global pivots. These pivots are broadcast to all processes using `MPI_Bcast` to ensure that each process has the information necessary for partitioning its data in the next phase.

Data Partitioning and Exchange: Each process uses the global pivots to partition its sorted data into segments. These segments are then exchanged among the processes using `MPI_Alltoallv`, ensuring that each process receives data that falls within a specific range defined by the pivots. This step ensures that each process ends up with a portion of the dataset that is locally sorted and falls within a distinct range of values.

Merging Local Partitions: After exchanging data, each process merges the received segments using `std::sort`. This results in each process having a fully sorted local partition of the dataset.

Final Gathering: In the final phase, the MASTER process gathers all the sorted local partitions from the processes using `MPI_Gatherv`. The gathered data represents the fully sorted version of the original dataset.

Key Challenges and Solutions: (1) Load Balancing: Ensuring that each process received an equal amount of data was crucial for minimizing load imbalance. The use of regular sampling helped to determine effective pivot values, ensuring even data distribution. (2) Efficient Data Exchange: Data exchange between processes was managed using `MPI_Alltoallv`, which allowed each process to send and receive variable-sized segments. This approach was crucial for maintaining efficiency and ensuring that each process received data corresponding to the correct value range. (3) Communication Overhead: The use of collective communication functions such as `MPI_Gather`, `MPI_Bcast`, and `MPI_Alltoallv` helped manage the complexity of data exchange while ensuring that the overhead remained manageable.

3.4 Parallel Merge Sort using OpenMP

The Parallel Merge Sort algorithm was implemented using OpenMP to achieve thread-level parallelism. The approach involved recursively dividing the dataset into smaller sublists, sorting each sublist in parallel, and then merging them to obtain the final sorted output.

Recursive Division and Sorting: The dataset is recursively divided into smaller parts until each sublist contains only one element. This is achieved using a recursive function, `mergeSort()`, which takes the left and right bounds of the current sublist. To parallelize this operation, I utilized the OpenMP `#pragma omp parallel` and `#pragma omp task` directives. These directives allowed different portions of the dataset to be sorted simultaneously by multiple threads.

Merging Phase: Once the sublists are sorted, they need to be merged back together. The merging phase is also parallelized using OpenMP. The `merge()` function combines two sorted sublists into a single sorted list. By using `#pragma omp task` and `#pragma omp taskwait`, the merging operations are performed concurrently where possible, which helps to speed up the process.

Key Challenges and Solutions: (1) Thread Management: One key challenge was managing the number of threads to avoid oversubscription, which could degrade performance. By controlling the number of threads using `num_threads` and balancing the workload across threads, I ensured efficient use of computational resources. (2) Efficient Synchronization: Proper synchronization was necessary to ensure that merging operations were completed in the correct order. The use of `#pragma omp taskwait` helped manage dependencies between sorting and merging tasks, ensuring correctness without introducing significant overhead. (3) Load Balancing: Dividing the dataset evenly among threads was important to prevent any single thread from becoming a bottleneck. The recursive division approach naturally distributed the workload, while OpenMP's dynamic scheduling further helped in balancing the load.

4 Experimental Results and Analysis

The performance results for both uniform and normal datasets were gathered across various worker configurations (1, 4, 16, and 32 workers) for each parallel sorting algorithm. Here, I analyze the execution time, speedup, efficiency, and other metrics such as CPU cycles, cache misses, and page faults.

4.1 Uniform Dataset Analysis

The performance of each algorithm on the uniform dataset is summarized in Figure 1.

Best and Worst Algorithms: The PSRS algorithm showed the best performance in terms of speedup and efficiency, particularly at 16 workers, where it achieved a speedup of 7.7114 and an efficiency of 0.482. On the other hand, Merge Sort consistently showed the worst performance, with very low speedup values (0.1081 with 32 workers) and efficiency nearing 0.0034 at 32 workers. This indicates that Merge Sort was not well-suited for the workload due to significant synchronization overhead and poor parallel scalability.

Execution Time: PSRS had the lowest execution time among all algorithms at 2148 ms with 16 workers, making it the most effective for large-scale parallel sorting on uniform data. Conversely, Merge Sort took significantly longer, even with increased worker counts, demonstrating its limitations in the context of parallel execution.

Speedup and Efficiency: PSRS displayed the most promising scalability, while the efficiency values for Merge Sort suggest that adding more workers beyond a small count did not yield proportional gains. This is likely due to the high synchronization cost and inherent dependencies in the merge phase, which hinder parallelization.

	# of workers	execution time (milliseconds)	cpu-cycles	cache-misses	page-faults	speedup	efficiency
standard	1	12315	42,831,703,469	74,118,682	361,798		
bucket sort	1	33347	134,456,535,787	508,466,375	1,534,594	0.3673	0.3673
bucket sort	4	24162	121,449,673,951	409,268,574	1,003,616	0.6349	0.1587
bucket sort	16	16817	91,293,268,474	51,048,075	280,250	0.9304	0.0582
bucket sort	32	24600	122,507,599,166	36,873,272	285,817	1.5188	0.0475
quick sort	1	13933	82,725,140,265	157,377,352	1,005,599	0.8887	0.8887
quick sort	4	9270	85,375,817,356	126,544,133	793,836	1.7828	0.4457
quick sort	16	8229	81,356,735,332	11,453,433	249,411	2.023	0.1264
quick sort	32	9002	83,106,885,012	8,670,518	246,743	1.8461	0.0577
psrs	1	17795	99,283,048,298	273,356,790	1,088,111	0.8294	0.8294
psrs	4	7007	76,741,275,246	56,619,802	122,716	2.318	0.5795
psrs	16	2148	63,952,618,162	10,663,545	27,694	7.7114	0.482
psrs	32	2974	69,511,517,652	6,278,975	19,721	7.2448	0.2264
merge sort	1	195923	307,990,370,378	239,632,839	52,741	0.0604	0.0604
merge sort	4	130396	456,620,154,824	905,531,562	102,190	0.0909	0.0227
merge sort	16	132401	462,546,095,201	921,308,032	119,203	0.0894	0.0056
merge sort	32	109853	368,499,076,876	509,976,862	148,739	0.1081	0.0034

Figure 1: Performance for uniform distributed dataset

Resource Utilization: Quick Sort also showed good performance, especially with 4 and 16 workers, balancing speedup and efficiency better than Merge Sort or Bucket Sort. Bucket Sort exhibited a modest speedup, but its efficiency decreased rapidly beyond 16 workers, suggesting that communication overhead was a limiting factor in performance gains.

4.2 Normal Dataset Analysis

The performance of each algorithm on the normal dataset is summarized in Figure 2.

Best and Worst Algorithms: Again, PSRS emerged as the best performer in terms of speedup and efficiency, particularly with 32 workers (speedup of 6.6323 and efficiency of 0.2073). Merge Sort, on the other hand, exhibited the worst performance, with negligible speedup and extremely low efficiency values, reaffirming its limitations in a parallel context.

Execution Time: PSRS achieved the best execution time of 2775 ms with 16 workers, while Merge Sort struggled with over 118345 ms even at higher worker counts. The consistent poor performance of Merge Sort highlights its unsuitability for high-level parallelism, especially in distributed memory environments.

Scalability and Efficiency: The efficiency of PSRS remained reasonable up to 16 workers, whereas other algorithms, particularly Merge Sort and Bucket Sort, experienced significant drops in efficiency as worker counts increased. The improved performance of PSRS can be attributed to its efficient load balancing using pivots and minimal communication compared to the other algorithms.

Resource Utilization: Quick Sort also performed well on the normal dataset, showing good scalability up to 16 workers before efficiency started declining. Bucket Sort, while moderately effective, did not scale well beyond 16 workers, which suggests bottlenecks due to load imbalance or communication overhead.

The results demonstrate that the PSRS algorithm is the most suitable for parallel sorting on both uniform and normal datasets. It achieved the highest speedup and efficiency among the tested algorithms. Quick Sort also showed reasonable performance and scalability, especially at lower worker counts.

Merge Sort consistently performed poorly in both datasets, suffering from high synchronization overhead and

	# of workers	execution time (milliseconds)	cpu-cycles	cache-misses	page-faults	speedup	efficiency
standard	1	11700	50,616,129,269	52,905,765	2,109		
bucket sort	1	14625	91,735,333,926	335,586,683	322,849	0.8064	0.8064
bucket sort	4	17534	113,423,526,517	83,317,248	55,965	0.8583	0.2146
bucket sort	16	11724	98,125,785,862	45,249,024	24,604	1.29	0.0806
bucket sort	32	11543	98,264,229,470	37,087,384	13,554	1.3161	0.0411
quick sort	1	13388	90,413,420,013	150,983,242	9,148	0.9053	0.9053
quick sort	4	8616	92,484,637,215	28,942,677	5,477	1.8792	0.4698
quick sort	16	7887	90,343,497,732	14,931,739	15,790	2.0616	0.1289
quick sort	32	8573	92,916,308,889	13,104,373	4,669	1.9302	0.0603
psrs	1	16511	99,544,656,383	265,630,619	1,066,366	0.7655	0.7655
psrs	4	7355	89,286,946,926	59,501,065	124,239	2.2054	0.5514
psrs	16	2775	76,918,314,967	10,902,470	27,183	5.8699	0.3669
psrs	32	3386	76,028,508,495	6,060,326	19,709	6.6323	0.2073
merge sort	1	202096	324,889,556,463	347,036,438	1,109,121	0.0606	0.0606
merge sort	4	120122	402,975,941,981	568,224,046	1,289,023	0.1027	0.0257
merge sort	16	118345	392,919,368,962	382,236,983	1,233,185	0.1181	0.0074
merge sort	32	119395	397,069,448,576	484,631,792	1,298,186	0.1039	0.0032

Figure 2: Performance for normal distributed dataset

poor scalability. Bucket Sort showed decent performance initially but struggled with scalability, particularly beyond 16 workers. The reduced performance of these algorithms can be attributed to several factors: Merge Sort's synchronization issues leading to excessive waiting times among threads, and Bucket Sort's communication overhead when distributing data and collecting results.

5 Optimizations Applied to Improve Performance

5.1 Data Partitioning and Load Balancing

In all the MPI-based algorithms, careful data partitioning was crucial to ensure balanced workloads among the different processes. Here are some of the approaches I used:

- Dynamic Data Partitioning in Bucket Sort: During Bucket Sort, I ensured that the range of values was dynamically divided into buckets based on the min and max values of the input. This helped in preventing load imbalance where some buckets might have significantly more elements compared to others.
- Regular Sampling in PSRS: The PSRS algorithm used regular sampling to determine global pivots that helped in evenly dividing the data among all processes. By selecting pivots based on sampled data, I ensured that each process received approximately the same amount of data, which greatly reduced load imbalance.

5.2 Communication Optimization

The key to improving performance in MPI-based parallel sorting was reducing communication overhead. In Bucket Sort, Quick Sort, and PSRS, I used non-blocking communication where feasible and minimized the frequency of data exchange. For example:

- `MPI_Scatterv` and `MPI_Gatherv`: In PSRS, `MPI_Scatterv` and `MPI_Gatherv` were used for distributing and gathering variable-sized data, ensuring that communication was optimized based on the actual amount of data each process needed to send or receive. This reduced unnecessary waiting and helped balance communication load.
- Collective Communication for Pivot Broadcast: During PSRS, pivots were broadcasted using `MPI_Bcast`, which is more efficient than point-to-point communication. This collective operation helped to reduce the time spent in synchronizing all processes for the pivot information.

5.3 Task Parallelism with OpenMP

In the Merge Sort implementation using OpenMP, I employed task parallelism to enhance performance. Here are the key optimizations:

- Recursive Task Parallelism: I used `#pragma omp task` to create tasks for recursive calls to `mergeSort()`. By allowing the recursive splitting to happen in parallel, the workload was distributed among available threads, which sped up the sorting of individual sublists.
- Task Synchronization: Using `#pragma omp taskwait`, I ensured proper synchronization between tasks to maintain data consistency. However, I carefully minimized synchronization points to reduce overhead, which could otherwise negate the performance gains from parallelization.

5.4 Minimizing Synchronization Overheads

- Reducing Lock Contention in Merge Sort: One of the key challenges in Merge Sort is merging sorted sublists concurrently. Instead of using a global lock (which could create significant contention), I allowed each thread to work on independent segments and merged them hierarchically, reducing the need for synchronization.
- K-Way Merge with Min-Heap for Quick Sort: For merging sorted segments in Parallel Quick Sort, I implemented a K-Way merge using a min-heap (priority queue). This approach ensured that the merging phase was efficient, with logarithmic complexity relative to the number of segments being merged. By using a heap, I minimized the overhead compared to a linear merge, thereby improving performance.

5.5 Memory Optimization

- In-Place Sorting in Bucket Sort: To reduce memory overhead, I used in-place sorting techniques within each bucket, which lowered the auxiliary memory requirement. This was particularly beneficial for larger datasets where memory usage could easily become a bottleneck.
- Minimizing Temporary Buffers in PSRS: During the partitioning and merging phases of PSRS, I minimized the use of temporary buffers by directly working with the data structure used for gathering and merging. This reduced memory footprint and improved cache locality, leading to faster access times.

6 Answers for the TODO in tutorial slides

For Task 1, apply Bucket Sort on both uniform and non-uniform datasets to see the difference.

In the case of the uniform dataset, Bucket Sort performed relatively well, as the data distribution across buckets was more balanced. This allowed each bucket to contain roughly an equal number of elements, thereby reducing the time required for sorting within each bucket.

On the other hand, the non-uniform dataset led to significant load imbalance, with some buckets containing many more elements than others. This imbalance resulted in higher sorting times for those overloaded buckets, leading to a decrease in overall efficiency. The non-uniform distribution negatively impacted performance, primarily because the bucket assignment was heavily skewed, which is a fundamental challenge of Bucket Sort when the data is not uniformly distributed.

For Task 2, measure the time consumption for local std::sort and the final merge. Do experiments on multi-processes to see how merge becomes the bottleneck.

The profiling results indicated that, as the number of processes increased, the time spent on the final merge phase began to dominate the overall runtime, effectively becoming the bottleneck.

With fewer processes (e.g., 4 workers), the local std::sort operations took a substantial portion of the overall execution time. However, as the number of processes increased (e.g., 16 or 32 workers), the merging step required a significant amount of time due to the increased overhead of combining more partitions. This merging phase involved sequential operations that limited the achievable parallel speedup, especially since merging inherently requires coordination among multiple processes.

The bottleneck was observed primarily due to the linear nature of the merging operation, which could not be parallelized as effectively as the initial sorting of local data segments. The K-Way merging process introduced communication and coordination overhead, which increased with the number of processes. This experiment demonstrated that, while local sorting scales well with more workers, the final merge phase presents challenges in scalability due to its dependence on the number of processes and the associated synchronization requirements.

For Task 2, how about the performance for evenly and unevenly distributed data?

For evenly distributed data, the workload was well-balanced across the processes, leading to efficient local sorting and merging phases. This resulted in relatively shorter overall execution times since each process had a similar amount of work.

In contrast, the unevenly distributed dataset led to unbalanced workloads, where some processes were assigned significantly larger segments of data than others. This imbalance increased the overall runtime as some processes took longer to complete their local sorting. Moreover, the final merging phase was more challenging, as the uneven sizes of the segments led to an increased load on the merging process, further emphasizing it as a bottleneck. The uneven workload distribution also increased synchronization overhead, negatively impacting scalability and overall efficiency.

For Task 3, measure the time consumption for each of the four phases. Which phase is the bottleneck?

The results indicated that the merging phase was consistently the bottleneck. During the data distribution phase, the time taken was relatively small compared to the other phases and scaled well with the number of processes. In the local sorting phase, each process used std::sort to sort its assigned data, and this phase took a moderate amount of time, benefiting significantly from parallelism as each process worked independently. The pivot selection phase involved gathering regular samples and determining global pivots, which introduced some communication overhead but did not dominate the overall execution time.

The merging phase, however, required extensive inter-process communication and synchronization, making it the most time-consuming part of the PSRS workflow. During this phase, each process had to exchange and merge data partitions, which led to significant synchronization and communication overhead, particularly as the number of processes increased. The uneven distribution of data across processes further exacerbated this issue, leading to increased waiting times for some processes. Overall, the merging phase was the primary bottleneck, limiting the scalability and efficiency of the PSRS algorithm.

For Task 3, how about the performance for evenly and unevenly distributed data?

For evenly distributed data, the pivot selection was highly effective in creating balanced partitions, which led to efficient sorting and merging. Each process received a similar amount of data, and the merging phase was completed relatively quickly since the workload was evenly spread.

However, with an unevenly distributed dataset, some processes ended up with much larger portions of data to sort and merge, leading to substantial load imbalance. This imbalance caused increased waiting times during the merging phase, which amplified the bottleneck effect previously observed. The uneven distribution also made the pivot selection less effective, resulting in less optimal partitions and further inefficiencies during the final merging step. Consequently, the overall performance suffered due to both increased synchronization requirements and prolonged merge times.

7 Conclusion

This project explored the implementation and optimization of various parallel sorting algorithms, including Bucket Sort, Quick Sort with K-Way Merge, PSRS, and Merge Sort. Through careful application of optimizations such as load balancing, communication minimization, task parallelism, and profiling-based tuning, I was able to significantly enhance the performance of these algorithms. The PSRS algorithm emerged as the best performer, demonstrating excellent load balancing and scalability. The challenges faced, such as synchronization overhead and communication costs, provided valuable lessons on the complexities of parallel programming. Future work can focus on hybrid approaches and more sophisticated data distribution strategies to further improve performance and scalability in parallel sorting tasks.