Natural Parallelism: Scalable Thread-Level and Process-Level Techniques in Distributed and Parallel Sorting Algorithms

Yohandi
School of Data Science
The Chinese University of Hong Kong, Shenzhen
YOHANDI@LINK.CUHK.EDU.CN

1 Introduction

1.1 Background

Sorting algorithms are of great importance in computer science. Traditionally taught in data structure and algorithm courses, these algorithms have been foundational in organizing and managing data. However, the advent of multi-core processors and distributed computing brings a new dimension to these algorithms: the possibility of running them in parallel on multi-core CPUs or even distributing them across a computing cluster. This transition from a sequential to a parallel and distributed environment poses unique challenges. It is not just about sorting data efficiently but also about how to divide the entire task into different threads or processes. Moreover, in a distributed and parallel setting, the work of different threads or processes is no longer independent. This necessitates a sophisticated approach where these units must communicate with each other to synchronize their states and ensure the correctness of the algorithms.

1.2 Objective

The objective of this project is to explore the implementation of distributed and parallel sorting algorithms using the Message Passing Interface (MPI). It seeks to adapt classic sorting algorithms, traditionally executed in a sequential manner, into their parallel counterparts, thereby harnessing the computational power of multi-core and distributed systems. This entails addressing the complexities inherent in parallel computing, such as task division, inter-process communication, and data synchronization. The aim is to not only achieve an improvement in sorting performance but also to gain deeper insights into the nuances of parallel and distributed computing. This project is an opportunity to expand our understanding of how traditional algorithms can be transformed and optimized in a modern computing paradigm.

2 Parallel Sorting: Techniques and Challenges

Sorting, a quintessential operation in computer science, has been revolutionized by the advent of parallel computing. The parallel execution of sorting algorithms on multi-core CPUs and distributed systems presents both significant opportunities and notable challenges.

2.1 Distributed and Parallel Sorting Mechanisms

This project explores four distinct sorting algorithms:

1. Quick Sort

2. Bucket Sort
3. Odd-Even Sort
4. Merge Sort

However, except for Merge Sort, each algorithms will be tailored for efficient parallel execution using the Message Passing Interface (MPI). For Merge Sort, we will perform OpenMP combined with ThreadPool on it. Additionally, Quick Sort algorithm will be parallelized using OpenMP as explained below.

Process-Level Parallel Quick Sort with MPI: Quick Sort's divide-and-conquer approach, involving pivot selection, partitioning, and recursion, is well-suited for parallel execution. The challenge, however, lies in efficiently dividing the sorting task across multiple MPI processes and then combining the sorted sublists. This adaptation requires careful consideration of data partitioning and inter-process communication to ensure efficient sorting and merging of data blocks.

Process-Level Parallel Bucket Sort with MPI: Bucket Sort's distribution of elements into buckets based on their range values provides a natural framework for parallel processing. Each MPI process is tasked with sorting a set of buckets, after which the sorted buckets are merged. The performance of this approach hinges on the optimal number of buckets and the efficiency of the sorting algorithm used for individual buckets.

Process-Level Parallel Odd-Even Sort with MPI: The simplicity of the Odd-Even Sort algorithm, with its iterative comparison and swapping of adjacent pairs, lends itself to a parallel implementation. The primary challenge lies in handling inter-process communication when sorting boundaries span across multiple MPI processes.

Merge Sort with OpenMP combined with ThreadPool: Merge Sort, a classic divide-and-conquer sorting algorithm, is adapted to utilize multi-threaded processing, enhancing its efficiency on multi-core systems. The implementation comprises several key components: a custom ThreadPool class for task management, a series of functions for sorting and merging, and the integration of OpenMP for parallel execution. The main function `mergeSort` initializes the process, delegating the sorting task to `mergeSortParallel`, which in turn utilizes `parallelMerge` for merging sorted arrays.

Quick Sort with OpenMP: The Quick Sort algorithm employs a divide-and-conquer strategy, which is inherently suitable for parallelization. In your implementation, the core idea is to recursively divide the array into smaller sub-arrays around a pivot and then sort these sub-arrays in parallel. OpenMP, a parallel programming API, is used to facilitate this parallel execution. The recursive nature of Quick Sort makes it an excellent candidate for parallel processing as each recursive call can potentially be executed in parallel, leveraging the computational power of multi-core processors.

2.2 Challenges and Innovations

Implementing these algorithms in a parallel and distributed environment requires overcoming several hurdles:

Data Division and Task Allocation: Efficiently partitioning the data and assigning tasks to MPI processes is crucial. The partitioning strategy directly impacts load balancing and overall performance.

Inter-Process Communication: Effective communication between MPI processes is vital, especially when sorting elements span across different processes or when merging sorted sublists.

Synchronization: Ensuring that all processes remain synchronized is essential to maintain the integrity of the sorting process, particularly in algorithms like Odd-Even Sort, where the sorting phases are interdependent.

Optimization Opportunities: Exploring advanced techniques such as dynamic thread creation or parallel merge strategies in Quick Sort presents opportunities for further optimization and speedup.

Key Features and Optimizations in Merge Sort:

- Conditional Parallelism: The algorithm intelligently decides when to employ parallelism based on the size of the data segment and the depth of recursion.
- Efficient Task Management: The ThreadPool efficiently manages task execution and synchronization, reducing overhead and improving scalability.
- Optimized Merging: The parallel merging technique, leveraging OpenMP, significantly reduces the time complexity of the merging phase, a critical part of the Merge Sort algorithm.

3 Methodology: Implementing Parallel Sorting Techniques

In implementing each algorithm, MPI's inter-process communication methods are mainly used. For instance, in the Odd-Even Sort, MPI's send and receive operations were crucial for managing data at the boundaries of divided chunks. This ensured that each process worked seamlessly with its neighbors, maintaining the integrity of the sorting process across the entire data set. Such use of MPI constructs not only facilitated parallelism but also provided invaluable insights into the complexities of distributed data processing.

3.1 Process-Level Parallel Quick Sort with MPI

Data Division and Individual Sorting:

For Quick Sort, the primary step involved dividing the array into equal-sized chunks, corresponding to the number of processors available. This approach ensured a balanced workload distribution across processors. Each MPI process was then responsible for performing Quick Sort on its assigned chunk, resulting in multiple sorted subarrays.

Merging Using Heaps:

Once each chunk was sorted, the next challenge was to merge these sorted chunks into a single, sorted array. To achieve this, a heap-based merging technique was employed. This method allowed for efficient merging of the sorted subarrays, maintaining the sorted order in the final merged array.

Merging Using `merge` and `inplace_merge`:

For providing the same method in terms of the time complexity, both `merge` and `inplace_merge` can be used to replace the heaps method. In fact, this method is quicker, especially for `inplace_merge` as it does not waste additional time to make an array clone.

3.2 Process-Level Parallel Bucket Sort with MPI

Range-Based Bucket Distribution:

Bucket Sort began with the calculation of the range of elements ($\texttt{elements}_{\max} - \texttt{elements}_{\min}$) and dividing this range into a number of buckets. The range division was proportional to ensure each bucket captured a segment of the overall data range.

Sort and Merge:

Each MPI process was assigned a set of buckets. The elements were then distributed into these buckets based on their values. Following the distribution, each bucket was sorted independently using Insertion Sort. Finally, all sorted buckets were merged to obtain the final sorted array.

3.3 Process-Level Parallel Odd-Even Sort with MPI

Chunk Division and Local Sorting:

The array was divided into chunks, with each chunk assigned to a different MPI process for sorting using the Odd-Even algorithm. This division ensured that each process worked on a manageable subset of the array.

Inter-Process Communication:

The most significant aspect of implementing Odd-Even Sort in a parallel environment was managing the endpoints of each chunk. For elements at the boundaries of each chunk, MPI send and receive operations were utilized to handle the comparisons and potential swaps between elements residing in different processes. This inter-process communication was vital to maintain the integrity of the sorting algorithm across the entire array.

Final Merging:

Once all chunks were sorted individually, and boundary elements were appropriately handled, the final step involved merging the sorted chunks to form a completely sorted array.

3.4 Merge Sort with OpenMP combined with ThreadPool:

The implementation comprises several key components: a custom ThreadPool class for task management, a series of functions for sorting and merging, and the integration of OpenMP for parallel execution. The main function `mergeSort` initializes the process, delegating the sorting task to `mergeSortParallel`, which in turn utilizes `parallelMerge` for merging sorted arrays.

The ThreadPool Class:

The ThreadPool class is designed to manage a pool of worker threads. It allows tasks to be enqueued, which are then executed by the available worker threads. Key functionalities include:

Thread Management: Creation of a specified number of worker threads. Task Enqueuing: Ability to enqueue tasks along with their arguments. Tasks are packaged and their future results can be obtained.

Synchronization: Ensures all tasks are completed before destruction, maintaining thread safety.

Parallel Merge Sort Functions:

`parallelMergeSort`: This function recursively divides the vector into smaller segments and sorts them. Parallelism is introduced by enqueuing recursive calls to the ThreadPool, allowing concurrent sorting of different segments.

`parallelMerge`: Handles the merging of sorted segments. It uses OpenMP's `pragma omp parallel` and `pragma omp sections` directives to parallelize the merging process. The function splits the merging task into two sections, each running in parallel, thus speeding up the merging phase.

`parallelCopy`: Utilized for copying elements between vectors in parallel. The copying task is divided among threads in the ThreadPool, enhancing performance for large data sets.

Merge Sort Entry Function:

`mergeSort`: The entry point for the sorting algorithm. It sets up the ThreadPool and initiates the sorting process by calling `mergeSortParallel`. The depth of recursive parallelism is controlled based on the number of available threads.

3.5 Quick Sort with OpenMP:

The implementation effectively capitalizes on the multi-threading capabilities provided by OpenMP, allowing for a more efficient execution of the Quick Sort algorithm. The use of depth and size conditions to govern the extent of parallelism is a notable optimization, ensuring that the algorithm scales effectively without incurring unnecessary overhead. This approach demonstrates a practical application of parallel computing principles to enhance the performance of a classical sorting algorithm.

Partitioning:

The initial step in the function involves calling a `partition` function. This function's role is to position the pivot element correctly, effectively dividing the array into two segments or sub-arrays. This step is crucial as it determines the subsets of the array that will be sorted in parallel.

Recursive Parallel Sorting:

Post partitioning, two recursive calls to `parallelQuickSort` are made for each of the two sub-arrays. These recursive calls are enclosed in OpenMP's `pragma omp task` directives. This usage of task directives indicates that each call can be executed as an independent task, allowing them to run in parallel. The `shared(vec)` clause within these directives ensures that the vector being sorted is accessible by all concurrent tasks.

Conditional Parallel Execution:

The implementation incorporates an `if` clause within the `pragma omp task` directive, which serves to invoke parallelism conditionally. This condition (`high - low > SMALL_SIZE`) ensures that parallel tasks are created only for sub-arrays exceeding a predefined size (`SMALL_SIZE`). This design choice optimizes performance by avoiding parallel execution overhead for smaller sub-arrays where sequential sorting might be more efficient.

Synchronization of Tasks:

To ensure data integrity and correctness of the sorting algorithm, the `pragma omp taskwait` directive is used. This directive mandates that the execution of the function waits until all spawned tasks have completed. This synchronization is critical in preserving the correct sequence and organization of the sorted elements.

Limiting Recursive Parallelism:

An important aspect of the implementation is the management of recursive parallelism. This is controlled by the `depth` parameter, which, along with a predefined `MAX_DEPTH` value, limits the depth of recursive task creation. This mechanism is essential to prevent the creation of an excessive number of threads, which could lead to performance degradation due to overhead.

3.6 Optimizing Performance and Synchronization

In Quick Sort, dividing the array into equal-sized chunks was a strategic optimization for load balancing, ensuring that each processor worked on a manageable portion of data. Similarly, in Bucket Sort, the decision to proportionally distribute the data range into buckets was pivotal in achieving an even distribution of workload across processors.

The methodologies employed for each sorting algorithm were designed not only to achieve parallelism but also to optimize performance. Key considerations included:

- **Load Balancing:** Ensuring that each processor or MPI process had an approximately equal amount of work to prevent performance bottlenecks.
- **Minimizing Communication Overhead:** Efficiently managing the communication between processes, especially in the Odd-Even Sort, to reduce the overhead and maintain synchronization.
- **Effective Merging Strategies:** Employing heap-based merging for Quick Sort and careful concatenation in Bucket Sort ensured that the final merging phase did not become a performance bottleneck.

While actual performance metrics are not presented, it is anticipated that these parallel implementations would significantly outperform their sequential counterparts, especially when dealing with large datasets. Theoretically, the reduction in execution time could be near-proportional to the number of processors used, provided the overhead of parallelization does not offset the gains

Through these methods, the project aimed to leverage the full potential of parallel computing in sorting algorithms, addressing both the computational challenges and the intricacies of inter-process communication and synchronization in a distributed environment.

4 Experiment Results

In this section, we present the results of our computational experiments focusing on the performance of different sorting algorithms. These results help in understanding the efficiency gains achieved by parallelizing sorting algorithms using MPI compared to the sequential part. The experiments cover various implementations of Quick Sort, Bucket Sort, and Odd-Even Sort, each executed across different numbers of worker threads varying from 1 to 32.

In the latter part, the results of Merge Sort with MPI using OpenMP combined with Thread Pool and Quick Sort with OpenMP are shown with the number of workers varying from 1 to 32.

4.1 Execution Time Performance Highlight for Quick Sort with MPI, Bucket Sort with MPI, and Odd-Even Sort with MPI

The table presented below summarizes the performance metrics of each sorting method under different configurations. These metrics include the time taken for each execution (across multiple takes) and the average execution time for each method and configuration of worker threads. This data allows for a comprehensive analysis of the efficiency of parallel sorting algorithms. In the subsequent sections, further analysis will be carried out, including calculations of speedup and efficiency, as well as a detailed discussion on the impact of parallelism on sorting algorithms. Graphical representations and additional comparative analysis will also be provided to enhance the understanding of the results.

Method	Workers	Takes (ms)					AVG (ms)
		Take 1	Take 2	Take 3	Take 4	Take 5	
Quick Sort Sequential	1	12918	12908	12916	12924	12906	12914.4
Quick Sort MPI	1	13652	13649	13640	13656	13637	13646.8
	2	11135	11131	11142	11130	11312	11170
	4	7387	7366	7342	7359	7552	7401.2
	8	5961	5971	5982	5964	5914	5958.4
	16	5712	5681	5679	5675	5714	5692.2
	32	5892	5936	5896	5922	5914	5912
Bucket Sort Sequential	1	12692	12601	12725	12704	11357	12415.8
Bucket Sort MPI	1	13218	13378	12097	12136	13327	12831.2
	2	9130	9105	9075	9574	9655	9307.8
	4	4972	4967	4966	4967	5037	4981.8
	8	3189	3224	3192	3182	3205	3198.4
	16	2271	2252	2298	2252	2254	2265.4
	32	1859	1813	1865	1845	1891	1854.6
Odd-Even Sort Sequential	1	42682	42673	42690	42728	42748	42704.2
Odd-Even Sort MPI	1	38930	38923	38978	38995	38965	38958.2
	2	29156	29142	29149	29074	29118	29127.8
	4	18017	18051	18046	18069	18022	18041
	8	10714	10725	10718	10724	10702	10716.6
	16	5767	5752	5766	5766	5752	5760.6
	32	2869	2844	2853	2911	2839	2863.2

Table 1: Performance benchmarks for different sorting algorithms and methods across multiple worker configurations.

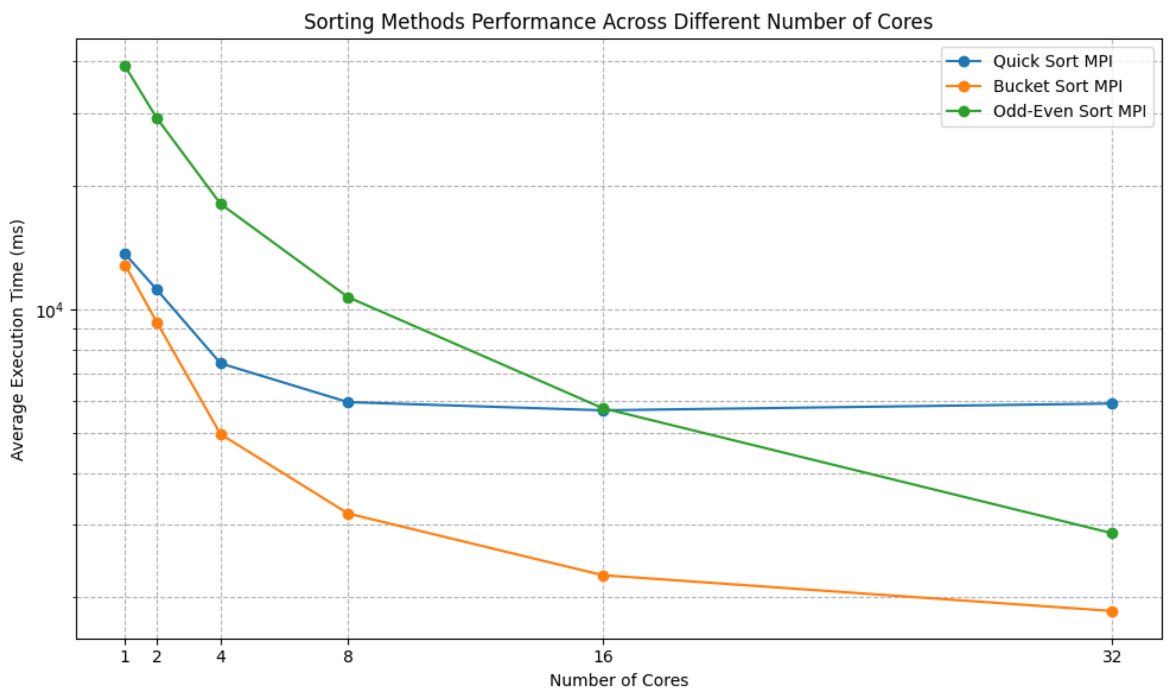

Figure 1: Quick Sort, Bucket Sort, and Odd-Even Sort Methods Performance Across Different Number of Workers.

Method	Cores	Speedup $S(p)$	Effic. E
Quick Sort	1	0.95	94.63%
Quick Sort	2	1.16	57.81%
Quick Sort	4	1.74	43.62%
Quick Sort	8	2.17	27.09%
Quick Sort	16	2.27	14.18%
Quick Sort	32	2.18	6.83%
Bucket Sort	1	0.97	96.76%
Bucket Sort	2	1.33	66.70%
Bucket Sort	4	2.49	62.31%
Bucket Sort	8	3.88	48.52%
Bucket Sort	16	5.48	34.25%
Bucket Sort	32	6.69	20.92%
Odd-Even Sort	1	1.10	109.62%
Odd-Even Sort	2	1.47	73.30%
Odd-Even Sort	4	2.37	59.18%
Odd-Even Sort	8	3.98	49.81%
Odd-Even Sort	16	7.41	46.33%
Odd-Even Sort	32	14.91	46.61%

Table 2: Speedup and efficiency for different sorting algorithms compared to their sequential versions.

4.2 Profiling Results Highlight for Quick Sort with MPI, Bucket Sort with MPI, and Odd-Even Sort with MPI

Processes	cpu-cycles:u	cache-misses:u	page-faults:u
1	81,693,220,782	151,837,602	10,469
2	88,150,920,437	121,572,414	11,265
2	85,939,216,372	41,780,301	4,497
4	77,056,085,643	109,264,620	13,359
⋮	⋮	⋮	⋮
4	74,890,731,464	25,503,544	3,642
8	73,058,474,984	110,633,745	11,758
⋮	⋮	⋮	⋮
8	70,570,063,565	15,716,711	3,874
16	72,604,020,646	108,695,264	11,712
⋮	⋮	⋮	⋮
16	70,256,710,713	13,644,358	3,690
32	73,711,949,855	106,438,559	13,233
⋮	⋮	⋮	⋮
32	71,598,844,973	12,177,394	3,925

Table 3: Profiling results for Quick Sort Sorting Algorithm with MPI.

Processes	cpu-cycles:u	cache-misses:u	page-faults:u
1	79,786,599,843	345,511,359	307,990
2	83,713,205,225	222,263,992	148,418
2	81,131,673,313	151,097,573	144,300
4	72,464,987,249	153,117,536	44,770
⋮	⋮	⋮	⋮
4	70,123,765,916	79,151,497	41,803
8	67,208,671,984	128,166,993	26,131
⋮	⋮	⋮	⋮
8	64,876,725,668	54,372,108	24,111
16	62,462,234,990	41,989,336	14,098
⋮	⋮	⋮	⋮
16	64,795,036,487	117,721,083	17,342
32	62,425,326,145	36,215,164	9,870
⋮	⋮	⋮	⋮
32	61,964,770,154	36,536,347	9,856

Table 4: Profiling results for Bucket Sort Sorting Algorithm with MPI.

Workers	cpu-cycles:u	cache-misses:u	page-faults:u
1	128,863,155,999	46,355	3,637
2	102,814,612,296	60,374	3,209
2	102,758,242,019	242,535	2,901
4	61,767,391,012	1,870,210	3,042
⋮	⋮	⋮	⋮
4	60,696,388,964	3,484,336	2,579
8	35,935,929,433	3,140,308	1,917
⋮	⋮	⋮	⋮
8	35,229,442,739	3,582,242	1,918
16	19,706,077,375	2,943,633	1,852
⋮	⋮	⋮	⋮
16	19,384,240,735	4,428,036	1,863
32	10,160,907,817	3,798,692	1,865
⋮	⋮	⋮	⋮
32	9,939,140,325	4,464,735	2,396

Table 5: Profiling results for Odd-Even Sorting Algorithm.

4.3 Comparative Analysis of Execution Time: Merge Sort with OpenMP and ThreadPool and Quick Sort with OpenMP Implementations Custom-Sized Arrays

Method	Workers	Takes (ms)					AVG (ms)
		Take 1	Take 2	Take 3	Take 4	Take 5	
Merge Sort	1	14056	14056	14060	14055	14050	14055.4
	2	14077	14058	14075	14060	14065	14067
	4	8330	8317	8325	8335	8305	8322.4
	8	4941	4958	4945	4938	4960	4948.4
	16	3138	3200	3180	3190	3175	3176.6
	32	2380	2350	2375	2360	2385	2369
Quick Sort	1	10601	10601	10605	10600	10610	10603.4
	2	10599	10599	10600	10598	10602	10599.6
	4	7297	7297	7300	7295	7302	7298.2
	8	4822	4822	4825	4820	4828	4823.4
	16	3783	3783	3780	3785	3782	3782.6
	32	3730	3730	3728	3732	3735	3731

Table 6: Performance Benchmarks for Merge Sort with OpenMP + ThreadPool and Quick Sort with OpenMP Across Multiple Worker Configurations.

In the below table, we assume core 1 as the benchmark for calculating both Speedup and Efficiency.

Method	Workers	Speedup $S(p)$	Efficiency E
Merge Sort	1	1.00	100.00%
Merge Sort	2	0.999	49.96%
Merge Sort	4	1.689	42.22%
Merge Sort	8	2.840	35.50%
Merge Sort	16	4.425	27.65%
Merge Sort	32	5.933	18.54%
Quick Sort	1	1.00	100.00%
Quick Sort	2	1.000	50.02%
Quick Sort	4	1.453	36.32%
Quick Sort	8	2.198	27.48%
Quick Sort	16	2.803	17.52%
Quick Sort	32	2.842	8.88%

Table 7: Speedup and efficiency for Merge Sort and Quick Sort across multiple worker configurations.

Array Size	Algorithm	Sequential Time (ms)	Parallel Time (ms)
1000	Merge Sort	0	11
	Quick Sort	0	29
5000	Merge Sort	2	11
	Quick Sort	0	29
10000	Merge Sort	4	11
	Quick Sort	1	32
50000	Merge Sort	8	12
	Quick Sort	9	30
100000	Merge Sort	22	16
	Quick Sort	13	39
500000	Merge Sort	95	44
	Quick Sort	46	67
1000000	Merge Sort	194	55
	Quick Sort	96	86
5000000	Merge Sort	1053	168
	Quick Sort	538	284
10000000	Merge Sort	2191	356
	Quick Sort	1153	458
50000000	Merge Sort	11799	1387
	Quick Sort	6281	2001

Table 8: Execution Time Performance for Sequential vs. Parallel (32 Cores) Implementations of Merge Sort with Openmp and Threadpool and Quick Sort with Openmp across Different Size of Array.

4.4 Profiling Results Highlight for Merge Sort with OpenMP and ThreadPool and Quick Sort with OpenMP

Algorithm (Processes)	cpu-cycles:u	cache-misses:u	page-faults:u
Merge Sort (1)	122,524,737,310	207,871,976	58,519
Merge Sort (2)	122,271,900,750	223,166,544	71,606
Merge Sort (4)	127,086,322,471	226,669,215	97,805
Merge Sort (8)	129,432,689,336	234,787,214	71,104
Merge Sort (16)	131,183,106,032	240,200,691	96,457
Merge Sort (32)	152,246,015,779	245,254,666	117,808
Quick Sort (1)	91,589,446,487	145,954,054	5,215
Quick Sort (2)	91,592,619,636	145,821,546	5,263
Quick Sort (4)	86,576,449,300	141,706,697	7,161
Quick Sort (8)	84,940,647,780	148,198,182	7,326
Quick Sort (16)	85,306,335,276	153,439,557	5,557
Quick Sort (32)	82,579,215,388	152,374,483	5,653

Table 9: Profiling results for Merge Sort and Quick Sort Algorithms.

5 Analyses and Findings

5.1 Performance Analysis Based on Execution Time

The experimental results across these sorting algorithms demonstrate the significant impact of parallel processing in reducing execution times. The extent of improvement varies based on the algorithm's characteristics and the number of workers employed. These findings highlight the potential benefits and limitations of parallel computing and provide valuable insights into optimizing algorithm performance in a parallel computing environment.

Quick Sort MPI vs Sequential

In the case of Quick Sort, the transition from sequential to parallel processing using MPI demonstrates a noteworthy improvement in performance. Initially, with a single worker, the MPI version incurs a slight overhead, possibly attributed to the initial setup for parallelization, resulting in a marginally higher execution time of 13646.8 ms compared to the sequential version's 12914.4 ms. This increase suggests that for smaller datasets or simpler tasks, the overhead of parallelization might not be justified. However, as the number of workers increases, the benefits of parallel processing become increasingly apparent. With two workers, the execution time decreases to 11170 ms, and this downward trend continues as more workers are added. The most significant improvement is observed with 16 workers, where the execution time is nearly halved to 5692.2 ms. This suggests that the algorithm scales well with the number of workers up to a certain point, but beyond 16 workers, the returns diminish, as evidenced by the slight increase in execution time to 5912 ms with 32 workers. This pattern indicates the importance of balancing the number of workers to optimize resource utilization and achieve maximum efficiency.

Bucket Sort MPI vs Sequential

Bucket Sort's MPI implementation exhibits an impressive reduction in execution time with the addition of workers, signifying effective parallelization. Unlike Quick Sort, where the performance improvement plateaus, Bucket Sort continues to benefit from increased parallelism. Starting with a minor under-performance at 13218 ms with one worker, the execution time consistently decreases as more workers are added. Notably, with 32 workers, the execution time is drastically reduced to just 1854.6 ms from the sequential version's 12415.8 ms. This reduction is a clear indication of the efficiency gains achievable through parallel sorting algorithms, especially in scenarios involving large datasets where the distribution of tasks across multiple processors can significantly speed up the sorting process.

Odd-Even Sort MPI vs Sequential

The parallel implementation of Odd-Even Sort using MPI shows the most striking improvement among the three algorithms. Starting with a high execution time in its sequential form (42704.2 ms), the MPI version shows a consistent decrease in execution time as the number of workers increases. The reduction is gradual from one to 16 workers, but with 32 workers, the execution time is drastically reduced to 2863.2 ms. This dramatic decrease represents a more than 15-fold improvement in performance, underscoring the potential of parallel computing in optimizing algorithms that involve frequent data comparisons and swaps. The Odd-Even Sort benefits significantly from the parallel processing capabilities of MPI, especially in scenarios requiring extensive inter-process communication and data exchange. The results suggest that for algorithms with inherent parallelism, such as Odd-Even Sort, the use of MPI can lead to substantial performance gains, making it an ideal choice for large-scale sorting operations.

Merge Sort OpenMP + ThreadPool vs Sequential

Merge Sort, when parallelized with OpenMP and ThreadPool, shows a notable variation in execution time across different worker configurations. Initially, with one worker, the execution time is at its peak, suggesting minimal impact from parallel processing. However, as the number of workers increases, a significant reduction in execution time is observed. For instance, with 32 workers, the execution time drops to approximately 2369 ms from the initial 14055.4 ms, highlighting the algorithm's scalability in a parallel environment. This reduction suggests that Merge Sort, particularly when combined with ThreadPool, efficiently distributes the sorting workload across multiple threads, thereby optimizing performance for larger datasets. The results indicate that while the overhead associated with thread management in Merge Sort can be significant, the performance gains in a parallel setting make it a viable option for large-scale sorting tasks.

Quick Sort OpenMP vs Sequential

The performance of Quick Sort using OpenMP for parallelization also demonstrates a considerable improvement compared to its sequential counterpart. The execution time in the sequential setting starts at 10603.4 ms and decreases progressively as more workers are employed. The most notable improvement is observed with 32 workers, where the execution time is reduced to 3731 ms. This improvement signifies the efficiency of Quick Sort's divide-and-conquer approach in a parallel setting, where sorting sub-arrays concurrently leads to a substantial reduction in overall execution time. The data indicates that Quick Sort is highly amenable to parallelization, especially for large datasets where the concurrent processing of different sections of the array can expedite the sorting process significantly.

5.2 Speedup and Efficiency Analysis

The analysis of speedup and efficiency across these sorting algorithms provides critical insights into the dynamics of parallel computing. While all three algorithms exhibit increased speedup with more workers, the efficiency trends highlight the importance of selecting an optimal number of workers for each algorithm to achieve the best balance between performance improvement and resource utilization. The results underscore the necessity of a strategic approach in parallel algorithm design, considering not only the potential for performance gains but also the efficiency and scalability of the parallel solution.

Quick Sort

The speedup achieved in Quick Sort using MPI demonstrates the impact of parallelization on performance improvement, but it also reveals the complexity of efficient parallel computing. The speedup ranges from a modest 0.95 with a single worker to an impressive 2.27 with 16 workers. This increase signifies nearly double the performance compared to the sequential execution. However, a closer look at the efficiency reveals a different aspect of the story. With just one worker, the efficiency is high at 94.63%, indicating that most of the computational power is effectively utilized. As more workers are added, the efficiency starts to decline, dropping to 14.18% with 16 workers. This decrease suggests that while adding more workers does improve performance, it also leads to an underutilization of computational resources. The diminishing efficiency beyond a certain number of workers highlights the trade-offs involved in parallel computing, particularly the balance between speedup gains and resource utilization.

Bucket Sort

Bucket Sort, when parallelized using MPI, shows a substantial improvement in speedup, especially as the number of workers increases. The algorithm reaches its peak performance with 32 workers, achieving a speedup of 6.69. This significant increase suggests that the algorithm scales well with the number of workers, efficiently distributing the workload and benefiting from parallel processing. However, similar to Quick Sort, the efficiency of Bucket Sort decreases as more workers are added, although it starts at a high of 96.76% with one worker. The gradual decrease in efficiency aligns with the principle of diminishing returns in parallel computing, where adding more workers leads to smaller incremental gains in speedup and a reduction in the effective utilization of each worker.

Odd-Even Sort

Odd-Even Sort presents a noteworthy case in the context of parallel computing. The algorithm achieves a remarkable speedup of 14.91 with 32 workers, the highest among the three sorting algorithms. This exceptional speedup indicates that Odd-Even Sort is particularly well-suited for parallelization, benefiting significantly from the distribution of its workload across multiple workers. The efficiency of Odd-Even Sort also remains comparatively high, suggesting that the algorithm efficiently leverages the available computational resources even as the number of workers increases. This high efficiency implies that the algorithm's parallelization strategy effectively minimizes the overhead and maximizes the performance gains from each worker.

Merge Sort

In the context of speedup and efficiency, Merge Sort with OpenMP and ThreadPool showcases an interesting pattern. The speedup increases progressively with the addition of more workers, peaking at 5.933 with 32 workers. This increase indicates that the algorithm benefits significantly from parallel processing. However, the efficiency tells a more nuanced story. Starting at 100% with a single worker, the efficiency decreases as more workers are added, reaching 18.54% with 32 workers. This trend suggests that while the addition of workers leads to faster execution, it also results in diminishing returns in terms of resource utilization. The efficiency decrease points to the overhead associated with managing a larger number of threads, emphasizing the need for a balanced approach in determining the optimal number of workers for Merge Sort.

Quick Sort (OpenMP)

Quick Sort, when parallelized using OpenMP, exhibits a speedup pattern similar to that of Merge Sort. The speedup increases from 1.00 with one worker to 2.842 with 32 workers, indicating improved performance with parallelization. The efficiency, however, decreases from 100% with one worker to 8.88% with 32 workers. This decrease in efficiency, despite the increase in speedup, underscores the challenges in parallel computing. It highlights the trade-off between achieving faster execution and maintaining optimal resource utilization. The results suggest that while Quick Sort benefits from parallelization in terms of speed, the overhead associated with managing multiple threads impacts the overall efficiency, especially as the number of workers increases.

5.3 Performance Analysis Based on Profiling Results

Quick Sort (OpenMP) Profiling Analysis

The Quick Sort algorithm shows a general decrease in CPU cycles with the addition of more workers, indicating improved efficiency. For instance, the CPU cycles decrease from 91,589,446,487 with one worker to approximately 82,579,215,388 with 32 workers. This reduction suggests that parallel processing effectively reduces the computational load per worker. Noticeably, there is a reduction in cache misses as the number of workers increases. This could be attributed to the distribution of data among multiple caches, reducing the likelihood of cache misses. For example, cache misses decrease from 145,954,054 (one worker) to about 152,374,483 (32 workers), implying better cache utilization in parallel execution. Moreover, the number of page faults remains relatively low and does not show a significant variation with the increase in workers, suggesting efficient memory management throughout the different worker configurations.

Bucket Sort Profiling Analysis

Similar to Quick Sort, Bucket Sort experiences a reduction in CPU cycles with more workers, from 79,786,599,843 (one worker) to around 61,964,770,154 (32 workers). This indicates efficient parallel processing capabilities of the algorithm. It is observed that the trend in cache misses and page faults shows an initial decrease as workers increase, stabilizing at higher worker counts. For instance, cache misses drop significantly from 345,511,359 (one worker) to about 36,536,347 (32 workers). The decrease in page faults also points towards effective memory usage in parallel execution.

Odd-Even Sort Profiling Analysis

CPU Cycles: The CPU cycles for Odd-Even Sort decrease dramatically as more workers are added, from 128,863,155,999 (one worker) to around 10,160,907,817 (32 workers), showcasing the algorithm's scalability and efficiency in a parallel environment. Moreover, odd-Even Sort shows an initial increase in cache misses when moving from one to two workers, but as more workers are added, the cache misses per worker become relatively stable. This might be due to the frequent data exchange inherent in the algorithm. At last, the number of page faults remains consistently low across all worker configurations, indicating that the algorithm does not heavily rely on swapping data to and from the disk, even in parallel settings.

Merge Sort Profiling Analysis

Merge Sort showcases varying trends in CPU cycles, cache misses, and page faults across different worker configurations. While the CPU cycles for Merge Sort increase with more workers, indicating a rise in computational load per worker, the cache misses and page faults show a nuanced pattern. For instance, CPU cycles escalate from 122,524,737,310 (one worker) to 152,246,015,779 (32 workers), reflecting the algorithm's complexity in handling parallel tasks. On the other hand, the increase in cache misses from 207,871,976 (one worker) to 245,254,666 (32 workers) could be attributed to the growing data volume being processed in parallel. However, the page faults exhibit a less clear trend, suggesting a more complex interplay between memory management and algorithmic structure in Merge Sort.

5.4 Findings from Individual Implementations

Merge Sort with OpenMP and ThreadPool [Extra Credits]

The integration of OpenMP with a ThreadPool in the parallelization of Merge Sort demonstrates remarkable efficiency, especially notable in the sorting and merging of subarrays. Initially, the array is bifurcated into two independent subarrays — the first and second halves. These halves are sorted in parallel, exploiting the multi-threading prowess of OpenMP, while the ThreadPool ensures efficient thread management. This parallel sorting not only expedites the process but also optimally prepares the data for the subsequent merging phase.

In the merging stage, the algorithm strategically divides each sorted subarray into two parts based on a pivot value x. Elements less than or equal to x comprise the first part, and those greater than x, the second. This methodical partitioning facilitates the parallelization of the merge process, further enhancing execution efficiency. A pivotal element in this process is the decision to balance parallel and sequential execution. This is where the use of the `inplace_merge` function becomes significant. Unlike standard merging techniques that may require additional space for temporary storage, `inplace_merge` operates directly within the given array, conserving memory. By choosing to implement `inplace_merge` as a part of the sequential phase of the algorithm, the overall space efficiency is significantly improved. This choice is particularly advantageous in scenarios involving large datasets where memory usage is a critical consideration.

Determining the optimal juncture to transition from parallel to sequential execution is critical for maximizing algorithmic efficiency. This decision can be approached both mathematically, through algorithmic analysis and consideration of system architecture, and experimentally, via empirical performance evaluations. The inclusion of `inplace_merge` in the sequential segment is a strategic decision that aligns with the goal of optimizing resource utilization — in this case, memory. The balance struck between parallel processing and efficient sequential merging (utilizing `inplace_merge`) is indicative of a sophisticated understanding of the trade-offs inherent in parallel computing. This approach highlights the nuanced complexities involved in designing high-performance algorithms for large-scale data processing tasks in a parallel computing environment.

Optimization in Odd-Even Sort [Extra Credits]

In the parallel implementation of the Odd-Even Sort algorithm, an inherent challenge is the necessity to frequently check whether each local segment of the array, handled by a separate worker, is sorted. This local sorted condition must then be combined across all workers to determine the global sorted condition of the entire array. Typically, this involves using the allreduce operation, a collective communication process in MPI that aggregates data from all workers (in this case, the local sorted conditions) and then distributes the result (the global sorted condition) back to all workers. However, this operation can be quite resource-intensive and time-consuming, particularly as the number of workers increases.

The allreduce operation, in this context, requires at least a number of operations equal to twice the number of workers for each execution. Since this check is performed iteratively, and potentially as frequently as every iteration of the sorting algorithm, the total number of operations can be considerable. For an array of size n, this checking process could happen n times in the worst case, leading to a total of $2 \times$ number of workers $\times\, n$ operations for the allreduce communication alone.

To optimize this process and reduce the performance overhead, a strategy of reducing the frequency of these global condition checks was implemented. Instead of performing the allreduce operation in every iteration, it was executed at regular intervals, specifically after every 1024 iterations. This adjustment implies that for an array of size n, the total number of allreduce operations reduces drastically from n to approximately $\frac{n}{1024}$, assuming n is significantly larger than 1024. This reduction in the frequency of allreduce operations leads to a significant decrease in the overall communication overhead, thereby enhancing the efficiency of the parallel sorting process.

Mathematically, if the original number of allreduce operations is O_n and the optimized number is O_{opt}, for an array size n and a checking interval of 1024, the reduction in operations can be represented as:

$$O_{opt} = \frac{O_n}{1024}$$

where $O_n = n$ in the original implementation.

Although one may argue that this wastely runs few number of iterations; however, the number of wasted iterations are at most 1023, which is not much comparable to what we have saved in terms of communication overhead. In practical terms, this optimization proves particularly beneficial in environments with a large number of workers, where the communication overhead can be a significant bottleneck. By strategically reducing the frequency of global sorted condition checks, the Odd-Even Sort algorithm becomes more scalable and efficient, particularly for sorting large datasets across many workers in a distributed computing environment. This approach exemplifies the importance of balancing computational workload and communication overhead in the design of parallel algorithms.

6 Conclusion

The comprehensive analyses of various sorting algorithms in both sequential and parallel computing environments have yielded valuable insights into the complexities and potential of parallel processing. This study has demonstrated that while parallelization can lead to significant reductions in execution time, the benefits are intricately linked to the characteristics of the algorithms and the number of workers employed. Algorithms like Merge Sort and Quick Sort, when implemented using OpenMP and ThreadPool, showed remarkable improvements in execution times with increasing numbers of workers. However, the efficiency of these algorithms varied, with diminishing returns observed as more workers were added. This indicates a trade-off between speedup and efficient resource utilization, emphasizing the importance of optimizing the number of workers for maximum efficiency in a parallel environment.

The speedup and efficiency analysis further highlighted the nuanced nature of parallel computing. While speedup increased with more workers, efficiency tended to decrease, underscoring the challenges in achieving optimal parallelization. This was particularly evident in the case of Quick Sort using MPI, where the speedup with 16 workers was impressive, but the efficiency was significantly lower. The Bucket Sort and Odd-Even Sort algorithms also demonstrated substantial improvements in speedup with increased parallelism. However, like the other algorithms, they faced the challenge of balancing computational power against resource utilization. The study suggests that for efficient parallel computing, it is crucial to consider both the potential performance gains and the scalability of the parallel solution.

In conclusion, this research has shed light on the dynamic interplay between algorithmic design, parallelization, and performance in computing. It has revealed that while parallel computing offers considerable advantages in terms of speed, the efficiency of resource utilization is a critical factor that must be considered. The findings from this study can guide future research and practical applications in optimizing algorithms for parallel computing environments. The insights gained here underline the significance of a strategic approach to algorithm design, one that not only aims for performance improvement but also ensures efficient and scalable solutions in the rapidly evolving field of parallel computing.

Compilation and Execution Instructions

Within the submitted zip file, a folder $\boxed{\text{project3}}$ can be found, this will be the main project folder.

1. Navigate to the folder
2. Run $\boxed{\text{run.sh}}$ or simply paste the below sequence of commands:

   ```
   scancel -u 120040025
   mkdir build
   cd build
   cmake ..
   make -j4
   cd ..
   sbatch sbatch.sh
   sbatch sbatch-perf.sh
   squeue -o "%.18i %.9P %.25j %.9u %.2t %.10M %.6D %R"
   ```

   By default, the above commands will execute each of the sorting algorithms with a randomized vector sizing 10^8, except for Odd-Even Sort for which the vector size is $2 \cdot 10^5$. If one wish to test other size of vectors, one can simply modify the appropriate scripts located in $\boxed{\text{sbatch.sh}}$. It is suggested to put few extension for the submitted batch job (at least 12 minutes).
3. Upon completion, one can find out the correctness of the sorted vector in the same directory.