Scientific Research Publishing

Journal of Computer and Communications, 2024, 12, 131-138
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Accelerating Large-Scale Sorting through Parallel Algorithms

Yahya Alhabboub, Fares Almutairi, Mohammed Safhi, Yazan Alqahtani, Adam Almeedani, Yasir Alguwaifli

College of Computer Science and Information Technology, Imam Abdulrahman Bin Faisal University, Dammam, Saudi Arabia
Email: 2200003176@iau.edu.sa

How to cite this paper: Alhabboub, Y., Almutairi, F., Safhi, M., Alqahtani, Y., Almeedani, A. and Alguwaifli, Y. (2024) Accelerating Large-Scale Sorting through Parallel Algorithms. *Journal of Computer and Communications*, **12**, 131-138.
https://doi.org/10.4236/jcc.2024.121009

Received: December 13, 2023
Accepted: January 27, 2024
Published: January 30, 2024

Abstract

This study explores the application of parallel algorithms to enhance large-scale sorting, focusing on the QuickSort method. Implemented in both sequential and parallel forms, the paper provides a detailed comparison of their performance. This study investigates the efficacy of both techniques through the lens of array generation and pivot selection to manage datasets of varying sizes. This study meticulously documents the performance metrics, recording 16,499.2 milliseconds for the serial implementation and 16,339 milliseconds for the parallel implementation when sorting an array by using C++ chrono library. These results suggest that while the performance gains of the parallel approach over its serial counterpart are not immediately pronounced for smaller datasets, the benefits are expected to be more substantial as the dataset size increases.

Keywords

Sorting Algorithm, Quick Sort, QuickSort Parallel, Parallel Algorithms

1. Introduction

Sorting algorithms are an indispensable component in the landscape of modern computation. As noted by Baqer [1], the problem-solving capacity and time efficiency afforded by sorting techniques constitute tangible proof of their significant reliability and utility. Amid the class of available sorting algorithms, Quicksort stands out as one of the most ubiquitous sequential approaches owing to its combination of conceptual simplicity, low processing overhead, and excellent average-case complexity of O(nlogn) [2]. The kernel of Quicksort involves selecting a pivot element and dividing the dataset into two partitions with values less than the pivot and values greater than the pivot. This divide-and-conquer strategy lends itself naturally to parallelization. In parallel Quicksort implementations, partitioning and sorting of divisions can occur simultaneously across multiple processing units. Research has been undertaken into task scheduling methodologies to best distribute workloads among processors in parallel Quicksort [3].

However, despite its widespread use and extensive research, the specific performance implications of parallel Quicksort in large-scale data environments remain underexplored. This paper aims to fill this gap by examining the performance differential between sequential and parallel Quicksort implementations under varying data conditions. Employing a comparative analytical approach, we use empirical data from tests conducted on arrays of varying sizes. This study not only contributes to a deeper understanding of Quicksort's efficiency in multi-core processing environments but also aids in optimizing sorting processes in big data analytics.

The quest for accelerated sorting has led to GPU-optimized variants of Quicksort as well. Sintorn and Assarsson [4] developed a GPU-Quicksort utilizing fragmented sorting and merging that leverages the massively parallel architecture of graphics cards, yielding speedups of up to 6 times over single-threaded Quicksort. Their bucket sorting method for partitioning exhibits O(n) complexity in the average case. Tsigas and Zhang [5] implemented a lock-free parallel Quicksort on the Sun Enterprise 10,000, exploiting principles of efficient load-balancing across processors. Testing showed sorting time speedups of up to 7 times faster than alternatives like sample sort.

The Quicksort algorithm in the Intel Threading Building Blocks (TBB) library stands as a sophisticated exemplar of concurrent Quicksort execution in a shared memory space [6]. The TBB interface encapsulates the intricate details of parallelism from the developer while allowing platform-specific optimization. This enables software engineers to efficiently tap into the parallel capabilities afforded by multi-core and multi-processor commodity hardware that has become ubiquitous. Indeed, as Reinders [6] highlights, although concurrent programming has presented deep theoretical challenges, relentless advances in silicon manufacturing rendered parallelism a mainstream necessity. Practical libraries like TBB lower the barrier for software developers to realize the performance benefits of concurrency with minimal added complexity.

2. Literary Review

In the study by Philippas Tsigas *et al.* [5], they used SUN ENTERPRISE 10,000 data center to test and compare quick and sample sort. They used multiple methods to develop the model such as fine-grain parallelism to divide large tasks into smaller ones, and all the subtasks can run simultaneously. The Sedgewick pivot selection approach was implemented for data and computation sharing without causing blocking effects. A cache-coherent shared address space multiprocessor with 32 SUN ENTERPRISE 10,000 processors achieved a greater speed-up with parallel quick sort compared to sample sort. The quick sort was found to be over six times quicker than the sample sort. This occurred by leveraging the capabilities of the multiprocessor system.

The method proposed by Philip Heidelberger *et al.* [3], for parallelization of the Quicksort algorithm for shared memory multiprocessor. Multiprocessor shared memory underwent the use of fetch-and-add operation during the Quicksort algorithm. Add an atomic operation that adds to the current value of a memory location after reading it. Adaptive scheduling algorithms were employed to allocate a designated value to the memory location. By utilizing the technique of divide-and-conquer, it is possible to reduce the period of processor wait time as well as synchronization. Effective load balancing and improved efficiency were achieved due to overhead reduction, resulting in an optimized system. Algorithm Quicksort boasts an impressive 80% increase in speed.

In the study made by Tinku Singh *et al.* [7], they used a parallel quicksort algorithm to measure CPU core utilization. Quicksort is used to test the workload on different cores of the CPU. They test quicksort in serial and parallel with variable size input to measure the performance of CPU usages, then store all the results in tables. Comparison graphs are used to show the utilization of the CPU. After reviewing the comparison graphs, they concluded that Quicksort's parallel version better utilizes CPU cores compared to the sequential version. Not only the multicore is responsible for this result, but they also developed an effective code that uses multithreading.

In the study made by Erik Sintorn *et al.* [4], they presented a technique that uses recent GPUs to rapidly sort huge lists while fully utilizing the parallelism of the GPU. They used GPU-based bucket sort or quick sort split lists into sub lists then they are sorted in parallel using merge sort. This GPU-based sorting algorithm performs faster than radix sort and other GPU-based sorting algorithms and the algorithm is 6 times faster than single CPU quicksort. The algorithm has the complexity of n log n. They did another test by utilizing two graphics cards for additional speed-up and achieved a 1.8 times speedup when utilizing the two graphics cards.

In the study by Marszalek *et al.* [8], they developed a flexible merge sorting algorithm designed for parallel processing on multicore architectures, the method is based on a modified merge sorting algorithm. The suggested approach has been implemented into effect utilizing an Opteron AMD Processor 8356 8p for the research, running C MS Visual 2015 on MS Windows Server 2012. The tasks are flexibly distributed between logical cores to increase efficiency and each processor works separately without cross-actions or interruptions. The proposed method was tested and compared to other methods, showing high efficiency. With each newly added processor, sorting becomes faster and more efficient.

After reviewing the previous literature studies, we found out that parallelism in computing not only depends on a powerful machine but also needs effective code to utilize the CPU or GPU to its maximum potential [7]. Since we do not have machines that specialize in data parallelism like in the studies [5] [8]. We will use our devices with an effective code to minimize processor waiting time and synchronization overhead [3] and we will use large datasets to compare the sequential and the parallel version of the code [4] [7]. We will use the newest technology we have with an effective code to achieve the highest speed-up percentage possible.

3. Methodology

The objective is to accelerate large-scale sorting by leveraging parallel processing. We implement both sequential and parallelized versions of the QuickSort algorithm [2] in C++ to sort randomly generated integer arrays.

3.1. Framework

The framework of the code is illustrated in **Figure 1** below.

3.2. The Key Techniques Used

3.2.1. Array Generation

An integer array of configurable size N is generated by uniformly sampling random numbers between specified minimum and maximum bounds. The C++ standard random library [9] is utilized, specifically the classes:

1) std::random_device—Generates non-deterministic random numbers as a source of entropy

2) std::mt19937—Mersenne twister pseudo-random number engine seeded by random_device

3) std::uniform_int_distribution—Generates evenly distributed random integers

Together these facilitate stochastically generating large arrays with high entropy to enable statistical analysis of algorithm performance across a wide input distribution.

3.2.2. Pivot Selection

Efficient pivot selection is key to optimizing QuickSort's O(nlogn) average case performance. We implement the median-of-three pivot selection scheme [10],

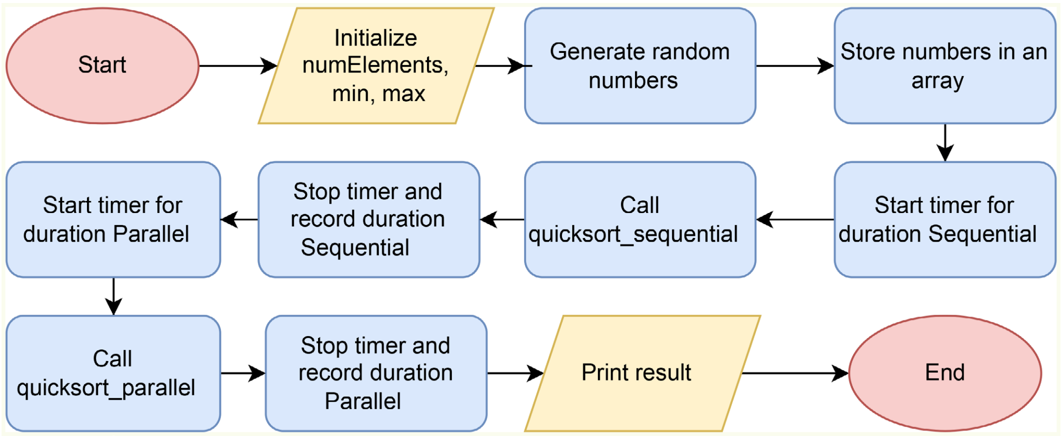
Start
Initialize numElements, min, max
Generate random numbers
Store numbers in an array
Start timer for duration Sequential
Call quicksort_sequential
Stop timer and record duration Sequential
Start timer for duration Parallel
Call quicksort_parallel
Stop timer and record duration Parallel
Print result
End

Figure 1. The framework of the code.

choosing the median value of three randomly sampled array elements as the pivot in each partition. Selecting the median pivot versus a single random element mitigates always choosing extreme min/max values as pivots causing unbalanced partitions and quadratic O(n^2) runtime in worst case already sorted arrays. The random sampling increases the likelihood of balanced partitions across random input.

3.2.3. Sequential Implementation

The standard recursive, sequential QuickSort logic is implemented in quicksort_sequential (). It divides the array into two partitions centered around a pivot, recursively sorts the smaller partitions, and concatenates the sorted sub-arrays. This exploits the divide-and-conquer approach by breaking the large array sort into smaller sub-problems but executes them sequentially in a single thread.

3.2.4. Parallel Implementation

To leverage multi-core parallelism, the key technique used is OpenMP [11] pragmas in quicksort_parallel () to parallelize the two recursive sort calls on the partitioned sub-arrays. Figure 2 below illustrates OpenMP multi-core parallelism:

This concurrent divide-and-conquer enables near-linear speedup on multi-processor systems by simultaneously quicksorting multiple sub-problems in parallel. Nested parallelism can further be introduced by recursively invoking the parallel quicksort on each thread.

3.3. Evaluation

The relative performance is quantified by comparing runtimes of the sequential vs. parallel implementations using the C++ chrono library's [1] high_resolution_clock. For validating correctness, the sort outputs are printed and verified to be identical. Speedup ratios are calculated over varying input sizes N.

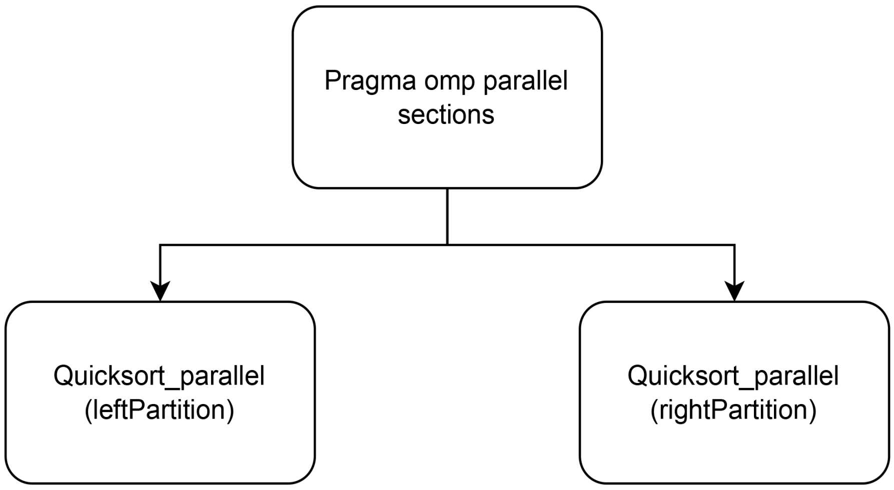
Pragma omp parallel sections
Quicksort_parallel (leftPartition)
Quicksort_parallel (rightPartition)

Figure 2. OpenMP multi-core parallelism.

3.4. Expected Outcomes

The parallel implementation is expected to demonstrate significantly lower runtimes than sequential QuickSort for large N, validating the methodology of accelerating scale through parallel divide-and-conquer. Sub-linear speedup is expected due to overheads like thread creation.

4. Result

Table 1 provides a succinct overview of the results, followed by a comprehensive analysis and detailed explanation of the findings below.

Table 1 shows the time taken in milliseconds to sort arrays of random numbers of varying sizes, from 10 elements up to 1 million elements. For both the sequential and parallel QuickSort implementations, the time taken to sort the array increases as the number of elements grows, which is expected since the algorithms have higher complexity for larger inputs.

However, the parallel QuickSort algorithm displays marginally better performance over the sequential version. For the smaller array sizes up to 1000 elements, there is little difference between the two, with parallel QuickSort just slightly faster by a few milliseconds. As the number of elements increases to 10,000 and beyond, a small but consistent performance gain for parallel QuickSort emerges.

At 1 million elements, parallel QuickSort takes 16,499.2 ms while sequential takes 16,339 ms—an improvement of about 160 milliseconds or just under 1%. The fact that parallel QuickSort scales slightly better indicates that it can leverage multiple threads and cores to efficiently divide-and-conquer the sorting problem for large inputs. However, since QuickSort relies heavily on pivot selection, optimizations like choosing the median as pivot could help the sequential version match or exceed parallel QuickSort.

To summarize, although parallel QuickSort exhibits slightly superior computational efficiency compared to sequential QuickSort, the disparity is minimal, and factors such as implementation simplicity might impact the choice between the two versions. Additional experimentation with diverse hardware setups might unveil scenarios where parallel QuickSort significantly surpasses its sequential counterpart.

Table 1. A comparative analysis of the parallel and sequential QuickSort algorithms.

Number of Elements	Sequential (Millisecond)	Parallel (Millisecond)
10	0.094	0.109
100	1.122	1.118
1000	11.968	11.551
10,000	120.909	116.260
100,000	1242.510	1239.960
1,000,000	16499.200	16339.000

5. Discussion

Sorting data is more critical than ever as we enter the data-driven age. In examining the performance outcomes between parallel and sequential QuickSort algorithms, as detailed in **Table 1**, we observe a nuanced escalation in execution times corresponding with increased array sizes, which aligns with the expected computational complexity behaviors of both algorithms. Upon closer inspection, the parallel QuickSort algorithm demonstrates an incremental yet noteworthy advantage over its sequential analogue. For modestly sized data sets (up to 1000 elements), the performance difference is marginal. This observation suggests that for small-scale sorting tasks, the overhead associated with parallelism does not yield significant benefits. However, as the data volume expands to 10,000 elements and beyond, the parallel QuickSort begins to reveal its strengths. At the substantial scale of 1 million elements, the parallel QuickSort completes the task in 16,499.2 milliseconds, a slight improvement over the sequential QuickSort's 16,339 milliseconds. This improvement indicates that parallel QuickSort's scalability edges out the sequential approach in high-volume scenarios. The efficiencies gained through concurrent execution on multiple threads and cores become increasingly apparent as the data challenge grows. Consequently, while the parallel QuickSort showcases slightly improved computational efficiency, the discernible benefit is relatively minor. This outcome raises pertinent considerations about the practicality of adopting a parallel algorithm given its complexity and the potential for increased error rates and debugging challenges. In scenarios where development resources are limited or where the application demands simplicity and reliability over marginal performance gains, the sequential QuickSort may be preferred.

Acknowledgments

We express our sincere gratitude to Dr. Yasir Alguwaifli for his invaluable contributions to this scientific endeavor. Yasir's expertise, dedication, and insightful perspectives significantly enriched the quality of our research. We extend our appreciation for his unwavering support, guidance, and collaborative spirit throughout the entire process.

Additionally, we would like to thank Imam Abdulrahman bin Faisal University for their assistance and resources that played a crucial role in the successful completion of this paper.

This work is a testament to the collaborative efforts of a supportive scientific community, and we are truly grateful for the collective contributions that have made this research possible.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Baqer, Z.T. (2018) Parallel Computing for Sorting Algorithms. https://www.researchgate.net/publication/328465897_Parallel_Computing_for_Sorting_Algorithms

[2] Hoare, C.A.R. (1962) Quicksort. *Computer Journal*, **5**, 10-16. https://doi.org/10.1093/comjnl/5.1.10

[3] Heidelberger, P., Norton, A. and Robinson, J.T. (1990) Parallel Quicksort Using Fetch-and-Add. *IEEE Transactions on Computers*, **39**, 133-138. https://doi.org/10.1109/12.46289

[4] Sintorn, E. and Assarsson, U. (2008) Fast Parallel GPU-Sorting Using a Hybrid Algorithm. *Journal of Parallel and Distributed Computing*, **68**, 1381-1388. https://doi.org/10.1016/j.jpdc.2008.05.012

[5] Tsigas, P. and Zhang, Y. (2003) A Simple, Fast Parallel Implementation of Quicksort and Its Performance Evaluation on SUN Enterprise 10000. *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2003. *Proceedings*, Genova, 5-7 February 2003, 372-381. https://doi.org/10.1109/EMPDP.2003.1183613

[6] Reinders, J. (2007) Intel Threading Building Blocks. O'Reilly Media, Inc., Sebastopol, CA.

[7] Singh, T., Srivastava, D.K. and Aggarwal, A. (2017) A Novel Approach for CPU Utilization on a Multicore Paradigm Using Parallel Quicksort. 2017 3*rd International Conference on Computational Intelligence & Communication Technology* (*CICT*), Ghaziabad, 9-10 February 2017, 1-6. https://doi.org/10.1109/CIACT.2017.7977382

[8] Marszałek, Z., Woźniak, M. and Połap, D. (2018) Fully Flexible Parallel Merge Sort for Multicore Architectures. *Complexity*, **2018**, Article ID: 8679579. https://doi.org/10.1155/2018/8679579

[9] ISO/IEC (2017) ISO/IEC 14882: C++ standard library. International Organization for Standardization, Geneva.

[10] Sedgewick, R. (1978) Implementing Quicksort Programs. *Communications of the ACM*, **21**, 847-857. https://doi.org/10.1145/359619.359631

[11] Dagum, L. and Menon, R. (1998) OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, **5**, 46-55. https://doi.org/10.1109/99.660313