

Comparing Parallel Sorting Algorithms on a High Performance Computing Cluster

Benjamin Pierce

Department of Computer and Data Sciences

Alberto Safra

Department of Computer and Data Sciences

Colin Causey

Department of Computer and Data Sciences

Jason Richards

Department of Computer and Data Sciences

Abstract—“Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms” (Introduction to Algorithms, pg. 148).

Sorting is commonly viewed as the most fundamental problem in the study of algorithms. Some cited reasons for this are that a great many software applications use sorting for various reasons, and a great many algorithms use sorting as a subroutine [1]. Given its ubiquity, therefore, it is valuable to be able to solve the sorting problem efficiently. For this reason, many efficient sorting algorithms have been developed (e.g., mergesort, quicksort, heapsort). Given the asymptotic lower bound of $\Omega(n \log n)$ for comparison-based sorting algorithms such as these, a natural route to take to achieve greater performance is parallel computing. In the interest of wanting to select the optimal sorting algorithm to run on a parallel computer, it is valuable to empirically compare the performance of various parallelized sorting algorithms. This is the aim of our research. We will conduct an empirical analysis and comparison of various parallel sorting algorithms. The criteria for evaluation will be (i) execution time, (ii) memory footprint, and (iii) scalability. The research will be conducted on Case Western Reserve’s high-performance computing (HPC) architecture. We will implement various parallel sorting algorithms and execute them with variously sized and randomly permuted input arrays. The execution times and memory footprints will be recorded for each run. Additionally, we will run the algorithms on a varying number of CPUs (e.g., one CPU, two CPUs, four CPUs) in order to assess their scalability. After these data have been collected, we will perform data analysis and use it to compare the various sorting algorithms. The comparison will facilitate making an informed choice about which sorting algorithm to use under various conditions (e.g., the number of CPUs available and the size of the input array).

I. INTRODUCTION

Sorting is a fundamental problem in many different fields. Any algorithm that depends on having some ordering to the data will likely deal with sorting in some form, and even simple tasks like “find the top n examples” are accomplished using sorts. However, today’s world of “Big Data” has led to an astronomical increase in the amount of computing power used to process it. As a response, the field of distributed computing has become more and more important. Distributed computing is a paradigm where computation is spread across many processors, working at the same time, rather than the usual sequential, single-core processing method. In this fashion, parallelized versions of different sorting algorithms have emerged. This study focuses on four of the most pop-

ular sorting algorithms: Quicksort, Mergesort, Heapsort, and Insertion sort.

II. BACKGROUND & THEORY

As mentioned before, all comparison based sorts have an asymptotic lower bound of $\Omega(n \log n)$. Indeed, Mergesort, and Heapsort achieve a $O(n \log n)$ upper bound as well, and Quicksort behaves as $O(n \log n)$ in the average case. Insertion sort, as a less advanced algorithm, has an upper bound of $O(n^2)$, as does Quicksort. [1] However, Quicksort’s upper bound is rarely the case, and tends to have smaller constant factors than either Mergesort or Heapsort. [2] Additionally, each algorithm is constructed differently, and some are more amenable to parallelization than others.

A. Insertion sort

Insertion sort is the simplest sorting algorithm discussed here. It works by iteratively building a sorted array one element at a time. That is, the algorithm loops through the list, creating a sorted and unsorted part, and inserting each element it encounters at the correct location. This procedure is $O(n^2)$ worst case, but tends to perform quite well on small datasets, and better than other $O(n^2)$ sorts, such as bubble sort. [3] For this reason, insertion sort is often used as a subroutine in hybrid sorts like Timsort, where it is called when the size of a subarray is smaller than some threshold. [4] Insertion sort can be parallelized

B. Mergesort

Mergesort is a divide and conquer sort that recursively divides an array into subarrays, and merges them together such that each subarray is sorted. This procedure can be seen in Figure 1. The recursive invariant is that each returned subarray is sorted, which can be proved by induction, as an array of length 1 is already sorted by definition. [1] Mergesort has a time complexity of $O(n \log n)$ in both the average and worst case, although constant factors can make it worse than Quicksort in practice. Mergesort is a natural candidate for parallelization; the divide and conquer nature of the algorithm is already suitable for this process.

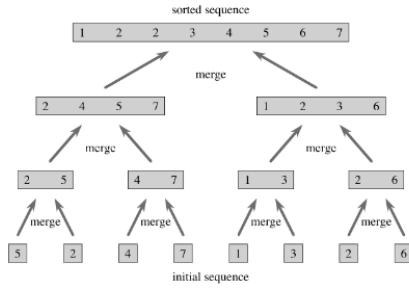


Fig. 1. Mergesort diagram from [1]

C. Heapsort

Heapsort is another $O(n \log(n))$ comparison sort. It, along with the heap data structure, was invented in 1964. [5] Heapsort first turns the dataset into a max heap, which is a binary tree where each parent node is greater than its children. This process, called *heapification*, is an $O(n)$ algorithm. Sorting is performed by repeatedly popping the root node (the maximum value) and re-heapifying. This procedure takes advantage of the binary tree structure, and is worst case $O(n \log(n))$ overall. [1] Unfortunately, Heapsort is a poor candidate for parallelization, as it does not partition into subarrays and depends on the root node being the absolute maximum.

D. Quicksort

The final sort discussed here is Quicksort, another $O(n \log(n))$ comparison sort. Developed in 1961 [2], Quicksort is a partitioning sort that works by selecting a pivot element in an array and partitioning based on a comparison to the pivot, as shown in Figure 2. In particular, Quicksort

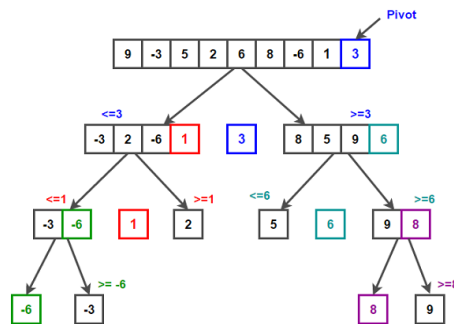


Fig. 2. Quicksort diagram. Source

is perhaps the most parallelizable sorting algorithm, and can achieve a linear speedup with few modifications. [6] This is because each subarray can be sorted independently, and this leads to speedup.

III. METHODOLOGY

In this study, the OpenMP [7] API is used for parallelization. It is an example of a fork-join methodology, where each parallel thread forks off from a main thread, then, the results are joined back together. This is implemented into the C

programming language via `#pragma` preprocessor directives. OpenMP is useful for obviously parallel cases, such as Mergesort and Quicksort, as the division of work is quite clear.

This work utilizes Case Western Reserve University's Markov cluster, which runs on Intel Xeon x86_64 Processors. Using the Simple Linux Utility for Resource Management [8] (SLURM), resources are allocated in batch mode, which enables repeatable, large scale experiments with the requested resources.

IV. RESULTS

V. CONCLUSION

As expected, the three $O(n \log(n))$ algorithms perform better than Insertion sort. Due to the more parallel nature of Quicksort and Mergesort, these algorithms benefit more from parallelization than the more sequential Heapsort. This is because the divide and conquer strategy is inherently more parallel, which should be taken into account when developing new algorithms to be run on parallel and distributed computing platforms.

REFERENCES

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, Jul. 2009, google-Books-ID: aefUBQAAQBAJ.
- [2] C. A. R. Hoare, "Algorithm 64: Quicksort," *Communications of the ACM*, vol. 4, no. 7, p. 321, Jul. 1961. [Online]. Available: <https://doi.org/10.1145/366622.366644>
- [3] D. E. Knuth, "The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition," p. 1061.
- [4] N. Auger, V. Jugé, C. Nicaud, and C. Pivoteau, "On the Worst-Case Complexity of TimSort," in *26th Annual European Symposium on Algorithms (ESA 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), Y. Azar, H. Bast, and G. Herman, Eds., vol. 112. Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018, pp. 4:1–4:13. [Online]. Available: <http://drops.dagstuhl.de/opus/volltexte/2018/9467>
- [5] G. E. Forsythe, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–349, Jun. 1964. [Online]. Available: <https://doi.org/10.1145/512274.512284>
- [6] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, vol. 39, no. 3, pp. 85–97, Mar. 1996. [Online]. Available: <https://doi.org/10.1145/227234.227246>
- [7] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," May 2008. [Online]. Available: <http://www.openmp.org/mp-documents/spec30.pdf>
- [8] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer, 2003, pp. 44–60.

APPENDIX