Comparing Parallel Sorting Algorithms on a High Performance Computing Cluster Using OpenMP

Benjamin Pierce
bgp12@case.edu
Department of Computer and Data Sciences

Alberto Safra
axs1249@case.edu
Department of Computer and Data Sciences

Colin Causey
cjc158@case.edu
Department of Computer and Data Sciences

Jason Richards
jdr145@case.edu
Department of Computer and Data Sciences

Abstract—**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 and studied. Three of the most popular and efficient sorting algorithms are Mergesort, Quicksort, and Heapsort. Given the asymptotic lower bound of $\Omega(nlog(n))$ 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 particular parallel computing architecture, it is valuable to empirically compare the performance of different parallelized sorting algorithms. This is the aim of our research. In this project, we conduct an empirical analysis and comparison of parallelized versions of two popular sorting algorithms: Mergesort and Quicksort. Heapsort and the difficulties of parallelizing it are also considered. The criteria for evaluation are (i) execution time and (ii) scalability. The research was conducted on Case Western Reserve's high-performance computing (HPC) architecture, specifically the Markov cluster. We implement parallel Mergesort and Quicksort and execute them with variously sized and randomly permuted input arrays. The execution times are recorded for each run. Additionally, we run the algorithms on a varying number of CPUs (e.g., one CPU, two CPUs, four CPUs) in order to assess their scalability. After collecting the data, we perform data analysis and use it to compare the algorithms according to the aforementioned criteria for evaluation. 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 to be solved in many algorithms and applications. Any algorithm or application, for instance, that depends on having some ordering to its data will likely deal with sorting in some form. Because of its pervasiveness, solving the sorting problem efficiently is highly desirable. For this reason, many efficient sorting algorithms have been developed. Three of the most popular are Mergesort, Quicksort, and Heapsort. All three of these algorithms have time complexities of $O(nlog(n))$ in the average case. Mergesort and Heapsort also achieve this time bound in the worst

case, while Quicksort can in rare situations exhibit a runtime of $O(n^2)$ in the worst case. Since the asymptotic lower bound for comparison-based sorting algorithms such as these is $\Omega(nlog(n))$, significantly improving the performance of them isn't feasible with serial computation. For this reason, we must turn to parallel computation to achieve significant performance gains. In order to take advantage of the performance increases enabled by parallel computing, parallelized versions of various sorting algorithms have been developed and studied. While theoretical analysis of parallel sorting algorithms is useful for understanding the asymptotic differences in the runtimes, determining and analysing the empirical performance of the algorithms on a specific computer architecture is valuable for making an informed choice about which algorithm to choose for that particular architecture and under various conditions (e.g., the number of CPUs available and the size of the input array to sort). This study focuses on an empirical evaluation of parallelized versions of two of the most popular sorting algorithms: Quicksort and Mergesort. Heapsort is also analyzed as it is comparable to Quicksort and Mergesort, although it is not parallelized due to difficulties that we will discuss. The algorithms are parallelized with OpenMP so as to take advantage of multithreaded, shared-memory parallelism on Case Western Reserve's HPC Markov cluster. The layout of the paper is as follows: In Section II, we give a brief overview of parallel computing and sorting algorithms. Section III...

II. BACKGROUND & THEORY

Today's world of "Big Data" has led to an astronomical increase in the amount of computing power needed to efficiently process data. As a result, parallel computing has become an important and necessary approach to solving computationally-intensive problems. Parallel computing is a paradigm where computation is spread across many processors working on multiple tasks and/or data at the same time. This is in contrast to serial computation in which each step of a computation is performed one after another. The processors used in parallel computation can be within a single node (as in multithreading and shared-memory parallel architectures) or they can be distributed across multiple nodes interacting

with each other (as in message-passing architectures). In order to take advantage of parallel computing, algorithms must be "parallelized," i.e., written in such a way as to take advantage of parallel computing. This usually involves specifying in the algorithm which parts can execute in parallel as well as dividing tasks the algorithm performs among multiple processors. Some algorithms are more inherently parallelizable than others. Algorithms, for instance, that operate by breaking a problem up into subproblems that can be solved independently (i.e., divide-and-conquer algorithms) are natural candidates for parallelization. The divide-and-conquer paradigm consists of three stages for solving a problem: the *divide* stage, the *conquer* stage, and the *combine* stage. In the divide stage, the problem is recursively divided up into subproblems until the subproblems become sufficiently simple such that they can be solved directly. In the conquer stage, the subproblems are solved. Finally, in the combine stage, the subproblems are combined in such a way that the original problem is solved. Because both Mergesort and Quicksort follow the divide-and-conquer paradigm, they are excellent candidates for parallelization. Algorithms that do not follow this paradigm can be more difficult to parallelize. Heapsort falls into this category. As we will discuss, Heapsort is much more difficult to parallelize than Mergesort and Quicksort.

As stated in the introduction, all comparison-based sorting algorithms have an asymptotic lower bound of $\Omega(nlog(n))$. Mergesort and Heapsort achieve a $O(nlog(n))$ upper bound as well, and Quicksort achieves $O(nlog(n))$ in the average case. In the worst case, Quicksort has an upper bound of $O(n^2)$; however, Quicksort's upper bound rarely occurs and tends to have smaller constant factors than either Mergesort or Heapsort. [2]

The following three subsections discuss the design and parallelization of Mergesort, Heapsort, and Quicksort.

A. Mergesort

Mergesort is a divide-and-conquer sorting algorithm that recursively divides an array into subarrays and merges them together in such a way that the original array is sorted. The operation of this procedure on an array of eight elements is depicted in Figure 1. Mergesort has a time complexity of
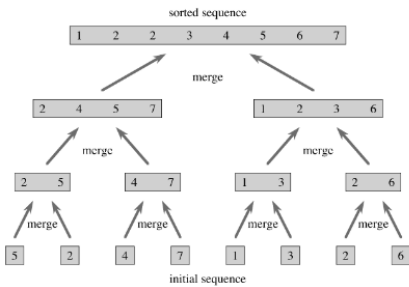

Fig. 1. Mergesort diagram from [1]

$O(nlog(n))$ in both the average and worst case, although constant factors can make it worse than Quicksort in practice. Given the divide-and-conquer paradigm that Mergesort uses, it is a natural candidate for parallelization. Mergesort follows this paradigm as follows: First the input array (of size n) is recursively divided in two until there are n single-element subarrays (divide). Each single-element array is trivially sorted (conquer). From there, the sorted subarrays are merged together, resulting in a single array that is equivalent to the input array but in sorted order (combine). Given the mechanics of the algorithm, there are two primary ways to parallelize Mergesort: (1) Parallelize the divide stage (i.e., the two recursive calls to the Mergesort procedure) and (2) Parallelize the combine stage (i.e., the Merge procedure). It is possible to implement either (1), (2), or both. According to [1], only parallelizing the recursive calls to Mergesort will result in diminishing returns on performance as the number of processors grows to more than a few dozen. In order to efficiently scale up to hundreds of processors, parallelizing the Merge procedure is necessary because it becomes the performance bottleneck. Our implementation of parallelized Mergesort is significantly based on [SOURCE]. The implementation parallelizes the recursive calls to Mergesort while implementing a serial Merge procedure. There are a couple of reasons for this design decision. First, the HPC architecture we are using (the Case Western Reserve Markov cluster) has well under one hundred CPU cores per node. Thus, parallel Mergesort with only the divide stage parallelized theoretically scales well within this core-count range. Second, parallelizing the Merge procedure proved to be difficult with OpenMP due to a lack of low-level control over threads.

B. Heapsort

Heapsort is another $O(nlog(n))$ comparison-based sorting algorithm. 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 then its children. This process, called *heapification*, is an $O(n)$ algorithum. 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 $O(nlog(n))$ overall. [1] Unfortunately, Heapsort is a poor candidate for paralleization, as it does not partition into subarrays and depends on the root node being the absolute maximum.

C. Quicksort

The final sort discussed here is Quicksort, another $O(nlog(n))$ 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 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.
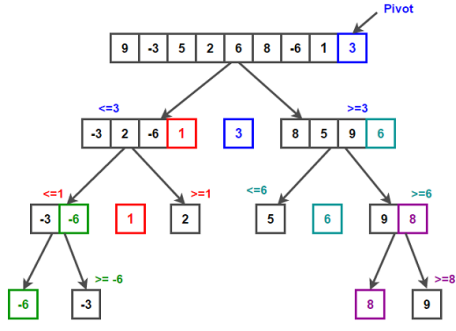
Fig. 2. Quicksort diagram. Source

III. METHODOLOGY

For our project, we used... For our project, we implemented our sorting algorithms in the C programming language. OpenMP [7] is used for achieving parallelization for our implementations of parallelized Mergesort and Quicksort. OpenMP facilitates multithreaded shared memory parallelism through the use of *#pragma* preprocessor directives in our C code. We ran our algorithms on Case Western Reserve University's HPC Markov cluster, which uses Intel Xeon x86_64 Processors.

The experiment consisted in running our sorting algorithms on array sizes of $10,000$, $100,000$, $1,000,000$, $10,000,000$, $100,000,000$, and $1,000,000,000$ while using CPU core numbers of 1, 2, 4, 8, and 16 for each array size. The execution times for each run of each array size/CPU count combination were collected and recorded in CSV files. From there, we analyzed the data in order to compare the performance and scalability of our sorting algorithms, producing several graphs in order to provide visuals for aiding in the analysis. The execution of our algorithms on the Markov cluster was achieved through the use of the Simple Linux Utility for Resource Management [8] (SLURM), which enables repeatable, large scale experiments to be performed with precisely configured resources. We wrote SLURM batch scripts that execute our sorting algorithms on Markov cluster compute nodes for each combination of array size and CPU count and return the execution times in output files.

IV. RESULTS

In this section, the results of each algorithm will be shown. The main variable of interest is how much time each algorithm takes as a function of the length of the input array n. We begin with an individual discussion of Quicksort.

A. Quicksort

There was great success with parallel Quicksort. As a divide-and-conquer algorithm, it is a naturally parallel algorithum. The results of this algorithum can be seen in Figure 3.
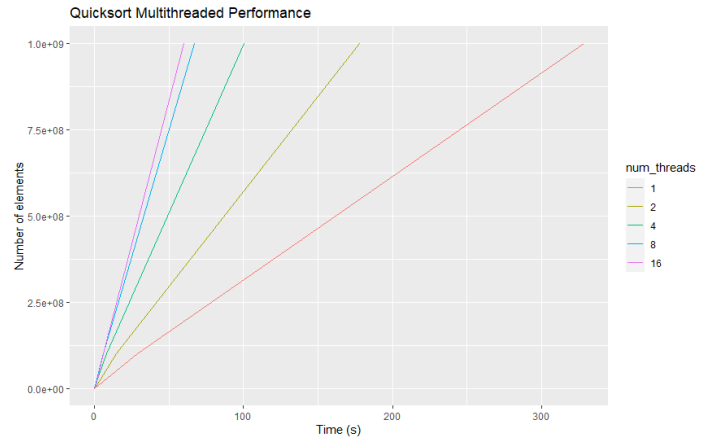

Fig. 3. Performance of Quicksort

B. Heapsort

Heapsort proved to be purely sequential, as the core of the algorithm depends on universal array access in the current form. The results of the nonparallel Heapsort are seen in Figure 4. As Figure 4 shows, the performance of Heapsort
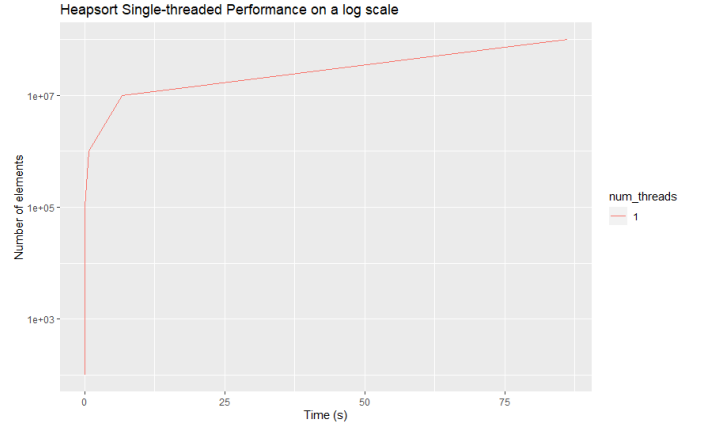

Fig. 4. Performance of Heapsort

becomes quite poor very rapidly. Although in its current form, Heapsort cannot be parallelized, we present an alternative algorithm utilizing the core ideas of Heapsort in a parallel manner.

Algorithm 1 presents a possibly parallel algorithm that utilizes the idea of Heapsort. Essentially, the algorithm divides the array into many smaller heaps, each managed by a single thread. This way, all heapification can be done in parallel, and the "max" item is selected from the number of heaps through a straightforward traversal of the roots of the sub-heaps. However, this algorithum has several downsides. One is that the end reheapification is still sequential; as by necessity, one heap must be popped, the algorithm must wait on the reheapification of that heap. This heap will be smaller by a constant factor then the traditional, single-threaded Heapsort,

Algorithm 1: Parallel-Heaps

Result: A sorted list

list a;

list $result$;

int $threads$;

let $lists$ be a partitioned into $threads$

while *every list in lists is not empty* **do**

    In Parallel;

        $heapify$ each list;

    pop the largest element of all sub-heaps into $result$;

    $re - heapify$ the heap that has been popped;

end

return $result$

Fig. 6. Comparative Performance with 8 threads

however. Therefore, $Parallel - Heaps$ will be at most a constant speedup for Heapsort.

$Parallel - Heaps$ was not implemented in this study, as it is not possible to do with the OpenMP framework; such a method would require POSIX threads. For the sake of comparison, this algorithm was thus excluded. Further investigation of this $Parallel - Heaps$ algorithm is a topic for further study.

C. Mergesort

As with Quicksort, Mergesort proved to be naturally parallel. The results of parallel Mergesort can be seen in Figure 5
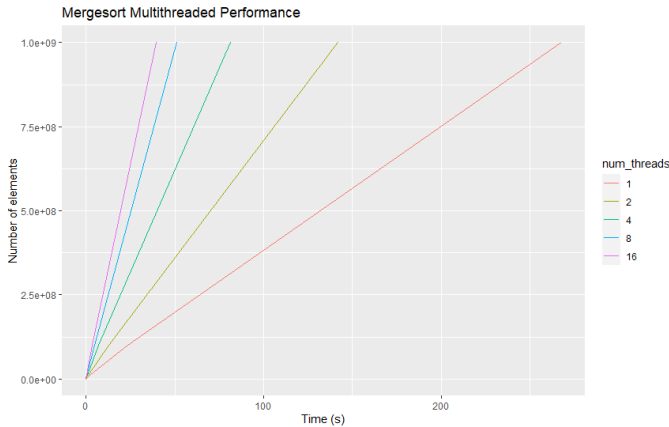

Fig. 7. Comparative Performance with for 10^4 elements

performance of Quicksort appears to worsen with the number of threads more then Mergesort; this suggests significant overhead introduced by parallelization. However, this effect goes away with an increase in the number of elements, as in Figure 8.

This suggests that there is significant overhead with smaller array sizes, but this is overcome by efficeicny gains on larger array sizes.

V. CONCLUSION

As expected, the three $O(nlog(n))$ algorithms perform better then Insertion sort. Due to the more parallel nature of Quicksort and Mergesort, these algorithms benefit more from parallization then 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.

Fig. 5. Performance of Mergesort

D. Shellsort

E. Comparative Results

Results are compared in a few different ways. One is to hold the number of threads constant, and plot the performance of each algorithm, as in Figure 6, which shows that Heapsort, which is not paralleized, proves to be much worse then either Quicksort or Mergesort. Additionally, it is useful to compare results on a single array size with multiple threads. This can be seen for 10^6 elements in Figure 7 Interestingly, the
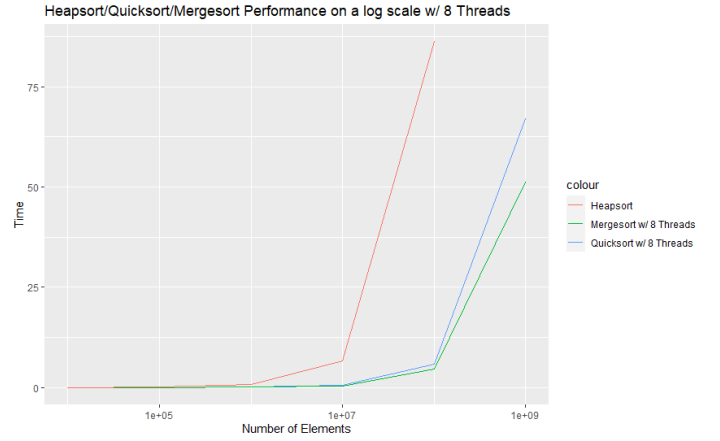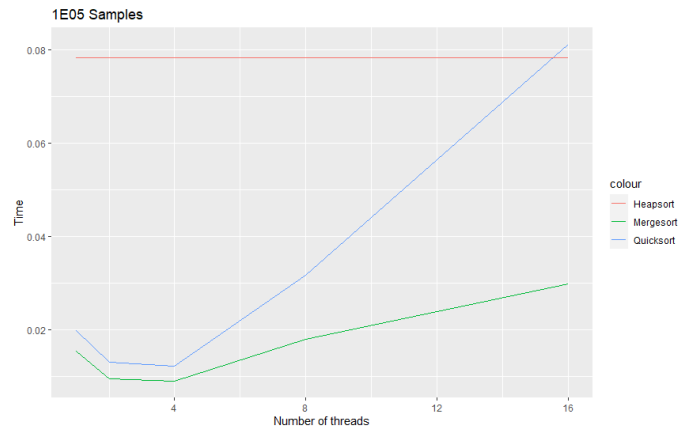
REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, Jul. 2009, google-Books-ID: aefUBQAAQBAJ.
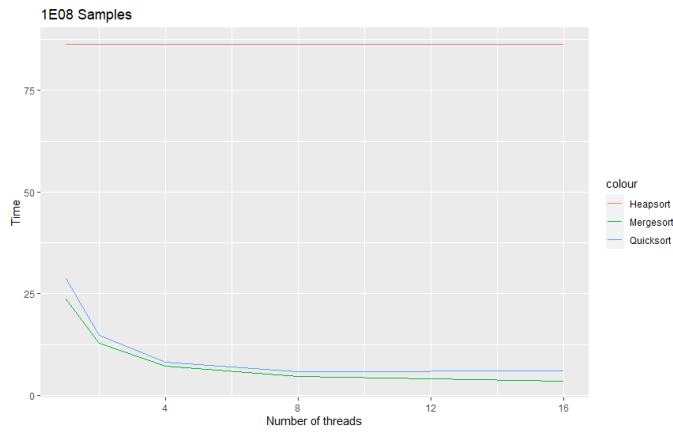
Fig. 8. Comparative Performance with for 10^7 elements

[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