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: Mergesort and Quicksort. Heapsort is also analyzed as it is comparable to Mergesort and Quicksort, although it is not parallelized due to difficulties that we will discuss. Instead, it is used as a reference serial algorithm to compare against our two parallelized algorithms. The algorithms are parallelized with OpenMP so as to take advantage of multi-threaded, shared-memory parallelism on Case Western Reserve University's HPC Markov cluster. The criteria for evaluation are (i) execution time and (ii) scalability, i.e., the extent to which adding more processors decreases execution time.

The layout of the paper is as follows: In Section II, we give an overview of parallel computing and the sorting algorithms that are included in our experiment and analysis. In Section III, we lay out our experimental methodology for comparing our sorting algorithms. Section IV presents our results and analysis. In Section V, we give our concluding remarks. Finally, in the Appendix, we provide information for how to access our code and data and how to run our experiment on the Case Western Markov cluster.

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 $O(nlog(n))$ in both the average and worst case, although

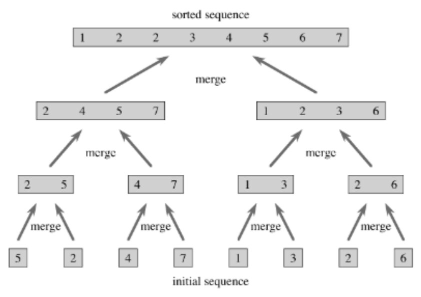

Fig. 1. Mergesort diagram from [1]

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 [3]. 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. OpenMP is used to parallelize Mergesort by making use of OpenMP's parallel sections construct. A parallel sections region is created, and each recursive call to Mergesort is enclosed in its own parallel section. This allows OpenMP to spawn multiple threads that can execute the recursive calls in parallel. Nested parallelism is used so that each level of the recursion can spawn more threads (if they are available). The performance gains from this parallelism are significant as will be demonstrated in our Results section.

B. Heapsort

Heapsort is another $O(nlog(n))$ comparison-based sorting algorithm. It, along with the heap data structure, was invented in 1964. [4] 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 repeatably 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 parallelization, as it does not divide or partition the input array into subarrays and depends on the root node being the absolute maximum.

C. Quicksort

The final sort discussed here is Quicksort, yet another $O(nlog(n))$ comparison-based sorting algorithm. 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. Similarly to Mergesort, Quicksort follows the divide-and-conquer paradigm. This makes Quicksort a naturally parallelizable sorting algorithm that can in theory achieve a significant performance increase with few modifications to its core logic. [5] This is because each subarray can be sorted independently, and this leads to speedup on parallel computers. The fundamental steps of Quicksort are as follows: The first step is to partition the array around a selected "pivot" point to produce two subarrays. These subarrays are then recursively sorted until the entire array of elements are in the correct order. Like Mergesort, there are two ways to parallelize Quicksort. We can parallelize the recursive calls to the Quicksort procedure, we can parallelize the Partition procedure, or both. To keep the design comparable to our Mergesort implementation, we opted to simply parallelize the recursive calls to Quicksort. OpenMP is used to parallelize Quicksort by making use of OpenMP task directives. In OpenMP, the task directive can tell OpenMP to dedicate a thread to a specific block of code labeled by the task. In this case, when the two subarrays are selected, each recursive call to the Quicksort method gets a dedicated OpenMP task initiated to create a parallel region of threads to be called later. This increases efficiency incredibly when allocating more and more OpenMP threads for Quicksort to use as is demonstrated in our Results section.

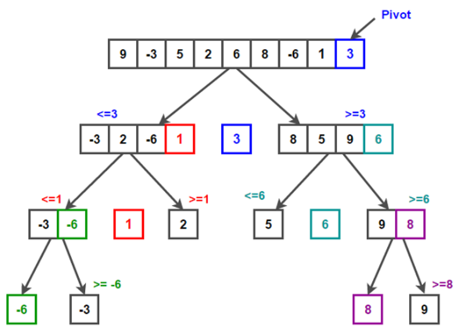

Fig. 2. Quicksort diagram. Source

III. Methodology

For our project and experiment, we implemented our sorting algorithms in the C programming language. OpenMP [6] 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 [7] (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 the execution times (measured in seconds) are returned in output files. The primary comparison is between Mergesort and Quicksort; however, Heapsort is included in our results as a reference serial algorithm. Heapsort is used as a sorting algorithm in the C standard library. It thus serves as an example of the performance that can be expected from calling a standard library sorting function, and so it is useful to see how its performance compares to our parallelized Mergesort and Quicksort.

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.

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 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.

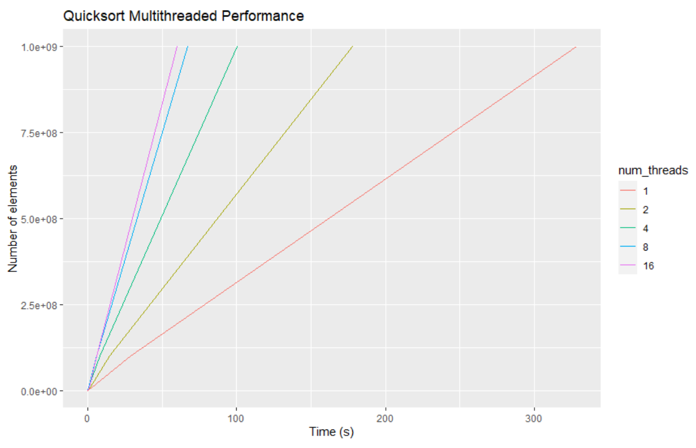

Fig. 3. Performance of Quicksort

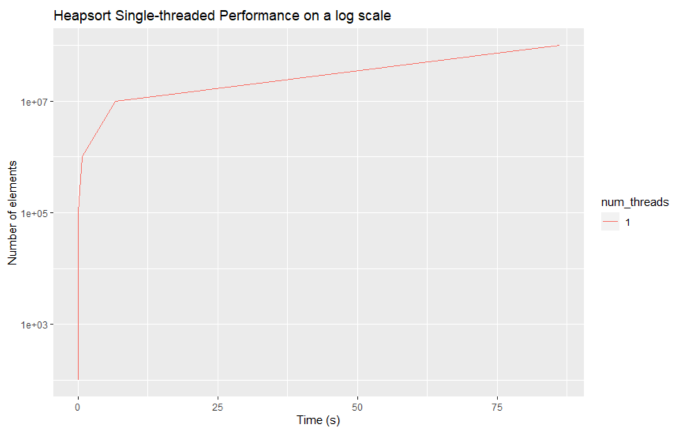

Fig. 4. Performance of Heapsort

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-

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 $result$
```

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, 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

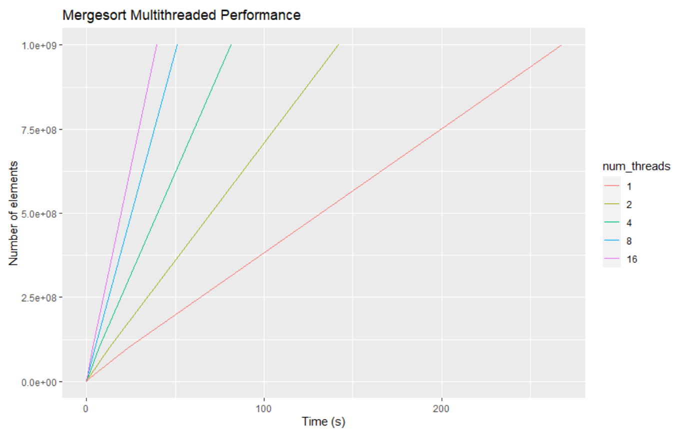

Fig. 5. Performance of Mergesort

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 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

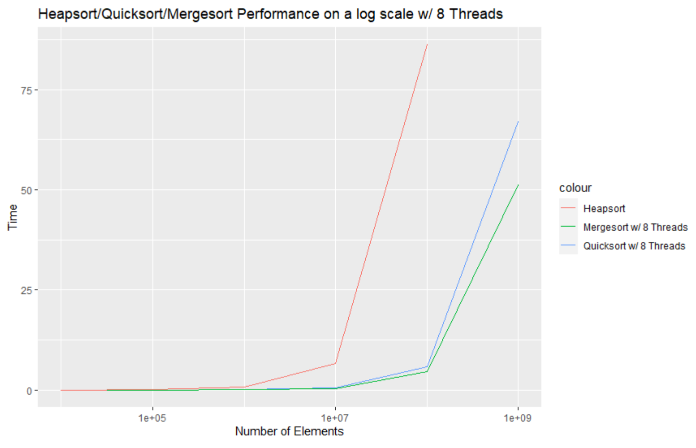

Fig. 6. Comparative Performance with 8 threads

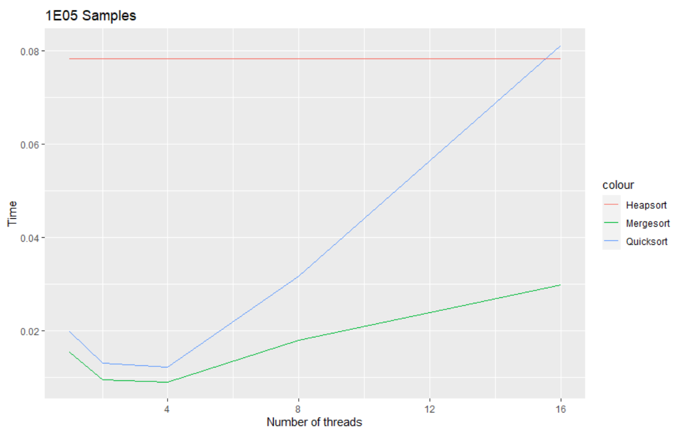

Fig. 7. Comparative Performance with for 10^4 elements

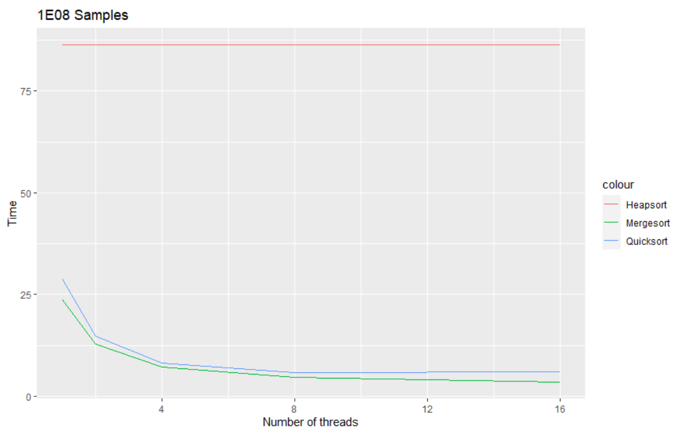

Fig. 8. Comparative Performance with for 10^7 elements

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.

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] A. Radenski, "Shared memory, message passing, and hybrid merge sorts for standalone and clustered smps," *International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 367–373, 2011. [Online]. Available: http://www1.chapman.edu/~radenski/research/papers/mergesort-pdpta11.pdf

[4] 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

[5] 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

[6] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," May 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[7] 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

All of our code and data can be found in our GitHub repo here. Our implementation of parallel Mergesort is located in mergesort/merge_sort_OpenMP.c. The parallel Quicksort implementation is in quicksort/quicksortOMP.c. Our serial Heapsort implementation is located in heapsort/heapsort.c. In order to run the experiment, clone the repo into a location on Case Western's HPC Markov cluster. From there, run the SLURM batch scripts. The first is **sorting.slurm** and is located in mergesort/. This script runs parallel Mergesort on a compute node on all combinations of array sizes and CPU core numbers (as explained in the main part of the paper). Once finished, an output SLURM file will be produced that contains the execution times of the sorts. *Mutatis mutandis* for Quicksort (the batch script is called **qs-job.slurm** and is located in quicksort/). CSV files that we manually created from the data produced from our SLURM jobs are located in mergesort/ and quicksort/ and are called "MergeSort Data.csv" and "project.csv", respectively.