Popular Sorting Algorithms in Parallel

Maria Bamundo, Hamza Memon and Megan Bayer
Department of Computer Science
Siena College
Loudonville NY 12211

Abstract

In this paper we will explore the development of parallel versions of the popular serial sorting algorithms Counting Sort, Merge Sort, and Quicksort. First we will provide a brief summary of the importance of sorting algorithms in computer science education and in the professional world. Then we will discuss how parallel programs are made as well as the advantages and disadvantages of this type of programming. Finally we will describe our six algorithms. We will assess the time each algorithm takes to sort an array of numbers and the serial algorithms will be compared to their parallel counterparts in order to determine which implementation is most efficient. Thus determining which algorithms are worth parallelizing,

1 Introduction

Sorting is one of the fundamental topics of any computer science student's curriculum. Students encounter many different sorting algorithms and examine how they can be used efficiently. This involves comparing the various algorithms with one another and examining both the cost and the times of such algorithms. Sorting algorithms can be used to organize elements into a specific order or to group similar elements. The ordering is dependent upon the data type for example when dealing with numbers, numerical sorting is used (Knuth, 420). If one wanted to sort words in an index however, they may choose to use lexicographical order. For the purposes of our paper we will be referring only to ascending numerical ordering as our arrays are filled with random numbers.

The main purpose of this project is to examine some serial sorting algorithms one might come across while taking an algorithms course and parallelize them using threads. The algorithms we will be focusing on are Counting Sort, Merge Sort, and Quicksort. Each parallel algorithm will be compared to its serial counterpart by means of speed. The arrays will increase in size with each test and we will also be experimenting with the thread count in the parallel algorithms. We will display the strong and weak scaling of our parallel algorithms. Pacheco defines strong and weak scalability as follows: “If when we increase the number of processes/threads, we can keep the efficiency fixed without increasing the problem size, the program is said to be strongly scalable. If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, then the program is said to be weakly scalable” [6]. The overall goal is to determine when the parallel implementation becomes more efficient than the serial implementation if it does at all.

2 Sorting in Computer Science

Before we delve into the mechanics and testing of our serial and parallel sorting algorithms it is imperative that we acknowledge their importance in computer science education as well as their real world applications. Some sorting algorithms are easy to develop. Therefore, students are encouraged to come up with them on their own thus enhancing their understanding of algorithm design. For those sorting algorithms that are not so intuitive, students are often asked to trace through these algorithms in order to better understand how they work. In analyzing complex sorting algorithms students may learn to think as a computer programmer. Furthermore, tracing through these algorithms sharpens a student's coding skills. It is similar to the notion that one must learn to read before one must learn to write. A student must first be able to trace through code in order to create their own.

The emphasis on the efficiency of sorting algorithms in computer science education is meant to develop a student's understanding of time complexity. First students must recognize the efficiency of an algorithm in order to realize when it would be advantageous to use said algorithm. Additionally, an understanding of the time complexity of sorting algorithms is expected to lead students to develop more efficient code with these algorithms as well as their advantages and disadvantages in mind.

Humans usually prefer to read sorted data. Therefore, in the real world, sorting algorithms serve many purposes across many different fields. The ability to sort data in any industry makes for more efficient work. For example, a company may keep records on each of its employees. If a manager wanted to access this information they could search their companies database. If the database was not organized it would take a very long time to find an employee's file. It would be like looking for the first instance of a term in a textbook without an index. You would literally have to read every page. However, if the database was sorted in lexicographical order by employees last name, or perhaps in numerical order by unique employee ID numbers this search would be much faster. Additionally, sorting makes it much easier to identify patterns in data. Sorting also helps in comparisons of data lists.

3 Parallel Programming

Before we describe how we parallelized Counting Sort, Merge Sort and Quicksort we will first provide a brief overview of how to program in parallel as well as some of the advantages and disadvantages of parallel programming. Libeskind-Hadas writes “today parallel computers are with hundreds and even thousands of processors are used in a broad range of applications” [5]. Parallelism is not only relevant in the world of computer science, it is utilized by many different industries. This was true in 1998 when Libeskind-Hadas’ article “Sorting in Parallel” was published and parallel computing is even more important now. The potential to devise an effective parallel sorting algorithm that could be used across disciplines was one of the driving factors of our research.

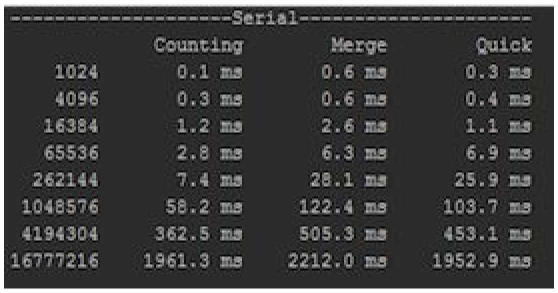
There are two commonly used approaches in parallel programming. These are data-parallelism and task-parallelism. Data-parallelism involves giving each processor a portion of the data. Then each processor performs similar operations on the data. In the end the now manipulated portions of the data are often brought back together. Whereas, task-parallelism, divides up the tasks amongst the processors.

Parallel programming has many advantages over serial programming. For example, it often results in enhanced speed, and better cost performance often as a result of the speedup. Since most computers are now built with multiple cores, parallel programs are becoming more popular since they take advantage of this advancement in technology.

While there appear to be many advantages to parallel programming, these programs are often difficult to write. One must divide the work or data as efficiently as possible amongst the processors. Load balancing may be difficult if some cores perform faster than others. Thus, even distribution may not alway be the most efficient distribution. Furthermore, communication is generally needed between the processors. This could cause a parallel program to perform slowly especially if the load balancing is inefficient and processes must wait for other processes to finish in order to complete their respective tasks. Lastly, parallel programs may consume more power than serial programs. We will touch upon power consumption later in the paper as it came up in our research when reading "Energy Consumption Analysis of Parallel Sorting Algorithms Running on Multicore Systems."

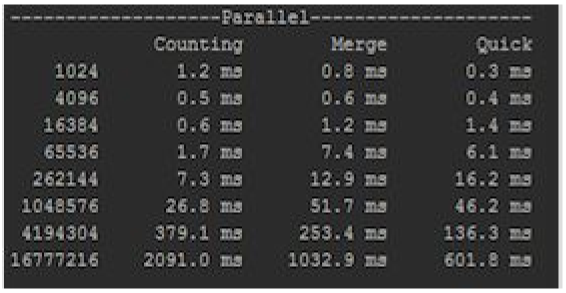
4 Data

Table A: Serial Results



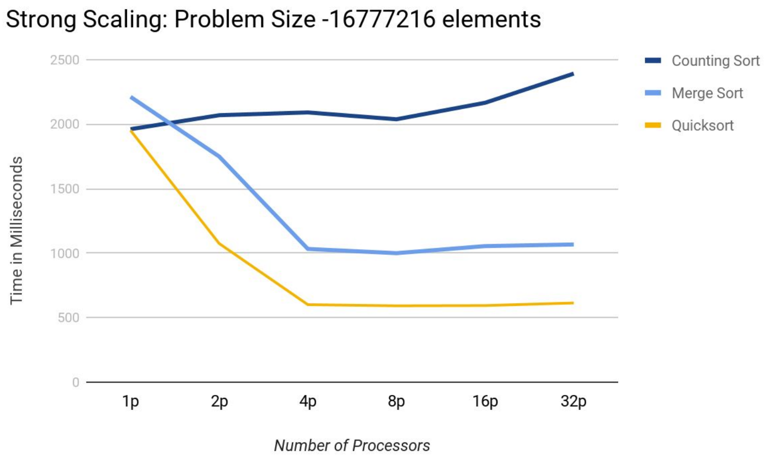
```
--------------------Serial---------------------
             Counting        Merge        Quick
    1024       0.1 ms       0.6 ms       0.3 ms
    4096       0.3 ms       0.6 ms       0.4 ms
   16384       1.2 ms       2.6 ms       1.1 ms
   65536       2.8 ms       6.3 ms       6.9 ms
  262144       7.4 ms      28.1 ms      25.9 ms
 1048576      58.2 ms     122.4 ms     103.7 ms
 4194304     362.5 ms     505.3 ms     453.1 ms
16777216    1961.3 ms    2212.0 ms    1952.9 ms
```

Table B: Parallel Results (4 Processors)

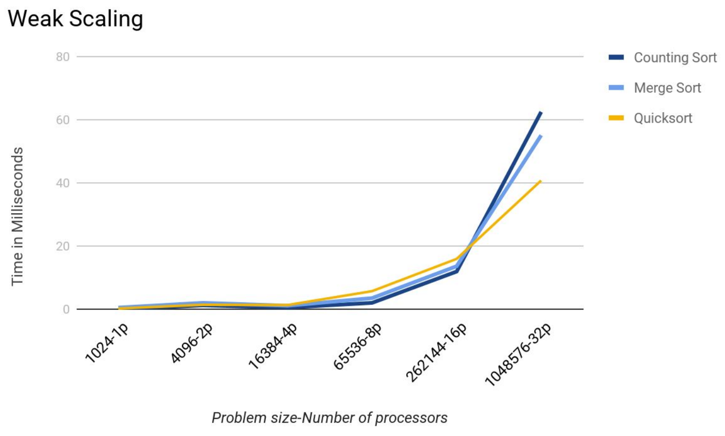


```
-------------------Parallel--------------------
             Counting        Merge        Quick
    1024       1.2 ms       0.8 ms       0.3 ms
    4096       0.5 ms       0.6 ms       0.4 ms
   16384       0.6 ms       1.2 ms       1.4 ms
   65536       1.7 ms       7.4 ms       6.1 ms
  262144       7.3 ms      12.9 ms      16.2 ms
 1048576      26.8 ms      51.7 ms      46.2 ms
 4194304     379.1 ms     253.4 ms     136.3 ms
16777216    2091.0 ms    1032.9 ms     601.8 ms
```

Graph A: Strong Scaling



Graph B: Weak Scaling



5 Counting Sort

Our serial Counting Sort utilizes four for loops and has a time complexity of O(n+k) where n is the number of elements in the array and k is the range of the data. The first for loop is used to find the maximum value in the array. The next loop counts the number of times each value is repeated and stores the number of instances of the repeated numbers in an array called Count. The index of the Count array corresponds to the number whose count it holds. Next, the count array is updated so that each intex stores the sum of the counts of the previous indexes. Thus the integers in the count array reflect the positions in which the numbers belong in the sorted array. Then the sorted array is created and the integers are placed in their corresponding positions and the count of each instance decreases by one. Finally you have your sorted array.

Much like the serial Counting Sort described above, our parallel implementation of Counting Sort also has four for loops. The sorting is actually the same. However instead of sorting the entire array, each thread is in charge of calling the sort method on a subset of the array. This method performs the same tasks as serial Counting Sort on the subset of the array, and returns the ordered subset to the thread.

The results of our Counting Sort comparison using four processors can be found above, in the Data section of our paper. We found that overall, the Parallel Counting Sort seriously underperformed when compared to the Serial Counting Sort regardless of the problem size. In fact, as the array size and processor count increases the Parallel time gets significantly worse due to message passing necessary to combine the subsets of the array. Akhriev and Pasetto further analyze this in their paper entitled “A Comparative Study of Parallel Sort Algorithms” in which they conclude that sorting data sets that do not fit in cache is difficult because of the memory transactions which must take place between the cores in order to bring the sorted portions of the array together [1]. Further testing of our parallel Counting Sort Algorithm showed that even as you increase the processor count, the results of the parallel implementation of Counting Sort do not undergo any significant improvement. In fact, we doubled the processor count until it reached thirty-two and in every instance the parallel implementation of Counting Sort took more than the serial Counting Sort. It was not surprising to find out that at thirty-two processes the results actually seem to get worse. This is due to the fact that we only have twenty cores at our disposal. This will come up with the following two algorithms as well. Table B implies that for the 3rd, 4th, and 6th sample size tests, the parallel merge sort is faster than the serial. However, we considered these to be outliers because none of our other tests which utilized four cores took less time than the serial Counting Sort. The apparent speedup may have been due to the randomized dataset already being partially sorted by chance. Charts A and B display that Counting sort is neither strongly or weakly scalable.

6 Merge Sort

Merge sort, the popular divide and conquer algorithm devised by John von Neumann, has a time complexity of O(nlog(n)) [4]. Akhriev and Pasetto assert that “Mergesort shortcomings comprise: requiring a separate output array; the partition procedure is based on an unbounded sorted container (e.g. priority queue), which should be carefully implemented to avoid memory fragmentation; and finally it is algorithmically and technically complex” [1]. The algorithm splits an array in half and then calls Merge Sort on those two halves. Merge Sort is recursive in that it calls itself on the portions of the array, halving it each time until there is one element in each. Once the array is divided into its smallest parts the merging begins. It is in the merge step that the elements of the portions of the array are compared so that they can be placed in numerical order.

The process of dividing the array several times is inherently parallel. Therefore, devising a parallel Merge Sort algorithm was quite intuitive. Each thread gets its own piece of the array and then performs a normal merge sort on these pieces and at the end the threads join their own sorted portions together. It is a perfect example of data parallelism. It may help to think of this algorithm as a tree in which each leaf node contains the smallest portion of the list and these are sorted up the tree in parallel [7]. This may not seem much different than the serial version but the test results show that at just four processors the speed is approximately doubled. This is due to the fact that we are performing the splits simultaneously across all of the processors.

The results of our Merge Sort comparison using four processors can be found in Table B pictured above. Further analysis showed that as you double the processor count the speed increases until you reach eight threads as pictured in Graph A. This was surprising to us because we thought that the increase in speed would stop at sixteen due to the fact that we only have twenty cores at our disposal as stated previously. The cause of the lack of consistent speed-up was the memory overhead due to the amount of message passing needed by this algorithm.

While our parallel Merge Sort provided a significant speed up as the size of the unordered array increased, it is important to acknowledge Rolfe's conclusions in his paper entitled "A Specimen of Parallel Programming: Parallel Merge Sort Implementation". He states, "If, however, there is significant message passing, the improvements promised by parallel processing can be greatly diminished by a mismatch between processing speed on each computer and the communications time for messages exchanged between them" [7]. This means that while parallel Merge Sort appears to be more efficient than its serial counterpart in our tests, it may actually prove to be less efficient on other machines if there is an excessive amount of communication between processors and the speeds of these processors do not match up.

7 Quicksort

Quicksort is another popular divide and conquer algorithm. The best case time complexity is O(nlogn) whereas, the worst case is O(n^2). Quicksort defines an element as *pivot* which divides the array into two parts such that elements in the left half are smaller than *pivot* while elements in the right half are larger than *pivot* (our *pivot* is always the last element in the array or portion of the array). Then we use recursion to perform the following steps: 1-*pivot* is brought to its correct position, 2-Quicksort the left part, 3-Quicksort the right part.The test condition for Quick Sorting the elements will be a comparison of the element in question to *pivot*.

Our parallel implementation of Quicksort clones the original array, and uses the Java utility ExecutorService to divide up the tasks amongst the threads. It also uses the Java utility Futures to obtain the portions of the array sorted using recursion. These are useful when doing things concurrently as in parallel programming.

Parallel Quicksort performed the best out of all three of our parallel sorting algorithms. At four processors it was approximately three times faster than Counting Sort, and nearly two times faster than Merge Sort. The strong scaling test demonstrated that like Merge Sort, Quicksort plateaued at eight processors due to too much passing between processors. When using both weak and strong scaling, Quicksort appears to be the best parallel sorting algorithm that we implemented.

8 Conclusion and Further Insights

As we were researching parallel implementations of serial algorithms we stumbled upon a paper about energy consumption. We may have made parallel versions of Merge Sort and Quicksort which appear to be more efficient than their serial versions due to the decrease in sorting time. However, if they consume a significantly larger amount of energy than their serial counterparts, then it may not be worth parallelizing them at all. We may utilize the conclusions of this paper to further our own research. How can we weigh speedup against energy consumption? When does the speedup become negligible compared to the excess amount of energy being consumed? Since we are unable to measure the power consumption of our algorithms these questions may be answered with further research and more testing.

In the end we determined that, despite the overhead, both Merge Sort and Quicksort are worth parallelizing if you are sorting large amounts of data. While Counting Sort, at least our implementation of it, would not be worth parallelizing because no matter how many processors you have or how large the data set is, the serial Counting Sort will always outperform the parallel Counting Sort. Strong Scaling depicted in Graph A gave a good visual representation of how Merge Sort and Quicksort outperformed Counting sort and plateaued at eight processes due to the increased message passing between processors.

Acknowledgments

The authors would like to thank Dr.James Teresco for providing inspiration, advice and direction in writing this paper.

References

[1] Akhriev, Albert. Pasetto, Davide. "A Comparative Study of Parallel Sort Algorithms" *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion.* 2011.

[2] Borodin, A ,Hopcroft, J.E. "Routing, merging, and sorting on parallel models of computation." *Journal of Computer and System Sciences,* vol 30, pp. 130- 145 1985.

[3] "Energy Consumption Analysis of Parallel Sorting Algorithms Running on Multicore Systems." *2012 International Green Computing Conference (IGCC), Green Computing Conference (IGCC), 2012 International*, 2012, p. 1. EBSCO*host*, doi:10.1109/IGCC.2012.6322290.

[4] Knuth, Donald E. *The Art of Computer Programming*. Vol 3. Chapter 5: Sorting

[5] Libeskind-Hadas, Ran. "Sorting in Parallel." *The American Mathematical Monthly*, vol. 105, no. 3, 1998, pp. 238–245. *JSTOR*, JSTOR, www.jstor.org/stable/2589078.

[6] Pacheco, Peter S. *An introduction to parallel programming*. Elsevier Science & Technology, 2013.

[7] Rolfe, Timothy J. "A Specimen of Parallel Programming: Parallel Merge Sort Implementation"