CMSC 180 Final Project

Comparative Analysis of Serial and Parallel Sorting Algorithms

Jose Enrique R. Lopez

June 12, 2022

1 Introduction

Sorting is a solved problem. In 1945, John von Neumann discovered merge sort [1], a divide-and-conquer algorithm that repeatedly subdivides an array, sorts the resulting subarrays, and consolidates them into the final sorted array. Merge sort achieved a worst-case running time of $O(nlogn)$, bringing down the run time of comparison-based sorting to its worst-case lower bound $\Omega(nlogn)$. However, sorting remains an active area of research [2][3][4]. One reason that sorting algorithms as a research topic persists is that certain use cases make certain algorithms more advantageous over others, which also calls forth the optimization of existing algorithms and invention of new ones. For example, the elementary algorithms bubble sort and insertion sort which have a quadratic worst-case running time run in $O(n)$ when applied on nearly sorted arrays [5]. On the other hand, the default out-of-place implementation of merge sort, while possessing a linearithmic run time, has a space complexity of $O(n)$ since it has to generate subarrays whose lengths total to n at each pass.

Parallelization is an especially important use case to consider due to the popularity of multi-core systems [3]. In general, the use of threads may make sorting algorithms more efficient as the runtime of the serial implementation is split among multiple threads. Moreover, the improved runtime brought about by threads may offset the efficiency-related disadvantages of elementary algorithms and make them competitive in their respective use cases.

2 Objectives

The purpose of this study is to compare the performance of serial and parallel sorting algorithms. Specifically, it sought to:

1. implement various serial and parallel sorting algorithms;
2. compare and contrast the performance of serial and parallel implementations;
3. recommend the most efficient sorting algorithm for general-purpose sorting.

3 Methodology

3.1 Implementation

Four sorting algorithms were implemented: merge sort, shell sort, bubble sort, and bucket sort to sort a random array of size N whose elements range from -N/2 to N/2. Here are the details of the subroutines implemented:

1. mergeSort(mode, index): This function accesses a global array and applies a bottom-up, iterative version of merge sort to it. In this algorithm, the array is first subdivided into subarrays of length 2 which are individually sorted, and are merged into subarrays of length 4. The larger subarrays are then repeatedly sorted and merged until the final sorted array is reached. In the serial version, mergeSort(SERIAL, 0) is called which performs the merging of the subarrays sequentially.

2. parallelMergeSort(k=2): Like its serial counterpart, this function accesses a global array but applies the merge sort function in parallel. Multiple calls to mergeSort(PARALLEL, i) are made by a given thread where i is the ith subarray with an offset of N//2. In my implementation I used only k=2 threads so that the left and right half are first sorted before applying the final merge sort.

3. shellSort(): This function accesses a global array and applies shell sort to it. In this algorithm, subarrays whose elements are gapped by k=N//2 slots are first sorted via insertion sort. At each pass, k is reduced by a factor of 2 until the gap can no longer be reduced and the array is sorted.

4. parallelShellSort(): This function parallelizes shellSort() by subdividing the set of subarrays at each pass equally between two threads. In effect, each thread is performing insertion sorts on one half of the set of subarrays at each pass.

5. void serialBubbleSort(): This function sorts a global array by repeatedly placing higher elements at the end of the array through consecutive swaps. This mechanism is implemented through a double loop where the primary loop considers each element and the secondary loop compares the element in question with every other element further down the list. The primary loop terminates when after one pass, no swap has been made, indicating that the list has already been sorted.

6. void parallelBubbleSort(): This function parallelizes serialBubbleSort() by dividing the array into two halves, each assigned to a separate thread which will perform bubble sort on that respective half. After the parallel execution, a final bubble sort pass is performed on the entire array. By first performing the parallel sorts of each half, it is expected that fewer swaps will be made on the last pass, decreasing the runtime.

7. void serialBucketSort(): This function creates square root of N buckets where each bucket i may contain values only from the ith percentile of the range of values between -N/2 to +N/2. Upon creation of buckets, each element in the global array is dumped in its corresponding bucket through a linked list. Each bucket/linked list is then sorted via bubble sort. The final sorted array is produced by iterating over all values in the buckets sequentially, placing them in the global array.

8. void parallelBucketSort(): This function parallelizes serialBubbleSort by dividing the buckets equally among two threads. In effect, each thread is sorting one-half of the total number of buckets with the expectation of bringing down the runtime by at most a factor of 2.

Functions 1-4 were implemented for Sam Gross's version of Python [6]. One relevant feature of this version is the absence of the Global Interpreter Lock (GIL). The GIL, featured in the default CPython implementation, is a mutex or lock which allows Python threads to be executed only one at a time [7]. By opting for Gross's implementation which does away with the GIL, we are able to use the threading library of Python and still enable true parallelism.

Meanwhile, functions 5-8 where implemented in C for a Linux environment. Threads were created through POSIX Threads.

Upon implementation, the functions' running times were recorded and graphed against each other.

For reference, here is the computing environment I had used.

- Processor: AMD A6-9220 RADEON R4, 5 COMPUTE CORES 2C+3G
- Clock Speed: 2.50 GHzRAM: 8.00 GB
- L1 Cache: 256 KB

3.2 Evaluation

Upon gathering the empirical running times, we use several metrics to evaluate the performance of the implementations, namely:

- Time complexity, the Big-Oh/worst-case running time of the algorithm expressed as the number of steps it takes to sort N elements. For our purposes, we further distinguish between the following metrics:
  - Serial runtime, the theoretical runtime for serial algorithms using the input size as the independent variable and the number of steps as the dependent variable
  - Parallel runtime, similar to serial runtime, but the input size and the number of processors constitute two independent variables.
- Space complexity, the Big-Oh/worst-case amount of memory used by the algorithm to sort N elements. For our purposes, we further distinguish between the following metrics:
  - Serial space complexity, the theoretical amount of space used by the algorithm, with the input size as the independent variable and the blocks of memory created/occupied as the dependent variable
  - Parallel space complexity, similar to serial space complexity, but the input size and the number of processors constitute two independent variables.
- Speedup, the ratio between the serial and parallel execution times,

$$\frac{T_s}{T_p}$$

- Efficiency, the ratio between the speedup and the number of processors,

$$\frac{Speedup}{p}$$

4 Discussion

Let us first consider the theoretical algorithmic complexities of the sorting algorithms as these figures will serve as our baseline for our empirical results. Here are some explanations of their theoretical runtimes of the serial algorithms:

- Merge sort has O(n log n) time complexity since there are log n number of levels from n=2 subarrays to n=N/2 subarrays, and per level we linearly sort the arrays, resulting to a linearithmic running time.
- Shell sort has O(n^2) time complexity since, in the worst case, the subarrays we may get are largely in descending order, hence maximizing the quadratic time of the individual insertion sorts.
- Similarly, bubble sort has O(n^2) time complexity since, in the worst case, our array is in descending order, and so each element must inevitably be compared with every other element to arrive at a sorted list, maximizing the runtimes of the nested loop.
- Bucket sort has O(n^2) time complexity since, in the worst case, the buckets would each have a descending order of elements, and so each value in the bucket will be compared with every other element in the bucket. The quadratic running times of bubble sorts are then multiplied by the number of buckets, resulting to O(n^2)

Meanwhile, as per our implementation, the running time of the parallel algorithms are given by the serial running time divided by k number of threads = 2 as the threads divide the effort of sorting in half for all the algorithms. For merge sort and bubble sort, we also add the final serial sort after the two threads execute. Here is a summary of all the theoretical runtimes:

Table 1.Theoretical Running Times of Various Sorting Algorithms

Algorithm	Theoretical Runtime
Merge Sort	**(n log n)**
Parallel Merge Sort	O((n log n)/2 threads) + O(n log n) = **O(n log n)**
Shell Sort	**O(n^2)**
Parallel Shell Sort	O(n^2/2 threads) = **O(n^2)**
Bubble Sort	**O(n^2)**
Parallel Bubble Sort	O(n^2/2 threads) + O(n^2) = **O(n^2)**
Bucket Sort	**O(n^2)**
Parallel Bucket Sort	O(n^2/2 threads) = **O(n^2)**

Meanwhile, here are some details regarding the space complexities of the algorithms:

- Merge sort has O(n) space complexity since, at each level, it has to produce subarrays whose total lengths sum up to N number of elements.
- Similarly, shell sort has O(n) space complexity. In my implementation, I had to record the indices of the subarrays at each pass in separate Python lists, inevitably occupying memory blocks to record all 0 to N-1 indices.
- On the other hand, bubble sort has O(1) space complexity since the implementation does not issue any additional arrays and merely swaps the elements in place.

- Finally, bucket sort has O(n) space complexity since the lengths of all linked lists/buckets would inevitable total the number of elements.

Meanwhile, the parallel implementations have space complexities identical to their serial counterparts. This is because each of the space complexities of the two separate threads are totalled to produce the final space complexity which is just equal to that of the serial implementation. For example, in merge sort, the space complexity of 1 thread is just N/2. But ultimately, both of the threads in total use N memory blocks, so the space complexity is still O(N). A summary of the space complexities is shown below:

Table 2.Theoretical Space Complexities of Various Sorting Algorithms

Algorithm	Theoretical Space Complexity
Merge Sort	O(n)
Parallel Merge Sort	O(n/2) + O(n/2) = O(n)
Shell Sort	O(n)
Parallel Shell Sort	O(n/2) + O(n/2) = O(n)
Bubble Sort	O(1)
Parallel Bubble Sort	O(1)
Bucket Sort	O(n)
Parallel Bucket Sort	O(n/2) + O(n/2) = O(n)

From the algorithmic complexities presented, it may be expected that the parallel implementations would perform faster by a factor of 2. If we compare the theoretical running times with their corresponding execution times (see Table 3), our hypothesis is only partly confirmed. For example, here are the empirical running times of the algorithms implemented in Python.

Table 3. Execution Times of Serial and Parallel Merge Sort and Shell Sort

Array Size	Execution Time (s)			
	Merge Sort		Shell Sort	
	Serial	Parallel	Serial	Parallel
50000	2.33	1.61	13.36	22.11
60000	4.16	2.03	3.87	6.54
70000	4.84	3.24	9.01	16.12
80000	5.17	3.65	6.44	11.40
90000	5.73	4.30	14.51	26.78
100000	6.38	8.46	16.25	25.01
200000	25.46	8.64	30.43	49.61
300000	98.27	80.56	69.86	111.19
400000	175.77	119.49	160.79	250.02
500000	280.62	188.40	402.64	693.66

As seen in Table 3, parallel merge sort performed better than serial merge sort for a sufficiently large N. Meanwhile, parallel shell sort performed worse than serial shell sort. A plausible reason why parallel merge sort performed better than its serial counterpart is that the threads shared the work of sorting the halves of the list, cutting down the runtime by at most a factor of 2. Yet, in the

figures, we see that this expected boost is not maximized likely due to the additional overhead of managing threads and disparities in the operating system's delegation of tasks. On the other hand, parallel shell sort had an apparent slowdown. This may be because each of the threads perform insertion sort on their subarrays which has a worst-case quadratic running time. Also, the work of starting and joining threads also contributed some overhead. These two factors, hence, down any apparent gains in speedup.

As the trendlines of the four algorithms show (see Figure 1), parallel merge sort is a clear winner.

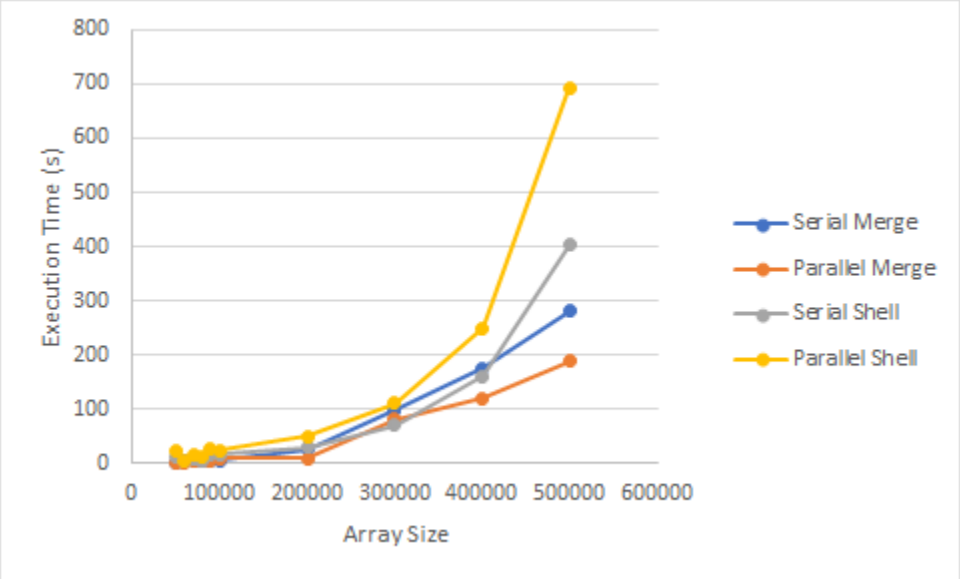

Figure 1: Trends in Running Time of Merge Sort and Shell Sort

Meanwhile, here are the execution times of the sorting algorithms implemented in C:

Table 4.Execution Times of Serial and Parallel Bubble Sort and Shell Sort

Array Size	Execution Time (s)			
	Bubble Sort		Bucket Sort	
	Serial	Parallel	Serial	Parallel
1000	0.57612	0.559864	7.262921	6.614251
2000	2.48219	2.467292	11.933451	17.02433
3000	6.18267	5.712658	16.326485	16.62865
4000	10.805	10.50512	22.297949	22.17428
5000	17.6144	16.25309	29.774258	28.97841
6000	25.0707	24.09342	40.938261	36.83896
7000	34.696	32.43832	45.288673	45.34233
8000	45.1135	43.35693	109.71722	104.3271
9000	57.7595	54.31708	192.35638	184.7697
10000	82.5566	66.93355	1212.4676	1194.404

As Table 4 shows, both parallel implementations performed slightly better than their serial counterparts. On the one hand, parallel bubble sort ran faster than its serial counterpart because

sorting each half of the list concurrently led to fewer swaps in the last bubble sort, cutting down on running time. On the other hand, parallel bucket sort also achieved speedup because the effort of sorting all the buckets were delegated amongst the threads, ensuring that this procedure would be faster by at most a factor of 2.

While the figures show slight improvements due to parallelization, the trendlines of the algorithms demonstrate almost no difference between the serial and parallel algorithms, as shown below:

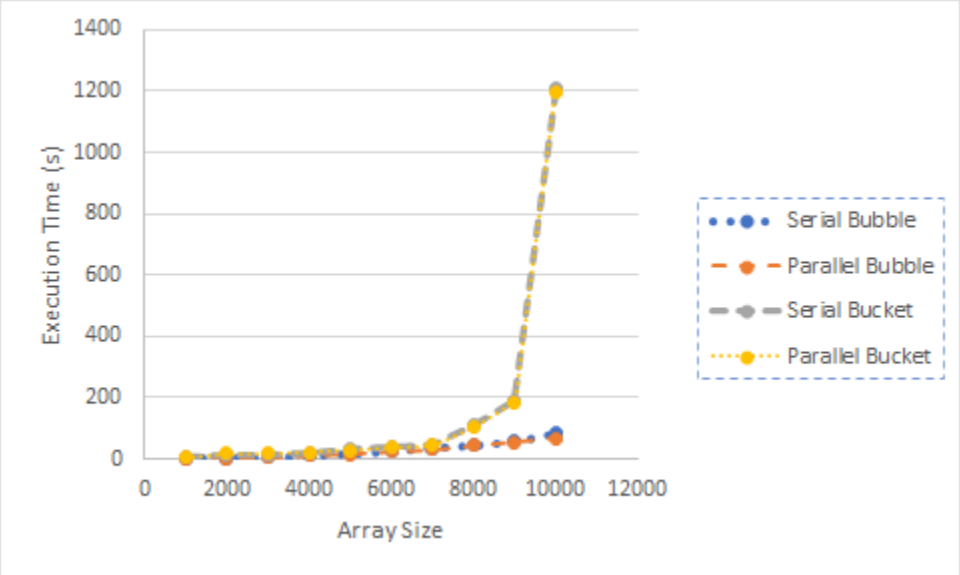

Figure 2: Trends in Running Time of Bubble Sort and Bucket Sort

The minimal improvements shown in Figure 2 is due to a variety of external factors, not the least of which is disparities in operating system scheduling. In this regard, the parallel algorithms may, in the future, benefit from application of core-affinity so that the tasks of sorting are uniformly delegated amongst threads.

To evaluate the performance of the algorithms, we compile their corresponding speedup and efficiency metrics, as detailed in the methodology section. The final figures, using the times with the largest array sizes and p processors = 2, are shown below:

Table 5.Speedup and Efficiency of the Parallelized Sorting Algorithms

Parallel Algorithm	Speedup	Efficiency
Merge Sort	1.489482	0.744741
Shell Sort	0.580455	0.290228
Bubble Sort	1.233411	0.616705
Bucket Sort	1.015123	0.507562

Merge sort emerges as the top-performing parallel algorithm. It has a speedup of approximately 1.5 which means that it is at most 50% times faster than the serial algorithm. It also logged an efficiency of 0.74 which implies that each processor devoted 74% of its time to sorting the array. Meanwhile, parallel shell sort experienced a slowdown, achieving a low efficiency of 0.29,

revealing that each processor was largely devoted to doing other tasks (e.g., managing threads, subdividing the array, and storing subarray indices) aside from the primary task of sorting the array. Meanwhile, bubble and bucket sort achieved only slight improvements in speedup and efficiency, possibly weighed down by the quadratic runtime of the serial algorithm executed by each thread.

Clearly, merge sort is the best algorithm to use for general-purpose sorting, and parallel merge sort should be used if the computing environment allows. The disadvantage of our merge sort implementation, however, is that it has a linear space complexity and is inefficient in memory use. To overcome this, we may instead implement an in-place version of merge sort which just involves using a temporary variable to swap elements in subarrays being compared instead of generating a brand new merged array.

On the other hand, the advantage of bucket sort is that it behaves optimally under uniformly distributed data since it produces balanced buckets [8], as has seen in our experimental results. Its disadvantage, however, is similar to out-of-place merge sort where we inevitably have to produce lists for each bucket, producing a linear space complexity.

Compared to merge sort and bucket sort, bubble sort behaves less optimally when sorting randomly generated arrays or inversely sorted arrays. As mentioned, it only has an advantage when the array is already nearly sorted. As such, future research must look into the effects of parallelizing elementary sorts under different array conditions.

Lastly, shell sort is generally advantageous because it nearly sorts the array by swapping elements far apart, then performs a single insertion sort [9]. The prior swaps in the subroutine ensures that the insertion sort would require lesser swaps, generating a faster overall runtime. However, our parallel implementation weighed down any apparent speedup possibly because each thread performs insertion sort for log n levels or passes. Since insertion sort is $O(n^2)$, the runtime of each thread is $O((n/2)^2 logn)$ Since there are two threads, the total runtime for the parallel implementation is $O((n/2)^2 * 2logn)$. Plus, the recording of subarray indices may have contributed to additional overhead at each level. This finding may call for a simpler parallel implementation whereby the threads work only on the first level of the array as in our implementation of bubble sort, as opposed to assigning threads at each level or each pass of the subroutine.

Given our findings, it is indeed worth the time to parallelize sorting algorithms. Our implementation of merge sort proves that speedups may be gained from parallelization. The reason why merge sort achieved immediate benefits from multithreading is that its recursive divide-and-conquer structure lent itself well to partitioning of work amongs threads. For the other algorithms, their implementations may require greater consideration and optimization of the computing environment, and accounting for which levels of the subroutine should be multithreaded.

5 Conclusion

The theoretical and empirical running times of the sorting algorithms were only partly consistent with each other. Merge sort, whose parallelization would theoretically divide runtime by a factor of 2, indeed gained speedups in its empirical runtime. Bucket sort and bubble sort gained only minimal improvements in empirical running time in part due to imbalances in operating system scheduling. In contrast, shell sort experienced a slowdown since the work of managing threads at each level contributed to overhead in addition to the quadratic run time of insertion sort per thread.

While merge sort is put forward as the best general-purpose sorting algorithm, it is recommended that future endeavors focus on optimizing the performance of the sorting other algorithms. For example, one may apply the parallel and serial algorithms on different kinds of arrays (e.g., nearly sorted, descending order, perfectly sorted). Also, the use of core affinity to bypass operating system scheduling may maximally boost the effects of parallelization, and reflect greater speedups.

6 List of Collaborators

The laboratory and research work were conducted independently by the author.

References

[1] D. Knuth, *Sorting by Merging*, ser. The Art of Computer Programming. Addison-Wesley, 1988, pp. 158–168.

[2] T. Fagbola and S. Thakur, "Investigating the effect of implementation languages and large problem sizes on the tractability and efficiency of sorting algorithms," *International Journal of Engineering Research and Technology*, vol. 12, pp. 196–203, 2019. [Online]. Available: http://www.ripublication.com/irph/ijert19/ijertv12n2_08.pdf

[3] D. P. Singh, I. Joshi, and J. Choudhary, "Survey of gpu based sorting algorithms," *International Journal of Parallel Programming*, vol. 46, pp. 1017–1034, 04 2017.

[4] Y. Ben Jmaa, R. Ben Atitallah, D. Duvivier, and M. Ben Jemaa, "A comparative study of sorting algorithms with fpga acceleration by high level synthesis," *Computación y Sistemas*, vol. 23, 03 2019.

[5] O. Astrachan, "Bubble sort," *ACM SIGCSE Bulletin*, vol. 35, p. 1, 01 2003.

[6] S. Gross, "nogil," 2022. [Online]. Available: https://github.com/colesbury/nogil

[7] IronPython, "Glossary — ironpython 2.7.2b1 documentation," ironpython-test.readthedocs.io, 2013. [Online]. Available: https://ironpython-test.readthedocs.io/en/latest/glossary.html?msclkid=272af30fb01211eca2036546ea05a7e3

[8] X. Shen and C. Ding, "Adaptive data partition for sorting using probability distribution." [Online]. Available: https://www.cs.rochester.edu/~cding/Documents/Publications/icpp04.pdf

[9] G. Abecasis, "Shell sort biostatistics 615/815 lecture 7," 2006. [Online]. Available: http://csg.sph.umich.edu/abecasis/class/2006/615.07.pdf