A Comparative Study of Sorting Algorithms

D.R. Aremu
Department of Computer Science
Faculty of Communication and Information Sciences
University of Ilorin
Ilorin, Nigeria
draremu2006@gmail.com

O.O. Adesina, O.E. Makinde, O. Ajibola & O.O. Agbo-Ajala
Department of Physical Sciences
Faculty of Natural Sciences
Ajayi Crowther University
Oyo, Oyo State, Nigeria
opeyemi.adesina@gmail.com, oladayo.makinde@yahoo.com, ajalabosun@gmail.com

ABSTRACT
The study presents a comparative study of some sorting algorithm with the aim to come up with the most efficient sorting algorithm. The methodology used was to evaluate the performance of median, heap, and quick sort techniques using CPU time and memory space as performance index. This was achieved by reviewing literatures of relevant works. We also formulated architectural model which serves as guideline for implementing and evaluating the sorting techniques. The techniques were implemented with C-language; while the profile of each technique was obtained with G-profiler. The results obtained show that in majority of the cases considered, heap sort technique is faster and requires less space than median and quick sort algorithms in sorting data of any input data size. Similarly, results also show that the slowest technique of the three is median sort; while quick sort technique is faster and requires less memory than median sort, but slower and requires more memory than heap sort. The number of sorting algorithms considered in this study for complexity measurement is limited to bubble, insertion, and selection sorting. Future effort will investigate complexities of other sorting techniques in the literature based on CPU time and memory space. The goal for this will be to adopt the most efficient sorting technique in the development of job scheduler for grid computing community.

Keywords— Sorting algorithms, Comparative Study, Heap Sort, Quick Sort, Median Sort

1. INTRODUCTION
This paper presents a comparative study of sorting algorithms. Sorting [4], a mechanism that organizes elements of a list into a predefined order is important for various reasons. For instance, numerical computation with estimated values (e.g. floating point representation of real numbers) is concerned with accuracy. To control accuracy in computations of this kind, scientists often apply sorting to actualize the goal. In string processing, finding the longest common prefix in a set of string and the longest repeated substring in a given string is often based on sorting. To develop jobs schedules, so as to maximize customer satisfaction by minimizing the average completion time, often requires methods (such as longest-processing-time-first rule) based on sorting. In the literature, various implementation solutions of algorithm exist for sorting. However, an important question of which sorting algorithm is the most-efficient or least-complex is yet to be fully addressed. The goal of this paper is to present a comparative study of some sorting algorithm with the aim to come up with the most efficient sorting algorithm. The methodology used was to evaluate the performance of median, heap, and quick sort techniques using CPU time and memory space as performance index. This was achieved by reviewing literatures of relevant works. We also formulated architectural model which serves as guideline for implementing and evaluating the sorting techniques. The techniques were implemented with C-language; while the profile of each technique was obtained with G-profiler. The results obtained show that in majority of the cases considered, heap sort technique is faster and requires less space than median and quick sort algorithms in sorting data of any input data size.
2. RELATED WORKS

In the literature, there are various researches on evaluating performance or complexity of sorting algorithms. In [4], a comparative study of some parallel sort algorithms such as: map sort, merge sort, Tsigas-Zhangs parallel quicksort, alternative quick sort, and STL sort was carried out. The goal was to introduce merge sort algorithm and its ability to sort n-sized array of elements in $O(n \log n)$ time complexity. The approach adopted in developing this algorithm was divide-and-conquer method. The empirical and theoretical analyzes of merge-sort algorithm was presented in addition to its pseudo codes. Merge sort algorithm was compared with its counterparts: bubble, insertion, and selection. It was recorded the other algorithms has quadratic $O(n^2)$ time. Results presented involves merge sort against insertion sort. This shows that merge sort algorithm is significantly faster than insertion sort algorithm for great size of array. Merge sort is 24 to 241 times faster than insertion sort using n-values of 10,000 and 60,000 respectively. Also, results show that the difference between merge and insertion sorts is statistically significant with more than 90 percent confidence.

Similarly, [5] gave a practical performance comparison of sorting algorithms like: odd-even transposition sort, parallel rank sort, and parallel merge sort. In [6], robustness was studied as a function of complexity of some sorting techniques. Finally, [4] evaluated merge sort empirically and theoretically to sort N-sized dataset in $O(n \log n)$ time as the most-efficient among bubble, insertion, and selection sort with quadratic time complexity.

According to [6], a great number of research works have addressed issues related to dedicated and parallel machines [12]; but only little research has been carried out on performance evaluations of parallel sorting algorithms on clustered station. Hence, the goal of [6] is to compare some parallel sorting algorithms with overall execution time as performance parameter. Algorithms considered include: odd-even transposition sort, parallel rank sort, and parallel merge sort. These algorithms were evaluated theoretically and empirically. In theory, the odd-even transposition has a complexity of $O(bn^2)$; such that $b = 1/2^n$. This implies that the time will be reduced by ‘b’. Similarly, in theory parallel rank sort has a complexity of $O(cn^2)$; $c = 1/p$. The theoretical complexity of parallel merge sort is $O(n/p \log n/p)$ [13]. ‘P’ is number of processes. Empirical results have shown that the fastest algorithm of three is the parallel merge technique. This is followed by the odd-even transposition algorithm; while the slowest is the parallel rank sorting algorithm.

The motivation of [7] was to research sorting algorithms for potential failing comparisons. It was expected that faulty comparisons tend to occur in sorting, due to random fluctuations in the evaluation functions that compares two elements. Hence, the aim of the work is research a method that is robust against faulty comparisons. The null hypothesis for the research is: a very efficient (i.e. low complexity order) sorting algorithm might be susceptible to errors from imprecise comparisons than the more efficient sorting algorithms which might implement a lot implicitly of redundant comparisons. The sorting algorithms considered include: bubble sort, merge sort, quick sort, heap sort and selection sort. The result of the research supports its null hypothesis. Bubble sort is the most robust sorting algorithms, followed by merge, quick, heap, and selection sorts. The work contributed to the state of the art by analyzing existing sorting algorithms based on robustness against imprecise and noisy comparisons.

In [14], performance evaluation of bubble, selection, and insertion sort techniques were conducted. The motivation of the work was based on the unprecedented growth of data in knowledge-based sectors and its requirements of mechanisms for analyzing these data which often demands sorting. Sorting techniques evaluated in this work were subjected to CPU time and memory space as performance index. Experimental results had shown that the most efficient of the techniques evaluated is the insertion sort technique in both cases of performance parameters. Similarly, it has been observed that selection sort is less efficient than insertion sort technique. The least efficient of the techniques is the bubble sort technique.

3. ARCHITECTURAL MODEL

The knowledge acquired from the literatures is being applied in designing the architectural model presented in Figure 1. It is divided into three basic components. These are: repository, shuffling module, and sorting module. This model was formulated to serve as benchmark of components to be developed when evaluating performance of sorting algorithms. The data designed for evaluating the algorithms is stored in the repository. This data may be ordered or unordered in nature.
However, since the aim of the work is to determine efficiency of sorting mechanisms in the solution space, it is important we realize a uniformly ruffled data. To realize this, a shuffling module (which randomizes positions of data) is introduced. The ruffled data which is a result of shuffling process is passed to the sorter. The sorter applies its logic to sort the input data. The architecture also shows the flow of control from a component to another. Control flow starts from the repository to shuffling module. Furthermore, the control flows from the shuffling module to the sorting module. The output obtained from sorting is passed into the repository for storage.

4. ALGORITHM DESIGN

This section describes and presents the basic algorithms which are the foundation of majority of the sorting algorithms. Algorithms under these categories include: shuffling and swapping algorithms.

4.1 Shuffling Algorithm

Figures 2 presents pseudo code of the shuffling algorithm. Its theoretical complexity is $O(n)$ and $O(1)$ in worst case and best case scenario respectively. We have assumed that swapping algorithm takes a constant time $O(1)$ to execute in theory.

```
shuffle( A, n)
1. i ← n – 1
2. while( i > 0 ) do
3. j ← rand() % (i + 1)
5. i ← i – 1
6. end-while
```

4.2 Sorting Algorithms

The sorting module component can be designed with any sorting technique in the solution space. To design this component, we will consider heap, quick, and median sort techniques. These algorithms will be discussed theoretically and with pseudo codes.

4.2.1 Heap Sort Algorithm

Figure 3 presents the pseudo code of make heap algorithm. This constructs the heap data to be sorted with the heap sort algorithm. The heap sort algorithm presented in figure 4 is applied to sort the heap constructed. According to [8], the heap sort algorithm has theoretical complexity of $O(n \log n)$ in its best, average and worst case scenario.

```
make-heap( A, n)
1. for i ← 1 to n – 1 do
2. val ← A[i]
3. s ← i
4. f ← (s - 1) / 2
5. while s > 0 AND A[f] < val do
7. s ← f
8. f ← (s - 1) / 2
9. end-while
10. A[s] ← val
11. end-for
```

Figure 3: Make Heap Algorithm
4.2.2 Quick Sort Algorithm

Figures 5, 6 and 7 collectively form quick sort algorithm. Figure 5 generates a pivot randomly for dividing data set into two distinct partitions. Figure 6 is the pseudo code of the algorithm for portioning a block of data set which is based on the pivot generated. Figure 7 presents quick sort algorithm. It is being developed with the divide-and-conquer approach. The method recursively calls itself for sorting once a partition has been created. According to [8], it has theoretical complexity of $O(n \log n)$ in its best and average cases and $O(n^2)$ in its worst case scenario.

\begin{verbatim}
heap-sort(A, n)
1. make-heap(A, n)
2. for i ← n - 1 to 1 do
3.   ivalue ← A[i]
4.   f ← 0
5.   if i == 1 then
6.     s ← -1
7.   else
8.     s ← 1
9.   end-if
11.   s ← 2
12.   while s >= 0 AND ivalue < A[s] do
14.     f ← s
15.     s ← s + f + 1
16.     if s + 1 <= i AND A[s] < A[s + 1] then
17.       s ← s + 1
18.     end-if
19.   end-while
20. end-if
21. end-for

pivot(left, right)
1. x ← right - left
2. y ← rand() % x
3. piv ← y - left
4. return piv

partition(A, left, right)
1. p ← pivot(left, right)
2. swap(A[p], A[right])
3. store ← left
4. for i ← left to right - 1 do
5.   if A[i] < A[right]
6.     swap(A[i], A[store])
7.     store ← store + 1
8. swap(A[store], A[right])
9. return store

quick-sort(A, left, right)
1. if left < right
2. p ← partition(A, left, right)
3. quick-sort(A, left, p)
4. quick-sort(A, p+1, right)
\end{verbatim}

4.2.3 Median Sort Algorithm

Figures 8, 9 and 10 collectively form median sort algorithm. Figure 8 generates a pivot by computing the average value of the left and right positions. This is used for dividing data set into two distinct partitions. Figure 9 is the pseudo code of the algorithm for portioning a block of data set which is based on the pivot generated. Figure 10 presents median sort algorithm. It is being developed with the divide-and-conquer approach. The method recursively calls itself for sorting once a partition has been created. According to [8], it has theoretical complexity of $O(n \log n)$ in its best and average cases and $O(n^2)$ in worst case scenario.
5. RESULTS AND DISCUSSION

This section presents the discussion of results obtained from the empirical evaluation of insertion, bubble, and selection sort algorithms. The efficiency of these algorithms was measured in CPU time. This was measured using the system clock on a machine with minimal background process running, with respect to the size of the input data. The algorithms were implemented in C-language. The tests were carried out using G-Profiler in the GCC suite on Linux Ubuntu 13.04. These were run on Dell Inspiron 6400 PC with the following specifications: Intel Dual Core CPU at 1.60 GHz and 1.00GB of RAM. Empirical results of bubble sort, insertion sort, and selection sort techniques using various data sizes and corresponding CPU time for the techniques is presented in Table 1. Similarly, table 2 presents the empirical results of the sorting techniques considered using various input data sizes and corresponding memory required for execution of the techniques.

<table>
<thead>
<tr>
<th>Data Size ($10^5$)</th>
<th>Heap Sort (s)</th>
<th>Quick Sort (s)</th>
<th>Median Sort (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>30</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>50</td>
<td>0.02</td>
<td>0.02</td>
<td>0.06</td>
</tr>
<tr>
<td>100</td>
<td>0.05</td>
<td>0.06</td>
<td>0.15</td>
</tr>
<tr>
<td>200</td>
<td>0.09</td>
<td>0.08</td>
<td>0.21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Size ($10^5$)</th>
<th>Heap Sort (Bytes)</th>
<th>Quick Sort (Bytes)</th>
<th>Median Sort (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.00</td>
<td>4.00</td>
<td>4.00</td>
</tr>
<tr>
<td>30</td>
<td>4.00</td>
<td>4.00</td>
<td>8.00</td>
</tr>
<tr>
<td>50</td>
<td>8.00</td>
<td>8.00</td>
<td>24.00</td>
</tr>
<tr>
<td>100</td>
<td>20.00</td>
<td>24.00</td>
<td>60.02</td>
</tr>
<tr>
<td>200</td>
<td>36.00</td>
<td>32.00</td>
<td>83.93</td>
</tr>
</tbody>
</table>

Also, graph 1 illustrates results of an experiment with complexities of the sorting techniques based on input data size against CPU time. In similar manner, M-Sort, H-Sort, and Q-Sort imply median, heap and quick sort techniques. We observed that for median and heap sorting techniques considered in this work, the memory space is directly proportional to the input data size, while the CPU time required for quick sort algorithm is the same for data sizes of 10,000 and 30,000. But for other data sizes, the memory space required is proportional to the input data size. Also we observed that the CPU time required by median sort is 2.5 times higher than its counterparts. Heap and quick sort algorithms compete in almost all the cases of input data size.
Also, graph 2 illustrates results of an experiment with complexities of the sorting techniques based on input data size against memory space. In similar manner, M-Sort, H-Sort, and Q-Sort imply median, heap and quick sort techniques. We observed that for median and heap sorting techniques considered in this work, the memory space is directly proportional to the input data size, while the memory space required for quick sort algorithm is the same for data sizes of 10,000 and 30,000. But for other data sizes, the memory space required is proportional to the input data size. Also we observed that the memory space required by median sort is 2.5 times larger than its counterparts. Heap and quick sort algorithms compete in almost all the cases of input data size.
6. CONCLUSION
In conclusion, we have evaluated the performance of median, heap, and quick sort techniques using CPU time and memory space as performance index. This was achieved by reviewing literatures of relevant works. We also formulated architectural model which serves as guideline for implementing and evaluating the sorting techniques. The techniques were implemented with C-language; while the profile of each technique was obtained with G-profiler [15].

Empirical results were tabulated and graphically presented. The results obtained show that in majority of the cases considered, heap sort technique is faster and requires less space than median and quick sort algorithms in sorting data of any input data size. Similarly, results also show that the slowest technique of the three is median sort; while quick sort technique is faster and requires less memory than median sort, but slower and requires more memory than heap sort. We infer from this study that empirical complexity can be determined in theory.

7. FUTURE THOUGHTS
We realized that in the solution space, numerous implementation solutions of sorting techniques exist. However, time constraint has limited this study to bubble, insertion, and selection sort techniques. We intend to investigate complexities of other sorting techniques in the literature based on CPU time and memory space. Also, we intend to adopt the most efficient sorting technique in the development of job scheduler for grid computing community.
REFERENCES