Our contributions include performance evaluation of bubble, insertion, and selection sort techniques [8] using CPU time and space as performance index. In addition to this, we will present these techniques with pseudo codes. Also, we will analyze the techniques empirically with G-Profiler [9].

The rest of this paper is organized as follows. The next section discusses literatures of related works. In section 3, we presented an architectural model designed in this work. Section 4 discusses results obtained from the evaluation of sorting techniques considered. In section 5, we presented our future thoughts.

2. RELATED WORKS

Sorting has been regarded as an algorithm with a great deal of research attention in computer science [3-6]. Hence, a great number of related works exist. However, due to inadequate access to these works, a few numbers of them will be examined in this work. These will be examined based on their goals, contributions, approaches, results and conclusions.

In [4], 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.

The goal of [4] was to provide a qualitative and quantitative analysis of the performance of parallel sorting algorithms on modern multi-core hardware. In the work, Mapsort [10],

Mergesort [11], Tsigas-Zhang's Parallel Quicksort [12], Alternative Quicksort [5], and STL sort [5] were the parallel algorithm studied. The experiments involved two machines (Core i7 architecture). One of the machines is Nehalem whose configurations are: Intel Xeon 5550, 2.67 GHz, 4 cores/8 threads, 6 Gb of memory, 3-channel memory controller, OS Linux Fedora 13 64-bit. The other machine is Westmere whose configurations are: Intel Xeon 5670, 2.93 GHz, 6 cores/12 threads, two sockets on board (24 threads in total), 24 Gb of memory, 3-channel memory controller, OS Linux RedHat 5.4 64-bit. Performance index include: sorting throughput; scalability, influence of CPU affinity; and micro-architecture analysis. The results of this study show that merge and map sort algorithms are memory intensive but faster compared with the quick sort methods which are slower than their counterparts.

According to [6], a great number of research works have addressed issues related to dedicated and parallel machines [13]; 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 = 1/2p^2. 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.

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.

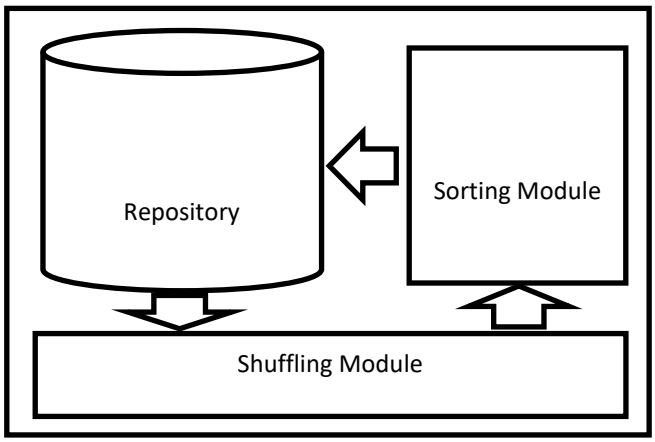

Figure 1: Architectural Model

3.1 Shuffling Algorithm

This section describes and presents of basic algorithms which are the foundation of majority of the sorting algorithms. Algorithms under these categories include: shuffling and swapping algorithms. Figures 1 presents pseudo code of the shuffling algorithm. 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)
4.        swap( A[i] , A[j] )
5.        i ← i – 1
6.    end-while
```

Figure 2: Shuffling Algorithim

3.2 Sorting Algorithms

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

3.2.1 Bubble Sort Algorithm

Figure 3 presents the pseudo code of bubble sort algorithm. According to [4,14], it has theoretical complexity of $\boldsymbol{O(n^2)}$ in its best, average and worst case scenario.

```
bubble-sort( A , n)
1.    for i ← 0 to n – 1 do
2.        for j ← 0 to n – i
3.            if A[j] < A[j+1] then
4.                swap( A[j] , A[j+1] )
5.        end-for
6.    end-for
```

Figure 3: Bubble Sort Algorithim

3.2.2 Selection Sort Algorithm

Figure 4 presents the pseudo code of selection sort algorithm. According to [15], it has theoretical complexity of $\boldsymbol{O(n^2)}$in its best, average and worst case scenario.

```
selection-sort( A, n )
1.    i ← 0
2.    while i < n do
3.        j ← i+1
4.        while j < n do
5.            if A[i] > A[j] then
6.                swap(A[i], A[j])
```

Figure 4: Selection Sort Algorithim

3.2.3 Insertion Sort Algorithm

Figure 5 presents the pseudo code of insert algorithm. The algorithm is a sub-function to the insertion sort algorithm and has a theoretical complexity of $\boldsymbol{O(n)}$ in its worst case scenario. Also, figure 6 presents the pseudo code of insertion sort algorithm. This method calls the

function insert whose pseudo code is presented in figure 5. According to [8], insertion sort algorithm has theoretical complexity of $\boldsymbol{O(n)}$ in its best and $\boldsymbol{O(n^2)}$ in its average and worst case scenario.

```
insert( A, pos , value)
1.      n ← pos - 1
2.      while n > 0 AND A[n] > value do
3.              A[n+1] ← A[n]
4.              n ← n – 1
5.      end-while
6.      A[n+1] ← value
```

Figure 5: **Insert Algorithm**

```
insertion-sort( A, n )
1.      x ← pos - 1
2.      for i ← i to n do
3.              insert(A , i, A[i])
4.      end-for
```

Figure 6: **Insertion Sort Algorithm**

4. RESULTS AND DISCUSSION

In section, we will be discussing results obtained from the empirical evaluation of insertion, bubble, and selection sort algorithms. The efficiency of these algorithms will be measured in CPU time which is 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 1: Results of CPU Time vs. Data Size

Data Size (10^3)	Bubble Sort (s)	Insertion Sort (s)	Selection Sort (s)
10	0.88	0.24	0.74
30	8.89	2.18	6.74
50	24.05	6.15	22.35
100	99.71	24.66	87.71
200	392.62	117.24	306.35

Table 2: Results of Memory vs. Data Size

Data Size (10^3)	Bubble Sort (Bytes)	Insertion Sort (Bytes)	Selection Sort (Bytes)
10	352.35	95.92	296.30
30	3560.27	869.57	2662.72
50	9868.69	2500.00	9995.53
100	39352.74	10000.00	40000.00

200	393357.55	39989.77	202013.61

Figure 7 illustrates the behaviour of the sorting techniques considered in this work. It measures the complexities of the algorithms in term of CPU time against input data sizes. B-Sort, I-Sort, and S-Sort mean bubble, insertion and selection sort techniques. We observed that for all sorting techniques considered in this work, the CPU time is directly proportional to the input data size. For input data sizes 10,000 and 30,000, the CPU time required is almost the same. But for data sizes > 50,000 sharp deviation is observed among the techniques.

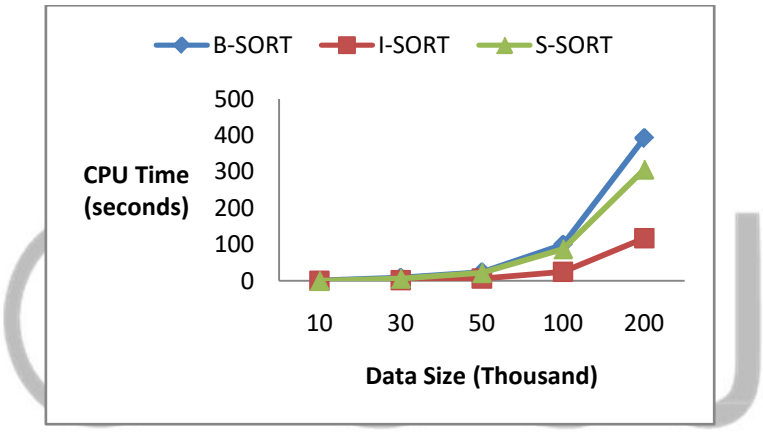

Figure 7: Graph of CPU Time Vs. Data Size

Also, figure 8 illustrates results of an experiment with complexities of the sorting techniques based on input data size against memory space. In similar manner, B-Sort, I-Sort, and S-Sort imply bubble, insertion and selection sort techniques. We observed that for all sorting techniques considered in this work, the memory space is directly proportional to the input data size. For input data sizes 10,000 and 30,000, the CPU time required is almost the same. But for data sizes > 50,000 sharp deviation is observed among the techniques.

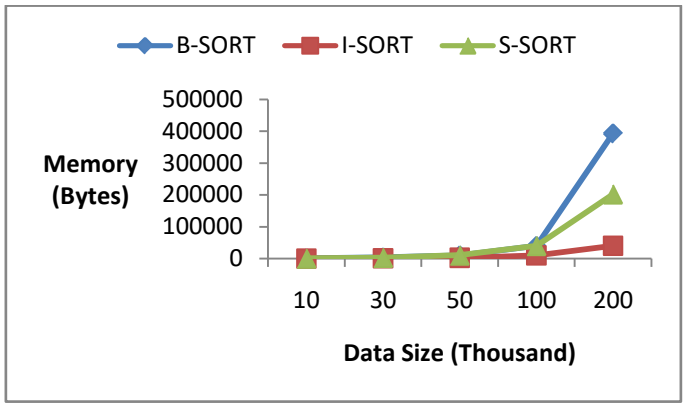

Figure 8: **Graph of Memory Space Vs. Data Size**

5. CONCLUSION

In conclusion, we have evaluated the performance of bubble, insertion, and selection 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. Empirical results were tabulated and graphically presented. The results obtained show that insertion sort technique is faster and requires less space than bubble and selection techniques in sorting data of any input data size. Similarly, results also show that the slowest technique of the three is bubble sort; while selection sort technique is faster and requires less memory than bubble sort, but slower and requires more memory than insertion sort. We infer from this study that empirical complexity can be determined in theory.

6. FUTURE THOUGHTS

We realized that in the solution space, numerous implementation solutions 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

[1] Wright A., Glut: Mastering Information through the ages, Henry, Washington D.C., 2007.

[2] Dubitzky W., Data mining techniques in grid computing environment, John Wiley and Sons, West Sussex, U.K., 2008.

[3] Aremu D.R., Adesina O.O., "Web services: A solution to interoperability problems in sharing grid resources", ARPN Journal of Systems and Software, vol. 1, no. 4, pp. 141 – 148, 2011.

[4] Qin M., "Merge Sort Algorithm", Department of Computer Sciences, Florida Institute of Technology, Melbourne, FL 32901.

[5] Qureshi K., "A Practical Performance Comparison of Parallel Sorting Algorithms on Homogeneous Network of Workstations", Department of Mathematics and Computer Science, Kuwait University, Kuwait.

[6] Pasetto D., Akhriev A., "A comparative study of parallel sort algorithms", IBM Dublin Research Laboratory, Dublin15, Ireland.

[7] Elmenreich W., Ibounig T., Fehérvári I., "Robustness versus performance in sorting and tournament algorithms", Acta Polytechnica Hungarica, vol. 6, no. 5, pp. 7 – 17, 2009.

[8] George T.H., Police G., Selkow S., Algorithms in a nutshell, O'Rielly, California, 2009.

[9] Blum R., "Professional Assembly Language", Wiley Publishing, Indianapolis, 2005.

[10] Edahiro M., "Parallelizing fundamental algorithms such as sorting on multi-core processors for EDA acceleration". Asia and South Pacific Design Automation Conference, pp. 230 – 233, 2009.

[11] Varman P.J., Scheufler S.D., Iyer B.R., Ricard G.R., "Merging multiple lists on hierarchical-memory multiprocessors", Journal of Parallel and Distributed Computing, vol. 12, no. 2, pp. 171 – 177, 1991.

[12] Tsigas P., Zhang Y., "A simple, fast parallel implementation of quicksort and its performance evaluation on Sun Enterprise 10000", 11th Euromicro Conference on Parallel Distributed and Network-based Processing, pp. 372 – 384, 2003.

[13] Bitton D., DeWitt D., Hsiao D.K., Menon J., "A taxonomy of parallel sorting", ACM Computing Surveys, vol. 16, no. 3, pp. 287 – 318, 1984.

[14] Leung J.Y-T, Handbook of scheduling algorithms models and performance analysis, CRC Press, Florida, 2004.

[15] Ahmad N.B., Jawawi D.N., "Selection sort", SCJ 2013 Data structure and algorithms, 2013.