*International Journal of Computer Applications (0975 – 8887)*
*Volume 145 – No.12, July 2016*

# AdaSort: Adaptive Sorting using Machine Learning

Somshubra Majumdar
D. J. Sanghvi College of Engineering
Mumbai, India

Ishaan Jain
D. J. Sanghvi College of Engineering
Mumbai, India

Kunal Kukreja
D. J. Sanghvi College of Engineering
Mumbai, India

## ABSTRACT

Sorting algorithms and their implementations in modern computing requires improvements in sorting large data sets effectively, both with respect to time and memory consumed. This paper is aimed at reviewing multiple adaptive sorting algorithms, on the basis of selection of an algorithm based on the characteristics of the data set. Machine Learning allows us to construct an adaptive algorithm based on the analysis of the experimental data. A review of algorithms designed using Systems of Algorithmic Algebra and Genetic Algorithms was performed. Both methods are designed to target different use cases. Systems of Algorithmic Algebra is a representation of pseudo code that can be converted to high level code using Integrated toolkit for Design and Synthesis of programs, while the Genetic Algorithm attempts to optimize its fitness function and generate the most successful algorithm.

## Keywords

Sorting, Machine Learning, Object oriented programming

## 1. INTRODUCTION

Sorting is defined as the operation of arranging an unordered collection of elements into monotonically increasing (or decreasing) order. Specifically, S = {a1, a2 ………….an} be a sequence of n elements in random order; sorting transforms S into monotonically increasing sequence S'= {a1 ', a2 '…………… an '} such that ai '≤ aj' for 1≤ i ≤ j ≤ n, and S' is a permutation of S [1].

There are certain characteristics of a data set that can be preprocessed to obtain some valuable information about the data set itself. A few characteristics obtained from a data set are its size and the degree of pre-sortedness. Different sizes of a data set necessitate utilization of different algorithms to sort them. Pre-sortedness can be described as the degree to which the initialized data set is already sorted. A sequence of integers to be sorted could be characterized by more than its length, but also by its degree of pre-sortedness. Three measures of pre-sortedness are used [2]:

- The number of inversions (INV),
- The number of runs of ascending subsequences (RUN)
- The length of the longest ascending subsequence (LAS)

RUN metric is shown to be the most efficient. RUN is calculated as number of subsets of the data set that are already sorted divided by the number of elements in the data set itself. A data set sorted in the descending order will have a pre-sortedness value of 1 while a data set sorted in the ascending order will have a value equal to 1/n.

Using the aforementioned characteristics of the data set, certain strategies can be used to optimize the performance of the algorithm used to sort the set. Two main strategies that can be used are Machine Learning and Genetic algorithm. Machine learning allows us to classify the data for making decisions on which algorithm is optimal, while Genetic Algorithm modifies itself to optimize the performance of the algorithm. These two are discussed in detail below.

Machine learning explores the study and construction of algorithms that can learn from and make predictions on data [3]. Such algorithms operate by building a model from example inputs in order to make data-driven predictions or decisions [4], rather than following strictly static program instructions. The machine learning techniques can be categorized based on the desired output of a machine-learned system [4]:

- Classification: Input sets are classified into one or more output classes based on a model
- Regression: the outputs are continuous rather than discrete values.
- Clustering: Inputs are divided into output groups. However, unlike classification, the groups are not known beforehand

Genetic algorithm is a search technique that simulates the process of natural selection. This technique is routinely used to obtain useful solutions to optimization and search problems. Genetic algorithm techniques are inspired by natural evolution - as in inheritance, mutation, selection and crossover. In such an algorithm, a set of possible candidate solutions to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties (its chromosomes or genotype) which can be mutated and altered [5].

In this paper, in section 2, features of the data sets used to perform analysis are studied. In section 3, The proposed model as well as methodology in creating the data sets and pre-sorting them are discussed. In section 4, The performance of the composite algorithms is compared with other base algorithms. In section 5, the conclusion is discussed.

## 2. LITERATURE SURVEY

An adaptive sorting algorithm is developed with the help of some pre-existing well known sorting algorithms, namely - Insertion sort, Shell sort, Heap sort, Merge sort and Quicksort. An adaptive sorting algorithm uses certain characteristics of the input dataset to select one or more sorting algorithms to sort the dataset. Generally pre-sortedness is computed using the RUNS metric, which is described as the number of sorted subsets or the data set, divided by the number of elements in that dataset.

Machine learning is similar to computational statistics in the sense that it also focuses on prediction-making. It has strong ties to mathematical optimization, which delivers implementations, theory and application domains to the field. Machine learning is used in a range of digital tasks where coding and implementing explicit algorithms is infeasible [4]. Example applications include spam filtering, speech and handwriting recognition, sentiment analysis, Natural Language Processing (NLP) and computer vision. Machine learning and pattern recognition can be viewed as two facets of the same field.

Algebraic algorithmics (AA in short) is an important domain of computer science, which was born from a collaboration of algebra, logic and algorithm schemes. It provides a standard for the knowledge about subject directions with the help of algebra and also deals with obstacles like standardization, specification of correctness and modification of algorithms. AA uses high-level abstractions of programs, represented by Systems of Algorithmic Algebras (SAA). One of the fundamental problems of AA is to increase the degree of flexibility of programs to particular use cases. Specifically, the problem can be solved at the disadvantage of argument-motivated creation of algorithm specifications by means of more complex algorithms [6].

Genetic algorithms are widely used in obtaining solutions for optimization and search problems. Evolution of the candidate solutions starts from a random population. This is an iterative process, where the population in each iteration is called as a generation. In every epoch, the fitness of every individual in the pool is evaluated such that the fitness is usually the value of the heuristic function in the optimization problem under consideration. A genetic algorithm requires a representation of the solution space (in the form of an array of bits) and a fitness function to test the solution space [8]. After the genetic representation and the fitness function is defined, a Genetic Algorithm begins by initializing a population of candidate solutions and then attempts to improve it through multiple applications of the mutation, crossover, inversion and selection operations on the population [8].

The authors of the paper "Optimizing Sorting with Machine Learning Algorithms" [7] talks about two methods of generating efficient sorting techniques. The first one uses machine learning algorithms to generate a function for specific target machine that is used to select the best algorithm. Their second approach builds on the first approach and constructs new sorting algorithms from a few fundamental operations. Using the array size, they are able to determine if the input set can fit into the cache memory and using the entropy of the input data, differentiate between the relative performances of radix sort with other comparison based sorting algorithms. Using sorting primitives as operations in genetic algorithms, they are able to create a composite algorithm which outperforms several other algorithms.

Yatsenko proposes the use of Decision Trees to classify the best sorting algorithm for the given data set [11]. They consider five sorting algorithms, mainly Insertion sort, Merge sort, Quick sort, Heap sort and Shell sort to analyses and decide the best algorithm for the given data set. They do this using various Decision Tree learning algorithms such as ID3, C4.5, NewId, ITrule and CN2. They then use SAA to formalize the algorithm used for conversion into C++ and Java code using IDS (Integrated toolkit for Design and Synthesis of programs). The drawbacks of this analysis is that there has been no consideration for multi-threaded algorithms which would drastically improve sorting time and also the fact that the study focuses on only smaller data sets (size < 100).

Before analyzing the adaptive sorting algorithm, an analysis of the performance of various standard sorting algorithms and their time and space complexities, expressed using the Big O notation, must be done.

**TABLE 2.1 Comparison of Sorting Techniques Based on the Parameters of the Proposed Model Based on Previous Studies and Experimental Results [10]**

| Sorting Method | Best Case Time Complexity | Worst Case Time Complexity | Space Complexity |
|---|---|---|---|
| Bubble Sort | O(n) | O($n^2$) | O(1) |
| Insertion Sort | O(1) | O($n^2$) | O(n) |
| Shell Sort | O(n $\log^2$ n) | O($n^2$) | O(n) |
| Heap Sort | O(n) | O(n log n) | O(1) |
| Quicksort | O(n log n) | O($n^2$) | O(log n) |
| Merge Sort | O(n log n) | O(n log n) | O(n) |

Based on a preliminary analysis of the various sorting algorithms widely known, algorithms that do not perform well for any case, or are too primitive to produce the output efficiently can be eliminated. Sorting techniques such as Bubble Sort and Selection sort have an average case complexity of O($n^2$) with best case complexity of O(n) and O($n^2$) respectively. Other considered sorting algorithms perform relatively better for all cases and as such negate the usage of Bubble and Selection Sort. The exception here is that while Insertion sort also has a complexity of O($n^2$), it's best case complexity is only O(1), which means that there are edge cases in which insertion sort produces the output in least time as compared to the other algorithms.

In addition to the above mentioned sequential sorting algorithms, a parallel quick sort algorithm is also incorporated into the testing procedure [12]. Utilizing a thread pool in order to limit the number of threads, the algorithm performs very well on large data sets and utilizes minimal additional memory resources. It is found to be more efficient than Parallel Merge Sort as the data set size increases above a few hundred thousand elements. In addition to the parallel quick sort algorithm, parallel merge sort implemented in the Java 8 Software Development Kit (JDK) is also incorporated as a test algorithm. It can be accessed using the Arrays.parallelSort() method, and is a Sort-Merge algorithm which uses the Fork-Join Common Pool introduced in Java 8 [14].

## 3. PROPOSED MODEL AND METHODOLOGY

### 3.1 Construction of dataset

Since example data set utilized for analysis needs to be an array of integers, one can generate a data set consisting of integers generated using a uniform Gaussian random number generator. Data sets of sizes varying sizes from 50 to 1 million uniformly distributed integers is generated. Due to the requirement of preprocessed arrays, 7 such replicas of each data set, whose subsets are then sorted according to the given input parameters of "pre-sortedness", are created. Thus each replicated dataset is pre-sorted according to the vector of 1/n to 1 where n is the size of the data set. Thus 42 data sets each having nearly 100 sample arrays are made. The exception to this is the data set of 1 million integers, of which there are far fewer samples. This is because the sorting time of such large data sets using Insertion Sort and

Shell Sort is far too large to be compared with faster sorting algorithms such as parallel merge sort and quicksort. It can be assumed that either parallel merge sort or quicksort will be the winning algorithm when such large data sets are given to the model.

Figure 3.1 describes the distribution of the generated arrays each having been pre-sorted to various degrees of pre-sortedness. Due to the large number of smaller sized arrays, the number of arrays which are almost completely sorted is higher than any other type of array.

## 3.2 Feature Selection

The proposed model will be using two features that are characteristic to each of the data sets. The first attribute will be the size of the array, as larger data sets cannot be sorted in an efficient manner using algorithms such as Insertion Sort and Shell Sort. The second attribute will be the pre-sortedness of the array, computed as the "RUNS" metric.

$$RUNS = \frac{Number\ of\ ascending\ subsequences}{Number\ of\ elements\ in\ the\ array}$$

Thus, for a completely sorted array, the RUNS value will be 1 / size of the array, whereas for an array sorted in descending order, RUNS value will be 1.

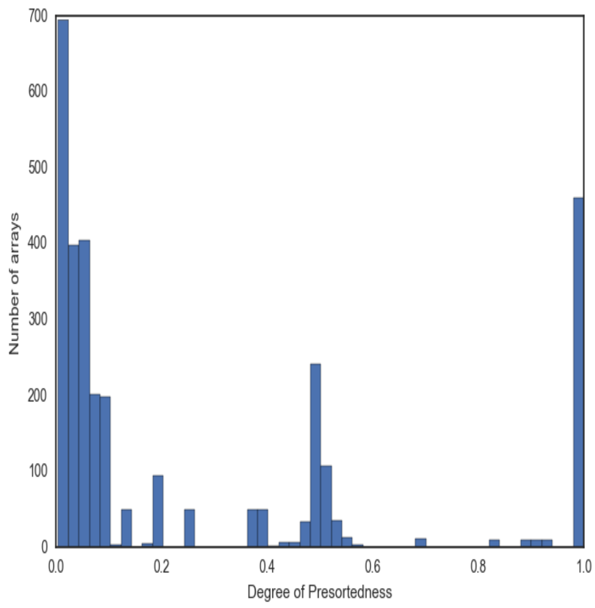

**Figure 3.1 Number of instances of arrays with different degrees of pre-sortedness (RUNS value)**

## 3.3 Preprocessing

These pre-sorted data sets are then sorted using Insertion Sort, Shell Sort, Heap Sort, Merge Sort, Quicksort, Parallel Merge Sort and Parallel Quick Sort. The sorting time for each algorithm on each array of each data set is computed and stored for later analysis. Alongside these results, the winning algorithm, that is the algorithm which uses the minimum time, can be determined to sort the given data set. In case of ties, one can select an algorithm which performs the best on similar data sets. Algorithms tie when data set size is small, as multiple algorithms sort the data set in almost exactly the same amount of time. In such cases, one can see which algorithm generally performs as the best algorithm in other data sets of the same size, and so assume it is also the current winner.

## 3.4 Optimizations

Each of the above algorithms was implemented using all feasible optimizations that would affect the sorting speed of any algorithm by a significant factor. Implementations of Quicksort were optimized according to the optimizations suggested by R. Sedgewick [13]. Parallel Quick Sort is implemented in accordance with the algorithm provided [12] and internally utilizes Dual Pivot Quicksort, available in the form of Arrays.sort() method in the Java 8 SDK to sort small subsets sequentially. Merge sort was tested in both iterative and recursive variants, and it was found that the recursive variant slightly outperformed the iterative counterpart.

The computed value of the threshold for Parallel Quick Sort was altered in order to reduce the execution time even further, while the general algorithm was implemented as is. Let data set size be given as 'n', and the number of available processors be 'p'.

The computation of the threshold can now be given as:

```
min_granularity := 8192
if n < min_granularity:
    SortDirectly(data_set)
else
    g := n / (p << 2)
    if g > min_granularity:
        threshold := g
    else:
        threshold := min_granularity
```

If the data set size 'n' is less than the minimum granularity, then a sequential sorting algorithm is applied. If n is larger, g is used to determine the minimum threshold.

By computing the threshold in this manner, the average performance of the Parallel Quick Sort algorithm was found to have increased, and thus this threshold computation was adopted.

## 3.5 Proposed Model

Following Yatsenko's model [11], a Decision Tree can be used to analyze and learn the features of the array size and degree of pre-sortedness and the winning algorithm as the target feature. Also use of the Gaussian Naive Bayes to classify the features, as well as train a multiclass Support Vector Machine with the Linear Dual Coordinate Descent as the optimization algorithm is made. The decision tree will also attempt to classify the data set with the splitting criterion as the gini score or with the information entropy score. The depth of the tree is also limited to prevent overfitting. Naive Bayes and multi class Support Vector Machine are also used to cross validate the classification results.

While Yatsenko's model [11] focused on smaller data sets of size smaller than 100 elements, the proposed model focuses on larger data sets of sizes in the range of 50 elements to 1 million elements. Focus is also on sorting large data sets using parallel algorithms and the threshold where thread creation cost is insignificant compared to the gain in sorting speed. The proposed model also attempt to measure the optimal threshold for dividing a data set into partitions for improving the performance of Merge Sort, Quick Sort and Parallel Merge Sort.

## 4. ANALYSIS OF PERFORMANCE OF ADASORT

### 4.1 Comparison of winning algorithms

Each of the seven sorting algorithms is utilized to sort the entire data set and provide information as to how much time in micro seconds was required to sort each array of each type in the data set by each algorithm. It was observed that each algorithm was a clear winner in a certain set of circumstances. However, there existed certain algorithms which performed poorly in every test condition.

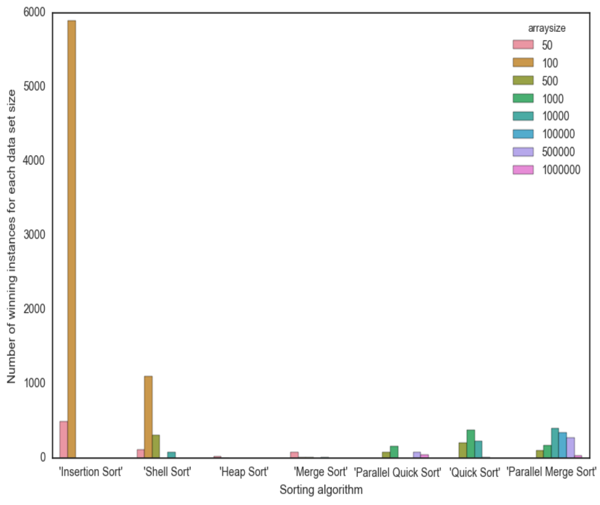

**Figure 4.1 Number of instances in dataset where algorithm is the winner**

In Figure 4.1, one can see that depending solely on the data set size, each algorithm is well suited to sort a given dataset of some size in the shortest time possible. It is seen that heap sort performs poorly for any given data set size, at least in comparison to other more efficient algorithms such as Parallel Merge sort and Quicksort. Due to this, a negligible number of instances where heapsort outperforms other algorithms from the final dataset are removed.

### 4.2 Machine learning to determine algorithm

According to Yatsenko [11], the use of a Decision Tree is optimal to understand and implement an equivalent sorting algorithm in any programming language, as the decision tree can be easily understood and implemented using if-else constructs available to all modern programming languages.

A decision tree is very prone to suffer from the overfitting problem. 10 fold cross validation has been used to limit the depth of the tree in the Machine Learning library, Accord.NET. Therefore even with a very large number of unique samples, the decision tree does not suffer from overfitting. The C4.5 learning algorithm was chosen for the Decision Tree Learning algorithm, due to its superior performance in comparison to the Iterative Dichometer 3 (ID3) Algorithm.

Since there exists no direct method to visualize the Decision tree in a graphical format using Accord.NET, the decision tree has been exported to a format that is suitable for the Java based machine learning tool, Weka, in order to obtain the graphical representation of the decision tree.

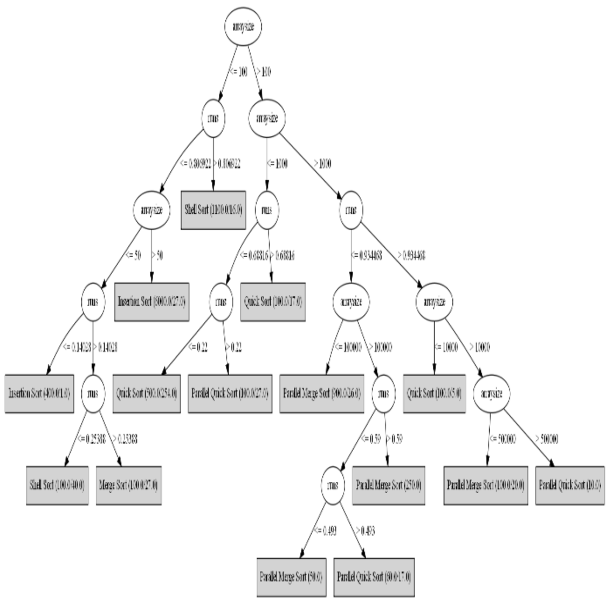

**Figure 4.2 Decision Tree visualization**

As seen in the decision tree, there are many redundant paths which can be pruned by logical inference. When the model is implemented to match the Decision Tree, the time required to compute the presortedness metric for very large size data sets (above 100,000 items) takes more time than to directly sort the data using a fast general algorithm such as Parallel Merge or Quick sort.

In order to optimize this algorithm, a few redundant branches of the tree can be pruned in order to avoid computation of the runs metric for large data sets. This technique reduces the accuracy of the final algorithm, but the composite algorithm is sufficiently fast enough in general and the gain in execution time is less than 5 % of the best execution time in most cases.

The classification accuracy of this decision tree to determine the winning algorithm is 98.3969 %, but practically another algorithm is superior in some cases. It is to be noted that in such cases, the difference in execution speed between the winning algorithm and the algorithm selected by the decision tree is small enough to be considered inconsequential. In most cases, the percentage difference between the two execution speeds is less than 5 % of the execution time of the faster algorithm.

### 4.3 AdaSort Algorithm

By reducing the various decision paths the algorithm must take, the execution speed of the adaptive algorithm is improved. The algorithm can be given as:

```
Algorithm AdaSort(data, n)
{
  if n <= 100:
        runs := computeRunsMetric(data, n)
        if runs > 0.68799:
                shellSort(data, n)
        else if n <= 50 and runs > 0.44:
                parallelMergeSort(data, n, p)
        else if n <= 50 and runs > 0.25388:
                mergesort(data, n)
        else:
                insertionSort(data, n)

    else if n <= 1000:
                quickSort(data, n)
    else if n <= 500000:
                parallelMergeSort(data, n, p)
    else:
                parallelQuickSort(data, n, p)
}
```

As seen by the algorithm, the decision path taken by the algorithm has been heavily pruned as the size of the data sets increases. It is observed in the decision tree that, when there are less than 1000 items in the array, Quicksort is almost always selected, and only under cases of almost completely sorted arrays does Parallel Quick Sort perform better. Removal of the overhead of computing the runs metric and generalization of the solution to use Quicksort all of the time when the data set size is between 100 and 1000 items.

Similarly, for data sets smaller than 500,000 items, it is noted that Parallel Merge Sort generally outperforms all others in a large variety of cases, so the algorithm can simply be generalized to use Parallel Merge Sort if the data set size if between 1,000 and 500,000. This also avoids the lengthy computation of the runs metric for such large data sets.

## 4.4 Optimizations to AdaSort

It must be noted that for small data sets of size 100 items are less are sorted in only a few microseconds. There are circumstances where optimizations native to the programming language are necessary in order for the composite algorithm to perform better than the core algorithm.

Computation of the runs metric is trivial for small data sets. However for larger data sets, the execution time required to compute the runs metric is often too large. The adaptive algorithm will perform poorly if the runs metric is computed for data sets larger than 100 items. Generally, computation of runs is unnecessary for such large data sets, since a single algorithm is generally fast enough for a certain data set size, and thus outliers can be neglected.

For data sets of size less than 100 items, even insertion sort completes the sorting process in less than 5 microseconds. As such, the comparison of multiple runs values can be a minor drawback. In such cases, a group of runs metrics can be merged into a single comparison in order to cover more search cases, at the expense of a small gain in sorting speed of the composite algorithm (1-2 microseconds).

The execution environment chosen for testing is a 64 bit 4 core (2 Physical, 4 Logical) Intel i5 processor and possessing 8 GB of RAM. Upon testing various conditions, it was found that comparison of 2 double type values was slightly slower than comparing 2 long type data values. While this difference is miniscule at execution speeds of less than 5 microseconds, it caused the composite algorithm to perform less efficiently. In order to correct this, the runs metric was multiplied by $10^5$ and stored in a variable of data type long. The comparisons were then made using this long variable and the 3 runs metrics which were also precomputed to long values by multiplying them by $10^5$. This small gain was sufficient in improving the execution speed of the adaptive algorithm and being comparable to the straight insertion sort algorithm.

## 4.5 Analysis of performance

The analysis of the composite adaptive algorithm was done as follows. Each of the arrays of the data set was sorted by the adaptive algorithm once, and the resulting execution times were compared with the winning algorithm identified by the lowest execution time among the rest 7 algorithms. The adaptive algorithm was considered to be the ‘best’ if its execution time was equal to or lesser than the winning algorithm. It was found that the optimized adaptive algorithm was as fast as / slightly slower than the winning algorithm nearly 96.3795 % of the time.

The small reduction in accuracy of the optimized algorithm was due to the pruned decision process in the adaptive sorting algorithm. However this reduction in classification of the best algorithm is acceptable, as the time required to predict the correct algorithm is reduced due to the pruning procedure.

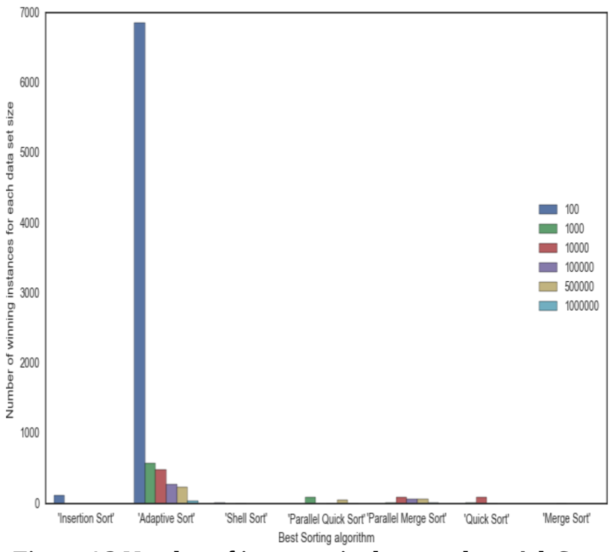

**Figure 4.3 Number of instances in dataset where AdaSort is the winner**

It is to be noted that on several occasions, the adaptive algorithm (AdaSort) was only a few micro seconds slower than the winning algorithm. Taking a 5 % margin of difference in execution speed between the winning algorithm and the adaptive algorithm, the adaptive algorithm performs as well as the winner nearly 96.5758 % of the time.

In table 4.1, the average sorting time in microseconds of all 7 general algorithms is provided, along with the sorting time required by the adaptive algorithm, taking into account a 5 % difference in execution time of the adaptive algorithm and the best algorithm for sorting that data set. The table considers only arrays which are partially sorted (runs value of approximately 0.8). This is because most data sets found in the real world are almost not sorted at all, and runs metric of 0.9 suggests an almost unsorted array.

**Table 4.1 Average sorting time in microseconds of various algorithms, partially sorted with runs metric close to 0.8, and taking 5 % margin of difference.**

| Array Size | Insertion Sort | Shell Sort | Merge Sort | Quick Sort | Parallel Merge Sort | Parallel Quick Sort | Adaptive Sort |
|---|---|---|---|---|---|---|---|
| 50 | **2** | 4 | 10 | 11 | 16 | 17 | **2** |
| 100 | **5** | 7 | 11 | 11 | 18 | 22 | **5** |
| 1,000 | 139 | 60 | 73 | **41** | 51 | 59 | **42** |
| 10,000 | 12,897 | 870 | 872 | 580 | **387** | 1,535 | **388** |
| 100,000 | 1,368,328 | 11,469 | 10,601 | 9,647 | **3,526** | 4,538 | **3,528** |
| 500,000 | 36,464,643 | 66,710 | 59,531 | 44,394 | **11,777** | 18,891 | **11,781** |
| 1,000,000 | 132,810,830 | 133754 | 115687 | 76270 | 34356 | **27983** | **27986** |

## 5. CONCLUSION

This paper is aimed at implementing a method to sort large data sets while having used Machine Learning for analysis. The analysis done previously is utilized in order to construct a general model for adaptively sorting data sets. As shown in Figure 4.1, certain sorting algorithms outperform others under certain circumstances. The AdaSort algorithm utilizes this property and performs efficiently under all circumstances.

The prospects of further investigations in this direction are

to integrate this technique into systems with larger and more complex datasets containing user defined objects as well as improve the stability of this sort.

Another important development prospect for this topic could be its usage in real scenarios such as sorting data based on physical or semi-physical relationships and forming or discovering useful patterns in them

## 6. REFERENCES

[1] M.J. Quinn, “Parallel Programming in C with MPand OpenMP” Tata McGraw Hill Publications, 2003, p. 338.

[2] Guo, H.: “Algorithm selection for sorting and probabilistic inference: A machine learning-based approach”. Ph.D. dissertation, Kansas State University (2003)

[3] Ron Kohavi; Foster Provost (1998). "Glossary of terms". Machine Learning 271–274.

[4] C. M. Bishop (2006). Pattern Recognition and Machine Learning. Springer.ISBN 0-387-31073-8.

[5] Mitchell, Melanie (1996). An Introduction to Genetic Algorithms. Cambridge, MA: MIT Press. ISBN 9780585030944.

[6] Doroshenko, A., Tseytlin, G., Yatsenko, O., Zachariya, L.: Intensional Aspects of Algebra of Algorithmics. Proceedings of International Workshop “Concurrency, Specification and Programming” (CS&P’2007), 27–29 September 2007, Lagow (Poland) (2007).

[7] Li, Xiaoming, Maria Jesus Garzaran, and David Padua. “Optimizing Sorting With Machine Learning Algorithms.” 2007 IEEE International Parallel and Distributed Processing Symposium (2007): n. pag. Web.

[8] Whitley, Darrell (1994). "A genetic algorithm tutorial". Statistics and Computing 4 (2): 65–85. doi:10.1007/BF00175354.

[9] "The Analysis of Heapsort". Journal of Algorithms 15: 76–100.doi:10.1006/jagm.1993.1031.

[10] Donald Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0.

[11] Olena Yatsenko. (2011). On Application of Machine Learning for Development of Adaptive Sorting Programs in Algebra of Algorithms, Concurrency, Specification and Programming, September 28-30, Pułtusk, Poland, pp. 577-588.

[12] Majumdar, Somshubra, Ishaan Jain, and Aruna Gawade. "Parallel Quick Sort Using Thread Pool Pattern." International Journal of Computer Applications IJCA 136.7 (2016): 36-41. Print

[13] R. Sedgewick. Implementing Quicksort Programs. Communications of the ACM, 21(10):847–857, October 1978.

[14] “Arrays (Java Platform SE 8)."Arrays (Java Platform SE 8). Web. 28 Mar,2015. <http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>.