Multiple Pivot Sort Algorithm is Faster than Quick Sort Algorithms: An Empirical Study

Salman Faiz Solehria¹, Sultanullah Jadoon²
¹. Faculty of Information Technology, Sarhad University, Peshawar,
². Department of Information Technology, Hazara University, Haripur Campus,

ss_suit@yahoo.com, sultanullah@gmail.com

Abstract—Multiple Pivot Sort is a new comparison-based sorting algorithm that has been developed to address shortcomings in current popular comparison-based sorting algorithms. The goal of this work is to perform an extensive empirical study of Multiple Pivot Sort against other established sorting algorithms including Quick Sort, Merge Sort, and Heap Sort. This research presents data to support the hypothesis that Multiple Pivot Sort is an extremely promising new algorithm in a critical field of Computer Science. The empirical study includes comparisons of the various sorts using randomly generated arrays of size 10 data items up to 1 million or more data items. In addition, integer arrays in which the data is in order and in reverse order are used in the study. Data items include integers, strings, and classes. The study keeps track of the number of comparisons, the number of data moves, and CPU time specific to the machine for each sort. Results of the empirical study performed in this research support the claim that Multiple Pivot Sort may well be the fastest sorting algorithm in existence.

Index Term — Multiple Pivot Sort, Quick Sort, Empirical Study, Sorting Algorithm.

I. INTRODUCTION

SORTING algorithms are the basic foundations of practical Computer Science, and consequently, the analysis and design of useful sorting algorithms has remained a priority in the field. However, despite several new algorithms like Three-Way Radix Quick Sort, LSD Radix Sort and Multikey Quick Sort being introduced, the majority of programmers in the field rely on one of the three staple comparison-based sorting algorithms: Quick Sort, Merge Sort, and Heap Sort. The new algorithms are generally not being used as much because they are not as general purpose as comparison-based sorts, usually require more tweaking to work with new classes and data types, and in most cases, do not perform as well as expected due to poorer locality of reference caused by linear passes through the array.

For a new algorithm to be accepted and used in the field of Computer Science, the method must be shown to have comparable performance to the aforementioned sorting algorithms and be easy to use, implement, and debug. Comparable performance will get programmers interested in the algorithm, and ease-of-use will be the final determinant in changing the programmer’s sorting preference. The easiest, most time efficient way to accomplish a task is usually preferred, and the comparison-based sorts highlighted here offer excellent performance and adaptability to any type of record or data type to be sorted.

Quick Sort, Merge Sort, and Heap Sort have been around for over 40 years and have been taught widely in the academic Computer Science community. These algorithms are established and well documented with literally dozens of thousands of published empirical analyses comparing and contrasting performances, highlighting theoretical boundaries, and recommending preferred solutions. This paper is no different in this aspect. However, this paper does divulge new information on a newly developed comparison-based sorting algorithm based on the partitioning scheme implemented in Quick Sort, and this algorithm has shown amazing promise by being able to meet and beat Quick Sort, Merge Sort, and Heap Sort in their own ideal sorting situations.

Multiple Pivot Sort, also referred to as Multiple Pivot Sort or Pivot Sort, is a new in place comparison-based sort that was developed in late 2004 and early 2005. The algorithm accomplishes sorting by grabbing a large sample of the array and isolating the sample by swapping it with records at the end of the list of records. Once the sample has been moved to the end of the list, the sample is sorted with Insertion Sort (an algorithm that works very well on lists smaller than 16 elements) and pivots are passively selected at locations that guarantee a partition of at least one record between the pivots. The list of records is then partitioned around the pivots (ie lesser records are placed before the pivots and greater records placed after.) Also, the algorithm calls for comparing pivots for equality and if found to be equal, placing all equal records remaining in the list between the two equal pivots, which solves a major problem with Quick Sort on lists with large numbers of duplicates. More information including a formal algorithm definition may be found in reference [4].

The basic outline of Multiple Pivot Sort is as follows:

a. Choose 7-15 equidistant pivot candidates from the list.
b. Isolate these pivot candidates by moving them to the end of the list.
c. Sort these pivot candidates with Insertion Sort.
d. Passively select 3 to 7 pivots starting from the second sorted candidate and every other candidate.

e. Partition the list around the pivots.

f. Recursively call Multiple Pivot Sort on any partition with more than 15 elements. Call Insertion Sort on any partition with 2-15 elements.

The following diagrams outline the major processes visually.

![Diagram 1](image1)

**Pivot Candidate Selection with 3 Pivots (25 elements)**

![Diagram 2](image2)

**Pivot Selection with 3 Pivots (25 elements)**

![Diagram 3](image3)

**Partition Phase with 3 Pivots (25 elements)**

The empirical study conducted here analyzes the performance of QuickSort, MergeSort, HeapSort, and PivotSort on near unique random lists of integers, strings, and a simple class of four string members as well as an ascending and descending list of integers, all of which are situations the former three algorithms are known to work well in. Two different methods for choosing pivots in QuickSort are included in these tests: the Median-Of-Three method (which does not use random selection because these are known random lists) and a simple pivot selection of the middle element of the list. All of the algorithms included are easy to implement and change for different data types and perform in \( O(n \log n) \) time, which has been shown using decision trees to be optimal [3]. Of the three established algorithms, QuickSort is known to be the fastest average case performer, and that is why the title is specific to that comparison.

### II. ANALYSIS

Sorting algorithms are usually ranked by numbers of data comparisons and moves, and the theoretical marks of QuickSort, HeapSort, and MergeSort are well documented. On average QuickSort performs very close to optimum in comparisons, performing about \( 2n \log n \) comparisons [8]. MergeSort requires just slightly more comparisons than optimal and performs just above \( n \log n \) consistently on any data set [8]. HeapSort performs between \( 2n \log n \) and \( 3n \log n \) comparisons [8]. With these figures in mind, MergeSort would appear to be fastest. However, comparisons are not the only gauge of speed, and QuickSort proves more efficient due to less average data movements, a very efficient partitioning scheme, and better locality of reference from not having to use an extra array.

Multiple Pivot Sort is a new algorithm that has received very little theoretical analysis. However, the worst case for comparisons and moves is known to hover around \( n^2 / 4 \), roughly half the worst case of QuickSort, due to a worst case of partitioning two records into order at a time rather than QuickSort’s one. Based on many empirical tests, the average case performance for Multiple Pivot Sort is 20% more comparisons than QuickSort, but QuickSort performs roughly 50% more data moves. Both data moves and comparisons hover at the \( 2n \log n \) mark, which is in contrast with the other three algorithms – all of which perform significantly more data movements.

Multiple Pivot Sort’s partitioning strategy is strengthened by the Strong Law of Large Numbers. Because sample median is considered to be an unbiased estimator and variance of sample median decreases as sample size increases, the Strong Law states that taking a larger sample size will produce a statistic closer to the actual population median, on the average. This correlation was incorrectly stated in a previous paper [4] as falling in line with the Central Limit Theorem.

Multiple Pivot Sort’s performance is also increased because it attempts to do more work per sample by placing pivot points throughout the sample and performing more partitions per level than QuickSort. This results in three optimal cases for Multiple Pivot Sort. The first case involves having every sample partitioned exactly like QuickSort, ie after partitioning, the first pivot would end up at the \( 1/2 \) mark, the second pivot would end up at the \( 3/4 \) mark, the third pivot at the \( 7/8 \) mark, etc. This case is extremely rare and has never been observed in any tests on random data. The second optimal case occurs when the partitioning phase results in equal partitions between each pivot. This special case for Multiple Pivot Sort can be observed in the ascending and descending list tests (also the optimal case for QuickSort). The third optimal case for Multiple Pivot Sort occurs when the list is composed entirely or nearly entirely with duplicates. Because Multiple Pivot Sort may be programmed to compare pivot values for equality and partition around a second equal pivot
differently (by moving equal elements between the equal pivots), Multiple Pivot Sort performs sorting in linear time. This case may be viewed by sorting a list based on a Boolean value like sex (male or female). Quick Sort’s performance on such lists quickly approaches $O(n^2)$ unless using a modification such as Bentley’s three-way partitioning strategy [7].

Another feature of Multiple Pivot Sort that complicates a thorough analysis is the ability to adapt and correct performance problems at runtime. Such a feature was not implemented in the code base for this testing suite as it wasn’t necessary for the types of tests performed here. However, in a real database situation, Multiple Pivot Sort could be tweaked to check the final partition locations of the pivots and change the next level of sample size by simply changing the number of pivots to be used. This is in sharp contrast to Quick Sort which has no runtime policies to correct poor performance.

III. TESTING

All testing was done on a Dell Core i3 Processor with three GBs of RAM running Windows 7. The testing suites were produced with Visual Studio .Net as C++ console applications. The strings tested were ANSI C++ strings available in the standard library. All graphed results are mean averages produced from ten runs of each sorting algorithm on the same ten data sets.

The goal of these tests is to determine Multiple Pivot Sort’s performance in comparison to Quick Sort, Heap Sort, and Merge Sort. Merge Sort was implemented in top down fashion as was Heap Sort. Two different pivot selection versions of Quick Sort were tested: the Median-of-Three method and a method that simply chooses the middle element of the partition as a pivot. As Quick Sort is generally considered the fastest general purpose comparison-based sorting algorithm, Multiple Pivot Sort is compared to it most and with good reason. Both versions of Quick Sort perform better than the implementations of Heap Sort or Merge Sort.

The first tests demonstrate performance on common patterns (ascending and descending lists) and are shown in Figures 4 through 7. These types of sorting operations are performed quite often in databases when either reversing the order of a list or just making sure that a list is sorted. Tests were performed on ascending and descending lists from sizes 10 to 100 million integers. However, only the results of the tests on 100 million integers are reported here for the sake of brevity, conciseness, and readability.

These kinds of sorts are Quick Sort’s optimal scenario, and no other sort beats it, but Multiple Pivot Sort comes close — finishing only one second later on both ascending and descending lists. Sorting performance on ascending and descending lists is not a major influence on algorithm selection, so not much will be elaborated here, but these tests do show that Multiple Pivot Sort performs very well on these common database operations.
The next tested scenario is much more practical. The following tests were performed on lists of 100 million near unique random integers, and the results are shown in Figures 8 through 11. The random integer generator was written by Andy Thomas [9].

Again, Quick Sort has been thoroughly noted by academia to perform near optimal data comparisons on near unique lists. However, Multiple Pivot Sort again shows its worth by meeting Quick Sort’s results in time completion and performing less total data comparisons and moves. More tests on integer lists between these sorting algorithms may be found in reference [4], including tests on lists of duplicates in which Quick Sort performs poorly. See the analysis section for more information about Quick Sort’s performance on duplicate records.

The next series of experiments gauge each algorithm’s performance on lists of randomized strings, and the results are shown in Figures 12 through 15. These tests are of immense practical importance, as a large proportion of sorting operations are done on text. Each sorting algorithm was tested on a list of 10 to 10 million randomly generated strings of 4 to 19 characters. Only the results of the tests on 10 million strings are reported here.
Multiple Pivot Sort performs better than the other sorting algorithms in these tests because Multiple Pivot Sort averages far less data movements. When dealing with larger classes like strings, the data movements performed weigh more heavily on algorithm performance than an equivalent number of data comparisons.

The final series of tests were performed on lists of one million randomly generated classes, each containing four string members of 7-17 characters. The results of these tests are shown in Figures 16 through 19. This type of situation is commonly found in most databases, like an accounting database of customers.

As noted in the string tests, Multiple Pivot Sort’s ability to minimize data movements results in better performance on large classes in contiguous array settings. The performance gap only increases as the size of the array grows larger. Many databases attempt to overcome this problem by converting the contiguous array into a queued list and converting data movements to pointer arithmetic, but this adds memory (in this case it would mean four extra megabytes of RAM.)

IV. CONCLUSION

In a previous paper [4], Multiple Pivot Sort is shown to perform faster than Quick Sort, Heap Sort, and Merge Sort when duplicates are present. The empirical study conducted in this paper shows that Multiple Pivot Sort meets or beats these same established sorting algorithms in unique random lists including strings and classes. As demonstrated in the string and class tests, Multiple Pivot Sort uses fewer moves and consequently should be preferred in situations where contiguous lists of large data structures are being used. Combined with excellent performance on ascending and descending lists and less total operations, these factors make Multiple Pivot Sort a viable general purpose sorting algorithm for all types of databases.

REFERENCES