Proposed Model for Sorting Algorithms

Malika Dawra
Department of Computer Science & Applications
M.D. University, Rohtak, Haryana, India
E-mail: er.malika@gmail.com

Priti
Asstt. Prof., Department of Computer Science & Applications
M.D. University, Rohtak, Haryana, India
E-mail: pritish80@yahoo.co.in

***Abstract*:**

An important issue in computer science is ordering a list of items. Sorting is the process of putting data in order; either numerically or alphabetically. Sorting problem has attracted a great deal of research because efficient sorting is important to optimize the use of other algorithms such as binary search. This paper presents a model that will split large array in sub parts and then all the sub parts are processed in parallel using existing sorting algorithms and finally outcome would be merged. To process sub parts in parallel multithreading has been introduced.

Keywords- algorithm, sorting, multithreading, splitting, merging.

INTRODUCTION

A sorting algorithm is an efficient algorithm, which perform an important task that puts elements of a list in a certain order or arrange a collection of items into a particular order. It is necessary to arrange the elements in an array in numerical or lexicographical order, sorting numerical values in descending order or ascending order and alphabetical value like addressee key [5, 6]. Sorting has been a profound area for the algorithmic researchers. And many resources are invested to suggest a more working sorting algorithm. For this purpose many existing sorting algorithms were observed in terms of the efficiency of the algorithmic complexity [9]. Sorting is used in many important applications and there have been a plenty of performance analysis [7]

All sorting algorithms are problem specific i.e. they are appropriate for specific kinds of problems. Some sorting algorithms work on less number of elements, some are used for huge number of data, some are used if the list has repeated values, and some are suitable for floating point numbers. Sorting algorithms are classified by:

- Computational complexity in terms of number of swaps. Sorting methods perform various numbers of swaps in order to sort a data.
- System complexity of computational. In this case, each method of sorting algorithm has different cases of performance. They are worst case, when the integers are not in order and they have to be swapped at least once. The term best case is used to describe the way an algorithm behaves under optimal conditions.
- Usage of memory and other computer resources is also a factor in classifying the sorting algorithms.
- Recursion: some algorithms are either recursive or non recursive, while others may be both.
- Whether or not they are a comparison sort examines the data only by comparing two elements with a comparison operator.

Sorting algorithms are sometimes characterized by big O notation in terms of the performances that the algorithms yield and the amount of time that the algorithm takes. The different cases in sorting algorithm are:

- O(n): is fair, the graph is increasing in the smooth path.
- $O(n^2)$: is inefficient, with larger input the graph shows significant increase.
- O(n log n): is efficient, it shows slower pace increase in graph as we increase the size of array or data.

There are literally hundreds of different ways to sort arrays. The basic goal of each of these methods is the same: to compare each array elements to another array element and swap them if they are in the wrong position. It continues executing over and over until the data is sorted.

EXISTING SORTING ALGORITHMS

There are several elementary and advance sorting algorithms available to sort the elements of an array in numerical or lexicographical order. Some algorithms work well on some specific problems and do not work for all the problems. Some of the sorting algorithms are:

- Bubble sort
- Selection sort
- Insertion sort
- Shell sort
- Quick sort
- Merge sort

Bubble Sort

In the bubble sort, as elements are sorted they gradually "bubble" (or rise) to their proper location in the array. Bubble sort was analyzed as early as 1956 [1]. The bubble sort compares adjacent elements of an array until the complete list gets sorted. The first and second elements are compared and swapped if out of order. Then the second and third elements are compared and this sorting process continue until the last two elements of the array are compared and swapped if out of order. When this first pass through the array is complete, the bubble sort starts the process all over again. The bubble sort knows that it is finished when it examines the entire array and no swaps are needed.

An array of numbers before, during and after a bubble sort for descending order:

Array at beginning :	52	14	30	57	75	63
After Pass 1:	52	30	57	75	63	14
After Pass 2:	52	57	75	63	30	14
After Pass 3:	57	75	63	52	30	14
After Pass 4:	75	63	57	52	30	14
After Pass 5:	75	63	57	52	30	14

The bubble sort is an easy algorithm to program, but it is slower than many other sorts. With a bubble sort, it is always necessary to make one final pass through the array to check to see that no swaps are made to ensure that the process is finished.

Selection Sort

The selection sort is a combination of sorting and searching. During each pass, the unsorted element with the smallest (or largest) value is moved to its proper position in the array. The number of times the sort passes through the array is one less than the number of items in the array. In the selection sort, the inner loop finds the next smallest (or largest) value and the outer loop places that value into its proper location. The selection sort has a complexity of $O(n^2)$ [3].

An array of numbers using selection sort for descending order:

Array at beginning :	52	14	30	57	75	63
After Pass 1:	52	63	30	57	75	14
After Pass 2:	52	63	75	57	30	14
After Pass 3:	57	63	75	52	30	14
After Pass 4:	75	63	57	52	30	14
After Pass 5:	75	63	57	52	30	14

While being an easy sort to program, the selection sort is one of the least efficient. The algorithm offers no way to end the sort early, even if it begins with an already sorted list.

Insertion Sort

The insertion sort, unlike the other sorts, passes through the array only once. The insertion sort splits an array into two sub-arrays. The first sub-array is sorted and increases in size as the sort continues. The second sub-array is unsorted, contains all the elements yet to be inserted into the first sub-array, and decreases in size as the sort continues.

An array of numbers using the insertion sort for descending order:

Array at beginning :	52	14	30	57	75	63
After Pass 1:	52	14	30	57	75	63
After Pass 2:	52	30	14	57	75	63
After Pass 3:	57	52	30	14	75	63
After Pass 4:	75	57	52	30	14	63
After Pass 5:	75	63	57	30	14	63

The insertion sort can be very fast and efficient when used with smaller arrays. If an application only needs to sort smaller amount of data, then it is suitable to use this algorithm [2]. But unfortunately, it loses this efficiency when dealing with large amounts of data.

Shell Sort

The shell sort is named after its inventor D.L. Shell. Instead of comparing adjacent elements, the shell sort repeatedly compares elements that are a certain distance away from each other (d represents this distance). The value of *d* starts out as half the input size and is halved after each pass through the array. The elements are compared and swapped when needed. The equation d = (N + 1)/2 is used, where N is the number of elements in the array.

An array of elements using shell sort for descending order:

Array at beginning :	52	14	30	57	75	63	D
After Pass 1:	57	75	63	52	14	30	3
After Pass 2:	63	75	57	52	14	30	2
After Pass 3:	75	63	57	52	30	14	1
After Pass 4:	75	63	57	52	30	14	1

The shell sort is a “diminishing increment sort”, better known as a “comb sort” [8].
The algorithm makes multiple passes through the list, and each time sorts a number of equally sized sets using the insertion sort [4]. This sorting process, with its comparison model, is an efficient sorting algorithm.

Quick *Sort*

The quick sort is considered to be very efficient with its "divide and conquer" algorithm. This sort starts by dividing the original array into two sections (partitions) based upon the value of the pivot (can be first element in the array). The first section will contain all the elements less than (or equal to) the pivot and the second section will contain all the elements greater than the pivot. This sort uses recursion – the process of "calling itself".

An array of elements using quick sort for descending order:

Array at beginning :	52	14	30	57	75	63
	57	75	63	52	14	30
	75	63	57	52	14	30
	75	63	57	52	14	30
	75	63	57	52	14	30
	75	63	57	52	30	14

Quick sort was considered to be a good sorting algorithm in terms of average theoretical complexity and cache performance [7].

Merge Sort

The merge sort combines two arrays into one larger array. The arrays to be merged must be sorted first. It follows "divide and conquer" strategy. Firstly divide the n-element sequence to be sorted into two subsequence of n/2 element each. Sort the two subsequences recursively using merge sort and then merge the two sorted subsequences to produce the sorted answer.

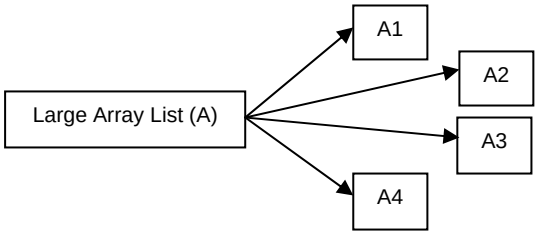
PROPOSED MODEL FOR SORTING ALGORITHMS

A large array is divided in sub parts and then all sub parts are processed in parallel using existing sorting algorithms and finally outcome would be merged. To process sub parts in parallel multithreading has been introduced. Various steps involved in this are:

- Divided array in sub parts
- Use of efficient algorithms on sub parts
- Algorithms are implemented in parallel.
- Parallel implementation can be done using multithreading.
- After sorting all sub parts are merged.

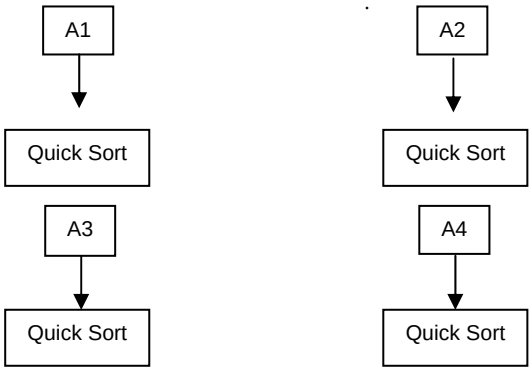
A. Divide Array in sub parts

Split the large array into smaller parts and store them in new arrays.



B. Use of Efficient Algorithms on sub parts

After creating new sub arrays, all sub arrays are processed using efficient algorithms



C. Algorithm are implemented in parallel

In order to implement sub arrays in parallel concept of multithreading would be used.

The multithreading paradigm has become more popular as efforts to further exploit instruction level parallelism have stalled since the late-1990s. This allowed the concept of throughout computing to re-emerge to prominence from the more specialized field of transaction processing:

- Even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multi-tasking among multiple threads or programs.
- Techniques that would allow speed up of the overall system throughout of all tasks would be a meaningful performance gain.

The two major techniques for throughout computing are multiprocessing and multithreading.

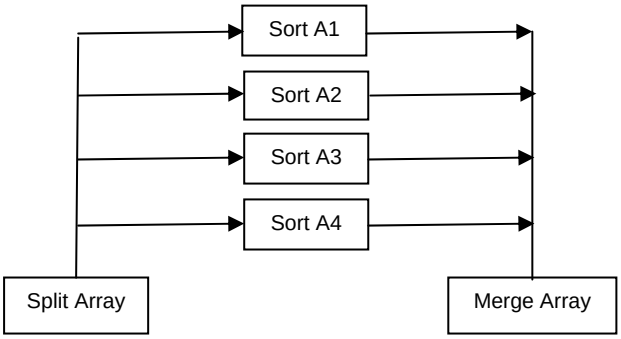
D. Parallel implementation can be done using multithreading

Multithreading is similar in concept to preemptive multitasking but is implemented at the thread level of execution in modern superscalar processors. Simultaneous multithreading (SMT) is one of the two main implementations of multithreading, the other form being temporal multithreading. In temporal multithreading, only one thread of instructions can execute in any given pipeline stage at a time. In simultaneous multithreading, instructions from more than one thread can be executing in any given pipeline stage at a time. This is done without great changes to the basic processor architecture.

- The main additions needed are the ability to fetch instructions from multiple threads in a cycle.
- A larger register file to hold data from multiple threads.

The two major techniques for throughout computing are multiprocessing and multithreading.

E. After sorting all sub parts are merged



After merging new array will store sorted information and would be displayed on screen.

CONCLUSION

Insertion sort is good only for sorting small arrays. The smaller the array the faster is the insertion sort. It becomes very slow when the size of array increases. Shell sort is slightly slower in average with very big arrays. Merge sort is a stable sort but its main problem is that it requires a second array with the same size as the array to be sorted, thus doubling the memory requirements. Quick sort is the most popular sorting algorithm. It is very fast algorithm and reason for its speed is that its inner loop is very short and can be optimized very well. The main problem with quick sort is that it is not trustworthy. But when we split a large array into equal parts and apply efficient sorting functions on sub arrays in parallel then parallel execution in faster processing. It takes less time to merge all sorted arrays that have been processed quickly in separate thread in parallel. One and only limitation is that the systems that does not support multithreading will not be eligible to get benefit. But most of the computers in this era are multithreaded based, so there are negligible technical issues.

REFERENCES

[1] Astrachanam O., Bubble Sort: An Archaeological Algorithmic Analysis, Duk University, 2003.
[2] Cormen T., Leiserson C., Rivest R. and Stein C., Introduction to Algorithms, McGraw Hill, 2001.
[3] Levitin A., Introduction to the Design and Analysis of Algorithms, Addison Wesley, 2007.
[4] Nyhoff L., An Introduction to Data Structures, Nyhoff Publishers, Amsterdam, 2005.
[5] G. Franceschini and V. Geffert, An In-place Sorting with O (n logn) comparisons and O(n) moves, In Proc. 44th Annual IEEE Symposium on Foundations of Computer Science, pages 242-250, 2003.
[6] Knuth D., The Art of Computer programming Sorting and Searching, 2nd edition, vol. 3. Addison – Wesley, 1998.
[7] J.L. Bentley and R. Sedgewick, Fast Algorithms for Sorting and Searching Strings, ACM-SIAM SODA" 97, 360-369, 1997.
[8] Box R. and Lacey S., A Fast Sort, Computer Journal of Byte Magazine, vol. 16, no. 4, pp. 315-315, 1991.
[9] Basti Shahzad and Muhammad Tanvir Afzal Enhanced Shell Sorting Algorithm. World Academy of Science, Engineering and Technology 27, 2007.