An Adaptive framework for Parallel Merge Sort Algorithm on Multicore Architecture

Ishwari Singh Rajput
Department of Computer Science and Engineering
Amity School of Engineering & Technology, Amity University, Noida, India

Deepa Gupta
Department of Computer Science and Engineering
Amity Institute of Information Technology, Amity University, Noida, India

Abstract- Sorting is among the most fundamental and well-studied problems within computer science and a core step of many algorithms. With the advancement in multi-core architectures many parallel sorting algorithms have been investigated which are implemented without a need for building interconnected machines. Present architectures have 2, 4, or 8 cores on a single die which demonstrates the benefits of parallelism at the chip level. In this paper, a comparative analysis of performance of three different types of sorting algorithms viz. merge sort, 3-way merge sort and parallel merge sort is presented. The merge sort algorithm to sort a sequence of n elements is based on divide and conquers approach of solving problems. The 3-way merge sort is also a sorting algorithm which follows divide and conquer approach and is an extension of simple merge sort. The comparative analysis is based on comparing average sorting time in parallel sorting over merge sort and 3-way merge sort. The time complexity for each sorting algorithm will also be mentioned and analyzed.

Keywords – Algorithm, merge sort, 3-way merge sort, and parallel sorting algorithms, parallel merge sort, performance analysis, multi core.

I. INTRODUCTION

Sorting is one of the core computational algorithms used in many scientific and engineering applications. All sorting algorithms are problem specific means they work well on some specific problem and do not work well for all the problems. Some sorting algorithms are applied to small number of elements, some sorting algorithms are suitable for floating point numbers, some are fit for specific range, some sorting algorithms are used for large number of data, and some are used if the list has repeated values. We sort data either in numerical order or lexicographical, sorting numerical value either in increasing order or decreasing order and alphabetical value like addressee key. Many sequential sorting algorithms consume O (nlogn) time to sort n keys [1].

Sorting [3, 4] is defined as the operation of arranging an unordered collection of keys or elements into monotonically increasing (or decreasing) order. Specifically, S= $\{a_1, a_2 \ldots\ldots\ldots\ldots a_n\}$ be a sequence of n elements in random order; sorting transforms S into monotonically increasing sequence S'= $\{a_1', a_2' \ldots\ldots\ldots\ldots\ldots a_n'\}$ such that $a_i' \leq a_j'$ for $1 \leq i \leq j \leq n$, and S' is a permutation of S.

Sorting has two different meanings ordering and categorizing, ordering means to order the list of same items and categorizing means grouping and labeling the same type of items [2].

Sequential sorting algorithms are classified into two categories. The first category, "distribution sort", is based on distributing the unsorted data items to multiple intermediate structures which are then collected and stored into a single sorted list. The second one, "comparison sort", is based on comparing the data items to find the correct relative order [3].We focus on comparison based sorting algorithms. These algorithms use various approaches in sorting such as exchange, partition, and merge. The exchange approach repeats exchanging adjacent data items to produce the sorted list as in case of bubble sort [4]. The partitioning approach is a "divide and conquer" strategy based on dividing the unsorted list into two sub-lists according to a pivot elements selected from the list of keys. The two sub-lists are sorted and then combined producing the sorted list as in case of quick sort [5].

Merge sort approach is also a divide and conquer strategy that does not depend on a pivot element in partitioning process. The approach repeatedly divides the original list into sub-lists until the sub-lists have only one data item. Then these elements are merged together given the sorted list as in case of merge sort [6, 7]. Merge sort is frequently employed in many applications.

A sorting algorithm [12] is said to be in-place whenever it does sorting with the use of constant extra memory. In this case, the amount of extra memory required to implement the algorithm does not depend on the number of keys in the input list. In addition, the sorting algorithm is stable if it keeps the indices of a sequence of equal values in the input list (in any of the input list) in sorted order at the end of sorting. Otherwise, the algorithm is said to be unstable.

In designing parallel sorting algorithms, the fundamental issue is to collectively sort data owned by individual processors in such a way that it utilizes all processing units doing sorting work, while also minimizing the costs of redistribution of keys across processors. In parallel sorting algorithms there are two places where the input and the sorted sequences can reside. They may be stored on only one of the processor, or they may be distributed among the processors. Parallel merge sort on PRAM model was reported to have fast execution time of O (logN) for n input keys using N processors [8].

Instead of using multiprocessors to achieve parallelism, multi-core architectures are used to implement and executes parallelized applications.

II. MERGE SORT ALGORITHM

Merge sort [12] is a standard sorting strategy that can be implemented in various ways: on lists, in a recursive top-down fashion or in an iterative bottom-up fashion. Till recently, the analysis was largely confined to dominating terms, the average-case complexity (i.e., number of key comparisons) being nlog2n + O (n) for all versions of the algorithm. Merge sort is an efficient 2-way divide-and-conquer sorting algorithm as it divides the original sequence of keys into two subsequences until each subsequence contains single element each. Because merge-sort is easier to understand than other useful divide-and-conquer methods, it is often considered to be a typical representative of such methods, and frequently used to introduce the divide-and-conquer approach itself [9].

Intuitively, merge sort operates on sequence of n objects as follows:

- If n > 1, divide the sequence into two subsequences of about half the size each;
- apply merge sort on each subsequence;
- Merge the two sorted subsequences from step 2 into one sorted array.

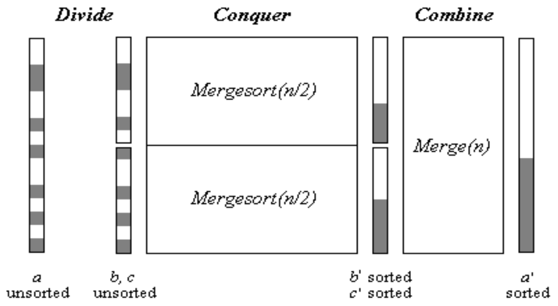

Figure 1. Graphical representation of the Merge sort algorithm

Pseudo Code:

Input: Array Arr [low...high], indices low, mid, high (low ≤ mid <high). Arr [low...high] is the array to be divided. Arr [low] is the beginning element and Arr [high] is the ending element

Output: Array Arr [low...high] in ascending order

.

```
private void MergeSort(T[] to, T[] temp, int low, int
high)
{
if (low >= high)
return;
var mid = (low + high) / 2;
MergeSort (temp, to, low, mid);
MergeSort (temp, to, mid + 1, high);
Merge (to, temp, low, mid, mid + 1, high, low);
}
```

Figure 2. Pseudo code for Merge Sort function

For small arrays, some implementations switch from recursive merge sort to non-recursive methods, such as insertion sort – an approach that is known to improve execution time.

Complexity Analysis

The average time complexity of merge sort is $O(n \log_2 n)$ [10], the same as quick sort and heap sort. In addition, best-case complexity of merge sort is only O(n), because if the array is already sorted, the merge operation perform only O(n) comparisons; this is better than best case complexity of both quick sort and heap sort. The worst case complexity of merge sort is $O(n \log_2 n)$ [10], which is the same as heap sort and better than quick sort. However, classical merge sort uses an additional memory of n elements for its merge operation (the same as quick sort), while heap sort is an in-place method with no additional memory requirements.

The average/best/worst asymptotic complexity of merge sort is at least as good as the corresponding average/best/worst asymptotic complexity of heap sort and quick sort; despite of this, merge sort is often considered to be slower than the other two in practical implementations. On the positive side, merge sort is a stable sort method, in contrast to quick sort and heap sort, which fail to maintain the relative order of equal objects. The practical performance of merge sort is known to improve with recursion removal and cache memory utilization [11].

For simplicity, assume that n is a power of 2 and each divide step yields two subsequences, both of size approximately n/2 [9].

The base case occurs when n = 1.
When $n \geq 2$, following are the merge sort steps:

Divide: Just compute mid as the average of low and high
Therefore, D (n) = Θ (1).

Conquer: Recursively solve 2 subsequences, each of size n/2
i.e. 2T (n/2).

Combine: MERGE on an n-element sub array takes Θ (n) time Therefore, C(n) = Θ (n).

As the general form of recursion is:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$
$$where\ f(n) = D(n) + C(n)$$

D(n) : cost of dividing the problem into sub-problems.
C(n) : cost of combining sub-solutions into original solution.
a: Number of sub arrays in which the original array is to be divided.
b: size of each sub array.

The recurrence relation for merge sort reduces to:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n)$$

$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ if n>1 …………………(i)

$T(n) - \Theta(1)$ if n=1(Base Case)

Equation (i) can also be written as $T(n) = 2T\left(\frac{n}{2}\right) + cn$ for some constant c.

The recursion tree looks like this.

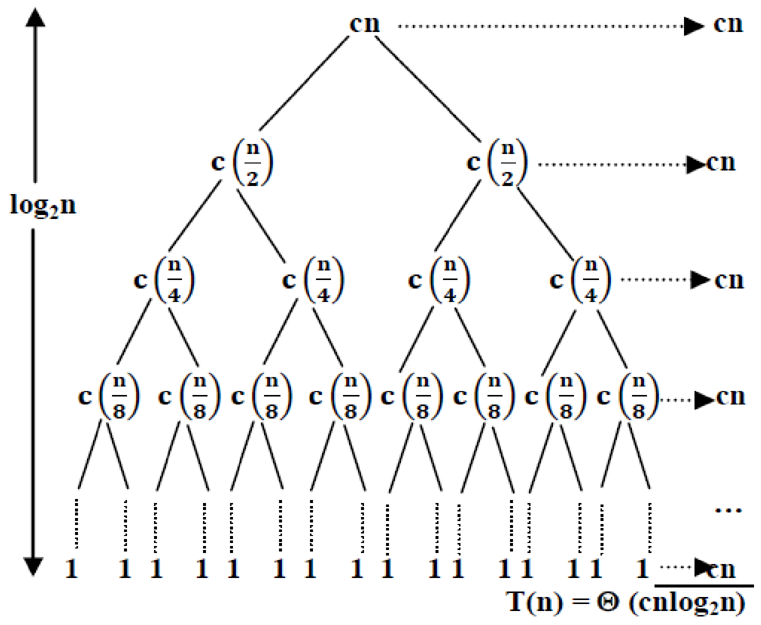

Figure 3. Recursion tree for merge sort algorithm

Therefore, the running time for merge sort is:
$T(n) = cnlogn$ i.e. $T(n) = \square(nlog_2n)$

III. 3-WAY MERGE SORT

3-way merge sort is an efficient divide and conquer sorting algorithm. It is an extension of simple merge sort algorithm. The sequence is divided into 3 equally sized subsequences and the generated sub-lists are further divided until each number is obtained individually. The numbers are then *merged* together as pairs to form sorted subsequences of length 3. The subsequences are then merged subsequently until the whole sequence is constructed.

3-way merge sort operates on a sequence of n objects as follows:

- If n > 1, divide the array into three sub-arrays of about one-third the size each;
- apply merge sort on each sub- array;
- Merge the three sorted sub-arrays from step 2 into one sorted array.

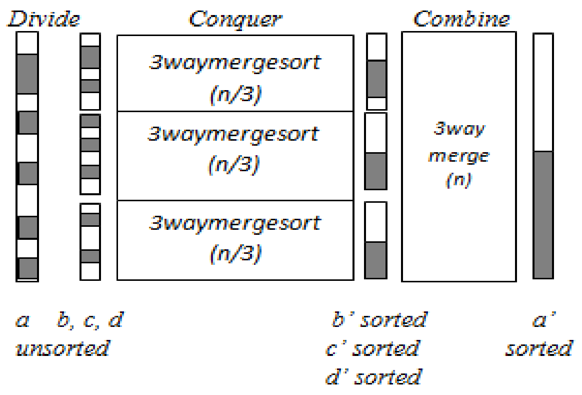

Figure 4. Graphical representation of the 3-way Merge sort algorithm

Pseudo Code

Input: Array Arr [low...high], indices low, mid1, mid2, high (low ≤ mid1<mid2 <high). Arr [low...high] is the array to be divided. Arr [low] is the beginning element and Arr [high] is the ending element

Output: Array Arr [low...high] in ascending order.

```
private void ThreeWayMergeSort (T[] to, T[] temp, int low, int
high)
{
if (low >= high)
return;
var mid1 = (high-low) / 3;
var mid2=2*mid1;
ThreeWayMergeSort (temp, to, low, mid1);
ThreeWayMergeSort (temp, to, mid1 + 1, mid2);
ThreeWayMergeSort (temp, to, mid2 + 1, high);
ThreeWayMerge (to, temp, low, mid1, mid2, high);
}
```

Figure 5. Pseudo code for 3-way Merge Sort function

Complexity Analysis

The average complexity of 3-way merge sort is same as that of basic merge sort i.e. of $O(n \log_2 n)$, the same as quick sort and heap sort. But the actual average running time of 3-way merge sort is less due to the reduced height ($\log_3 n$) of recursion tree. 3-way merge sort performs better for some extent due to reduced number of calls to ThreeWayMergeSort function.

For simplicity, assume that each divide step yields three sub-problems, each of size exactly n/3.

The base case occurs when $n = 1$.
When $n \geq 3$, following are the 3 way-merge sort steps:

Divide: Just compute q as the average of p and r. Therefore, $D(n) = \Theta(1)$.

Conquer: Recursively solve 3 sub-problems, each of size n/3
i.e. $3T(n/3)$.

Combine: MERGE on an n-element sub array takes $\Theta(n)$ time Therefore, $C(n) = \Theta(n)$.

Here a and b has value 3 as the array is partitioned into three sub arrays and each has size n/3. Therefore the recurrence relation for 3-way merge sort is:

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{n}{3}\right) + T\left(\frac{n}{3}\right) + \Theta(n)$$

$$T(n) = 3T\left(\frac{n}{3}\right) + \Theta(n) \qquad \text{if } n>1 \text{ …………………(i)}$$

$$T(n) = \Theta(1) \qquad \text{if } n=1 \text{(Base Case)}$$

Equation (i) can also be written as $T(n) = 3T\left(\frac{n}{3}\right) + cn$ for some constant c.

The recursion tree looks like this.

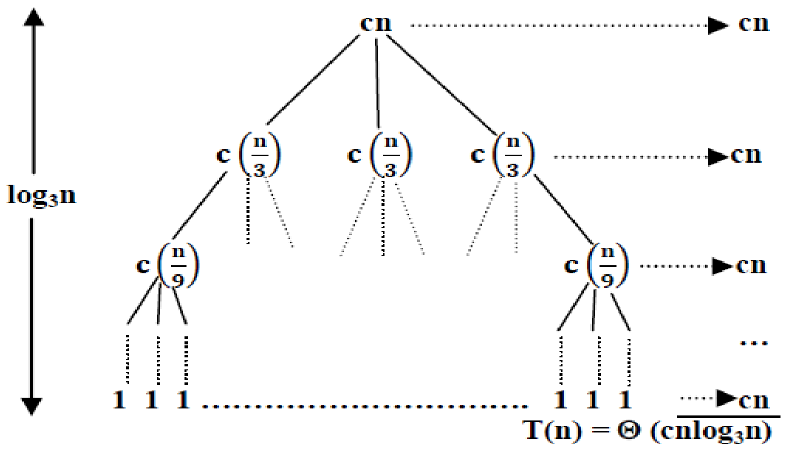
cn
cn
$c\left(\frac{n}{3}\right)$
cn
$\log_3 n$
$c\left(\frac{n}{9}\right)$
cn
...
1 1 1 ... 1 1 1
cn
$T(n) = \Theta(cn\log_3 n)$

Figure 6. Recursion tree for 3-way merge sort algorithm

Therefore, the running time for merge sort is:

$T(n) = cnlogn$ i.e. $T(n) =$ □$(n\log_3 n)$

IV. PARALLEL MERGE SORT

Advances in multi-core architectures are showing great promise at demonstrating the benefits of parallelism at the chip level. Current architectures have 2, 4, or 8 cores on a single die, but industry insiders are predicting orders of magnitude larger numbers of cores in the not too distance future [13, 14, 15].

Chip Multiprocessors (CMP) [16] is a multithreaded architecture, which integrates more than one processor on a single chip. In this architecture, each processor has its own L1 cache. The L2 cache and the bus interface are shared among processors. Intel Core 2 Duo [17] is an example of such architecture; it has two processors on a single chip, each of them has an L1 cache, and both of them are sharing the L2 cache

These architectures not only provide a facility for implementing and running the parallelized applications without a need for building interconnected machines but also enhance the data management operations among parallel processes due to the reliable utilization of hardware resources

Multi-core architectures are designed to provide a high performance feature on a single chip since that they do require neither a complex system nor increased power requirements [18]

On the other hand, many parameters such as latency, bandwidth, caches and even the system software [19] affect the performance of such systems. These challenges should to be studied to gain the objective of these architectures

In parallel merge sort a large array is partitioned into equal parts and efficient sorting functions is applied on sub arrays in parallel, then parallel execution results in faster processing. It takes less time to merge all sorted arrays that have been processed quickly in separate thread in parallel.

Most basic construct for task parallelism is: Parallel.Invoke (DoLeft, DoRight);
It executes the methods DoLeft and DoRight in parallel, and waits for both of them to finish. Invoke itself is a synchronous method, it will only return when it has executed all tasks. This method provides a simple way in which a number of tasks may be created and executed in parallel. As with other methods in the Parallel Task Library, Parallel.Invoke provides potential parallelism. If no benefit can be gained by creating multiple threads of execution the tasks will run sequentially.

Pseudo code
Input: Array Arr [low...high], indices low and high (low $\leq$ mid <high). Arr [low...high] is the array to be divided. Arr [low] is the beginning element and Arr [high] is the ending element
Output: Array Arr [low...high] in ascending order

```
private void ParallelMergeSort(T[] to, T[] temp, int low, int high, int
depth)
{
if (high - low + 1 <= SEQUENTIAL_THRESHOLD || depth <= 0)
  {
    MergeSort (to, temp, low, high);
    return;
   }
  var mid = (low + high) / 2;
  depth--;
  Parallel.Invoke (
  () => ParallelMergeSort (temp, to, low, mid, depth),
  () => ParallelMergeSort (temp, to, mid + 1, high, depth)
  );
  ParallelMerge (to, temp, low, mid, mid + 1, high, low, depth);
}
```

Figure 7. Pseudo code for Parallel Merge Sort function

Parallel.Invoke construct executes the Parallel merge sort on two sub arrays, partitioned around mid element in a parallel manner. Merging of two sub arrays can also be performed by invoking Parallel.Invoke for parallel execution which is as follows:

```
private void ParallelMerge(T[] to, T[] temp, int lowX, int highX, int lowY, int highY, int
lowTo, int depth)
 {
  ……………………
  ……...…………….
if (lengthX < lengthY)
{
ParallelMerge(to, temp, lowY, highY, lowX, highX, lowTo, depth);
return;
}
  …………………….
  …………………….
Parallel.Invoke(
() => ParallelMerge(to, temp, lowX, midX - 1, lowY, midY - 1,
lowTo, depth),
() => ParallelMerge(to, temp, midX + 1, highX, midY, highY, midTo + 1, depth)
);
}
```

Figure 8. Pseudo code for Parallel Merge Sort function

V. CASE STUDY

Consider a sequence S, consisting of 8 unordered data elements given as:
S= {91, 75, 64, 15, 21, 8, 88, 54}
This data set is sorted by the two, above mentioned sorting algorithms viz. merge sort, three-way merge sort.

5.1 Sorting by Merge sort

Perform the steps as mentioned in the pseudo code for merge sort to sort the data set as follows:

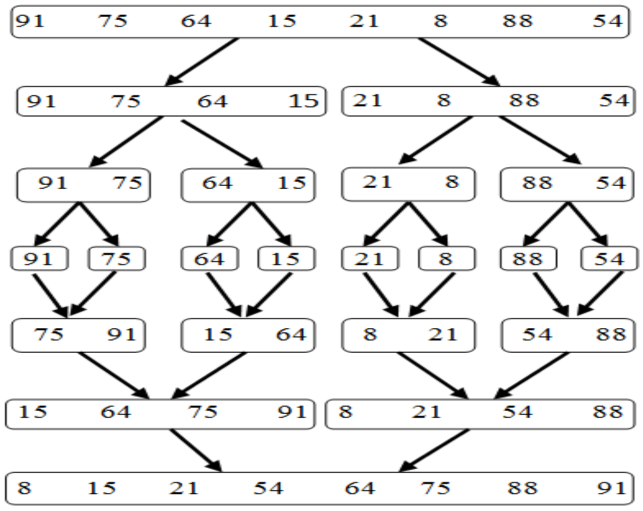

Figure 9. Sorting by Merge Sort

Firstly, the input sequence is partitioned into two subsequences of equal size and the procedure repeats it recursively until each subsequence contains a single element. Then the subsequences are merged together stepwise until sorted sequence is obtained.

5.2 Sorting by 3-way merge sort

Perform the steps as mentioned in the pseudo code for 3-way merge sort to sort the data set as follows:

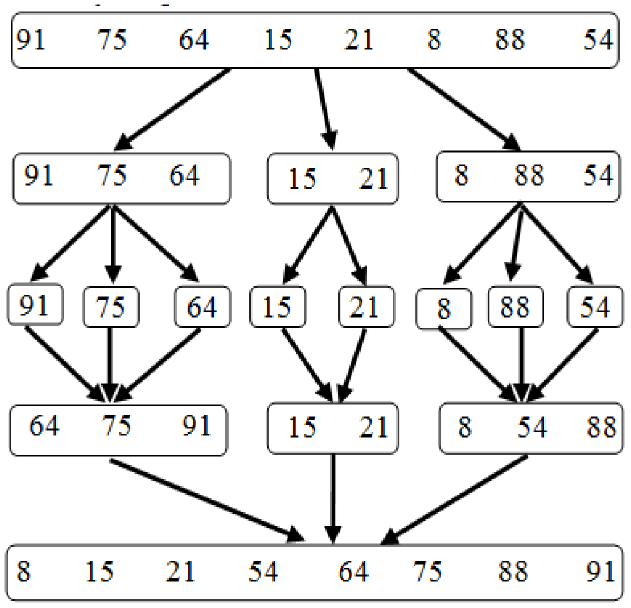

Figure 10. Sorting by Merge Sort

Firstly, the input sequence is partitioned into three subsequences of equal size and the procedure repeats it recursively until each subsequence contains a single element. Then the subsequences are merged together stepwise until sorted sequence is obtained.
It is observed that the height of the tree formed is less as compared to simple merge sort which reduces the average running time to some extent. 3-way performs better as number of input keys increases.

VI. COMPARISON OF RESULTS AND DISCUSSION

The performance of above mentioned algorithms can be analyzed by considering the average running time on 2-chip processor. Table1 shows the average running time (in seconds) of merge sort, 3-way merge sort and parallel merge sort with respect to increasing number of input values. It shows that parallel merge sort using Parallel.Invoke perform better over merge sort and 3-way merge sort, due to the use of parallelism and proper utilization of both cores of CPU.

Table1. Average Running Time of Merge sort, 3-way Merge Sort and Parallel Merge Sort

Input Size (n)	**Average Running Time(ms)**		
	Merge sort	**3-way Merge Sort**	**Parallel Merge Sort**
0.1 X 10^6	0.0592	0.0556	0.0419
3 X 10^6	1.5456	1.4745	0.9753
6 X 10^6	3.7465	3.5561	2.2787
9 X 10^6	5.7134	5.2987	3.3112
12 X 10^6	7.8280	7.3174	4.5527
15 X 10^6	9.8671	9.1287	5.8604
18 X 10^6	11.9538	11.0550	6.9699
21 X 10^6	14.0710	12.8127	8.5667
24 X 10^6	16.2288	14.8221	9.4809
27 X 10^6	18.3616	16.8638	10.7920

Figure 11 shows that the parallel merge sort performs better over merge sort as the tasks get executed in a parallel fashion on multiple cores.

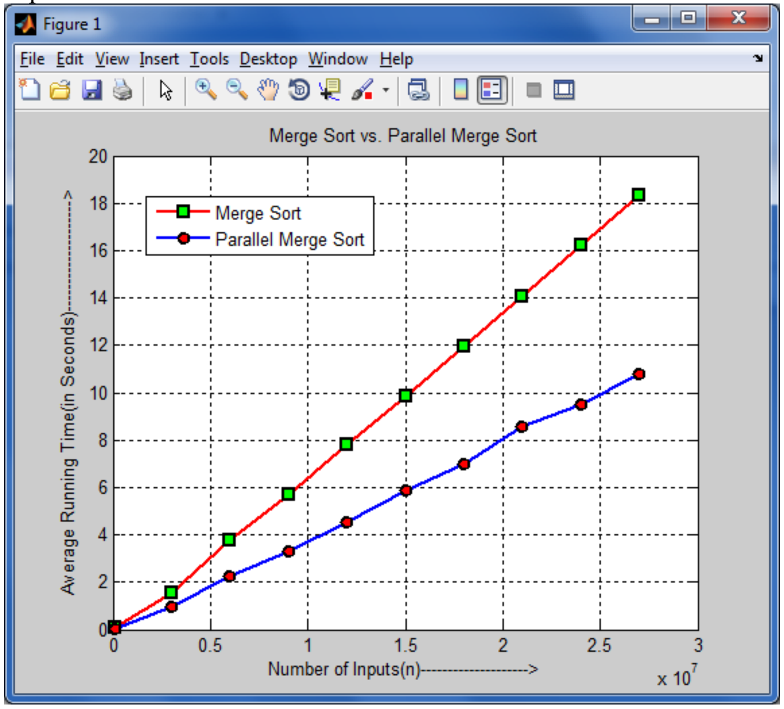

Figure 11. Merge Sort vs. Parallel Merge Sort

Fig. 12 shows the better performance of parallel merge sort over 3-way merge sort. 3-way merge sort has higher average running time than parallel merge sort due to the absence of parallelism.

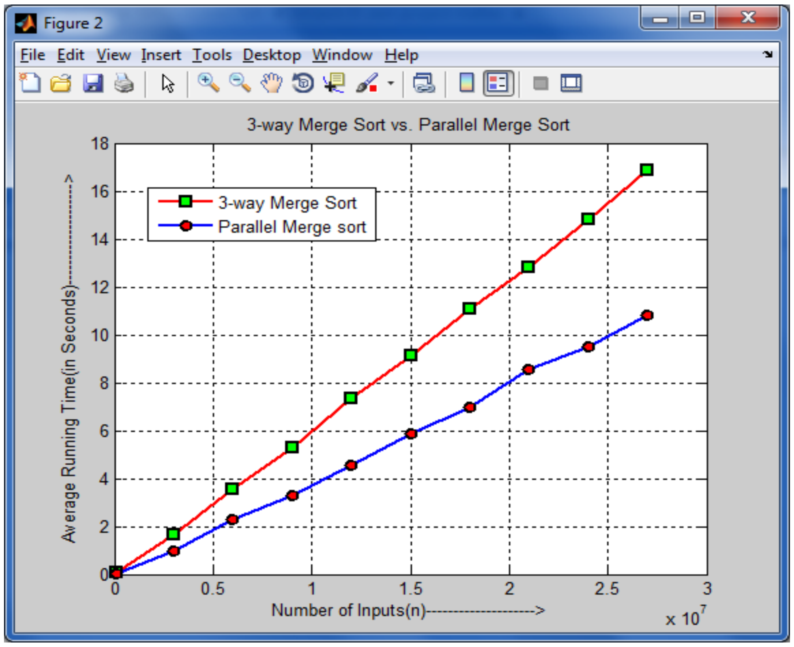

Figure 12. 3-way Merge Sort vs. Parallel Merge Sort

Figure 13 shows the comparison of average running time of all three sorting algorithms viz. merge sort, 3-way merge sort and parallel merge sort with increasing number of input size i.e. n. Figure shows that the parallel merge sort is best of all three sorting algorithms due to the use of parallelism.

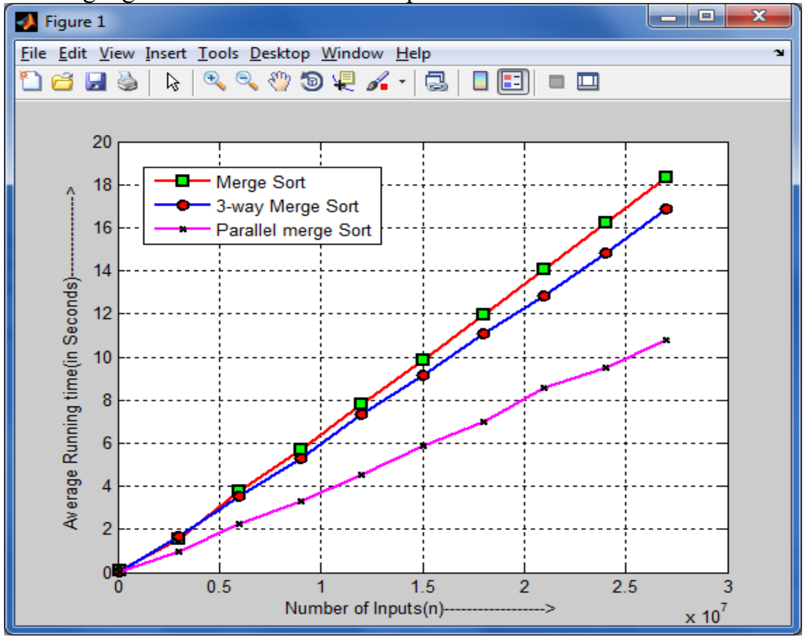

Figure 13. Merge Sort, 3-way Merge Sort and Parallel Merge Sort

VII.CONCLUSION

In this paper we considered the sorting problem for large data sets, and three sorting algorithms are compared successfully. The effect of the number of cores on the performance of merge sort has been theoretically and experimentally studied.

The basis of analysis is the average running time on double core processor. It is observed that parallel sorting algorithm i.e. parallel merge sort performs well in all respects in comparison to sequential merge sort and 3-way merge sort. The better performance is obvious because parallel sorting algorithm take the advantage of parallelism to reduce the average running time. In future, same analysis can be performed with parallel sorting algorithms (parallel quick sort and hyperquicksort) and parallel sorting by regular sampling algorithm (PSRS) for wide variety of MIMD architectures and the processors with more than two cores.

REFERENCES

[1] Minsoo Jeon and Dongseung Kim, Load-Balanced Parallel Merge Sort on Distributed Memory Parallel Computers, IEEE, 2002.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 1990. ISBN 0-262- 03293-7. Chapter 27: Sorting Networks, pp.704–724S

[3] L. Rashid, W. Hassanein, M. Hammad "Analyzing and enhancing the parallel sort operation on multithreaded architectures" Journal of Supercomputing, vol. 53, no. 2, pp 293-312, 2010.

[4] O. Astrachan, "Bubble sort: An Archaeological Algorithmic Analysis" ACM SIGCSE Bulletin, vol. 35 no. 1, 2003.

[5] P. Tsigas and Yi. Zhang " A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on Sun Enterprise 10000" Proceedings of the 11th EUROMICRO Conference on Parallel Distributed and Network-Based Processing (PDP). pp. 372 – 381, 2003.

[6] A. LaMarca and R. E. Ladner "The influence of caches on the performance of sorting" Proceedings of 8th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA97), pp.370–379, 1997.

[7] B.R. Lyer and D.M. Dias "System Issues in Parallel Sorting for Database Systems" Proceedings of the International Conference on Data Engineering, pp. 246-255, 2003.

[8] R. Cole, “Parallel merge sort,” SIAM Journal of Computing, vol. 17, no. 4, 1998, pp.770-785.

[9] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. Introduction to Algorithms (3rd ed.), MIT Press, 2009.

[10] Katajainen, Jyrki; Träff, Jesper L. A meticulous analysis of mergesort programs. Lecture Notes in Computer Science, 1997, Volume 1203/1997, 217-228.

[11] LaMarca, Anthony; Ladner, Richard. The influence of caches on the performance of sorting. Proc. 8th Ann. ACM-SIAM Symposium on Discrete Algorithms (SODA97), 370–379.

[12] D. E. Knuth, The Art of Computer Programming, Vol. III|Sorting and Searching, Addison- Wesley, 1973.

[13] D. Geer. Chip Makers Turn to Multicore Processors. IEEE Computer, 38(5):11–13, 2005.

[14] G. Lowney. Why Intel is designing multi-core processors. https://conferences.umiacs.umd.edu/ paa/lowney.pdf.

[15] J. Rattner. Multi-Core to the Masses. Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on, pages 3–3, 2005.

[16] L. Hammond, B. Nayfeh and K. Olukotun " A Single-Chip Multiprocessor" IEEE Computer, vol. 30 no. 9, pp 79-85, 1997.

[17] Intel Core2Duo.URL: http://www.intel.com/products/processor/core2duo/index.htm

[18] M. Kistler, M. Perrone, F. Petrini "Cell Multiprocessor Communication Network:

[19] Built for Speed," Micro, IEEE , vol.26, no.3, pp. 10- 23, May-June 2006.

[20] R. Kumar, V. Zyuban and D. Tullsen "Interconnections in multi-core architectures: understanding mechanisms, overheads and scaling" ISCA '05 Proceedings 32nd International Symposium on, Computer Architecture 2005.