Threshold Analysis and Comparison of Sequential and Parallel Divide and Conquer Sorting Algorithms

Tinku Singh
Department of Computer Science and Engineering
BRCM, College of Engineering and Technology,
Bahal, Haryana, India

Durgesh Kumar Srivastava
Department of Computer Science and Engineering
BRCM, College of Engineering and Technology, Bahal,
Haryana, India

ABSTRACT

One of the basic problems in computer science is sorting that need to be fast and efficient, since data is growing day by day. Various applications need fast sorting algorithms like Big Data analyses particularly in large scale scientific, social/web mining and commercial application domains. Divide and conquer Sorting Algorithms (Quick sort and merge sort) provides the best running time among all the sorting algorithms. When parallelism is applied to these algorithms, new performance leaps are accomplished. Recent parallel programming procedures and environment needs profound changes in programs to accomplish parallelism furthermore constitute puzzling, confounding and mistake inclined constructs and standards. When the number of processors utilization gets large, the overhead of thread synchronization and processor scheduling gets increase, this diminishes the speedup. In this paper, two algorithms are designed using C# viz. parallel quick sort and parallel merge sort that uses Parallel.Invoke() method. Both algorithms when executed over multicore architecture compute the threshold beyond which the above mentioned algorithms achieve speedup in comparison to its sequential version, Also threshold is calculated and compared for both the algorithms for uneven input size.

Keywords

Parallel, threshold, multicore, speedup, sorting, complexity, processor.

1. INTRODUCTION

1.1 Divide and conquer paradigm

Divide and conquer [1] is a design perspective that works with multi branch recursion. Since recursion is utilized as a part of divide and conquer paradigm for solving subproblems, so it's a need that each subproblem should be smaller enough in compared to the original problem and there should be a base case for subproblems. A problem is broken up into small subproblems of the same kind using the recursion; this practice is repeated until problem gets to be adequately basic to be explained easily as shown in figure 1. The solution of these small arrangements is done and it is joined to give arrangement of the first problem. Divide-and-conquer algorithms are finished in three sections:

- **Division of the problem** into a quantity of subproblems those are smaller in size of the original problem.
- **Conquer** the subproblems subsequent to illuminating them utilizing recursion. If the subproblem is small enough, solve this as base case.
- **Combine** the solutions to these subproblems to find the solution for the original problem.

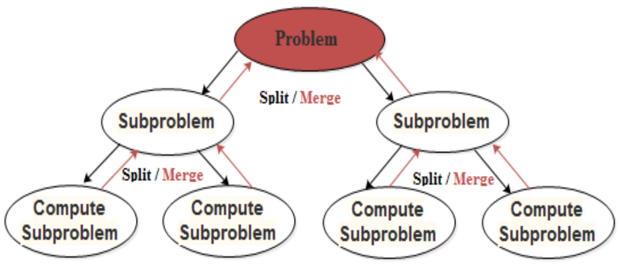


Fig 1: Divide and Conquer paradigm

Divide and conquer sorting algorithms are the best sorting algorithms(quick sort and merge sort) among all comparison based available sorting algorithms in terms of running time.

1.2 Parallelism in Programming-

As there is the limitation for achieving the CPU clock speed, manufacturers moved towards increasing the core counts. But the standard single-threaded code will not be able to utilize the CPU cores.

The server applications can easily handle the multiple cores assignment for the majority of the server applications, when a client sends the request; it can be separately handled by a thread. In the desktop applications however it is difficult - in light of the fact that computationally rigorous code usually requires that you do the following:

1. Partition the code into small partitions.
2. As a part of the parallel Execution, assign these small partitions to multiple threads.
3. Results of the parallel executions are gathered as it will be accessible, in a thread-safe manner.

"The idea of parallel programming comes from the multithreading that strengths multicore or various processors"

There work among the threads can be partitioned by two ways: task parallelism and data parallelism.

1.2.1 Task parallelism-

It is the type of unstructured parallelism. All the instructions of your program are not parallel. Parallel work is sprinkled over the complete program.

1.2.2 Data parallelism-

It's a form of parallelism where, same task is performed on different data items. It is a type of structured parallelism.

When, structured parallelism and unstructured parallelism are compared, structured parallelism is easy and less erroneous, it explains how to perform partitioning and it also has better techniques for thread coordination.

In this paper, parallel divide and conquer sorting algorithms (quick sort and merge sort) are presented and compared for performance with their sequential version performance. On comparing their performances, it is found that the size of the array that is called threshold value T, at which the parallel algorithm becomes slower than their sequential version. Because of load imbalance in parallel applications due to various reasons like- parallelism overhead, thread creation, time spent at synchronization, thread communication, granularity of task decomposition. This provides enough data to draw a conclusion about the threshold in performance when using the parallel sorting algorithm.

2. QUICKSORT

British computer scientist Tony Hoare in 1959/1960 developed Quicksort, it is a sorting technique that is based on divide and conquer paradigm [2]. The implementation of simple Quicksort algorithm is as follows:

- Choose an element from the array. It is said to be pivot element. Generally the last element out of the sorting section is selected.
- Iterate through the sorting section; place all numbers smaller to the pivot to a position on its left and all other numbers to the position on its right. This is achieved by swapping the elements.
- The pivot element is in sorted position after the iteration and this process carry on using recursion with the divide-and conquer approach, same approach is followed on the left subpart and right subpart, until the complete array is sorted.

For sorting n number of elements, the number of comparison made by quicksort will be O(nlog2n) in average case and it makes O(n2) comparisons, in the worst case, Although it occurs rarely. In practical aspects it is faster than other O(nlogn) algorithms [3]

2.1 Parallel Quicksort

In parallel quicksort, It is assumed that system has distributed memory [3]. Unsorted list is distributed by applying some approach of distribution on the threads.

Parallel quicksort algorithm is expected to produce the following result:

- The array stored on each subprocess is sorted.
- The last element on process *i*'s array is smaller than the first element on process *i + 1*'s array.

The first element is chosen as pivot element from the first process and places all the numbers smaller to the pivot to a position on its left and all other numbers to the position on its right. Now this process is divided into two sub processes using Parallel.Invoke() method in C#, to work parallel. This process is continued by applying the same algorithm on the left subpart and right subpart recursively as shown in figure 2. After $\log_2 P$ recursions, every subprocess has an unsorted list of values completely disjoint from the values held by the other sub processes. The largest value on subprocess *i* will be smaller than the smallest value held by subprocess *i + 1*.

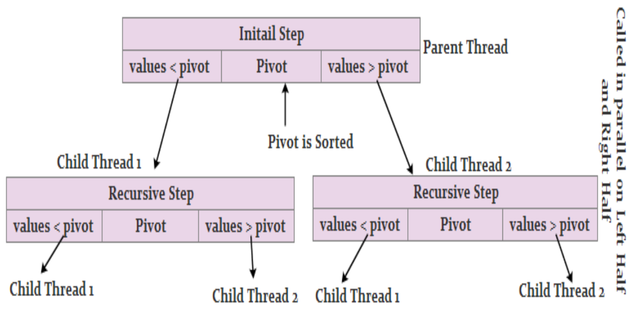


Fig 2: parallel quicksort

```
public  void QuickSortParallel<K>(K[] input, int leftOfArray,
int rightOfArray)  where K : IComparable<K>
        {
          if (leftOfArray >= rightOfArray)
          {
              return;
          }
  Swap(input, leftOfArray, (leftOfArray + rightOfArray) / 2);
          int lastElement = leftOfArray;
          for (int current = leftOfArray + 1; current <=
rightOfArray; ++current)
{
 if (input[current].CompareTo(input[leftOfArray]) < 0)
    {
     ++lastElement;
      Swap(input, lastElement, current);
     }
}
 Swap(input, leftOfArray, lastElement);
 Parallel.Invoke(
              () => QuickSortParallel (input, leftOfArray,
lastElement - 1),
              () => QuickSortParallel (input, lastElement + 1,
rightOfArray)
          );
}
Public void Swap<K>(K[] inputArr, int index1, int index2)
        {
          K temp = inputArr[index1];
           inputArr[index1] = inputArr[index2];
           inputArr[index1] = temp;
        }
```

3. MERGESORT

Mergesort is recursive algorithm that works on divide-and-conquer approach [3], it always partition the input array into 2 equal parts, this process partitioning continues until each sub array contains one element. Recursion is used for splitting the array into two equal arrays [4].

For a sequence of n items MergeSort works as follows:

Base case: If the array has more than one element n > 1, the array is splitted into two equal halves and merge sort is recursively called on them.

Merge: Both the sub arrays are **merged** during the merge step, into a single sorted array. As shown in figure-2, merging is the procedure of taking two smaller arrays and combining them together into a single, sorted array as shown in figure 3.

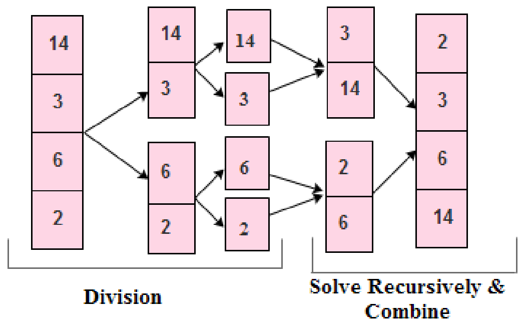


Fig 3: MergeSort with recursion

Recurrence relation for the mergesort based on the recursion tree shown in figure 4, is-

$T(n) = 2T(n/2) + cn$

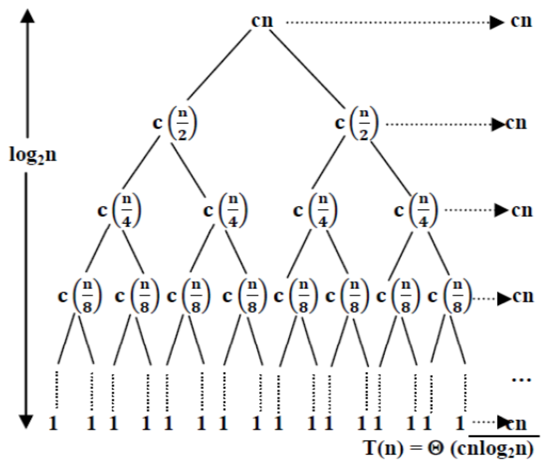

cn
cn
$\log_2 n$
$c\left(\frac{n}{2}\right)$
$c\left(\frac{n}{4}\right)$
$c\left(\frac{n}{8}\right)$
1
$T(n) = \Theta(cn\log_2 n)$

Fig 4: Recursion tree for Mergesort

Therefore, the running time for mergesort is: $\mathbf{O(n\log_2 n)}$

3.1 Parallel Mergesort

In parallel merge sort an array is partitioned into two equal parts and efficient sorting functions is applied on sub arrays in parallel [5], Most basic construct for the parallelism is: Parallel.Invoke() (Threading.Tasks);

Using Threading.Tasks two tasks are passed for the left subpart and right subpart in the parallel for the sorting, and wait for both of them to finish. Invoke is a synchronous method, it will return when it has executed all tasks. Invoke() method is used to create a number of tasks and execute them in parallel. Parallel.Invoke() offers promising parallelism, when used other methods in the Parallel Task Library,

Pseudo code

Input: Array A [starting...ending], indices starting and ending (ending >=mid >= starting). Arr [starting...ending] is the input array to be divided.

A [start] is the beginning element and A [ending] is the ending element

Output: Array A [starting...ending] in ascending order

```
Public void MergeSort_Parallel(k[] myArr, T[] temporary, int
begining, int ending, int coreCount)
{
if (ending - begining + 1 <= SEQUENTIAL_THRESHOLD ||
coreCount <= 0)
{
MergeSort (myArr, temporary, begining, ending);
return;
}
var mid = (begining + ending) / 2;
coreCount--;
Parallel.Invoke (
() => MergeSort_Parallel (temporary, myArr, begining, mid,
coreCount),
() => MergeSort_Parallel (temporary, myArr, mid + 1,
ending, coreCount)
);
Merge_Parallel (myArr, temporary, begining, mid, mid + 1,
ending, begining, coreCount);
}
```

Parallel.Invoke() is used for parallel execution of merging of two arrays. This is as follows:

```
public void Merge_Parallel(T[] myArr, T[] temporary, int
beginingX, int endingX, int beginingY, int endingY, int
beginingMyArr, int coreCount)
    {
      ..................
      ..................
      if (lengthX < lengthY)
      {
         Merge_Parallel(myArr, temporary, beginingY,
endingY, beginingX, endingX, beginingMyArr, coreCount);
      return;
      }
      var midX = (beginingX + endingX) / 2;
      var midY = BinarySearch(temporary, beginingY,
endingY, temporary[midX]);
              ..................
      ..................
      Parallel.Invoke(
      () => Merge_Parallel(myArr, temporary, beginingX,
midX - 1, beginingY, midY - 1,beginingMyArr, coreCount),
        () => Merge_Parallel(myArr, temporary, midX + 1,
endingX, midY, endingY, midMyArr + 1, coreCount)
            );
    }
```

4. RELATED WORK

Multicore processor models are intended to boost execution and minimize heat yield by coordinating two or more processor centers into a solitary processor socket [6]. Parallel programming can exploit multicore innovation. Current structures have 2, 4, or 8 centers on a solitary processor, however industry insiders are anticipating requests of greatness bigger quantities of centers in the not very separation in future.

It is thought that dual core processor will have two times faster execution speed as compared to the single core processor [7]. But answer is no. A processor with two cores is one and half time more effective than single core processor. Execution speed increases about fifty percent.

Chip multiprocessors - additionally called multicore microchips or CMPs is a multithreaded design, which incorporates more than one processor on a solitary chip [8]. In this engineering, every processor has its own L1 cache. The L2 cache and the bus interface are shared among processors. Intel Core 2 Duo is a case of such design that is shown in figure 5; it has two processors on a solitary chip, each of them has a L1 cache, and both of them are sharing the L2 cache [9]. These models not just give a facility to executing and running the parallelized applications without a requirement for building interconnected machines additionally improve the information administration operations among parallel procedures because of the solid usage of hardware resources.

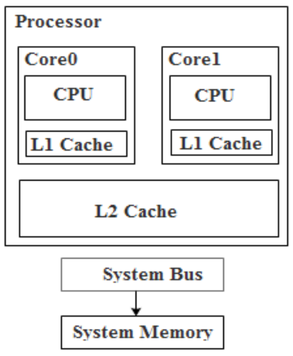


Fig 5: Core 2 Duo processor

4.1 Parallel with multicore

Higher performance can be achieved by executing parallel code on multiple cores, in comparison to single core processor [10]. Work is distributed among multiple cores using multithreading. In multicore CPU, a wide range of applications can be executed in parallel more efficiently because of low inter-processor communication latency between the cores.

Large uniprocessors are no more scaling in execution, since it is just conceivable to remove a restricted measure of parallelism from an average guideline stream utilizing usual superscalar direction issue techniques. On the other hand, numerous parameters, for example, transfer speed, latency, caches and even the framework programming influence the execution of such systems [11]. Parallel machines are produced using ware processors and information parallelism is not a decent model when the code has bunches of branches

The vital source of wastefulness in parallel codes:

- Parallelism overhead.
- Thread synchronization, correspondence and creation.
- Load irregularity because of various measures of work across the processors.
- Communication and computation.
- Time spent at synchronization is high and is uneven over processors, however not generally so straightforward
- Task conditions - Can all tasks be keep running in any request (counting parallel)?

With the parallel algorithms Because of load imbalance due to various reasons like- parallelism overhead, Thread creation, Time spent at synchronization, thread communication, extra cost of creating, monitoring and managing of the parallel tasks is added to the total computational cost [11].

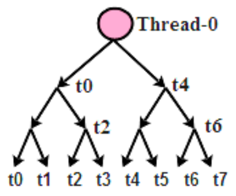


Fig 6: Division of work into threads

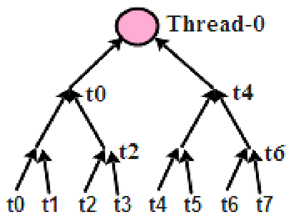


Fig 7: Combining the result of child threads into a parent thread.

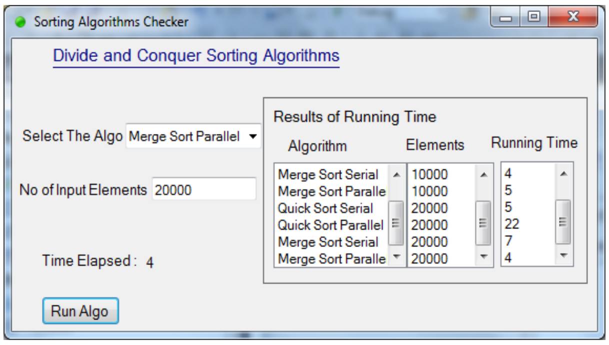
Based on these assumptions it can be said that if divide and conquer sorting algorithms are solved in parallel using the following steps-

- Division of an Array into multiple sub arrays using multithreading as shown in figure 6.
- Combining these sub arrays into a single array after solving them recursively as shown in figure 7.

Complete process takes extra computational time because of multiple thread creation and synchronization. This extra time is too costly for the small size of array that parallel version of divide and conquer sorting algorithms takes more time than their sequential version.

5. EXPERIMENTAL RESULT

A Graphical user Interface(GUI) based application is designed using Visual C# that will calculate the running time of the different sorting (Quicksort, parallel quick sort, merge sort and parallel merge sort) algorithms. The results for the running time along with Algorithm name and number of elements will be stored in the separate list box as shown in figure 8. Scientific lab (scilab) is application software that is used in this paper for drawing the graphs for the results.


Sorting Algorithms Checker
Divide and Conquer Sorting Algorithms
Run Algo

Select The Algo: Merge Sort Parallel
No of Input Elements: 20000
Time Elapsed : 4

Results of Running Time

Algorithm	Elements	Running Time
Merge Sort Serial	10000	4
Merge Sort Paralle	10000	5
Quick Sort Serial	20000	5
Quick Sort Parallel	20000	22
Merge Sort Serial	20000	7
Merge Sort Paralle	20000	4

Figure 8: GUI tool for running the sorting algorithms

Table1 shows the average running time (in milliseconds) of quicksort, parallel quicksort, mergesort and parallel mergesort with respect to increasing number of input values. It shows that using Parallel.Invoke() parallel quick sort performs better over quick sort after a threshold value and parallel merge sort performs better over merge sort after a threshold value. Serial version of quick sort and merge sort works better for small number of elements. When the number of elements are increased. After a value that is called the threshold value. Parallel version of quick sort and merge sort works better, due to the use of parallelism and proper utilization of CPU cores.

Table 1, average running time of different sorting algorithms

Input Size (n)	Average Running Time(ms)			
	Sequential Quick Sort	Parallel Quick Sort	Sequential Merge Sort	Parallel Merge Sort
5000	2	5	5	7
10000	4	6	7	9
15000	6	7	8	11
20000	8	9	11	12
25000	11	14	14	14
30000	12	14	17	15
35000	17	14	20	17
40000	21	19	24	19

Figure 8 shows that the quicksort performs better over parallel quicksort up to the threshold value as the tasks get executed in a parallel fashion on multiple cores.

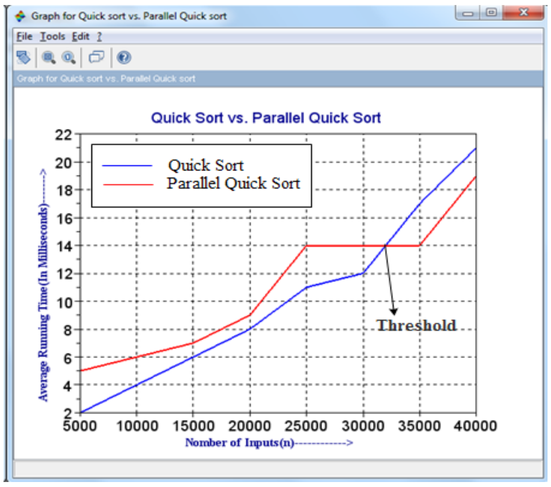


Fig 8: Quick Sort vs. Parallel Quick Sort

Figure 9 shows that the merge sort performs better over parallel merge sort up to the threshold value as the tasks get executed in a parallel fashion on multiple cores.

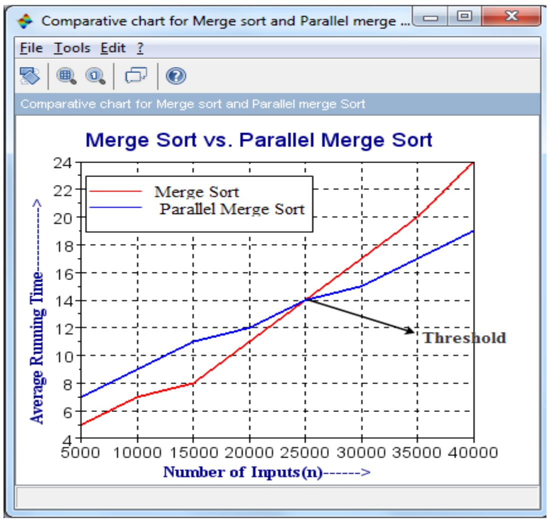


Fig 9: Mergesort vs. Parallel Mergesort

6. CONCLUSION

In this paper the divide and conquer sorting problem for large data sets are considered, and compared successfully. The effect of the number of cores on the performance of quicksort and mergesort has been theoretically and experimentally studied. The basis of analysis is the average running time on dual core processor. It is observed that parallel sorting algorithms i.e. parallel versions of quicksort and mergesort performs well for higher number of inputs in comparison to their sequential versions as shown in figure 8 and figure 9. For small size of input, sequential version of quicksort and mergesort is better to their parallel version, because of parallelism overhead, thread creation, time spent at synchronization, thread communication, granularity of task decomposition etc. In future, same analysis can be performed with parallel sorting algorithms for wide variety of MIMD architectures and the processors with more than two cores. In future, parallel sorting algorithms can be used for enhancing the performance of CPU and parallel divide and conquer sorting algorithms can be used for measuring the performance of CPU cores separately.

7. REFERENCES

[1] Rohit Yadav, Nitin Kr. Verma and Kratika Varshney, Volume 3, Issue 11, November 2013, Analysis of Recursive and Non-recursive Merge Sort Algorithm, International Journal of Advanced Research in Computer Science and Software

[2] Sabahat Saleem, M. IkramUllah Lali1, M. Saqib Nawaz1 and Abou Bakar Nauman, Vol.7, No.2 (2014), pp.151-164, Multi-Core Program Optimization: Parallel Sorting Algorithms in Intel Cilk Plus, International Journal of Hybrid Information Technology

[3] Alaa Ismail El-Nashar, Vol.2, No.3, May 2011 PARALLEL PERFORMANCE OF MPI SORTING ALGORITHMS ON DUAL–CORE PROCESSOR WINDOWS-BASED SYSTEMS, International Journal of Distributed and Parallel Systems (IJDPS)

[4] Ishwari Singh Rajput, Bhawnesh Kumar and Tinku Singh, *Volume 57– No.9, November 2012,* Performance Comparison of Sequential Quick Sort and Parallel Quick Sort Algorithms, International Journal of Computer Applications (0975 – 8887).

[5] Ishwari Singh Rajput, Deepa Gupta, Vol. 2 Issue 3 May 2013, An Adaptive framework for Parallel Merge Sort Algorithm on Multicore Architecture, International Journal of Latest Trends in Engineering and Technology (IJLTET).

[6] G. Koch, **2013** March 5, Multi-Core Introduction”, Intel Developer Zone, https://software.intel.com/en-us/articles/multi-core-introduction.

[7] Archana Ganesh Said, Volume 6, Issue 4, April 2016, Multi-core Processors – A New Approach towards Multiprocessing, International Journal of Advanced Research in Computer Science and Software Engineering.

[8] Abdulrahman Hamed Almutairi & Abdulrahman Helal Alruwaili, Volume 12 Issue 10 Version 1.0, Year 2012, Improving of Quicksort Algorithm Performance by Sequential Thread or Parallel Algorithms, Global Journal of Computer Science and Technology Hardware & Computation.

[9] Nitin Chaturvedi, S Gurunarayanan, Vol.4, No.4, July2013, STUDY OF VARIOUS FACTORS AFFECTING PERFORMANCE OF MULTI-CORE PROCESSORS, International Journal of Distributed and Parallel Systems (IJDPS).

[10] http://people.eecs.berkeley.edu/~yelick/cs194f07/lectures/lect01-whyparallel.pdf

[11] Christian Martin, embedded world 2014, Multicore Processors: Challenges, Opportunities, Emerging Trends, exhibition & conference.

[12] Dali Ismail, http://www.cse.wustl.edu/~jain/cse567-13/ftp/multicore.pdf