International Institute of Information Technology,Hyderabad

APS PROJECT REPORT

MONSOON 2018

Bitonic Sort and Odd-Even Sort Algorithms
and
Comparison with Merge Sort and Quick Sort Algorithms

Team members:

1) Ajinkya Rawankar (2018201020)
2) Amrit Kataria (2018201067)

TABLE OF CONTENTS

1. Introduction

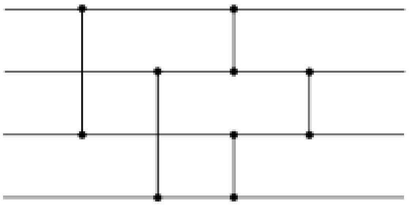
Sorting is a process of arranging elements in a group in a particular order, i.e., ascending order, descending order, alphabetic order, etc. Sorting a list of elements is a very common operation. A sequential sorting algorithm may not be efficient enough when we have to sort a huge volume of data. Therefore, parallel algorithms are used in sorting.

In computer science, a **parallel algorithm**, as opposed to a traditional serial algorithm, is an algorithm which can be executed a piece at a time on many different processing devices, and then combined together again at the end to get the correct result. Algorithms in which operations must be executed step by step are called serial or sequential algorithms. Algorithms in which several operations may be executed simultaneously are referred to as parallel algorithms. A parallel algorithm for a parallel computer can be defined as set of processes that may be executed simultaneously and may communicate with each other in order to solve a given problem. The term processmay be defined as a part of a program that can be run on a processor.

In designing a parallel algorithm, it is important to determine the efficiency of its use of available resources. Once a parallel algorithm has been developed, a measurement should be used for evaluating its performance (or efficiency) on a parallel machine. A common measurement often used is run time. Run
time (also referred to as elapsed time or completion time)refers to the time the algorithm takes on a parallel machine in order to solve a problem.
More specifically, it is the elapsed time between the start of the first
processor (or the first set of processors) and the termination of the last processor (or the last set of processors).

Various approaches may be used to design a parallel algorithm for a given problem. The approach iused here is to attempt to convert a sequential algorithm to a parallel algorithm.

A simple parallel sorting network



Here we will discuss the following −

- Bitonic Sort
- Odd-Even Sort
- Parallel Merge Sort
- Parallel Quick Sort

2. The Algorithms

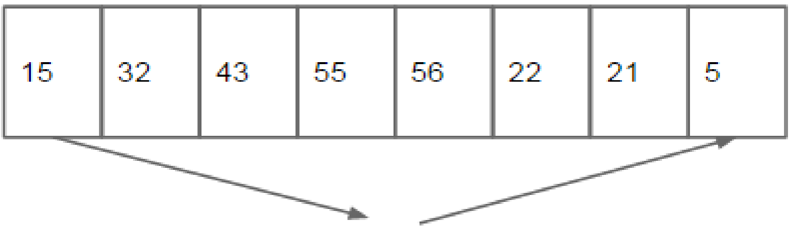
2.1 Bitonic Sort

Bitonic sort is a parallel sorting algorithm which performs O(n Log ^{2}n) comparisons. Although, the number of comparisons are more than that in any other popular sorting algorithm, It performs better for the parallel implementation because elements are compared in predefined sequence which must not be depended upon the data being sorted. The predefined sequence is called Bitonic sequence.

<u>Bitonic Sequence</u>

Bitonic sequence is the one in which, the elements first comes in increasing order then start decreasing after some particular index. An array A[0... i ... n-1] is called Bitonic if there exist an index i such that,

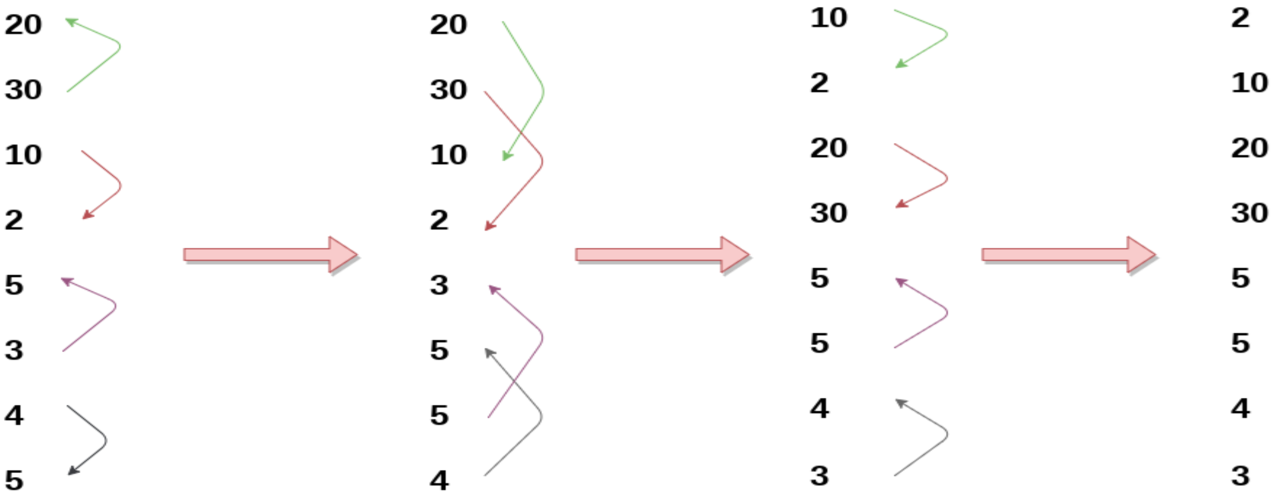
A[0]< A[1] < A[2] ....A[i-1] < A[i] > A[i+1] > A[i+2] > A[i+3] >...>A[n-1]



15	32	43	55	56	22	21	5

Forming a bitonic sequence from a random sequence :

Consider a sequence A[0 ... n-1] of n elements. First start constructing Bitonic sequence by using 4 elements of the sequence. Sort the first 2 elements in ascending order and the last 2 elements in descending order, concatenate this pair to form a Bitonic sequence of 4 elements. Repeat this process for the remaining pairs of element until we find a Bitonic sequence.



Constructing Bitonic Sequence

Bitonic Sorting

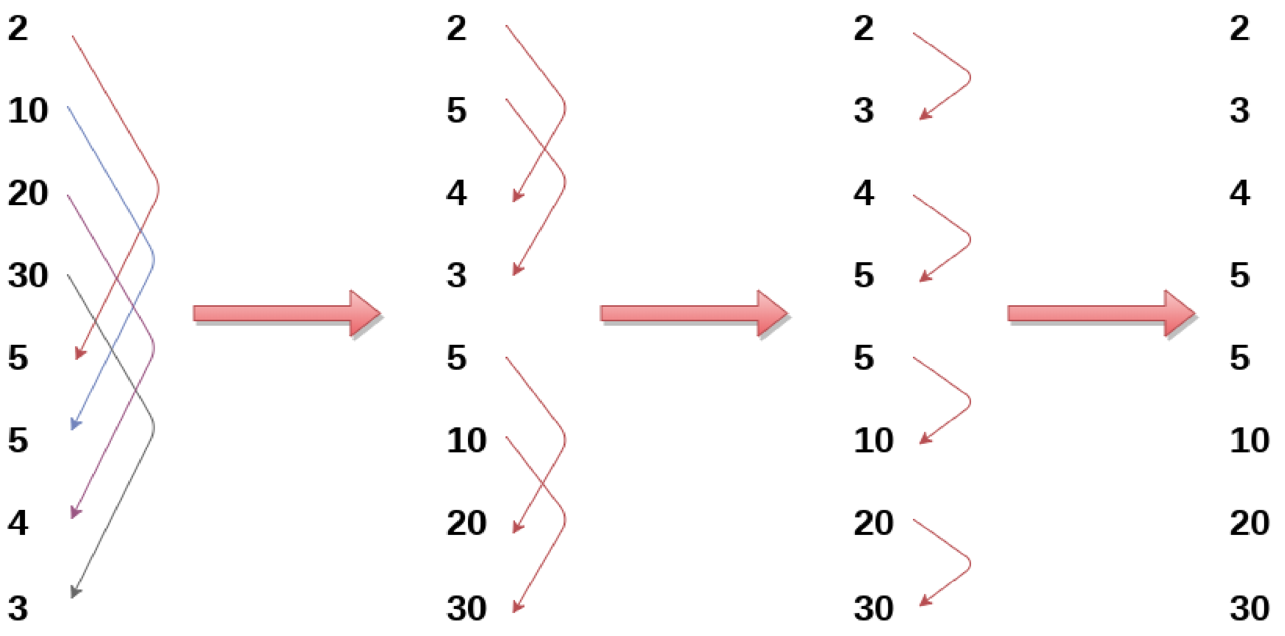
After we have obtained the bitonic sequence in which the first half is an increasing sequence and the second half is a decreasing sequence. We do the following :

- Compare the first element of first half with the first element of second half , second element of first half with the second element of second half and so on , and swap elements if an element in second half is smaller than element in first half.

-Now we repeat the above step for half the length i.e if in above step elements 1 and 5 were compared (length = 4), now we compare elements 1 and 3 , 2 and 4 , and so on (length =2).

- We follow the process until a sorted sequence with n elements is obtained.

-The above procedure is demonstrated in the image below.



Constructing Sorted Sequence

The Pseudocode

```
// Here we sort the array A
// count is the number of elements to be sorted
bitonicSort(A , low , count , direction )
    if count > 1
        temp = count/2
        bitonicSort (A, low , temp , 1 ) // (direction = 1 ,to sort in ascending order )
        bitonicSort (A, low , temp , 0 ) // (direction = 0 ,to sort in descending order )
        bitonicMerge(A, low , count , direction) // to merge the 2 sequences in
                                                 // ascending order
// It recursively sorts a bitonic sequence in ascending order,
// if direction = 1, and in descending order otherwise .
// The sequence to be sorted starts at index position low,

bitonicMerge(A, low , count , direction)
    if (count > 1)
        temp = count/2
        for i = low to (low+ temp)
            CompareAndSwap(A, i , i + temp , direction)
        bitonicMerge(A, low , temp , direction)
        bitonicMerge(A, low + temp , temp , direction)

// Here the function CompareAndSwap swaps elements A[ i ] and A[ i + temp ] if
// A[ i ] > A [ i + temp ] and direction is ascending or when A[ i ] < A [ i + temp ] and
// direction  is descending.
```

Time Complexity Analysis and comparison

When run in serial, the bitonic sorting completes its work in $O(n \log^2 n)$ comparisons, which falls short of the ideal comparison-based sort efficiency of $O(n \log n)$. Parallel versions of the sort, however, can lead to dramatic speedups, depending on the implementation.

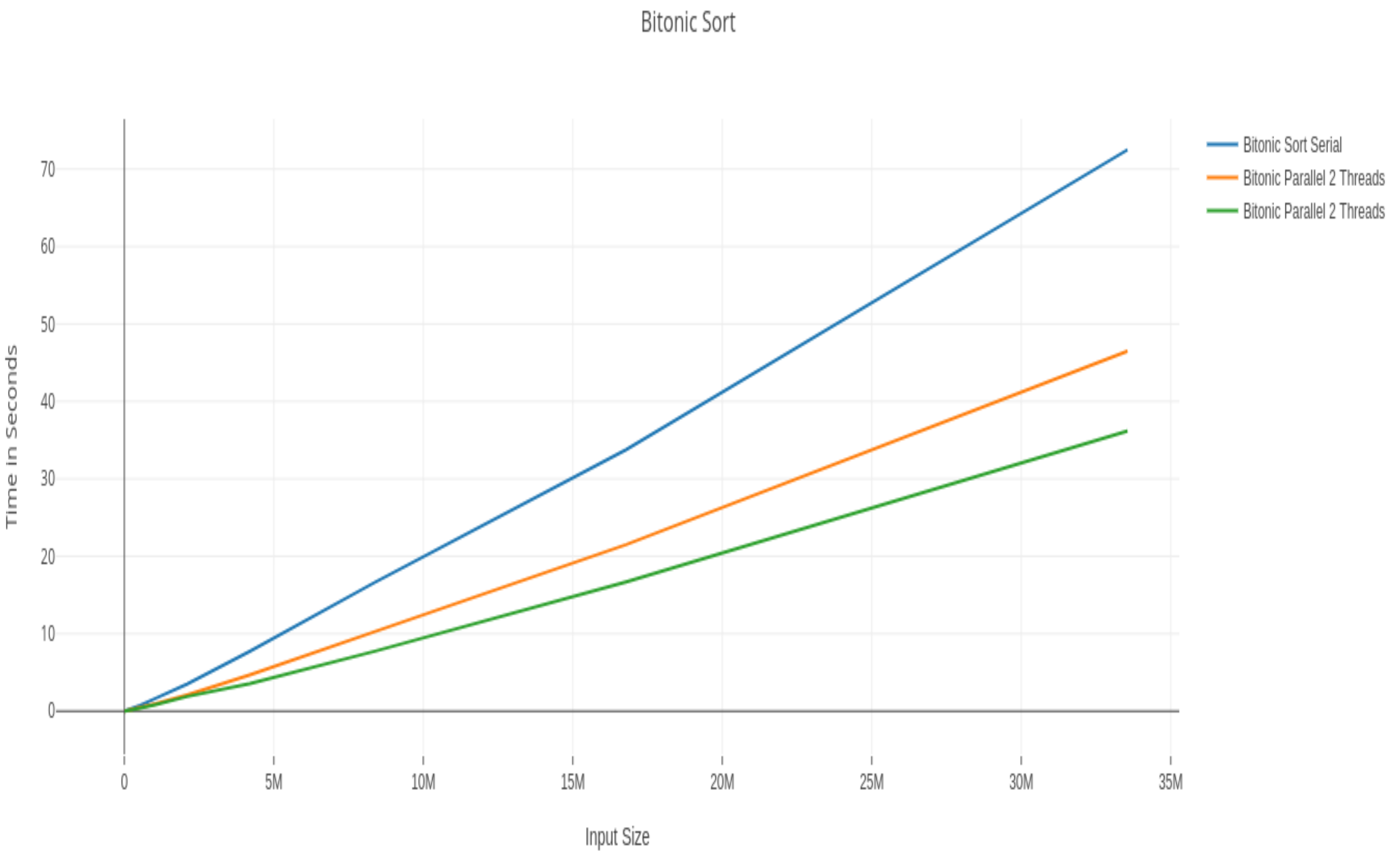
Here we have implemented parallel bitonic sort using OpenMP (discussed later) for 2 and 4 threads.
The openMP implementation consists of 2 main operations for the algorithm: one is called a bitonic split and other being bitonic merge.
- In the bitonic split operation we will keep splitting the input data array into two bitonic sequences until a sequence with only one element is obtained.
- In the bitonic merge operation we will merge and again sort the different parts of the input array in a bitonic fashion.

Here we present the comparison between parallel(with 2 and 4 threads) and serial bitonic sort -

Test Data

Serial No	i/p size	serial	2 threads	4 threads
1	4096	0.003	0.002	0.002
2	8192	0.007	0.005	0.004
3	16384	0.018	0.014	0.01
4	32768	0.03	0.021	0.017
5	65536	0.069	0.046	0.035
6	131072	0.153	0.1	0.085
7	262144	0.33	0.2	0.176
8	524288	0.75	0.44	0.39
9	1048576	1.662	0.97	0.85
10	2097152	3.49	2.11	1.89
11	4194304	7.71	4.67	3.56
12	8388608	16.65	10.28	7.75
13	16777216	33.75	21.5	16.7
14	33554432	72.5	46.5	36.19

Graph Plot


Bitonic Sort
Time In Seconds
70
60
50
40
30
20
10
0
0
5M
10M
15M
20M
25M
30M
35M
Input Size
Bitonic Sort Serial
Bitonic Parallel 2 Threads
Bitonic Parallel 2 Threads

2.2 Odd-Even Sort

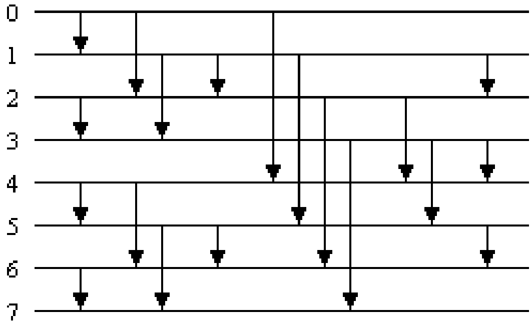
In computing, an **odd–even sort** or **odd–even transposition sort** (also known as **brick sort** is a relatively simple sorting algorithm, developed originally for use on parallel processors with local interconnections. It is a comparison sort related to bubble sort, with which it shares many characteristics. It functions by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. The next step repeats this for even/odd indexed pairs (of adjacent elements). Then it alternates between odd/even and even/odd steps until the list is sorted.

Odd-Even Merge Sort

A related but more efficient sort algorithm is the Batcher odd–even mergesort, using compare–exchange operations and perfect-shuffle operations. Batcher's method is efficient on parallel processors with long-range connections.

In contrast to mergesort, this algorithm is not data dependent, i.e. the same comparisons are performedregardless of the actual data. Therefore, odd even mergesort can be implemented as a sorting network.

Odd-even mergesort for n= 8



The idea behind Batcher’s algorithm is the following claim :

1. If you sort the first half of a list, and sort the second half separately, and then sort the odd-indexed entries (first, third, fifth, ...) and the even-indexed entries (second, fourth, sixth, ...)separately, then you need make only one more comparison-switch per pair of keys to completely sort the list.
2. We will assume that the length of our list,n, is a power of 2.
3. Let’s see what happens for a particular list of length 8. Suppose we are given the f ollowing list of numbers:

2 7 6 3 9 4 1 8

4. We wish to sort it from least to greatest. If we sort the first and second halves separately we obtain:

2 3 6 7 1 4 8 9

5. Sorting the odd-indexed keys (2,6,1,8) and then the even-indexed keys (3,7,4,9) while l eaving them in odd and even places respectively yields:

1 3 2 4 6 7 8 9

6. This list is now almost sorted: doing a comparison switch between the keys in positions (2 and 3),(4 and 5) and (6 and 7) will in fact finish the sort.

The Algorithm

By recursive application of the merge algorithm the sorting algorithm odd-even mergesort is formed.

Algorithm odd-even mergesort(n)

Input: sequence $a_0,....,a_{n-1}$ (n is a power of 2)

Output: the sorted sequence

```
Method:
    if n>1 then
        1.apply odd-even mergesort(n/2) recursively to the two halves
            $a_0,...,a_{n/2-1}$ and $a_{n/2},....,a_{n-1}$ of the sequence;
        2.odd-even merge(n);
```

Algorithm *odd-even merge*(*n*)

Input: sequence $a_0,....,a_{n-1}$ of length n>1 whose two halves $a_0,...,a_{n/2-1}$ and $a_{n/2},....,a_{n-1}$ are sorted (n is a power of 2)

Output: the sorted sequence

```
Method :    if $n$>2 then
                1. apply odd-even merge($n$/2) recursively to the even subsequence
                   $a_0, a_2,....,a_{n-2}$ and to the odd subsequence $a_1, a_3, ,.....,a_{n-1}$
                2. comparison [$i$ : $i$+1] for all $i \in$ {1, 3, 5, 7,...,$n$-3}
            else
                1. comparison [0 : 1]
```

Time Complexity Analysis and comparison

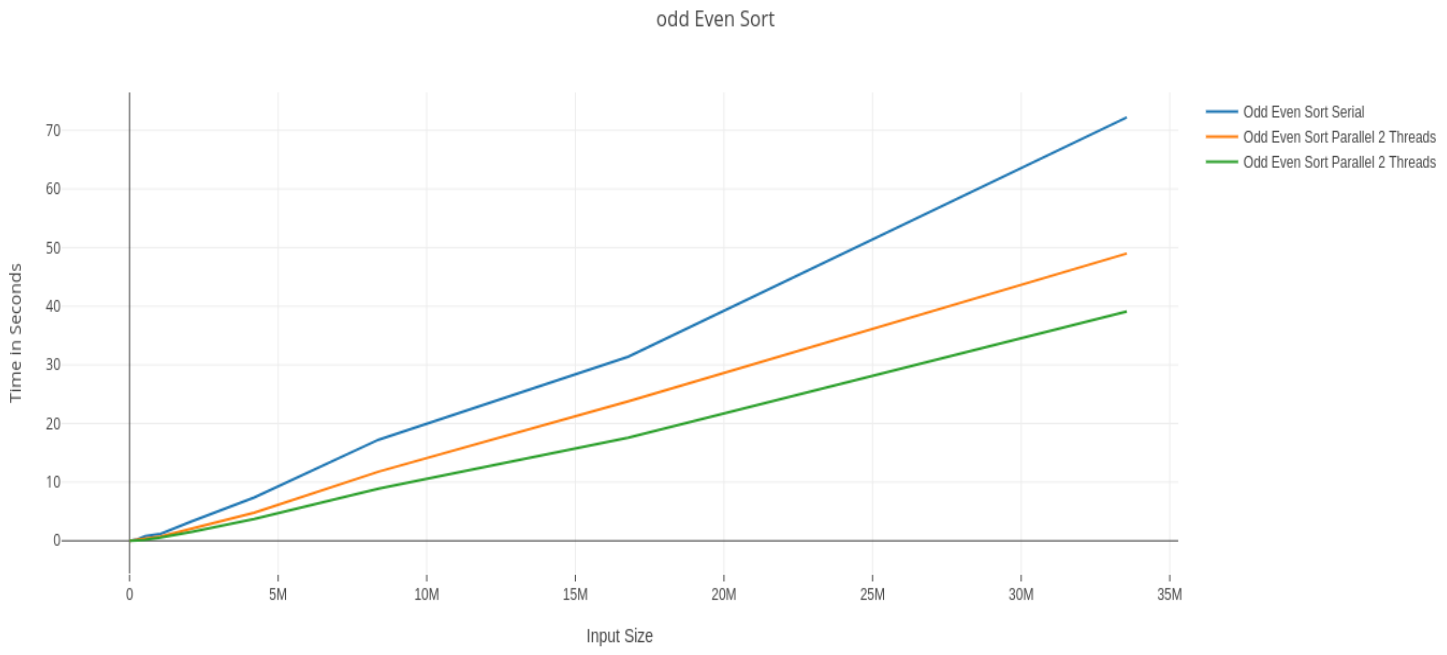
Batcher's odd–even mergesort is a generic construction which runs in O(n(log n)^2) and where n is the number of items to be sorted.
Here we have implemented parallel odd-even merge sort using OpenMP (discussed later) for 2 and 4 threads.
The recursive odd-even mergesort calls are executed in parallel sections for 2 and 4 threads to achieve speedup. Though the speed up achieved is not exactly twice or four times respectively due to other overheads.

Here we present the comparison between parallel(with 2 and 4 threads) and serial odd-even merge sort -

Test Data

Serial No	i/p size	serial	2 threads	4 threads
1	4096	0.002	0.001	0.001
2	8192	0.005	0.002	0.001
3	16384	0.012	0.01	0.001
4	32768	0.025	0.018	0.016
5	65536	0.054	0.04	0.016
6	131072	0.15	0.091	0.047
7	262144	0.256	0.189	0.078
8	524288	0.768	0.482	0.187
9	1048576	1.19	0.73	0.563
10	2097152	3.32	2.09	1.53
11	4194304	7.38	4.79	3.7
12	8388608	17.26	11.81	8.9
13	16777216	31.377	23.8	17.6
14	33554432	72.2	49	39.1

Graph Plot



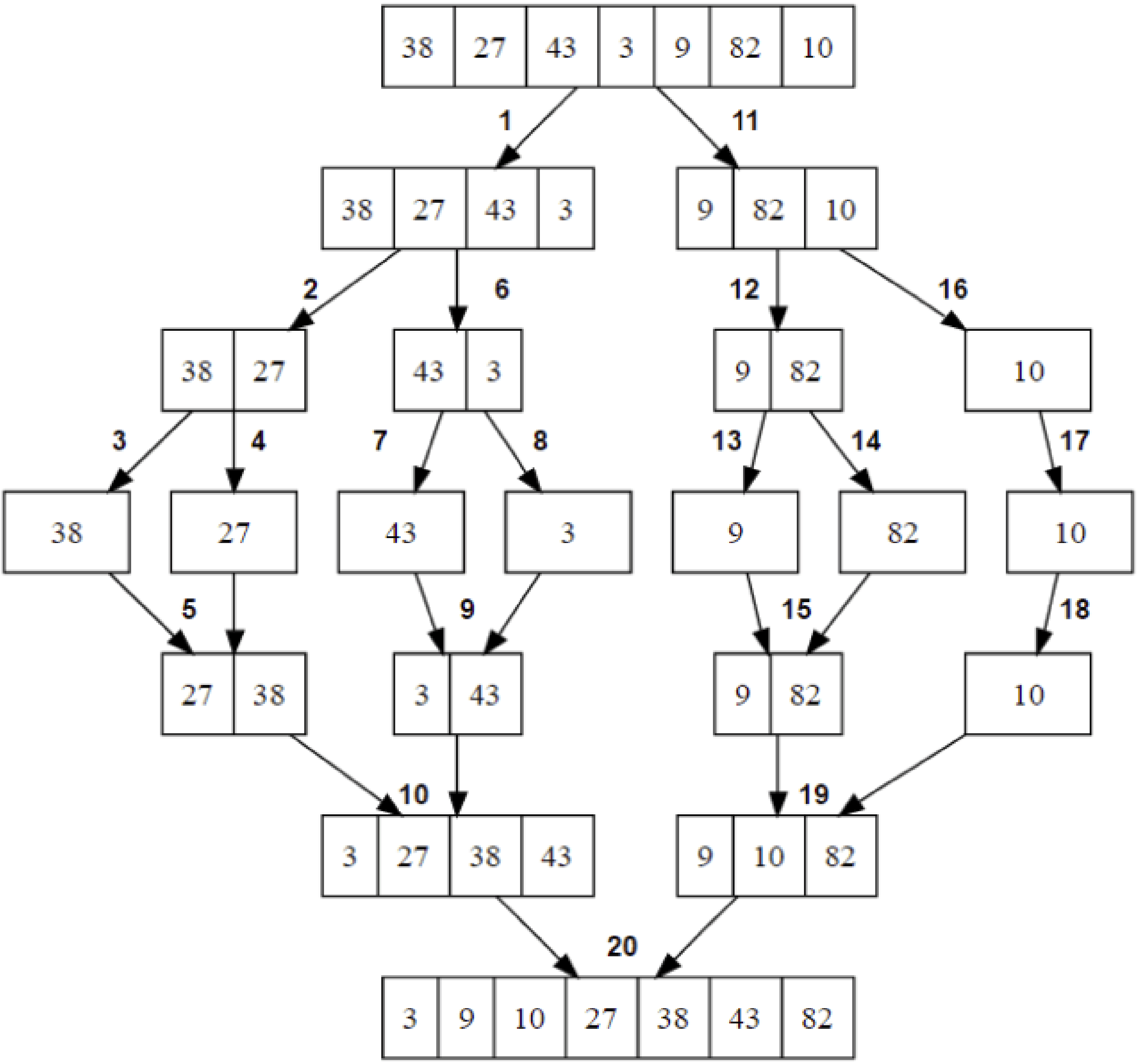
2.3 Merge Sort

The *mergesort* algorithm is based on the classical divide-and-conquer paradigm. It operates as follows:

DIVIDE: Partition the n-element sequence to be sorted into two subsequences of n/2 elements each.

CONQUER: Sort the two subsequences recursively using the mergesort.

COMBINE: Merge the two sorted sorted subsequences of size n/2 each to produce the sorted sequence consisting of n elements.



A merge sort works as follows:

1. Divide the unsorted list into n sublists, each comprising 1 element (a list of 1 element is supposed sorted).
2. Repeatedly merge sublists to produce newly sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

Merging of two lists done as follows:

The first element of both lists is compared. If sorting in ascending order, the smaller element among two becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the newly combined sublist covers all the elements of both the sublists.

The Algorithm

```
MergeSort (arr,left,right)

    If right > left

      1. Find the middle point to divide the array into two halves:
            middle  = (left+right)/2

      2. Call mergeSort for first half:
            Call mergeSort(arr, left, middle)

      3. Call mergeSort for second half:
            Call mergeSort(arr, middle+1, right)

      4. Merge the two halves sorted in step 2 and 3:
            Call merge(arr, left, middle, right)
```

```
Merge (arr,left,middle,right)

    1. Create 2 temporary arrays L and R to contain the 2 halves of the array.
    2. Initialize i = left and j = middle+1 and k = left
    3. while i <= middle and j <= right
    4.        if L[ i ] <= R[ j ]
                      arr[ k ] = L [ i ]
                      i = i+1
    5.        else
                      arr[ k ] = R[ j ]
                      j = j+1
    6.        k=k+1

    7. while i <= middle
            arr[k] = L[i];
            i=i+1;
            k=k+1;

    8. while j <= right
            arr[k] = R[j];
                      j=j+1;
                      k=k+1;
```

Parallel Merge Sort

The serial implementation of Merge Sort, is based on a divide and conquer approach. The data set is divided into recursively into smaller and smaller parts, i.e. the divide part, and then to conquer the data, the data is ordered, two elements at a time. The *merge* action is then called from bottom up, combining the different parts to form the final sorted array.
As deduced, this is a stable sort.

Here we have implemented the parallel merge sort using OpenMP by first writing the serial implementation of merge sort, then calling the divide step of algorithm parallely as the 2 smaller arrays are independent of each other.

Time Complexity Analysis and comparison

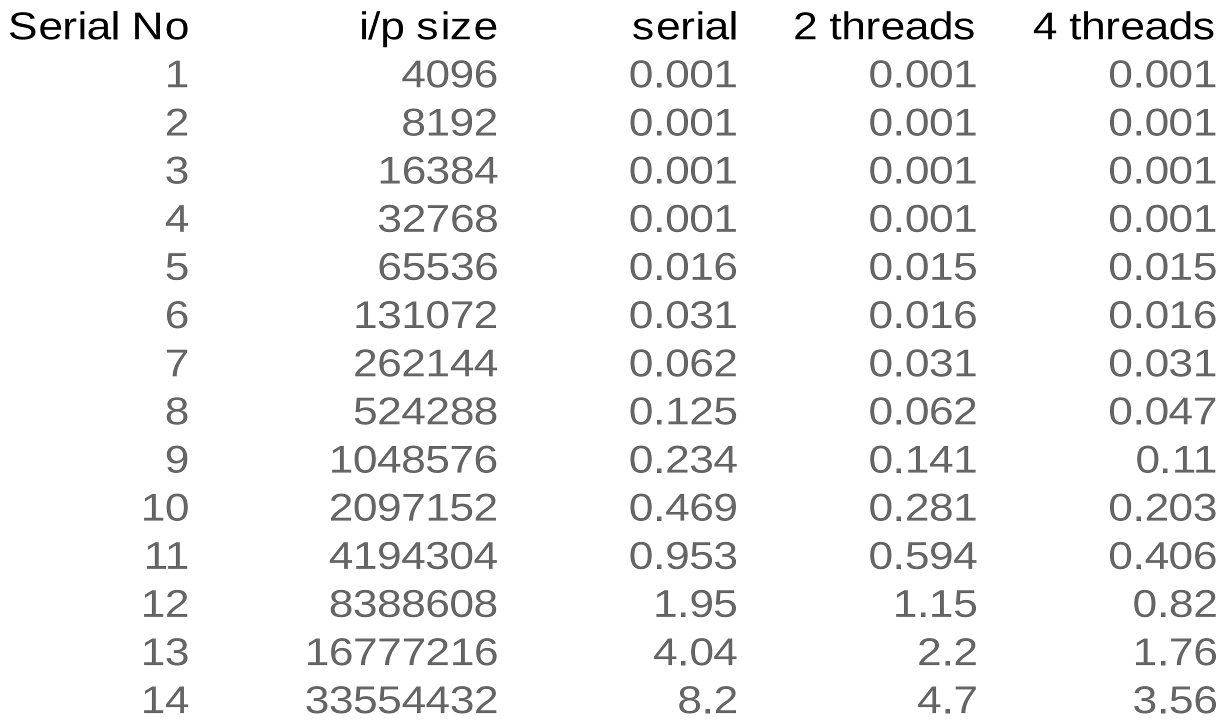
Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

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

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is O(nLogn)
Time complexity of Merge Sort is O(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.
Here our parallel algorithm runs in time O(n) as the merge algorithm takes O(n).

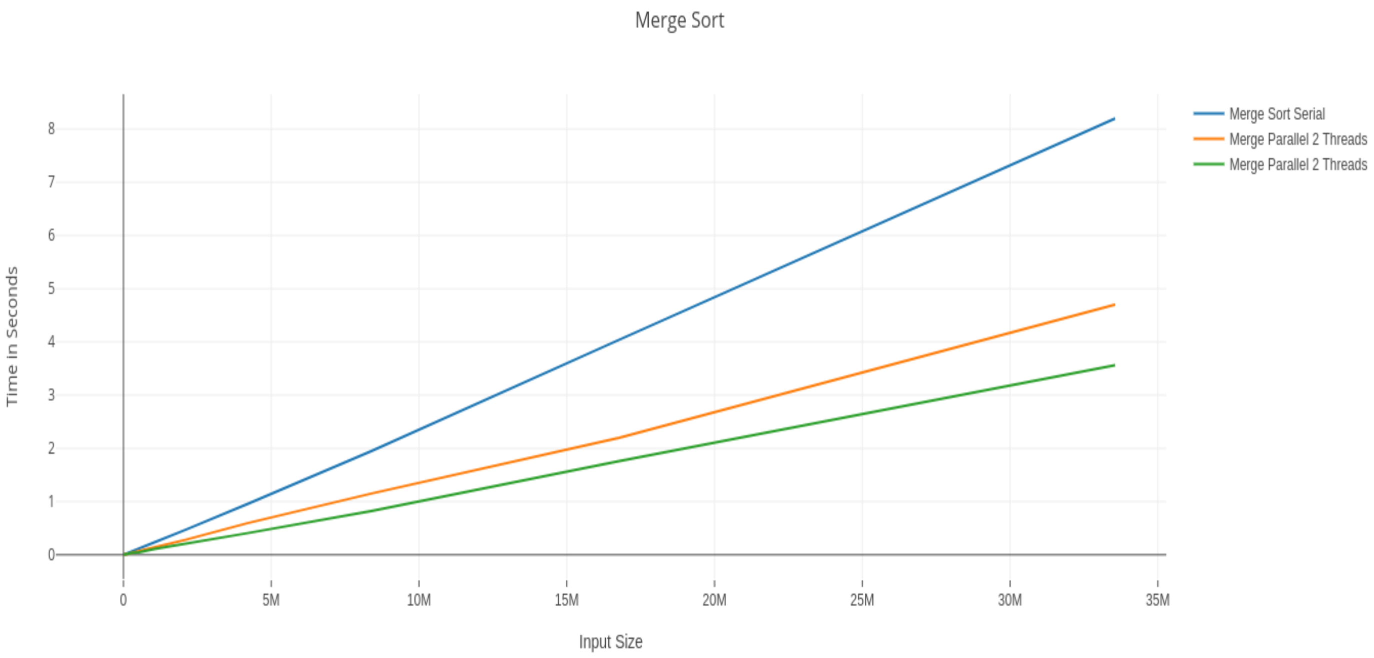
Now we present the comparison of parallel and serial mergesort -

Test Data



Serial No	i/p size	serial	2 threads	4 threads
1	4096	0.001	0.001	0.001
2	8192	0.001	0.001	0.001
3	16384	0.001	0.001	0.001
4	32768	0.001	0.001	0.001
5	65536	0.016	0.015	0.015
6	131072	0.031	0.016	0.016
7	262144	0.062	0.031	0.031
8	524288	0.125	0.062	0.047
9	1048576	0.234	0.141	0.11
10	2097152	0.469	0.281	0.203
11	4194304	0.953	0.594	0.406
12	8388608	1.95	1.15	0.82
13	16777216	4.04	2.2	1.76
14	33554432	8.2	4.7	3.56

Graph Plot



Merge Sort
Merge Sort Serial
Merge Parallel 2 Threads
Merge Parallel 2 Threads
Time in Seconds
8
7
6
5
4
3
2
1
0
0
5M
10M
15M
20M
25M
30M
35M
Input Size

2.4 Quick Sort

Quick Sort is also based on the concept of Divide and Conquer, just like merge sort. But in quick sort all the heavy lifting(major work) is done while dividing the array into subarrays, while in case of merge sort, all the real work happens during merging the subarrays. In case of quick sort, the combine step does absolutely nothing.

It is also called partition-exchange sort. This algorithm divides the list into three main parts:

1. Elements less than the Pivot element
2. Pivot element(Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as pivot.

For example: In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take 25 as pivot. So after the first pass, the list will be changed like this.

{6 8 17 14 25 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate sunarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

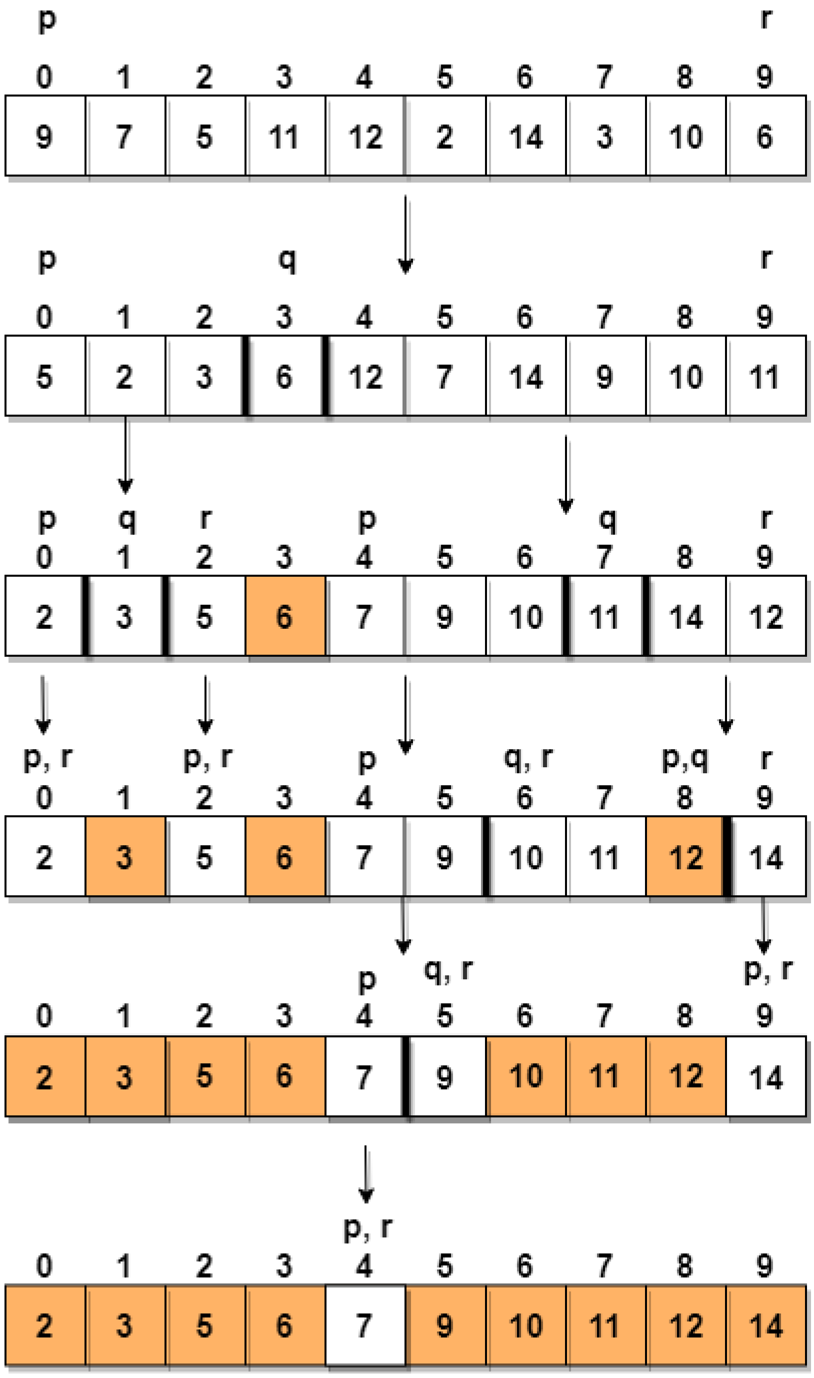
<u>How Quick Sorting Works?</u>

Following are the steps involved in quick sort algorithm:

1. After selecting an element as pivot, which is the last index of the array in our case, we divide the array for the first time.
2. In quick sort, we call this partitioning. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the pivot will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
3. And the pivot element will be at its final sorted position.
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, elements on the left of pivot and elements on the right of pivot, and we perform partitioning on them by choosing a pivot in the subarrays.

Let's consider an array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}

Below, we have a pictorial representation of how quick sort will sort the given array.



In step 1, we select the last element as the pivot, which is 6 in this case, and call for partitioning, hence re-arranging the array in such a way that 6 will be placed in its final position and to its left will be all the elements less than it and to its right, we will have all the elements greater than it.

Then we pick the subarray on the left and the subarray on the right and select a pivot for them, in the above diagram, we chose 3 as pivot for the left subarray and 11 as pivot for the right subarray.

And we again call for partitioning.

The Algorithm

```
quicksort(A, lo, hi)
    if lo < hi then
        p = partition(A, lo, hi)
        quicksort(A, lo, p - 1 )
        quicksort(A, p + 1, hi)

partition(A, lo, hi)
    pivot = A[hi]
    i = lo - 1
    for j = lo to hi - 1 do
        if A[j] < pivot then
            if i != j then
                i = i + 1
                swap A[i] with A[j]

    i = i + 1
    swap A[i] with A[hi]
    return i
```

Parallel Quick Sort

Quicksort can be parallelized in a variety of ways. Here , during each call of QUICKSORT, the array is partitioned into two parts and each part is solved recursively. Sorting the smaller arrays represents two completely independent subproblems that can be solved in parallel. Therefore, one way to parallelize quicksort is to execute it initially on a single process; then, when the algorithm performs its recursive calls, assign one of the subproblems to another process. Now each of these processes sorts its array by using quicksort and assigns one of its subproblems to other processes. The algorithm terminates when the arrays cannot be further partitioned.

Here we have used OpenMp to parallelize the quicksort algorithm. At each recursive call a new thread is created which makes the recursive call and both the arrays are sorted in parallel.

Time Complexity Analysis and comparison

Time taken by QuickSort in general can be written as following.

$T(n) = T(k) + T(n-k-1) + O(n)$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

$$T(n) = T(0) + T(n-1) + O(n)$$

which is equivalent to

$$T(n) = T(n-1) + O(n)$$

The solution of above recurrence is $O(n^2)$.

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

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

The solution of above recurrence is O(nLogn). It can be solved using case 2 of Master Theorem.

Average Case:To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn’t look easy.
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

$$T(n) = T(n/9) + T(9n/10) + O(n)$$

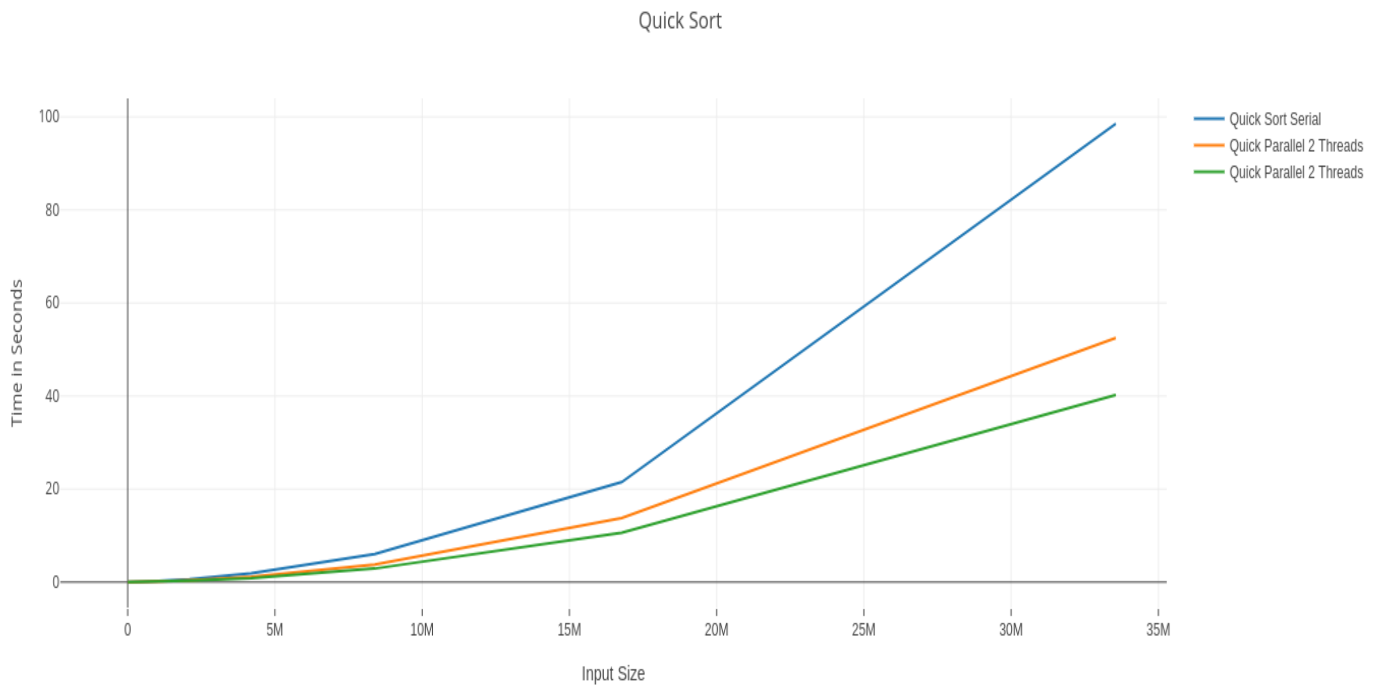
Solution of above recurrence is also O(nLogn)

Now we present the comparison of parallel and serial quicksort -

Test Data

Serial No	i/p size	serial	2 threads	4 threads
1	4096	0.001	0.001	0.001
2	8192	0.001	0.001	0.001
3	16384	0.001	0.001	0.001
4	32768	0.001	0.001	0.001
5	65536	0.016	0.015	0.015
6	131072	0.031	0.016	0.016
7	262144	0.047	0.031	0.031
8	524288	0.11	0.062	0.07
9	1048576	0.234	0.141	0.16
10	2097152	0.64	0.39	0.33
11	4194304	1.87	1.18	0.87
12	8388608	6.06	3.8	2.91
13	16777216	21.5	13.8	10.64
14	33554432	98.5	52.48	40.24

Graph Plot


Quick Sort
Time in Seconds
100
80
60
40
20
0
0
5M
10M
15M
20M
25M
30M
35M
Input Size
Quick Sort Serial
Quick Parallel 2 Threads
Quick Parallel 2 Threads

3. Parallelism with OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran,on most platforms, instruction set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

What is OpenMP

-A directive based parallel programming model

-OpenMP program is essentially a sequential program augmented with compiler directives to specify parallelism

-Eases conversion of existing sequential programs

Main concepts:

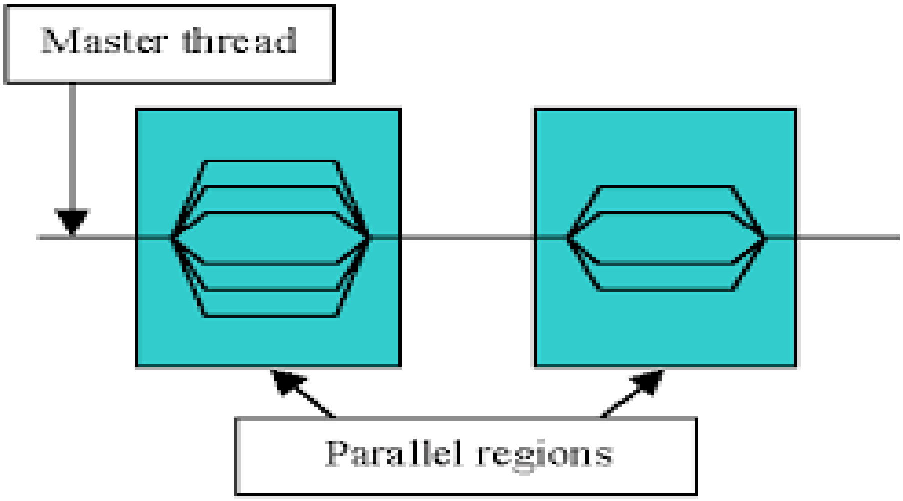
-Parallel regions: where parallel execution occurs via multiple concurrently executing threads.

-Each thread has its own program counter and executes oneinstruction at a time, similar to sequential program execution.

-Shared and private data: shared variables are the means of communicating data between threads.

-Synchronization: Fundamental means of coordinating execution of concurrent threads.

-Mechanism for automated work distribution across threads.



Core Elements

The core elements of OpenMP are the constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables.

In C/C++, OpenMP uses #pragmas. The OpenMP specific pragmas are listed below.

Thread creation

The pragma *omp parallel* is used to fork additional threads to carry out the work enclosed in the construct in parallel. The original thread will be denoted as *master thread* with thread ID 0.

Work-sharing constructs

Used to specify how to assign independent work to one or all of the threads.

- *omp for* or *omp do*: used to split up loop iterations among the threads, also called loop constructs.
- *sections*: assigning consecutive but independent code blocks to different threads
- *single*: specifying a code block that is executed by only one thread, a barrier is implied in the end
- *master*: similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.

4. Testing Environment

Processor – Intel Core i5 6th generation

Number of Cores - 4

RAM – 8 GB

OS – Windows 10

IDE – Dev C++

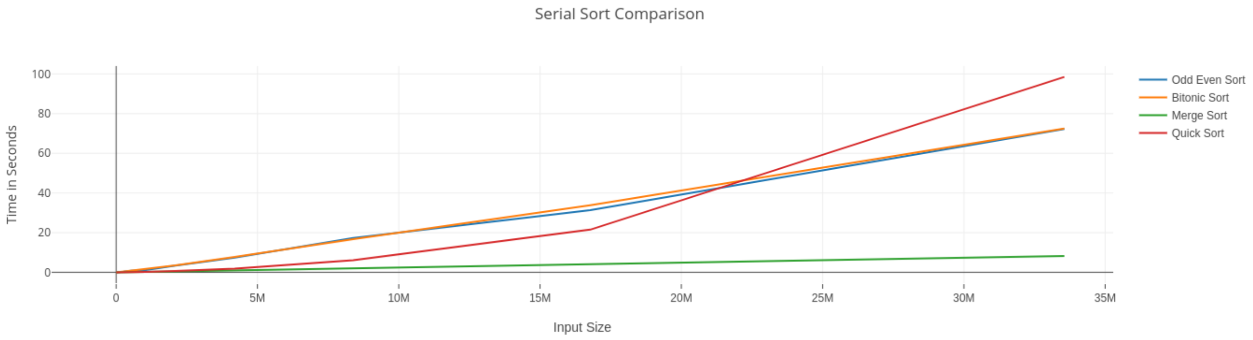
Test Cases – 2^{12} to 2^{25}

5. Results

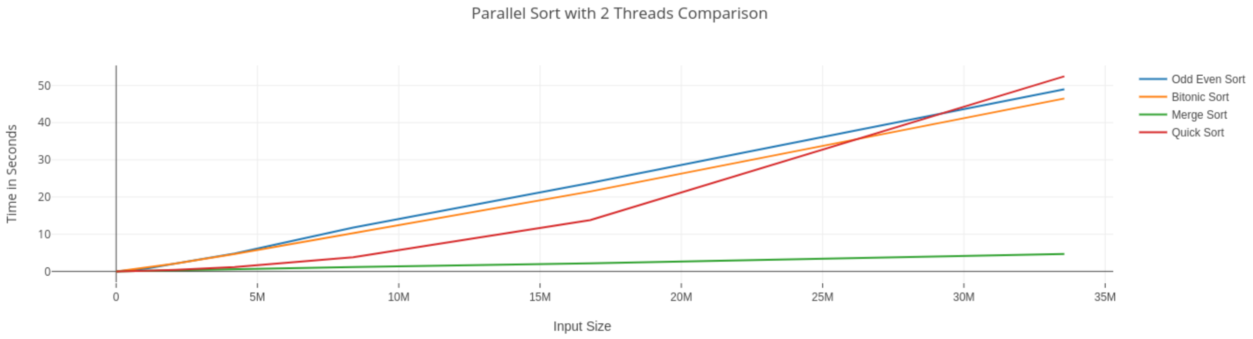
Following is the comparison of the above explained sorting algorithms running in serial and parallel for 2 and 4 threads when tested in above mentioned environment.

It is observed that bitonic sort, odd-even sort and merge sort are not dependent on input data whether it is sorted or not , the number of comparisons remain the same.

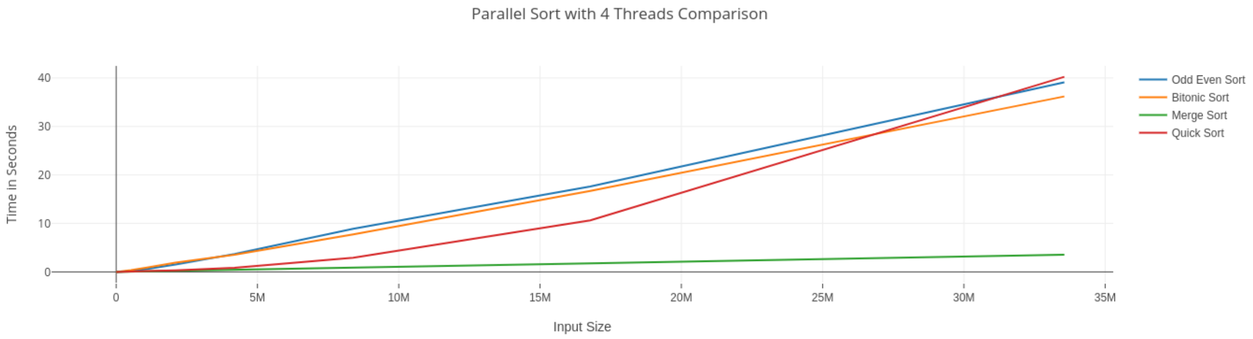
Comparison of various sorting algorithms running in serial :



Comparison of various sorting algorithms running in parallel for 2 threads :



Comparison of various sorting algorithms running in parallel for 4 threads :



6. Refrences

-https://en.wikipedia.org/wiki/Bitonic_sorter

-https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Mullapudi-Spring-2014-CSE633.pdf

-https://en.wikipedia.org/wiki/Batcher_odd%E2%80%93even_mergesort

-https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf

-https://www.tutorialspoint.com/parallel_algorithm/parallel_algorithm_sorting.htm

-http://selkie.macalester.edu/csinparallel/modules/ParallelSorting/build/html/MergeSort/MergeSort.html

-http://parallelcomp.uw.hu/ch09lev1sec4.html

7. User Guide

This guide explains the usage of the programs :

- For all the sorting algorithms seperate files have been provided i.e **bitonic.cpp , mergesort.cpp , oddeven.cpp and quicksort.cpp**.

- Each file contains the serial as well as parallel implementation of the sort.

- To compile a program use the command:

```
g++  <filename>.cpp  -fopenmp
```

-The -fopenmp flag is used to provide support for the OpenMP for parallel programming.

- Each file contains seperate functions for serial and parallel implementaion of the sort.

- In each file the number of threads for parallel implementation can be set by specifying the value of the variable **numThreads** in the begining of the program.

- When a program is run it first asks for the number of elements to be sorted.

- After entering the number of elements , it generates that many random values.

- Three copies of the above generated random numbers are stored in three different arrays.

- The first one is sorted with standard library function , the second with our serial implementaion , and the third with the parallel implementation.

- Time taken by these 3 algorithms is calculated and displayed.

- After that the checkResult function is called which matches the sorted result of our implementation with the sorted array of the standard library sort function.