A New Sorting Algorithm Based on Interpolation Merging and It's Parallel Implementation

By

Tarun Das

ID:2012-1-60-005

Supervised By

Dr. Md. Shamim Akhter

Assistant Professor

Department of Computer Science and Engineering

East West University

A Project Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelors of Science in Computer Science and Engineering

to the

Department of Computer Science and Engineering

East West University

Dhaka, Bangladesh

Abstract

We present a new modified merge sort algorithm (invented by John von Newmann in 1945) based on interpolation merging technique. Known to date, simple binary merging is the best general purpose merging algorithm. However, interpolation merging requires substantially fewer comparisons and increases the performance of modified merge sort algorithm. Results from a computer implementation of the new sorting algorithm is given and compared with merge sort implementation based on tape merging and binary merging algorithms. In addition, the parallel version of the proposed algorithm is implemented with openMP and pthreads frameworks.

Declaration

We hereby declare that, this project was done under CSE497 and has not been submitted elsewhere for requirement of any degree or diploma or for any purpose except for publication.

Tarun Das
ID: 2012-1-60-005
Department of Computer Science and Engineering
East West University

Letter of Acceptance

I hereby declare that this thesis is from the student's own work and best effort of mine, and all other source of information used have been acknowledged. This thesis has been submitted with my approval.

Dr. Md. Shamim Akhter **Supervisor**

Assistant Professor

Department of Computer Science and Engineering

East West University

Dr. Shamim Hasnat Ripon **Chairperson**

Chairperson & Associate Professor

Department of Computer Science and Engineering

East West University

Acknowledgement

First of all, I would like to thank almighty God for give me the strength & proper knowledge to complete my thesis work.

I would like to express my deep sense of gratitude and sincere thanks to my honorable supervisor Dr. Md. Shamim Akhter, Assistant Professor, Department of Computer Science and Engineering, East West University, Aftabnagar, Dhaka, Bangladesh for his kind guidance, sharing knowledge and constant inspiration throughout this thesis work.

My sincere gratefulness for the faculty of Computer Science and Engineering whose friendly attitude and enthusiastic support that has given me for four years.

I am very grateful for the motivation and stimulation from my friends and seniors.

Finally my most heartfelt gratitude goes to my beloved parents and brother for their endless support, continuous inspiration, great contribution and perfect guidance from the beginning to the end.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Objective of the Research

Sorting a huge data set in a nominal time is always a demand for almost all fields of computer science. Divide and conquer is a general and extremely successful strategy for the design and analysis of algorithm. It is the basis of infinitude of sorting algorithm for the usual comparison-based model of computation. Overall view, it is simply bottom up approach followed by a top down traverse. In the sorting technique arena, natural order and merging are taken into deep consideration. Measurements of disorderness through top down approach and suitable merging scheme for bottom up approach have been studied as a universal method for the development of sorting algorithms. Existing tape merge [1] and simple/generalized binary algorithms [1] are usually used to merge to disjoint linearly ordered sets.

1.2 Methodology of Research

In the proposed algorithm, at first the disorderness data is checked and partitions in a single pass over the data set. Thereafter, the partitions are selected according to their order and merged using interpolation searching technique. It has been ensured that the approach provides the optimum time while the bottom up merging tree is balanced.

1.3 Results of the Research

Our results, both theoretical and experimental, indicate a constant factor (~2.7) speed up over binary merging scheme. The parallel version of the proposed algorithm is implemented with OpenMP and Pthreads frameworks. It shows that, OpenMP implementation is performing better than Pthread implementation.

Chapter 2

Background Study

2.1 Measure of Disorderness

In this section, we used “log” to denote the base 2 logarithms, “n” is the total number of elements in the data set, “m” is the number of partition buffers required.

2.1.1 Natural Disorder

Any raw data set contains some natural order or sequence among them. Even in the most disordered situation at least two elements have an ordered sequence, may be increasing or decreasing. For an example, let’s consider the data set {9, 5, 3, 4, 10, 12, 8, 2}. Using these data we will get the following zig-zag diagram, Figure 2.1.1.1. Our goal is to make the data sorted, means the result set of the above data will be {2, 3, 4, 5, 8, 9, 10, 12}. Figure 2.1.1.2 presents the Zig-Zag diagram represents sorted data in natural order.

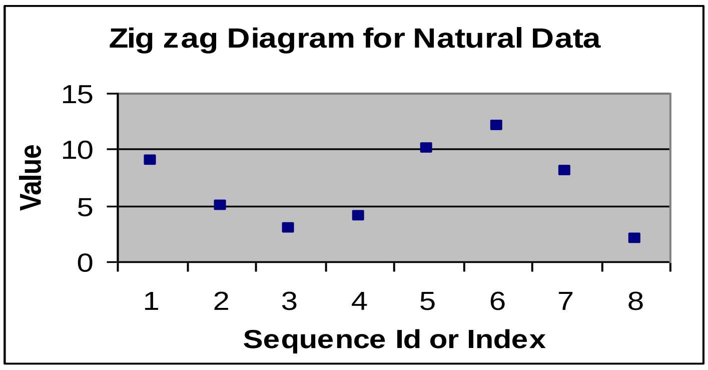


Figure 2.1: Zig-Zag diagram for natural disorder data set

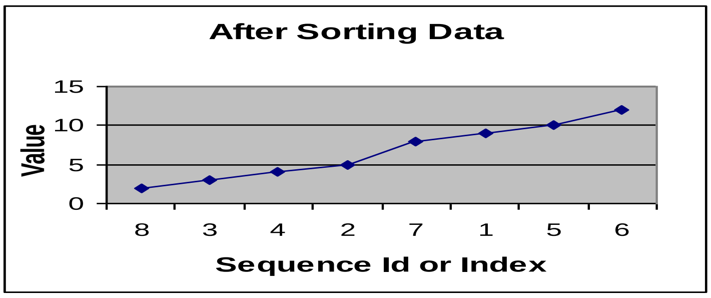


Figure 2.2: Zig-zag diagram for sorted data in natural order

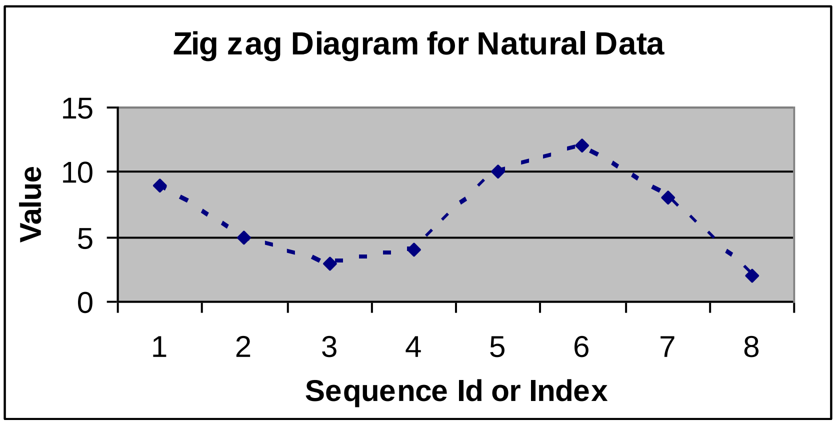


Figure 2.3: Revised Zig-Zag diagram for natural disorder data set.

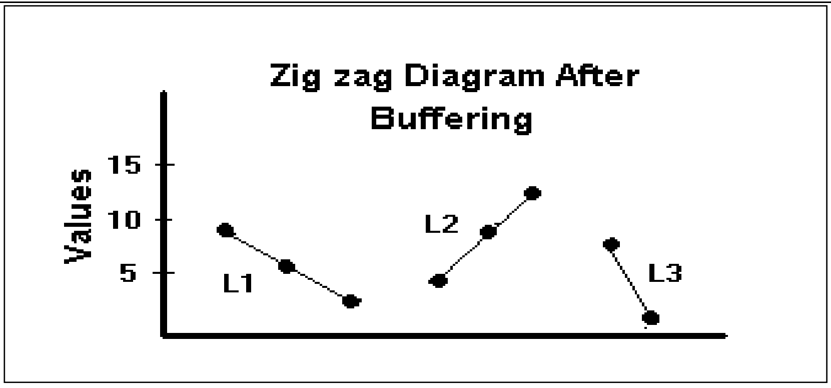


Figure 2.4: Zig-zag diagram after buffering.

Figure 2.1.1.3 and 2.1.1.4 will help to understand the difference between classical sorting algorithm and adaptive sorting algorithm. In all classical ways, sequence Ids are shifted to gain the sorted order. However, in adaptive sorting scheme, lines, connecting the points are taken into consideration. And by doing so, all the points on the two lines (currently under process), are in action. In natural order, in the proposed technique, at least two points are in one line and three comes the time complexity of proposed technique: where m<=n/2. According to the Fig. 2.1.1.4, we make lines L1 = <9, 5, 3>, L2 = <4, 10, 12>, L3 =<8, 2> and finally merging these lines using interpolation search we get approximately a straight line (Which will represent that the data are sorted) present in Fig 2.1.1.2.

2.1.2 Ordering complexity [2]

In order to express the performance of a sorting algorithm in terms of the length and the disorder in the input we must evaluate the disorder in the input. Intuitively, a measure of disorder is a function that is minimized when the sequence has no disorder and depends only on the relative order of the elements in the sequence.

There are several measures of disorder. We define the most three common measures of disorder. Runs(n) as the minimum number of contiguous up-sequences required to cover n data. A natural generalization of Runs is the minimum number of ascending subsequences required to cover the given sequence denoted by Shuffled Up-Sequences (SUS). We generalize again and define *SMS* (n) (for Shuffled Monotone Subsequence) as the minimum number of monotone (ascending or descending) subsequences required to cover the given sequence. For example W0= < 6, 5, 8, 7, 10, 9, 4, 3, 2, 1 > has Runs(W0)=8, while *SUS*(W0) = ||{<6,8,10>, <5,7,9>, <4>,<3>,<2>,<1>}|| = 7 and *SMS*(W0) = ||{<6,8,10>,<5,7,9>,<4,3,2,1>}||= 3. This technique also provide || {< 6, 5 >, < 8, 7 >, < 10, 9, 4, 3, 2, 1 >} || = 3. The number of ascending runs is directly related to the measure Runs, Natural Mergesort takes O(|n| (1+log[Runs(n) +1])) time. Quick sort takes O(|n| log[n+1]) running time in average case.

2.2 Merging schemes

In this section, we are going to describe tape merge algorithm, simple binary merge algorithm and the generalized binary algorithm.

2.2.1 Tape merge

The "tape merge" algorithm is the commonly used procedure to merge two tapes or lists of sorted items. It can be described by the following steps (details of storing and stop conditions are omitted):

TM1. Compare a_m with b_n.

TM2. If $a_m < b_n$, set n =n- 1 and go to TM1.

TM3. If $a_m > b_n$, set m=m-1 and go to TM1.

It can be easily shown that

$$K_t(m,n) = m + n - 1$$

And hence the "tape merge" algorithm is M-optimal for $n \leq m + 3$.

2.2.2 Simple binary merge

The "simple binary" algorithm can be described by the following steps:

SB1. Merge a_m into B by the binary search procedure.

SB2. Pull out a_m and elements of $B > a_m$. (These are already in order and larger than the rest of the elements of A U B.) Set m = m - 1 and redefine m and n. (The new n ≥ new m.) Go back to SB1.

It is clear that under the worst possible outcome, a_m is always larger than b_n, and hence no element of B is discarded. Therefore,

$$Ks(m, n) = m*ceil(log2\ (n + 1)).$$

For m=1, we have,

$$Ks(m, n)= K(m, n).$$

However, we shall show in the next section that

$$Ks(m, n) > K(m, n) \text{ for } m > 2.$$

The distinctive feature of these two algorithms is their simplicity, although in general, they are quite inefficient in the sense that both Kt(m, n) - K(m, n) or Ks(m, n) - K(m, n) can be very large.

2.2.3 The generalized binary algorithm

For the sake of simplicity, we shall assume that whenever we are required to merge two disjoint linearly ordered sets with cardinalities x and y respectively, n will always refer to max (x, y) and m, to min (x, y), so that n ≥ m.

The generalized binary algorithm may now be described as follows (again, details of storage and stop criteria are omitted):

GB1. Let α= floor(log2 (n/m)) and x = n - 2^α + 1.

GB2. Compare a_m with b_x. If $a_m < b_x$, pull out the set of all elements in $B \geq b_x$, say C. We are then left with the problem of merging two disjoint sets A and B C. Redefine m and n and go back to GB1. (Note that B - C has n – 2^α elements and we need to interchange the role of m and n if and only if n = m.)

GB3. If $a_m > b_x$, merge a_m into the set C - b_x by the simple binary algorithm. Note that C - b_x has exactly 2^α - 1 elements and a_m can be merged into the set in exactly α more comparisons. Pull out a_m and the set D of all elements in $B > a_m$.

We are then left with the problem of merging the set A - a_m with the set B - D. Redefine m and n and go back to GB1.

2.3 Parallel Tools

2.3.1 Thread

A thread is a semi-process, which has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads. A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel.

2.3.2 OpenMP

OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed. Each thread has an id attached to it which can be obtained using a function (called omp_get_thread_num()). The thread id is an integer, and the master thread has an id of 0. After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions. The OpenMP functions are included in a header file labeled omp.h in C/C++.

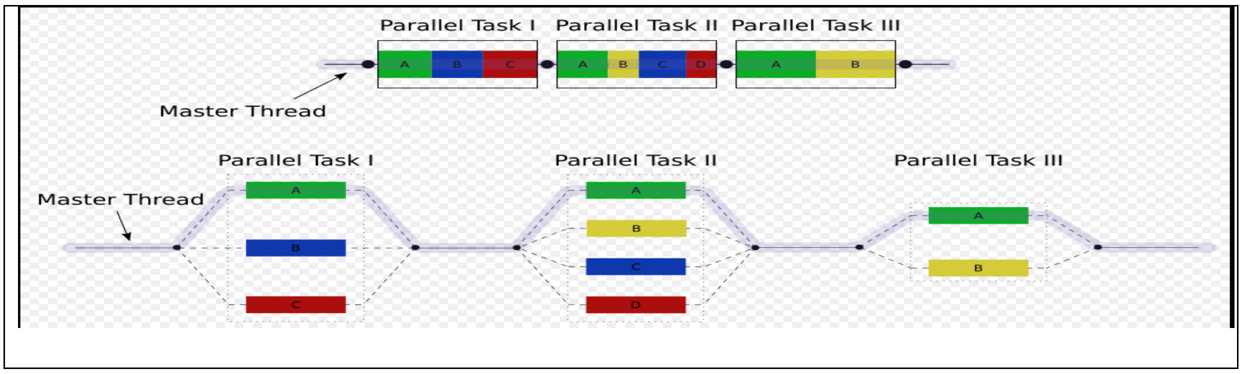


Figure 2.5: Example of OpenMP [3]

Work-sharing constructs:

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

- omp for : 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.

2.3.3 Pthreads

The Pthreads library is a POSIX C API thread library that has standardized functions for using threads across different platforms. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's.

Pthreads are defined as a set of C language programming types and procedure calls. Vendors usually provide a Pthreads implementation in the form of a include file and a library, which you link with your program. The primary motivation for using Pthreads is to realize potential program performance gains. When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes. All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication. Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:

- **Overlapping CPU work with I/O:** (for example), a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, other threads can perform CPU intensive work.
- **Priority/real-time scheduling:** tasks, which are more important, can be scheduled to supersede or interrupt lower priority tasks.
- **Asynchronous event handling:** tasks, which service events of indeterminate frequency and duration, can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests. Multi-threaded applications will work on a uniprocessor system; yet naturally take advantage of a multiprocessor system, without recompiling. In a multiprocessor environment, the most important reason for using Pthreads is to take advantage of potential parallelism.

Chapter 3

Related Works

Merging is a building block for other algorithms, most notably sorting. It is the process whereby two pre-sorted lists of m and n data values each are combine in a systematic manner to create a single sorted output list of m+n data values and requires O(N) time, which is optimal, but also uses O(N) additional space. John Von Neuman first proposed merging in 1945 as a method for sorting data on digital computer systems that use the stored program concept. Kronrod [4] derived a method of merging two sorted sequences of a total of N elements in O(N) time using only a constant amount of additional space. Unfortunately, Kronrod's merging algorithm was not stable. Horvath [5] managed to derive a stable algorithm with the same asymptotic complexity which, however, had the undesired characteristic of modifying the keys of the elements during the merging. Pardo [6] overcame this obstacle and finally derived an asymptotically optimal algorithm which did not use key modification. Even though asymptotically optimal, because of their complex structure and the large constant of proportionality, both of the algorithms of Horvath and Pardo are considered impractical. Dvorak and Durian [7], Mannila and Ukkonen [8] and Huang and Langston [9] derived linear time algorithms for unstable in-place merging. Even though there are algorithms that perform stable merging by using only a constant amount of extra space in linear time, no one succeeds in matching the lower bounds on both the number of comparisons Ω(m log(n/m)) and the number of element assignments (Ω(N)). The algorithms of Horvath [5], Pardo [6] and Huang and Langston [9] perform O(N) comparisons and element assignments. SPLITMERGE [10] matches the lower bounds but uses O(m)†extra space. The algorithm of Mannila and Ukkonen matches all the lower bounds (number of comparisons/ assignments and extra space) but is unstable. To achieve that, it uses the binary merge algorithm of Hwang and Lin [11]. In 1972, they presented a general purpose merging algorithm, also known as the binary merge algorithm, which combines the ideas of binary insertion and linear merge and reduce the number of comparisons of the merging algorithm from O(N) to O(m log(n/m+1). This paper serves two purposes. First it presents a collection of useful techniques used in merging and, second, it presents an optimal stable in-place merging algorithm.

Interpolation search is a searching method of retrieving a desired record by key in an ordered file by using the value of the key and the statistical distribution of the keys. It's average complexity is O(log logn) to retrieve a key, assuming that the n keys are uniformly distributed [12].The proposed new sorting algorithm which derives its motivation from interpolation search.

Chapter 4

Materials and Methods

In this chapter we are going to describe two implementations that are serial implementation and parallel implementation.

4.1 Serial implementation

In this section we are going to describe interpolation search based sorting and two methods of binary search based sorting algorithms.

4.1.1 Binary search based sorting algorithm Implementation 1

The algorithm can be better understood if preceded by an example. So this section will aim at explaining the algorithm with an example.

Suppose we want to merge following array using binary search approach.

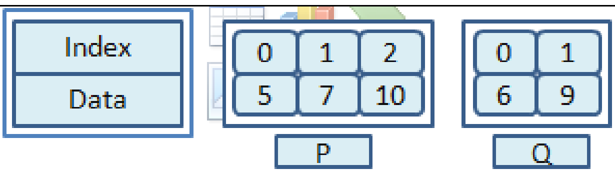


Figure 4.1: Merging Array for Implementation 1

Here 5 is less than the first value of the Q array so 5 is directly copy into B array. The index for value 5 in P buffer is 0. So copy 5 into the position 0 in B array.

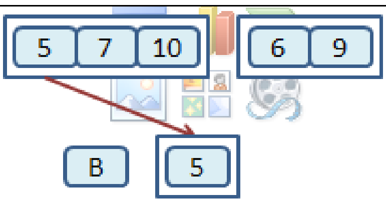
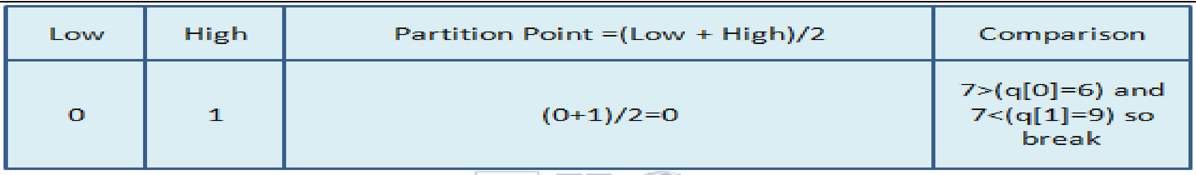


Figure 4.2: Step 1 of Implementation 1

7 will search the appropriate position in the buffer Q using Binary search from the position 0 to 1. The Binary search returns position 0 for value 7. The index for value 7 in P buffer is 1. So 7 push in the index (0+1+1=2) in the B array.



Low	High	Partition Point =(Low + High)/2	Comparison
0	1	(0+1)/2=0	7>(q[0]=6) and 7<(q[1]=9) so break

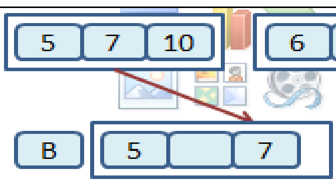


Figure 4.3: Step 2 of Implementation 1

If the search value of the P is greater than the last position value of the Q then copy search value into the location (last index of the Q array - first index of the Q array + search value position in P array + 1). Here 10 is greater than the last position value of the Q array. The index for value 10 in P buffer is 2. So 10 push in the index (1-0+2+1=4) of the B array.

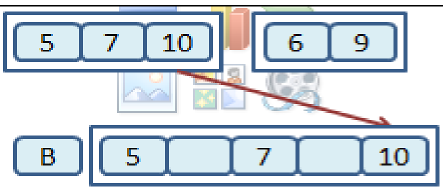
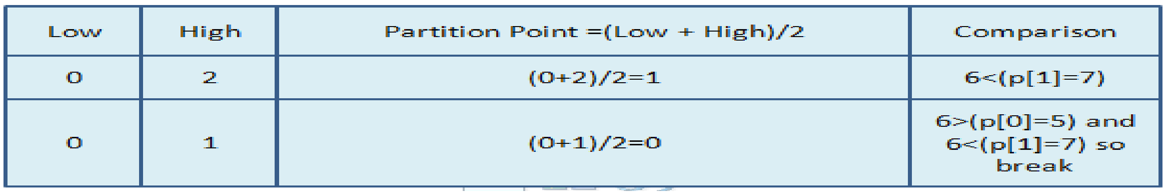


Figure 4.4: Step 3 of Implementation 1

Then 6 will search the appropriate position in the buffer P using Binary search from the position 0 to 2. The Binary search returns position 0 for value 6. The index for value 6 in Q buffer is 0. So 6 push in the index (0+0+1=1) in the B array.



Low	High	Partition Point =(Low + High)/2	Comparison
0	2	(0+2)/2=1	6<(p[1]=7)
0	1	(0+1)/2=0	6>(p[0]=5) and 6<(p[1]=7) so break

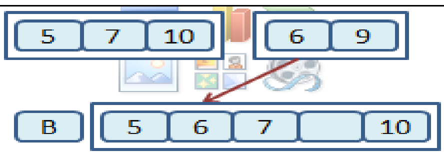


Figure 4.5: Step 4 of Implementation 1

Then 9 will search the appropriate position in the buffer P using Binary search from the position 0 to 2. The Binary search returns position 1 for value 9. The index for value 9 in Q buffer is 1. So 9 push in the index (1+1+1=3) in the B array.

Low	High	Partition Point =(Low + High)/2	Comparison
0	2	(0+2)/2=1	9>(p[1]=7) and 9<(p[2]=10) so break

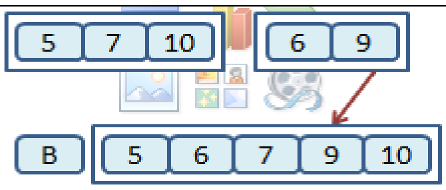


Figure 4.6: Step 5 of Implementation 1

4.1.2 Binary search based sorting algorithm Implementation 2

Now we implement binary search based sorting algorithm like our interpolation search based sorting algorithm method. The algorithm can be better understood if preceded by an example. So this section will aim at explaining the algorithm with an example.

Suppose we want to merge following array.

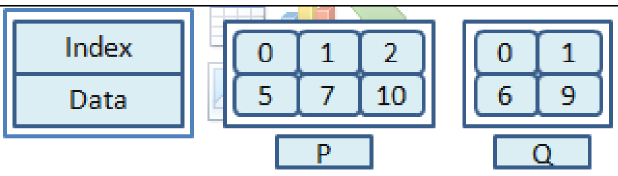


Figure 4.7: Merging Array for Implementation 2

Here 5 is less than the first value of the Q array so 5 is directly copy into B array at position 0.

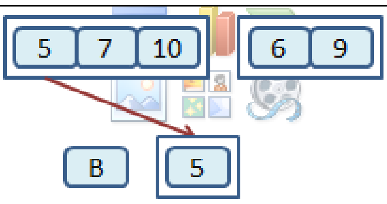
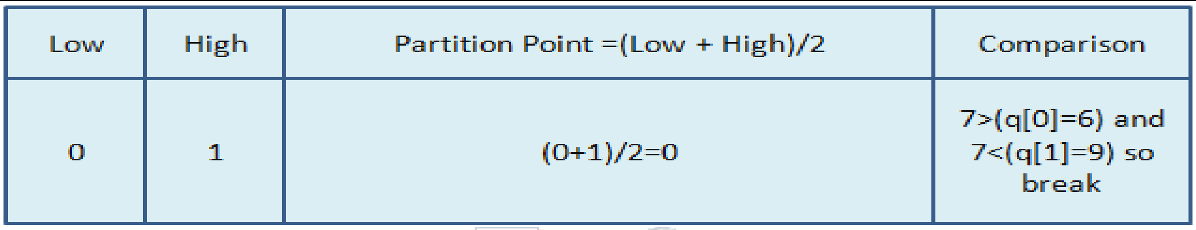


Figure 4.8: Step 1 of Implementation 2

Now 7 will search the appropriate position in the buffer Q using Binary search from the position 0 to 1. Then pull out all the element of buffer Q that is less than 7 and search value of buffer P into another array B.



Low	High	Partition Point =(Low + High)/2	Comparison
0	1	(0+1)/2=0	7>(q[0]=6) and 7<(q[1]=9) so break

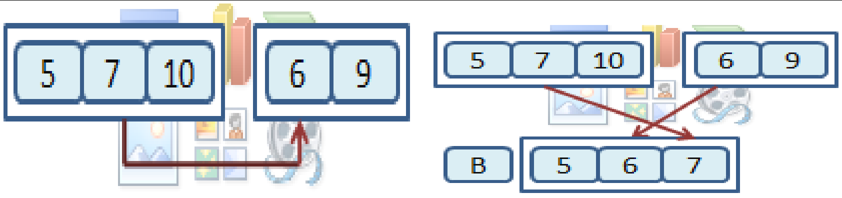


Figure 4.9: Step 2 of Implementation 2

Here, 10 is greater than the last element of the Q buffer so all of the elements of the Q buffer are less than 10. So we directly copy rest of the elements of the Q array into B. then copy 10 into the B array.

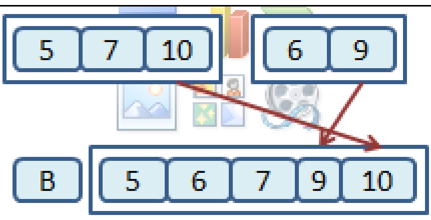


Figure 4.10: Step 3 of Implementation 2

4.1.3 Interpolation Search based Sorting Algorithm

We propose a new modified merge sort algorithm based on interpolation merging technique. The algorithm can be better understood if preceded by an example. So this section will aim at explaining the algorithm with an example.

For n element data set, we first make some buffers according to their sequential order, the order may be in ascending or descending, the running time will need O(n). Each buffer will get information about the starting index and the ending index of the sequential sorted data and also a flag which will provide us the order of the sequential data (flag 0 means ascending, flag 1 means descending order), this flag will be needed only in the first level comparisons but others time no need to check.

Suppose we have a dataset {7,10,12,15,2,9,11,16,20,19,14,8,17,3,1}. So the partition will be {7, 10, 12, 15}, {2, 9, 11, 16, 20}, {19, 14, 8}, {17, 3, 1}.

In table 1, we consider the Figure 4.1.1.1, the buffer L1 = <7, 10, 12, 15>, where 7 is the first and 15 is the last element of this particular dataset. So the starting index is 0 and ending index is 3. Dataset 7 to 15 is in ascending order, so the flag is set to 0.

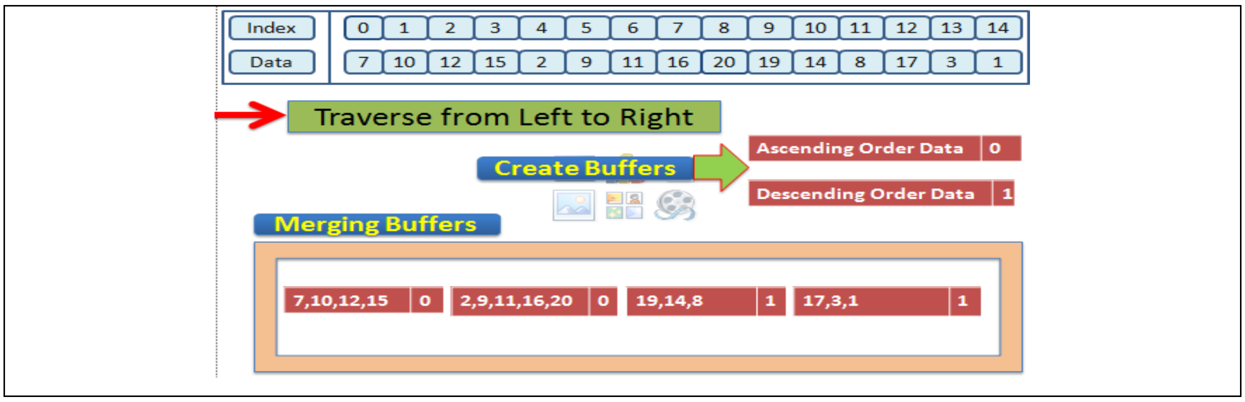


Figure 4.11: Create Buffer of Unsorted Data for Serial Implementation

Table 1: The buffer information

Starting index	0
Ending index	3
Flag	0

Now, we pick first two buffer say P and Q.

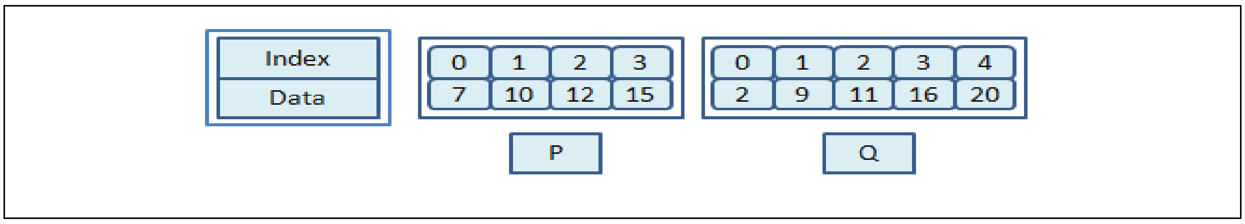


Figure 4.12: First two Buffer for Serial Implementation

Here P and Q array are ascending order. If P array is ascending then we start from upward. Now 7 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 4. Then pull out all the element of buffer Q that is less than 7 and search value of buffer P into another array B.

Low	High	Partition Point = Low + (value – q[Low])*(High-Low)/(q[High]-q[Low])	Comparison
0	4	0+(7-2)*(4-0)/(20-2)=1	7<(Q[1]=9)
0	1	0+(7-2)*(1-0)/(9-2)=0	7>(q[0]=2)and 7<(Q[1]=9)so break.

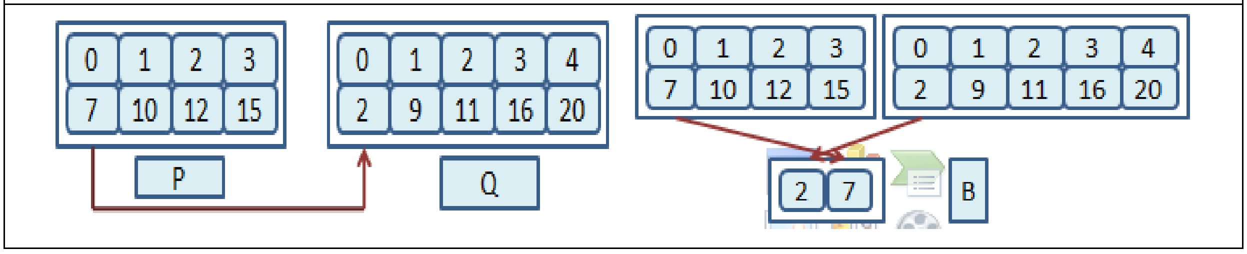
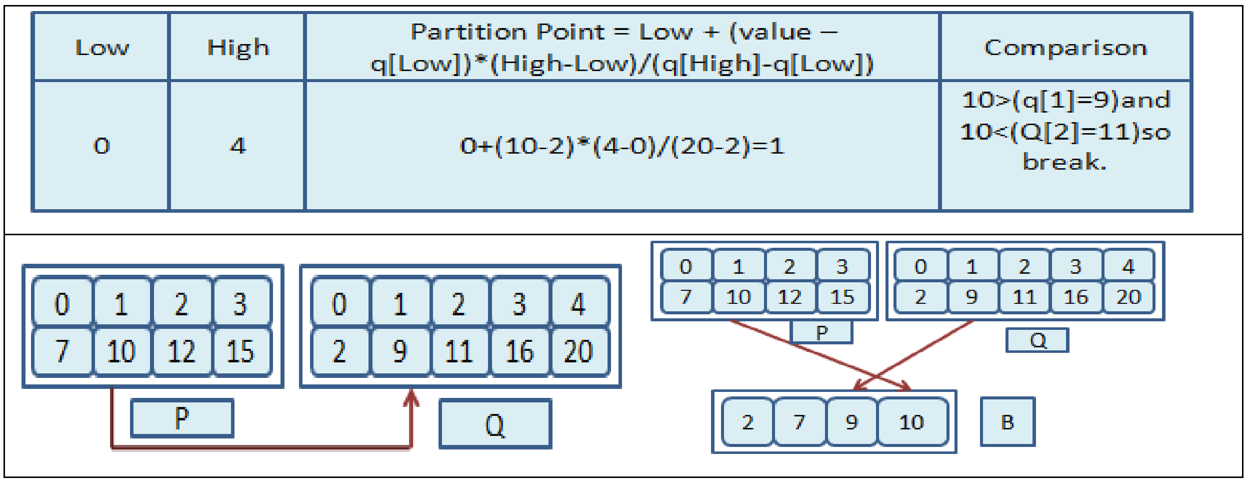


Figure 4.13: Step 1 of Serial Implementation

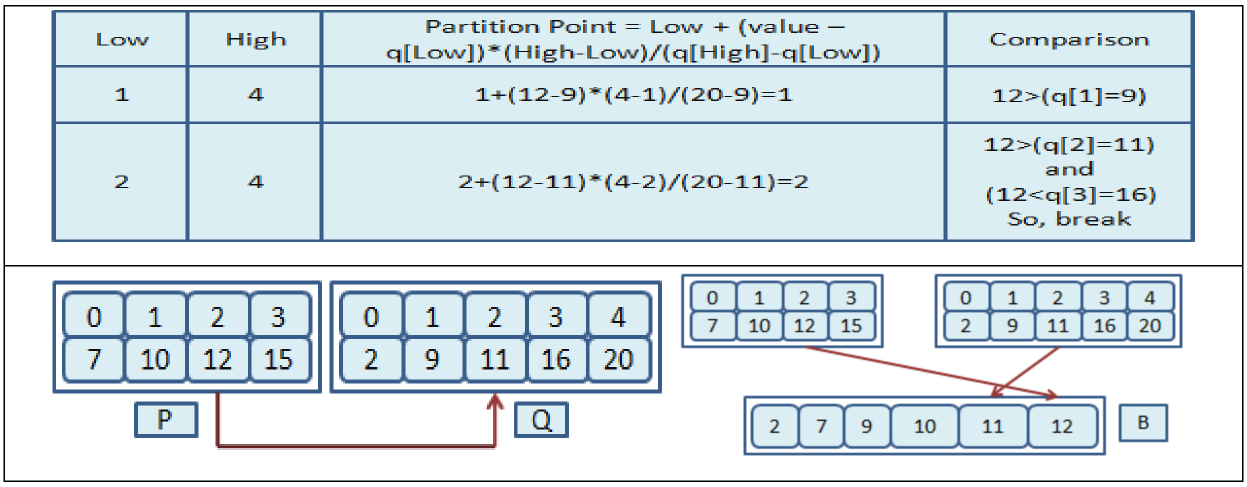
Now 10 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 4. Then pull out the elements of buffer Q that starts from last position in the Q buffer that already copy + 1 to the new position that will find by interpolation search and search value of buffer P into another array B. here 2 is already copy into B array so, copy the elements of Q that starts from 9. Here only 9 will be copied because the new search position will be 1.



Low	High	Partition Point = Low + (value – q[Low])*(High-Low)/(q[High]-q[Low])	Comparison
0	4	0+(10-2)*(4-0)/(20-2)=1	10>(q[1]=9)and 10<(Q[2]=11)so break.

Figure 4.14: Step 2 of Serial Implementation

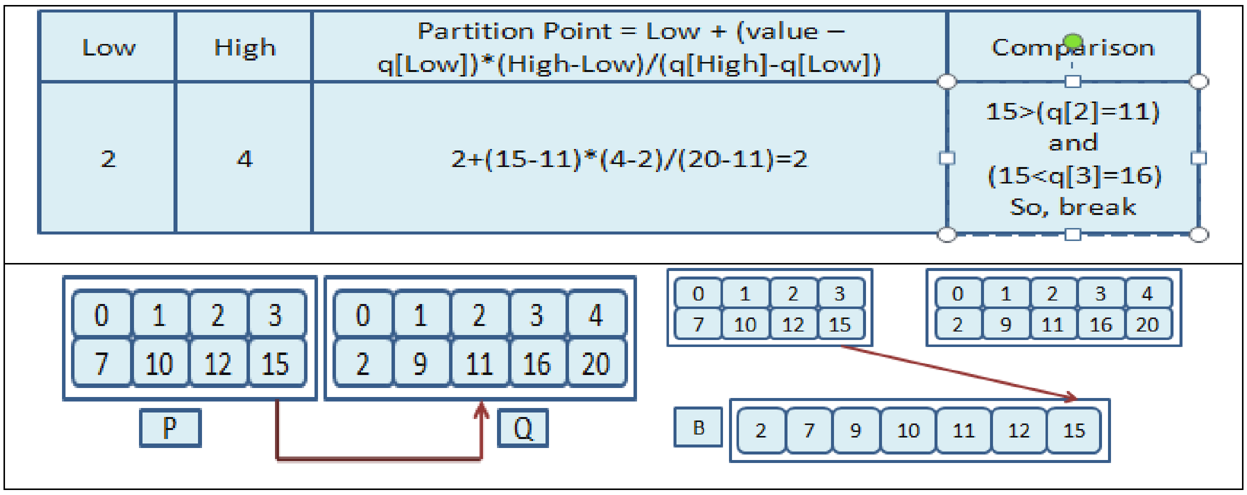
Now 12 will search the appropriate position in the buffer Q using interpolation search from the position 1 to 4. Here 2 and 9 are already copy into B array so, copy the elements of Q that starts from 11. Here only 11 will be copied because the new search position will be 2.



Low	High	Partition Point = Low + (value – q[Low])*(High-Low)/(q[High]-q[Low])	Comparison
1	4	1+(12-9)*(4-1)/(20-9)=1	12>(q[1]=9)
2	4	2+(12-11)*(4-2)/(20-11)=2	12>(q[2]=11) and (12<q[3]=16) So, break

Figure 4.15: Step 3 of Serial Implementation

Now 15 will search the appropriate position in the buffer Q using interpolation search from the position 2 to 4. Here 2, 9 and 11 are already copy into B array so, only search value 15 will be copied into B array.



Low	High	Partition Point = Low + (value – q[Low])*(High-Low)/(q[High]-q[Low])	Comparison
2	4	2+(15-11)*(4-2)/(20-11)=2	15>(q[2]=11) and (15<q[3]=16) So, break

Figure 4.16: Step 4 of Serial Implementation

Now we can see that all the elements of the P array are copied into the B array. But some of the element is not copied of the Q array. So rest of the elements of the Q array is copied into B array.

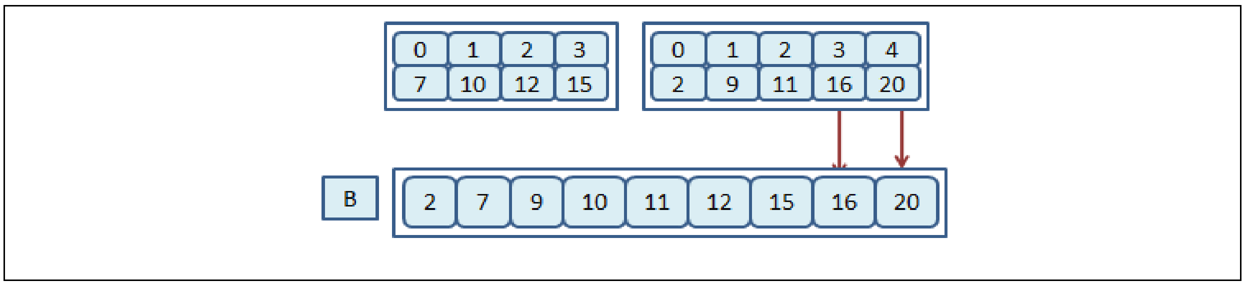


Figure 4.17: Step 5 of Serial Implementation

Now we pick last two buffer say P and Q.

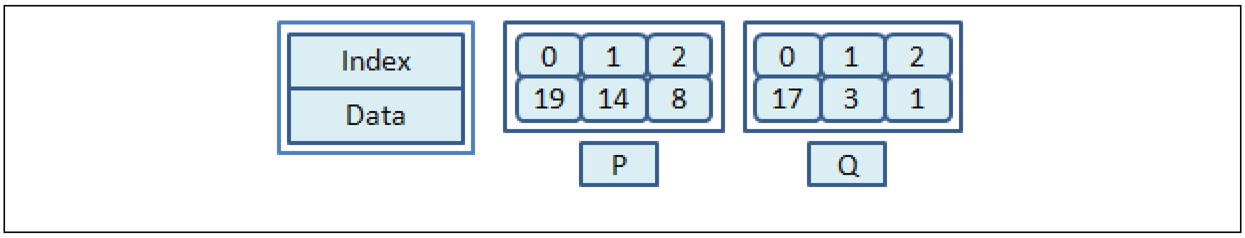


Figure 4.18: Last Two buffer for Serial Implementation

Here P and Q array are descending order. If P array is descending then we start from backward. So, 8 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 2. Note: Here interpolation search return the position 1that is count from the backward in Q array. Then pull out all the element of buffer Q that is less than 8 and search value of buffer P into another array B.

Low	High	Partition Point = Low + (value – q[Low])*(High-Low)/(q[High]-q[Low])	Comparison
0	2	0+(8-1)*(2-0)/(17-1)=0	8>(q[0]=1)
1	2	1+(8-3)*(2-1)/(17-3)=1	8>(q[1]=3) and 8<(q[2]=17) so break

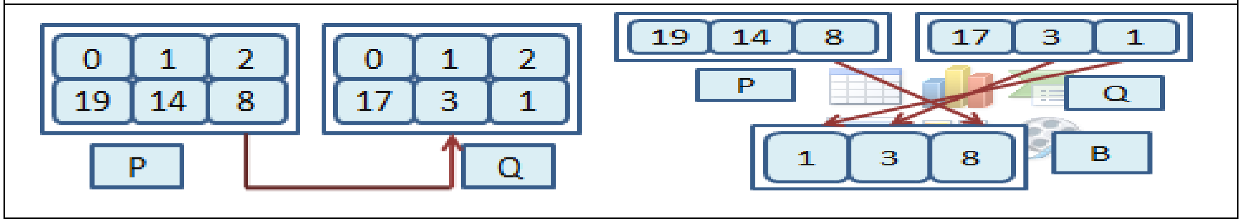


Figure 4.19: Step 6 of Serial Implementation

Now 14 will search the appropriate position in the buffer Q using interpolation search from the position 1 to 2. Here, 1 and 3 are already copy into the B array. So, only 14 will be copied into the B array.

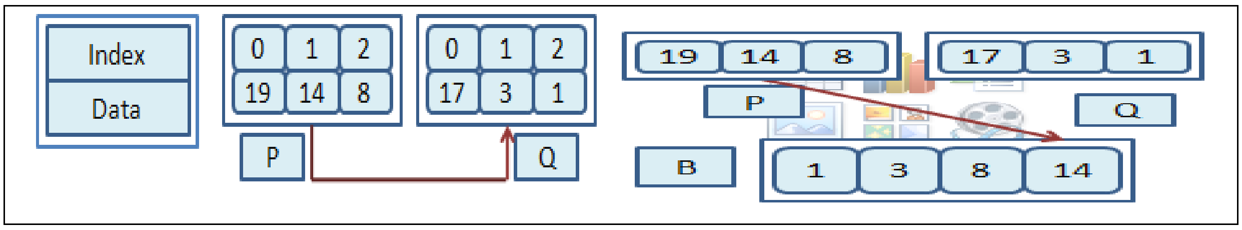


Figure 4.20: Step 7 of Serial Implementation

Here, we can see that 19 is greater than the first element of the second buffer and Q array is descending. All the elements of the Q array are less than 19. So we directly copy rest of the elements of the Q array into B array. And then 19 will be copied into the B array.

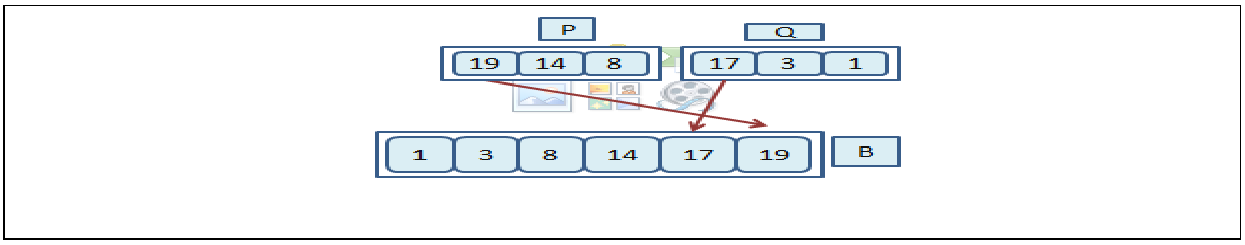


Figure 4.21: Step 8 of Serial Implementation

Then we merge two sorted array in previous way. So finally we get,

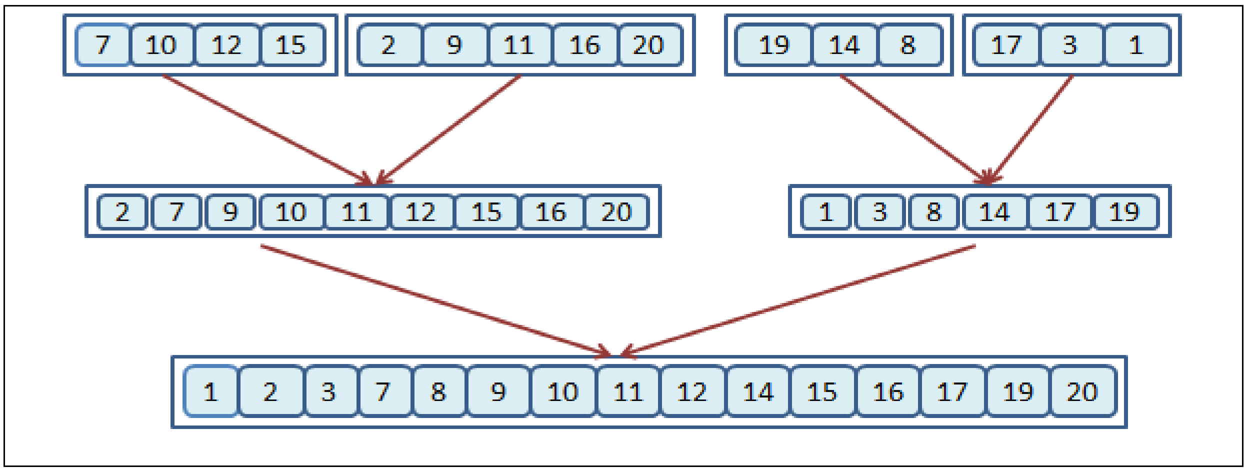


Figure 4.22: Final Sorted Array of Serial Implementation

4.2 Parallel Implementation

In this section we are going to describe interpolation search based sorting algorithm with openMP, binary search based sorting algorithm with openMP, interpolation search based sorting algorithm with two threads, binary search based sorting algorithm with two threads, interpolation search based sorting algorithm with multi-threads and binary search based sorting algorithm with multi-threads.

4.2.1 Interpolation Search based Sorting Algorithm with OpenMP

Now we implement interpolation search based sorting algorithm with openMP. The algorithm can be better understood if preceded by an example. So this section will aim at explaining the algorithm with an example.

For n element data set, we first make some buffers according to their sequential order, the order may be in ascending or descending, the running time will need O(n). Each buffer will get information about the starting index and the ending index of the sequential sorted data and also a flag which will provide us the order of the sequential data (flag 0 means ascending, flag 1 means descending order), this flag will be needed only in the first level comparisons but others time no need to check.

Suppose we have a dataset {7,10,12,15,2,9,11,16,20,19,14,8,17,3,1}. So the partition will be {7, 10, 12, 15}, {2, 9, 11, 16, 20}, {19, 14, 8}, {17, 3, 1}.

In table 2, we consider the Figure 4.2.1.1, the buffer L1 = <7, 10, 12, 15>, where 7 is the first and 15 is the last element of this particular dataset. So the starting index is 0 and ending index is 3. Dataset 7 to 15 is in ascending order, so the flag is set to 0.

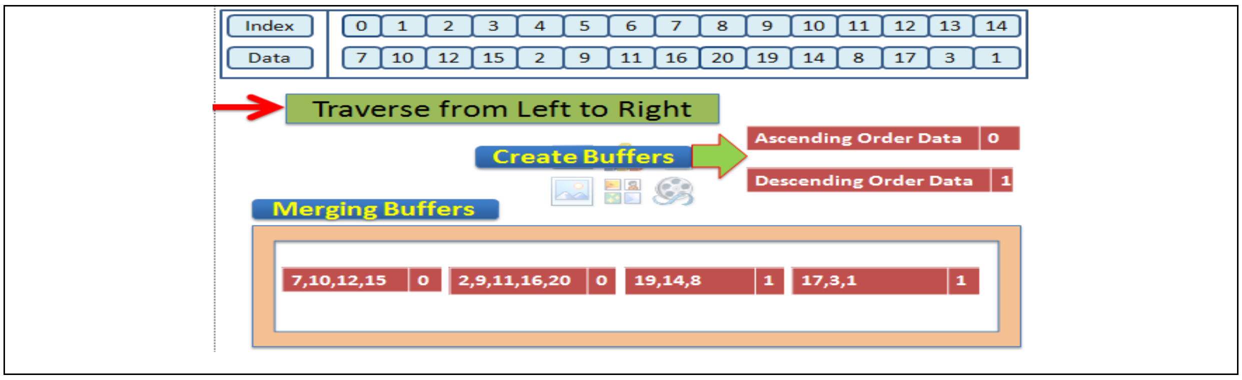


Figure 4.23: Create Buffer for OpenMP Implementation

Table 2: The buffer information

Starting index	0
Ending index	3
Flag	0

Now, we pick first two buffer say P and Q.

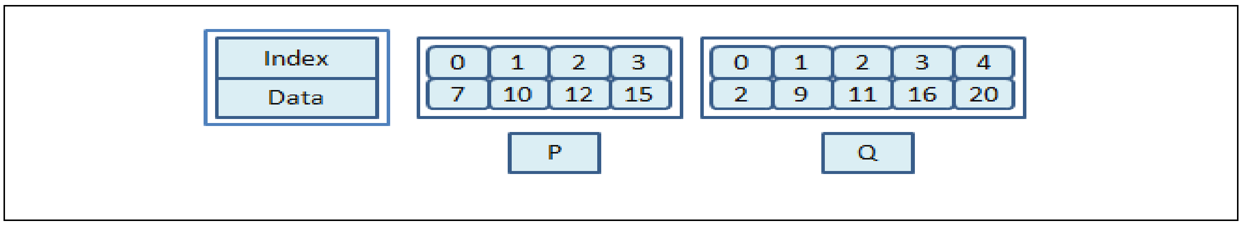
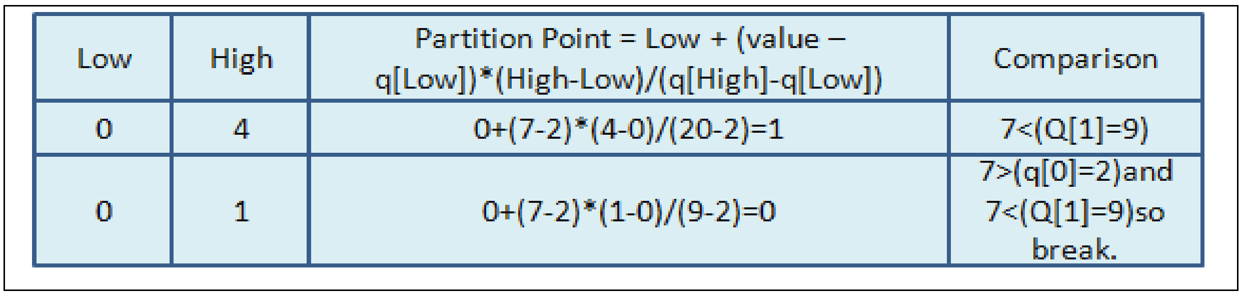


Figure 4.24: First Two Buffer for OpenMP Implementation

There are many way for merging this array P and Q when we use openMP. One way is explained below:

Suppose, 7 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 4.

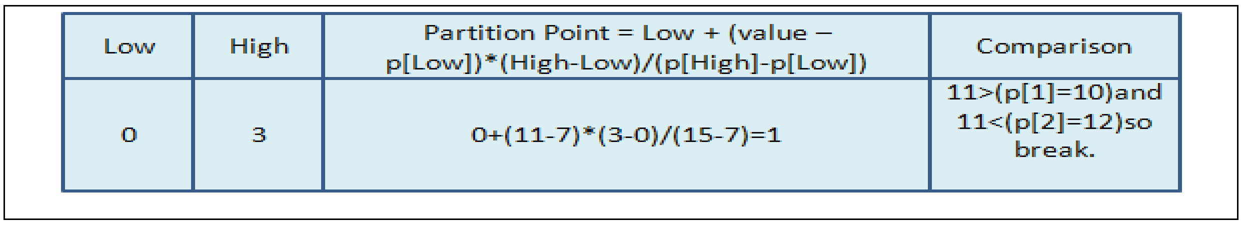


Low	High	Partition Point = Low + (value – q[Low])*(High-Low)/(q[High]-q[Low])	Comparison
0	4	0+(7-2)*(4-0)/(20-2)=1	7<(Q[1]=9)
0	1	0+(7-2)*(1-0)/(9-2)=0	7>(q[0]=2)and 7<(Q[1]=9)so break.

Figure 4.25: Searching Mechanism using Interpolation Search Ex. 1.

So, the position is 0.

Another example, 11 will search the appropriate position in the buffer P using interpolation search from the position 0 to 3.



Low	High	Partition Point = Low + (value – p[Low])*(High-Low)/(p[High]-p[Low])	Comparison
0	3	0+(11-7)*(3-0)/(15-7)=1	11>(p[1]=10)and 11<(p[2]=12)so break.

Figure 4.26: Searching Mechanism using Interpolation Search Ex. 2.

So, the position is 1.

At first, 7 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 4. At the same time 2 will search the appropriate position in the buffer P using interpolation search from the position 0 to 3. Here 2 is less than the first value of the P array so 2 is directly copy into B array. the index for value 2 in Q buffer is 0. So copy 2 into the position 0 in B array. The interpolation search return position 0 for value 7. The index for value 7 in P buffer is 0. So 7 push in the index (0+0+1=1) in the B array.

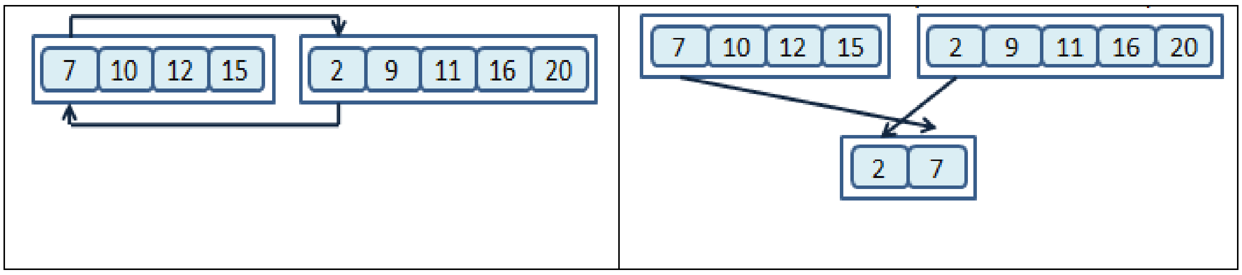


Figure 4.27: Step 1 of OpenMP Implementation using Interpolation Search

Then, 10 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 4. The interpolation search returns position 1 for value 10. The index for value 10 in P buffer is 1. So 10 push in the index (1+1+1=3) in the B array.

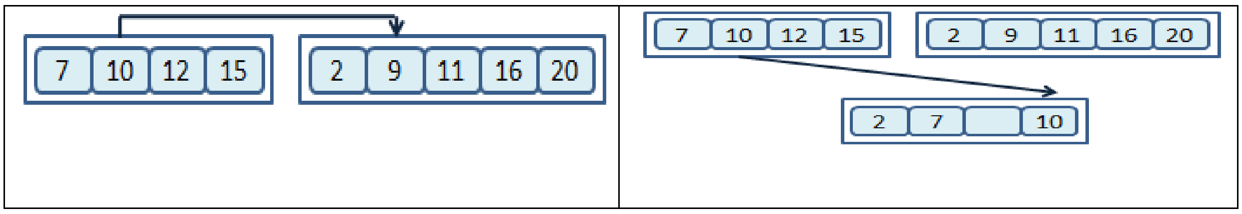


Figure 4.28: Step 2 of OpenMP Implementation using Interpolation Search

12 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 4. At the same time 9 will search the appropriate position in the buffer P using interpolation search from the position 0 to 3. The interpolation search returns position 2 for value 12. The index for value 12 in P buffer is 2. So 12 push in the index (2+2+1=5) in the B array. And the interpolation search return position 0 for value 9. The index for value 9 in Q buffer is 1. So 9 push in the index (0+1+1=2) in the B array.

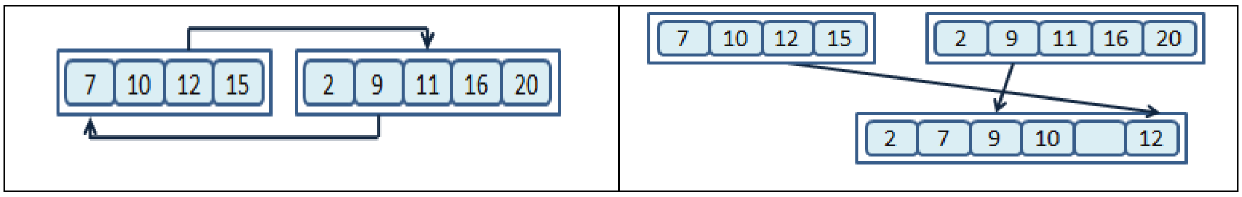


Figure 4.29: Step 3 of OpenMP Implementation using Interpolation Search

11 will search the appropriate position in the buffer P using interpolation search from the position 0 to 3. The interpolation search returns position 1 for value 11. the index for value 10 in Q buffer is 2. so 10 push in the index (1+2+1=4) of the B array.

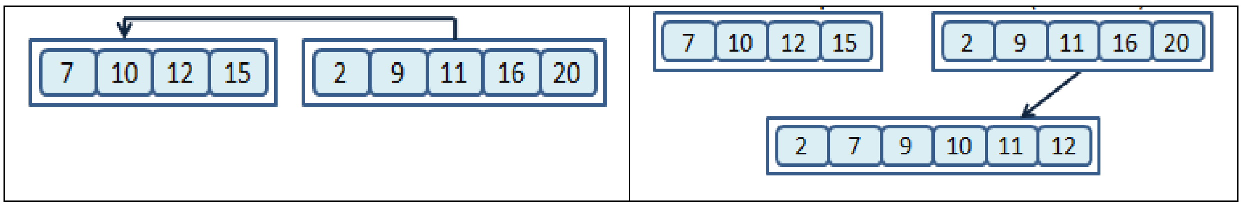


Figure 4.30: Step 4 of OpenMP Implementation using Interpolation Search

15 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 4. At the same time 16 will search the appropriate position in the buffer P using

interpolation search from the position 0 to 3. If the search value of the Q is greater than the last position value of the P then copy search value into the location (last index of the P array - first index of the P array + search value position in Q array + 1). Here 16 is greater than the last position value of the P array. The index for value 16 in Q buffer is 3. So 16 push in the index (3-0+3+1=7) of the B array. And 15 push in the index (2+3+1=6) of the B array.

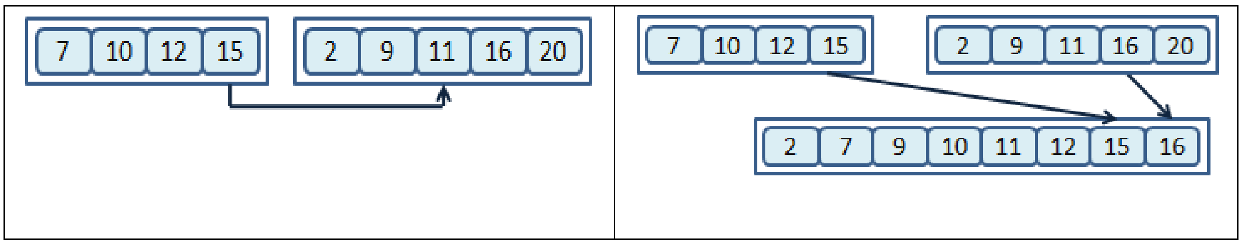


Figure 4.31: Step 5 of OpenMP Implementation using Interpolation Search

Here 20 is greater than the last position value of the P array. The index for value 20 in Q buffer is 4. So 20 push in the index (3-0+4+1=8) of the B array.

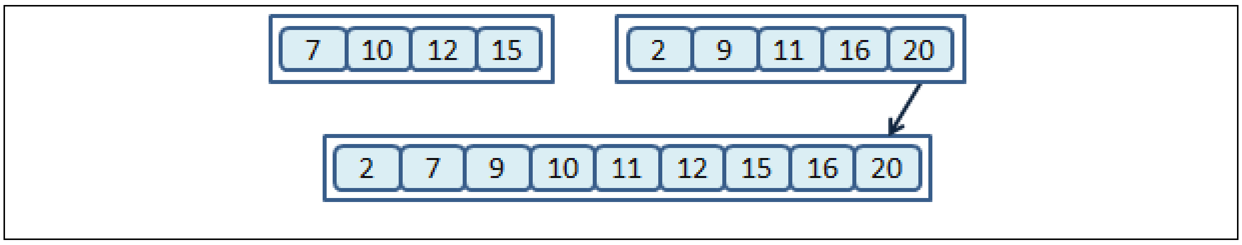


Figure 4.32: Step 6 of OpenMP Implementation using Interpolation Search

So, two threads had done their work. Then last two block goes to another two threads. So finally, we get,

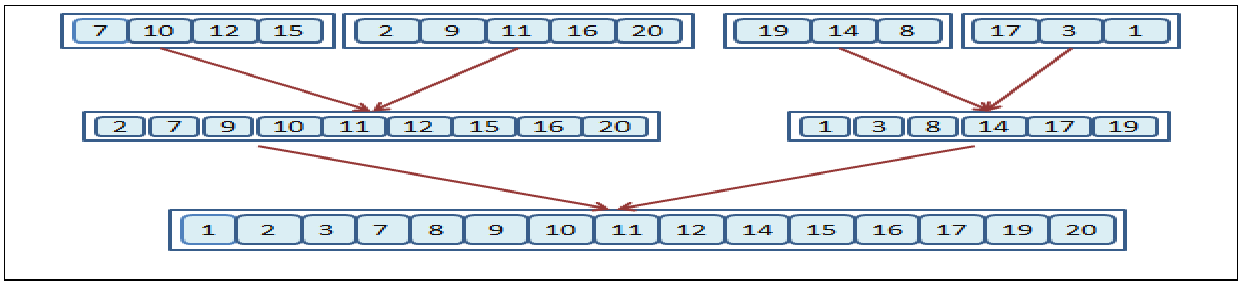


Figure 4.33: Final Sorted Array of openMP Implementation using Interpolation Search

4.2.2 Binary Search based Sorting Algorithm with OpenMP

Now we implement binary search based sorting algorithm with openMP. The algorithm can be better understood if preceded by an example. So this section will aim at explaining the algorithm with an example.

Suppose we want to sort this two array using binary search with openMP.

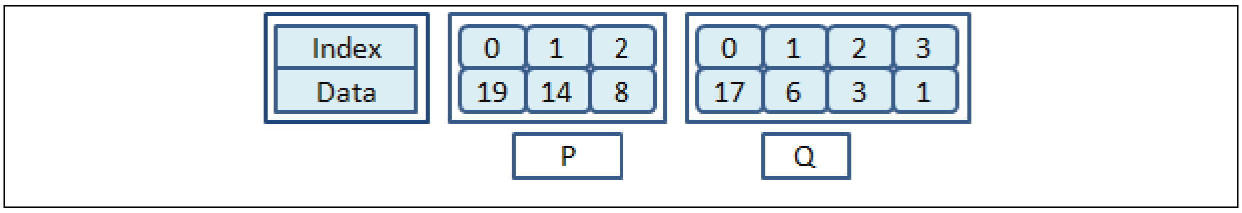
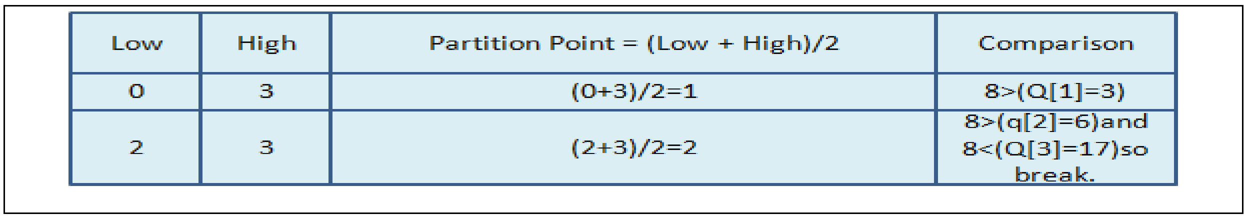


Figure 4.34: Array for Merging

There are many way for merging this array when we use openMP. One way is explained below:

Suppose, 8 will search the appropriate position in the buffer Q using binary search from the position 0 to 3.



Low	High	Partition Point = (Low + High)/2	Comparison
0	3	(0+3)/2=1	8>(Q[1]=3)
2	3	(2+3)/2=2	8>(q[2]=6)and 8<(Q[3]=17)so break.

Figure 4.35: Searching Mechanism using Binary Search.

Note: Partition point in the array is counted from backward.

So, the partition point is 2.

8 will search the appropriate position in the buffer Q using binary search from the position 0 to 3. At the same time 1 will search the appropriate position in the buffer P using binary search from the position 0 to 2. The binary search return position 2 for value 8.we count to from backward in the Q array. the index for value 8 in P buffer is (total size of the P array-index of the Search value) 0. so 8 push in the index (2+0+1=3) in the B array. And 1 is less than the last position value of the P array. So all the value of P array is greater than 1. The

index for value 1 in Q buffer is (total size of the q array-index of the Search value) 0. So 1 directly pushed at position 0 in the B array.

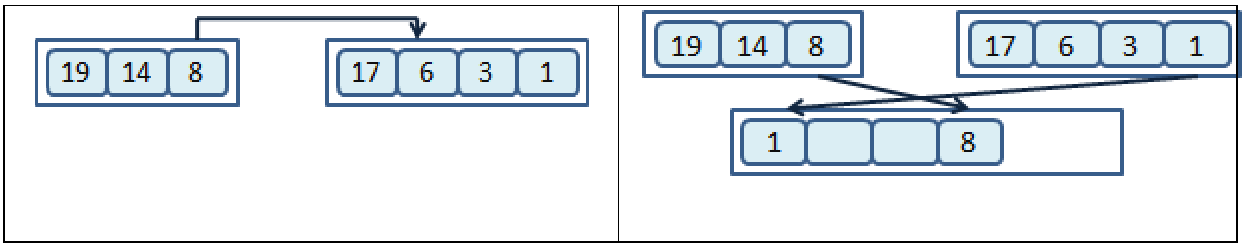


Figure 4.36: Step 1 of OpenMP Implementation using Binary Search

3 will search the appropriate position in the buffer P using binary search from the position 0 to 2. here 3 is less than the last position value of the P array. the index for value 3 in Q buffer is (3-2) 1. so 3 directly pushed at position 1 in the B array.

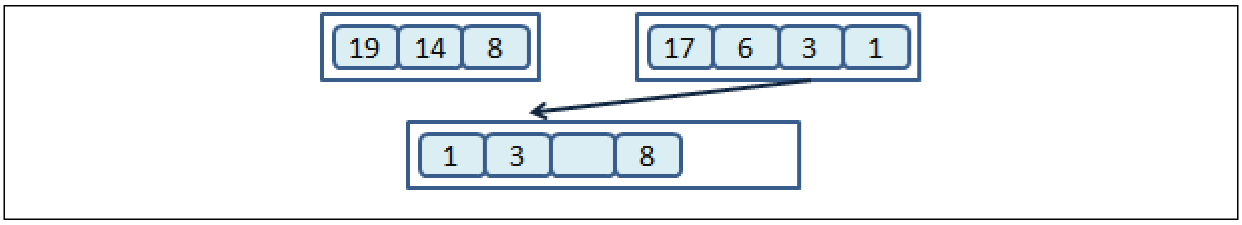


Figure 4.37: Step 2 of OpenMP Implementation using Binary Search

6 will search the appropriate position in the buffer P using binary search from the position 0 to 2. Here 6 is less than the last position value of the P array. The index for value 6 in Q buffer is (3-1) 2. So 6 directly pushed at position 2 in the B array.

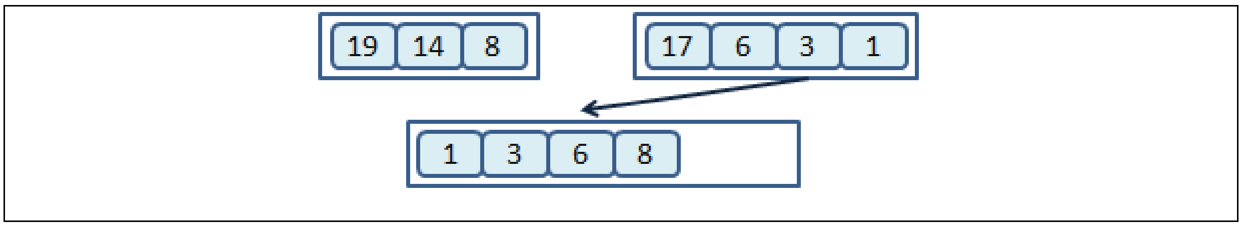


Figure 4.38: Step 3 of OpenMP Implementation using Binary Search

Then, 14 will search the appropriate position in the buffer Q using binary search from the position 0 to 3. At the same time 17 will search the appropriate position in the buffer P using binary search from the position 0 to 2. The binary search return position 2 for value 14. The index for value 14 in P buffer is (total size of the P array-index of the Search value) that is (2-

1) 1. so 14 push in the index (2+1+1=4) in the B array. And The binary search return position 1 for value 17. The index for value 17 in Q buffer is (total size of the P array-index of the Search value) that is (3-0) 3. so 17 push in the index (1+3+1=5) in the B array.

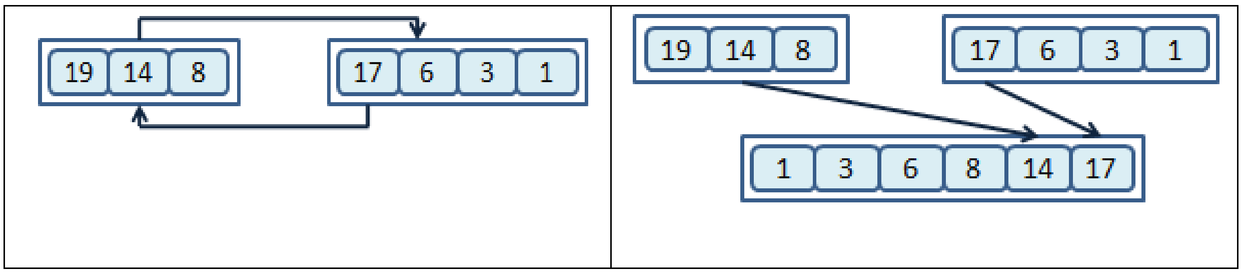


Figure 4.39: Step 4 of OpenMP Implementation using Binary Search

If the search value of the P is greater than the first position value of the Q then copy search value into the location (last index of the Q array - first index of the Q array + search value position in P array + 1). Here 19 is greater than the first position value of the Q array. the index for value 19 in P buffer is (2-0) 2. so 19 push in the index (3-0+2+1=6) of the B array.

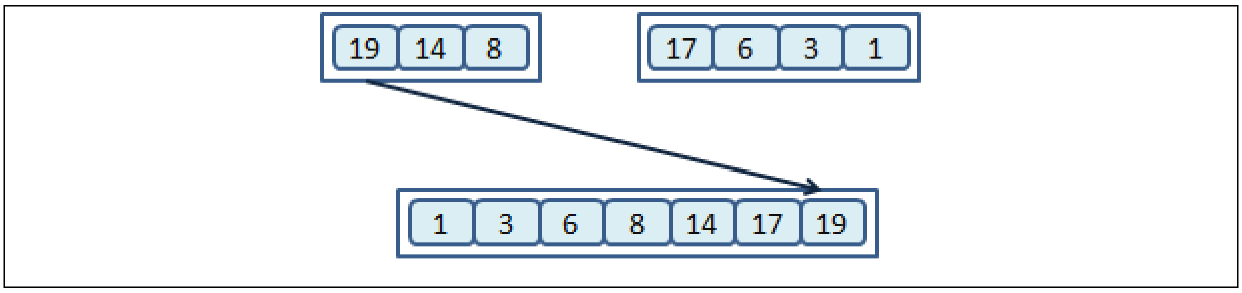


Figure 4.40: Step 5 of OpenMP Implementation using Binary Search

So finally we get,

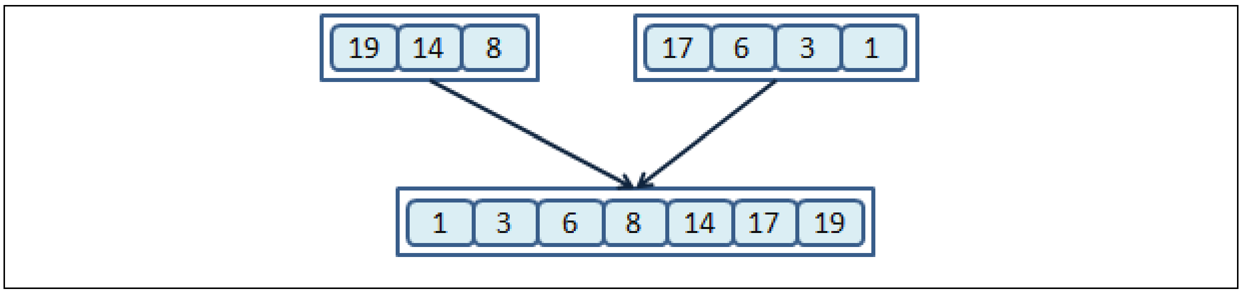
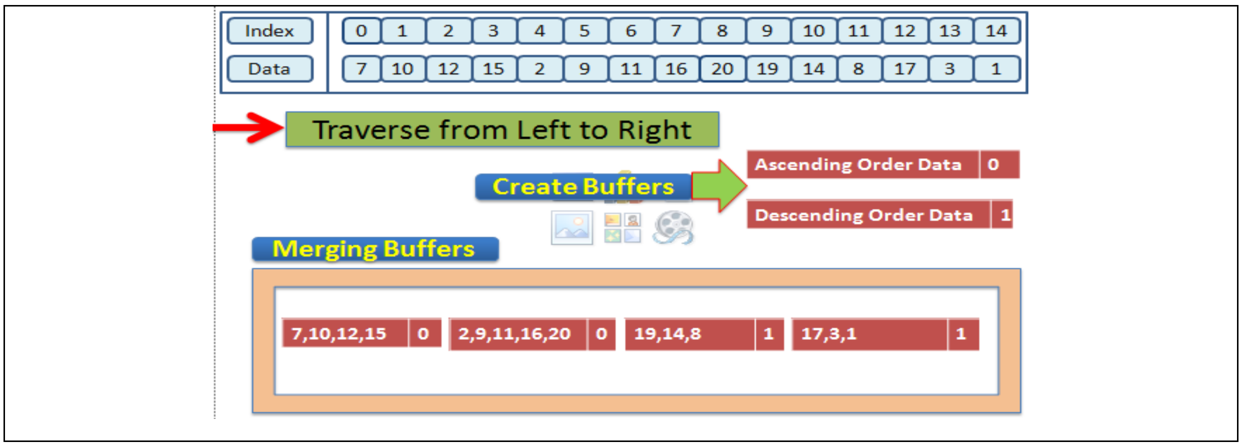


Figure 4.41: Final sorted array of OpenMP Implementation using Binary Search

4.2.3 Interpolation Search based Sorting Algorithm with Two Threads

Now we implement interpolation search based sorting algorithm with two threads. The algorithm can be better understood if preceded by an example. So this section will aim at explaining the algorithm with an example.



Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14
Data | 7 | 10 | 12 | 15 | 2 | 9 | 11 | 16 | 20 | 19 | 14 | 8 | 17 | 3 | 1
Traverse from Left to Right
Create Buffers
Ascending Order Data | 0
Descending Order Data | 1
Merging Buffers
7,10,12,15 | 0
2,9,11,16,20 | 0
19,14,8 | 1
17,3,1 | 1

Figure 4.42: Create Buffer for Two Threads Implementation

We merge this two array using interpolation search with two threads. At first P goes to one thread. At the same time Q goes to another thread. Now P and Q merge together. Because P and Q are independent.

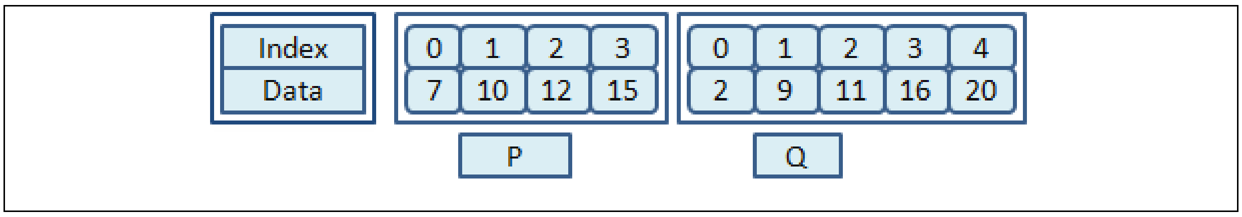


Figure 4.43: Array for Merging

Suppose, 7 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 4.

Low	High	Partition Point = Low + (value – q[Low])*(High-Low)/(q[High]-q[Low])	Comparison
0	4	0+(7-2)*(4-0)/(20-2)=1	7<(Q[1]=9)
0	1	0+(7-2)*(1-0)/(9-2)=0	7>(q[0]=2)and 7<(Q[1]=9)so break.

Figure 4.44: Searching Mechanism using Interpolation Search Ex. 1.

So, the position is 0.

Another example, 11 will search the appropriate position in the buffer P using interpolation search from the position 0 to 3.

Low	High	Partition Point = Low + (value – p[Low])*(High-Low)/(p[High]-p[Low])	Comparison
0	3	0+(11-7)*(3-0)/(15-7)=1	11>(p[1]=10)and 11<(p[2]=12)so break.

Figure 4.45: Searching Mechanism using Interpolation Search Ex. 2.

So, the position is 1.

At first, 7 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 4. At the same time 2 will search the appropriate position in the buffer P using interpolation search from the position 0 to 3. Here 2 is less than the first value of the P array so 2 is directly copy into B array. the index for value 2 in Q buffer is 0. So copy 2 into the position 0 in B array. The interpolation search return position 0 for value 7. The index for value 7 in P buffer is 0. So 7 push in the index (0+0+1=1) in the B array.

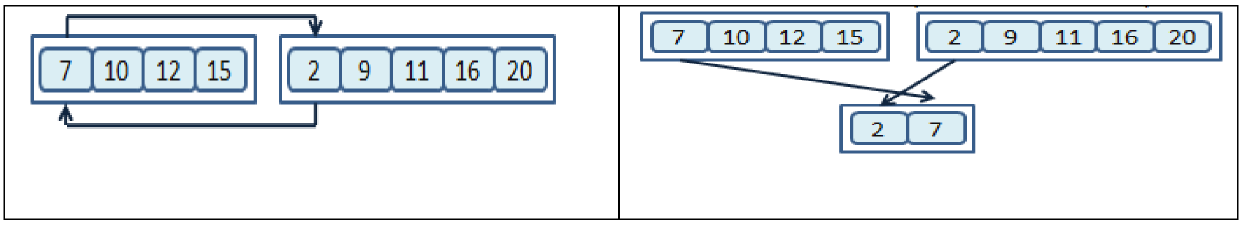


Figure 4.46: Step 1 of Two Threads Implementation using Interpolation Search

Then, 10 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 4. The interpolation search returns position 1 for value 10. The index for value 10 in P buffer is 1. So 10 push in the index (1+1+1=3) in the B array.

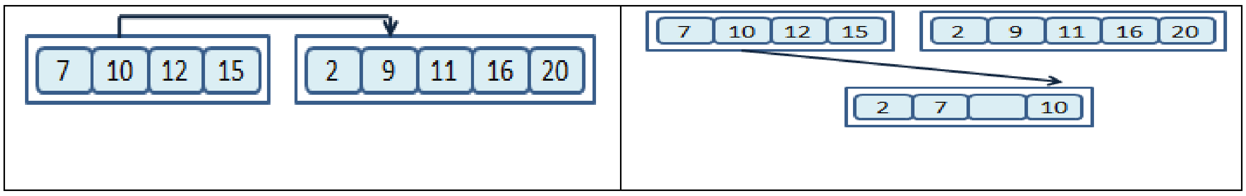


Figure 4.47: Step 2 of Two Threads Implementation using Interpolation Search

12 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 4. At the same time 9 will search the appropriate position in the buffer P using interpolation search from the position 0 to 3. The interpolation search returns position 2 for value 12. The index for value 12 in P buffer is 2. So 12 push in the index (2+2+1=5) in the B array. And the interpolation search return position 0 for value 9. The index for value 9 in Q buffer is 1. So 9 push in the index (0+1+1=2) in the B array.

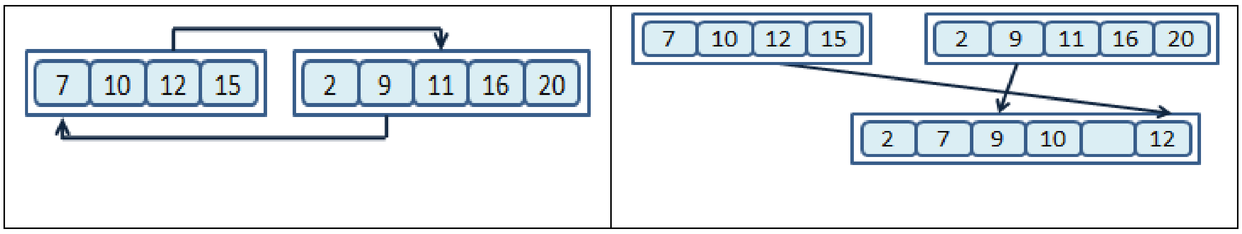


Figure 4.48: Step 3 of Two Threads Implementation using Interpolation Search

11 will search the appropriate position in the buffer P using interpolation search from the position 0 to 3. The interpolation search returns position 1 for value 11. the index for value 10 in Q buffer is 2. so 10 push in the index (1+2+1=4) of the B array.

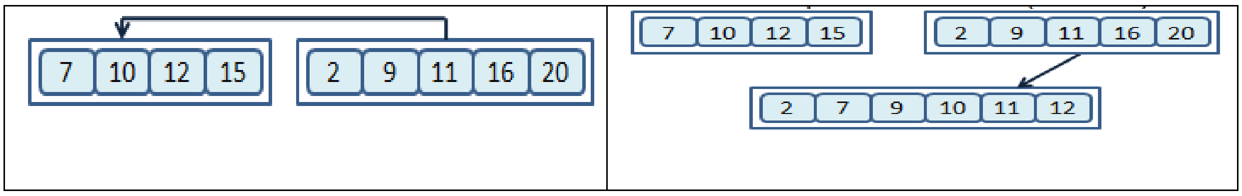


Figure 4.49: Step 4 of Two Threads Implementation using Interpolation Search

15 will search the appropriate position in the buffer Q using interpolation search from the position 0 to 4. At the same time 16 will search the appropriate position in the buffer P using interpolation search from the position 0 to 3. If the search value of the Q is greater than the last position value of the P then copy search value into the location (last index of the P array - first index of the P array + search value position in Q array + 1). Here 16 is greater than the last position value of the P array. The index for value 16 in Q buffer is 3. So 16 push in the index (3-0+3+1=7) of the B array. And 15 push in the index (2+3+1=6) of the B array.

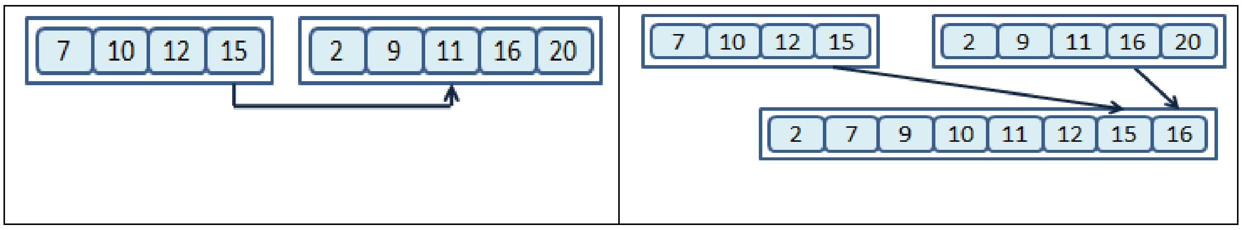


Figure 4.50: Step 5 of Two Threads Implementation using Interpolation Search

Then one thread is done his work but another thread is not done his work. So, one thread is waiting for another thread until his work is not done.

Here 20 is greater than the last position value of the P array. The index for value 20 in Q buffer is 4. So 20 push in the index (3-0+4+1=8) of the B array.

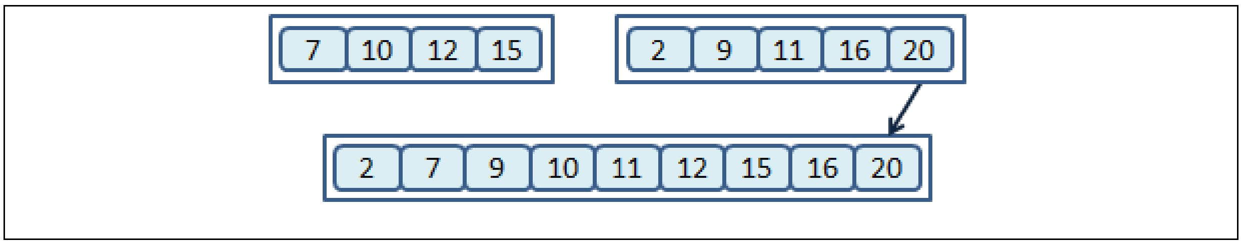


Figure 4.51: Step 6 of Two Threads Implementation using Interpolation Search

When two threads had done their work. Then copy the B array in the main array. So we get,

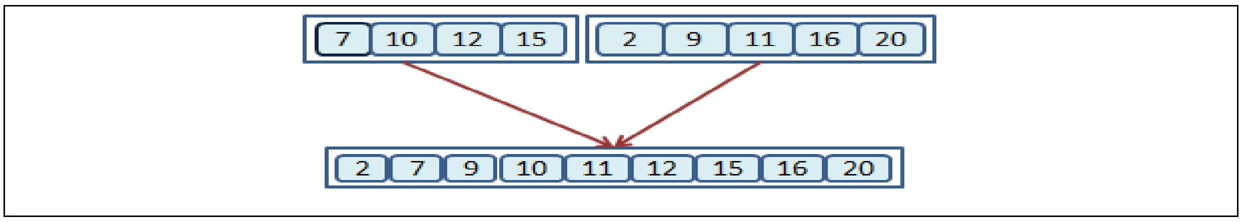


Figure 4.52: First array of Two Threads Implementation using Interpolation Search

Then we pick last two buffers. Then P goes to one thread and Q goes to another thread. So finally we get,

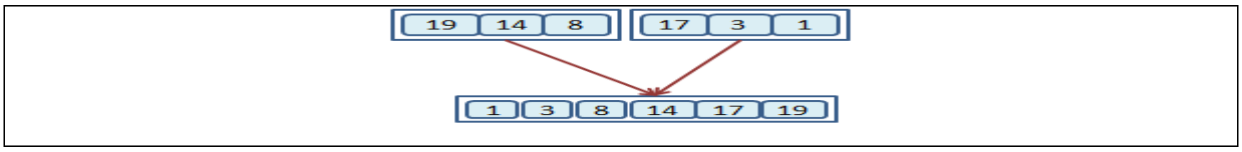


Figure 4.53: Second array of Two Threads Implementation using Interpolation Search

Now we pick first sorting buffer and second sorting buffer say P and Q. P goes to one thread and Q goes to another thread.

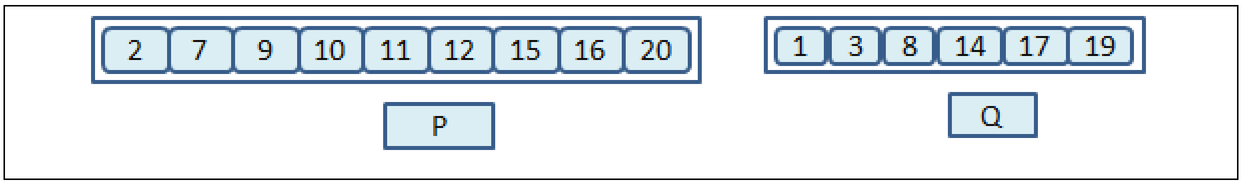


Figure 4.54: Array for merging

So finally we get,

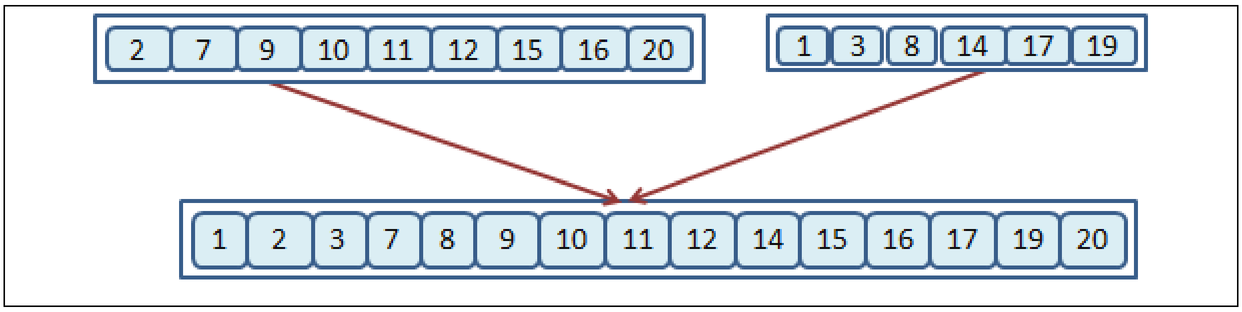


Figure 4.55: Final sorted array of Two Threads Implementation using Interpolation Search

4.2.4 Binary Search based Sorting Algorithm with Two Threads

Binary search with two threads procedure is same as the interpolation search with two threads procedure but only the difference is search mechanism is based on Binary search that is already discussed.

4.2.5 Interpolation Search based Sorting Algorithm with Multi-Threads

Suppose, There are 4 block in the dataset. We want to merge them. So at first 2 block goes to one thread. At the same time second two blocks goes to another thread. So we get,

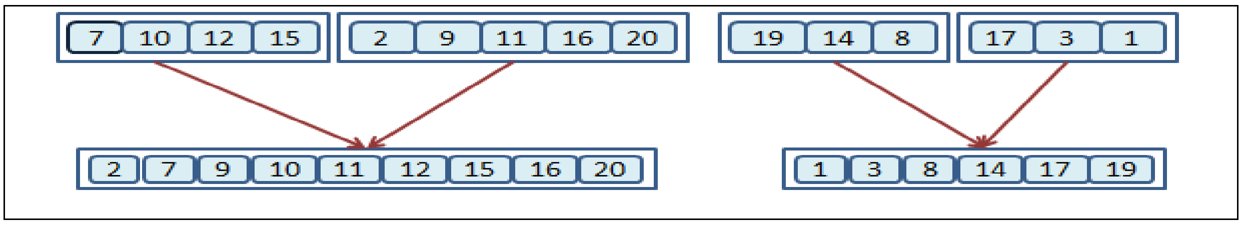


Figure 4.56: Step 1 of Multi-Threads Implementation

If one thread is done his then thread is wait until all of the thread is not done.

So finally remaining two buffers go to one thread for merging. So we get,

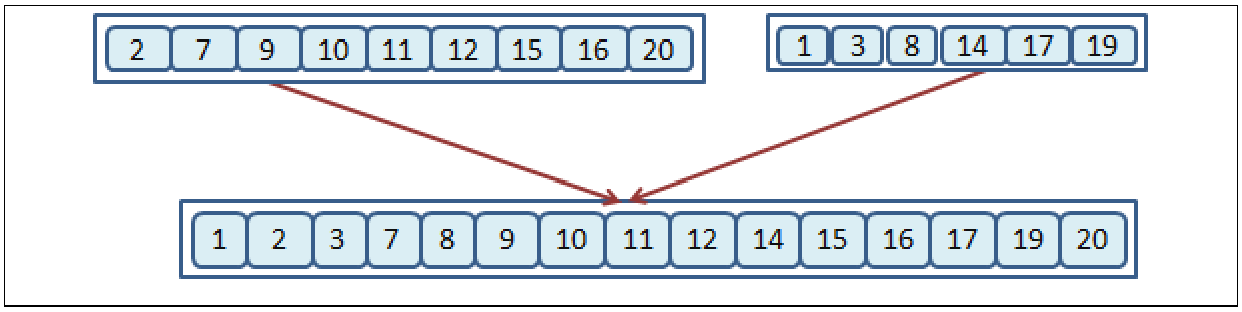


Figure 4.57: Final sorted array of Multi-Threads Implementation

4.2.6 Binary Search based Sorting Algorithm with Multi-Threads

Binary search with multi-thread procedure is same as the interpolation search with multi-thread procedure but only the difference is search mechanism is based on Binary search that is already discussed

Chapter 5

Experimental results and Analysis

A code implementing the above algorithm has been made and the working of the algorithm is checked. In this chapter we are going to describe the running time and the number of iteration's to sort an array of given size.

5.1 Results of serial implementation

The run time is compared with that of Binary Merging, Tape Merging, and Interpolation Merging.

Table 3: Run time statistics of Binary Merging, Tape Merging, and Interpolation Merging.

Number Of Data	Binary Merging	Tape merging	Interpolation Merging
50	0.0000	0.0000	0.0000
100	0.0000	0.0015	0.0000
500	0.0000	0.0015	0.0000
1000	0.0000	0.0016	0.0000
5000	0.0025	0.0093	0.0028
10000	0.0057	0.0187	0.0078
50000	0.0349	0.1013	0.0312
100000	0.0811	0.2013	0.0702
200000	0.1770	0.3884	0.1575

Here, we can see that interpolation merging is performing better than binary merging and tape merging scheme.

The expected speed-up of interpolation merge over binary merge implementation is about 2.7. To confirm this we run a simulation experiment and obtained result. The process was as follows: ten sets each of 1000 pseudorandom numbers chosen between 0 and 1000 were generated with the random number generator function Rand. Each set was sorted and merged using our proposed technique and the number of comparisons noted. The results of the experiment are shown in the following table.

Table 4: Comparison the average number of iterations of Binary merging and interpolation merging.

Data set Number	Comparison in Binary merging	Comparison in interpolation merging	Ratio of comparisons in interpolation merging to comparisons in Binary merging
1	18218	6698	2.72
2	18511	6795	2.72
3	18363	6742	2.72
4	18046	6718	2.69
5	18194	6736	2.70
6	17955	6578	2.73
7	18046	6594	2.74
8	18796	6937	2.71
9	18236	6769	2.69
10	18437	6807	2.71

5.2 Result of parallel Implementation

In this section we are going to describe result of OpenMp's implementation, result of Pthreads implementation.

Table 5: Run time statistics of OpenMP implementation and Pthread implementation of interpolation merging.

Number Of Data	OpenMP implementation	Pthread Implementation
50	0.0000	0.0010
100	0.0000	0.0020
500	0.0005	0.0100
1000	0.0007	0.0250
5000	0.0054	0.1470
10000	0.0122	0.2690
50000	0.0861	1.4360
100000	0.2945	3.0570
200000	1.0855	5.0940

Here, we can see that, OpenMP implementation is performing better than Pthread implementation of Interpolation merging.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We explore the new sorting algorithm based on interpolation merging. The expected speed up both theoretical and experimental, indicate a constant factor (~2.7) speed up over binary merging scheme. Also interpolation merging based sorting algorithm is implemented in parallel system with OpenMP and Pthreads. Openmp implementation is performing better than Pthread implementation.

6.2 Future work

If we implement interpolation search based sorting algorithm in GPU based system then performance will be improved.

References

[1] F. K. Hwang and S. Lin, “A simple algorithm for merging two disjoint linearly-ordered sets,” SIAM J. Comput ., Vol. 1, No. 1, March 1972.

[2] Dr. Md. Shamim Akhter and M. Tanveer Hasan, “Sorting N-Elements Using Natural Order: A New Adaptive Approach,” Journal of Computer Science 6(2): 163-167, 2010, ISSN: 1549-3636.

[3] https://en.wikipedia.org/wiki/OpenMP, 10-1-2016.

[4] Kronrod, M. A. (1969) ‘An optimal ordering algorithm without a field operation’, Dokladi Akad. Nauk SSSR, 186, 1256–1258.

[5] Horvath, E. C. (1978) ‘Stable sorting in asymptotically optimal time and extra space, Journal of the ACM, 25, 177–199.

[6] Pardo, L. T. (1977) ‘Stable sorting and merging with optimal space and time bounds, SIAM Journal on Computing, 6, 351–372.

[7] Dvorak, S. and Durian, B. (1988) ‘Unstable linear time O(1) space merging’, The Computer Journal, 31, 279–282.

[8] Mannila, H. and Ukkonen, E. (1984) ‘A simple linear-time algorithm for in situ merging’, Information Processing Letters, 18, 203–208.

[9] Huang, B.-C. and Langston, M. A. (1988) ‘Practical in-place merging, Communications of the ACM, 31, 348–352.

[10] Carlsson, S. (1986) ‘SPLITMERGE: a fast stable merging algorithm’, Information Processing Letters, 22, 189–192.

[11] Hwang, F. K. and Lin, S. (1972) ‘A simple algorithm for merging two disjoint linearly ordered sets’, SIAM Journal on Computing, 1, 31–39.

[12] Yehoshua Perl, Alon Ital and Haim Avni, “Interpolation search- A Log LogN search,” Communications of the ACM, July 1978, Volume 21, Number 7.