CSE4001 Literature Review

A Project

Submitted by

JayaSurya (20BCE1626)
Hari Krishna (20BCE1684)
Adrian. V. Martin (20BCE1847)
Akash (20BCE1919)

to

Dr. Venkataraman S
School of Computer Science and Engineering

in partial fulfillment of the the requirements for the course

of

CSE4001 – Parallel and Distributed Computing

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

NOVEMBER 2022

Base Paper: An Experimental Analysis of Parallel Sorting Algorithms by G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha

Abstract:

In computer science, a sorting algorithm is an algorithm that puts elements of a list into an order. Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists. Sorting is also often useful for canonicalizing data and for producing human-readable output.

The main goals of this research paper was to –

1. To implement as fast a sorting algorithm as possible for integers and floating-pont numbers.
2. To gain insight into practical sorting algorithms in general

The authors of this paper felt that the three most promising parallel sorting algorithms are Bitonic sort, Radix sort and Sample sort. When the number of keys per processor is large, it was discovered that Sample sort is the fastest. However it is not as efficient with small keys and is complicated to code in which case the Radix sort performs better. Even though the Bitonic sort seems to be the slowest when the no. of keys per processor is high, it is the most space efficient algorithm.

The running times of the sorts are modeled using equations based on problem size, number of processors, and a set of machine parameters. These equations serve several purposes. First, they make it easy to analyze how much time is spent in various parts of the algorithms. Second, they make it easy to generate good estimates of running times on variations of the algorithms. Third, it can be determined how various improvements in the architecture would improve the running times of the algorithms.

The remainder section of this research paper discusses the various applications and also considers other sorting algorithms. It concludes that with many keys per processor, the sample sort is approximately three times faster than the other two sorts and the Bitonic is superior if space is a major concern

Literature Survey:

1. Quick-Merge Sort Algorithm Based on Multicore Linux:

In this paper, the quick sort and merge sort has two type of execution, one is serial and parallel. In quick sort the reading time is reduced exponentially since it is executed parallelly. The block is split in each partition execution. Parallel quick sort achieves 40% to 50% better performance than serial execution. In Merge Sort, we split the data and merge two sorted arrays repeatedly to get the complete data. Parallel merge sort achieves 10% to 15% better performance than serial execution. In result it shows that parallel sort is always better than non-parallelized sort in quick and merge sort.

2. The performance analysis and research of sorting algorithm based on OpenMP:

In this paper, the bubble sorting is performed in OpenMP with different OMP scheduling like static, dynamic, guided and trapezoid, with and without chunk size. The performance of different combination is tabulated with different number of threads. The static scheduling shows lowest performance while dynamic scheduling shows the highest performance. The block size shows reduce in performance.

3. Communication-Efficient Bitonic Sort on a Distributed Memory Parallel Computer:

Bitonic sorting algorithm by Batcher produces a sorted sequence after a few iterations of bitonic merge which converts two bitonic sequences of size in each to one monotonic sequence of size 2m. However, if it is implemented on a distributed memory parallel computer, a great portion of overall execution time should be paid for the inter-processor communication. It is because the algorithm demands to exchange all the keys each processor has throughout the merge-and-split steps

In this paper, the authors have improved bitonic sort by optimizing the communication process of merge-and-split steps. Three communication pattems are identified to shorten the communication and the associated computation times. They have achieved a maximal reduction of 48% in total execution time when the input keys have uniform distribution, with 128M keys on 64-processor CRAY T3E. Also, it has been observed that the performance is not very sensitive to the key distribution. The improvement is also insensitive to the input size with the same number of processors, however, better improvement can be achieved as the number of processor is increased.

4. Sequential & Parallel Hybrid Approach for Non-Recursive Most Significant Digit Radix Sort:

Radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix. For elements with more than one significant digit, this bucketing process is repeated for each digit, while preserving the ordering of the prior step, until all digits have been considered. For this reason, radix sort has also been called bucket sort and digital sort.

The two goals of this research paper was to: come up with an alternative sequential most significant digit radix sort algorithm that performs better than the traditional recursive one and to implement the sequential and parallel versions of a hybrid combination of most significant digit radix sort algorithm.

In this paper, radix sort has two types of implementations. One is recursive MSD radix sort and the other one is non-recursive MSD radix sort. As the name suggests the first method is done using a recursive methodology. Numbers from the input vector are placed into buckets based on the value of the corresponding digits being examined and each of these buckets is sent to the recursive MSD function for further execution. In the second method a non-recursive methodology is followed. The process begins by calculating the MSD of each number in the main vector. After that it adds each number with corresponding digit value i(i=0…9) in the ith bucket of the first_bucket.

Hybrid Radix sort in sequential and parallel methods:

Hybrid radix sort involves a combination of MSD radix sort and quick sort algorithms. MSD radix sort creates individual vectors and quicksort is applied to sort each of the generated vectors.

In contrast with bitonic sort, radix sort is not a comparison sort; it does not use comparisons alone to determine the relative ordering of keys. Instead, it relies on the representation of keys as b-bit integers.

5. Performance Analysis of Counting Sort Algorithm using various Parallel Programming Models:

In this paper, Counting sort is implemented in OpenMp and MPI, and their performance in each is compared for their serial and parallel implementations. So, Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. It determines the amount of values fewer than each value after counting the instances of each value. The values are then organised according to their count in descending order. Place the Y value in the Xth position if there are X values that are smaller than Y. The temporal complexity of this approach, in both the average and worst cases, is O (N+K), where N is the total number of values in the array and K is the range of values that are present in the array. When compared, at the beginning the difference is not that much significant in the graph and when the values of array taken higher, then significant difference is observed, and speed up is discrete between the serial and parallel implementations.

Algorithms:

Bitonic sort:

The algorithm Bitonic sort was created by Ken Batcher in 1968. Bitonic sort is a comparison-based sorting technique· Elements are sorted depending on the bitonic sequence. A sequence is called Bitonic if it is first increasing, then decreasing. In other words, an array arr[0..n-i] is Bitonic if there exists an index i, where 0<=i<=n-1 such that

$$x0 <= x1 \ldots.. <= xi$$

and

$$xi >= xi+1 \ldots.. >= xn-1$$

Bitonic sequence can be formed by forming 4-element bitonic sequences from consecutive 2-element sequences. Consider 4-element in sequence x0, x1, x2, x3. We sort x0 and x1 in ascending order and x2 and x3 in descending order. We then concatenate the two pairs to form a 4 element bitonic sequence.

Next, we take two 4-element bitonic sequences, sorting one in ascending order, the other in descending order Bitonic Sort, and so on, until we obtain the bitonic sequence.

To form a sorted sequence of length n from two sorted sequences of length n/2; log(n) comparisons are required.

Pseudocode of Parallel Bitonic Sort:

Input:

Ar[n]; an integer array of size n

Output:

```
bitonicSortFromBitonicSequence(int startIndex, int lastIndex, int dir, int *ar)
  if dir = 1
```

```
    counter = 0
    noOfElements = lastIndex - startIndex + 1
    for  j = noOfElements / 2 to 0,  decrementing j by j/2
        counter = 0
        for i = startIndex to i + j <= lastIndex, incrementing i by 1
             if counter < j
                  ascendingSwap(i, i + j, ar)
               increment counter by 1
           else
             counter = 0
             i = i + j - 1
  else
   counter = 0
   noOfElements = lastIndex - startIndex + 1
   for j = noOfElements / 2 to 0, decrementing j by j/2
        counter = 0
       for i = startIndex to lastIndex – j, incrementing i by1
            if counter < j
               decendingSwap(i, i + j, ar)
               increment counter by 1
           else
               counter = 0
               i = i + j – 1

bitonicSequenceGenerator(int startIndex, int lastIndex, int *ar)
 int noOfElements = lastIndex - startIndex + 1
   for  j = 2 to noOfElements, incrementing j by 2*j
     # omp for
      for  i = 0  to noOfElements, incrementing I by  i + j
           if ((i / j) % 2) = 0 , bitonicSortFromBitonicSequence(i, i + j - 1, 1, ar)
           else, bitonicSortFromBitonicSequence(i, i + j - 1, 0, ar)
```

Quick sort:

The algorithm quick sort Developed by British computer scientist Tony Hoare in 1959. Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort. The sub-arrays are then sorted recursively.

On the average, it has O(n log n) complexity, making quicksort suitable

for sorting big data volumes. So, it is important to make it parallel.As the thread creation time is far longer than the time to compare and exchange two elements in most cases, the algorithm cannot start new threads for each partitioning.

Instead, the number of elements to be partitioned is compared to a hard-coded limit, and if the number of elements is sufficiently large, it creates a new thread, otherwise it just makes a recursive function call.

Pseudocode of Parallel Quick Sort:

Input:

a[rj: A list of sequence, start, end

Output:

```
int Partition(arr,start,end)
    Pivot<-arr[end]
    I<-start - 1
    for each j from start to (end – 1)
        if (arr[j] < pivot)
                i++
                swap(arr[i], arr[j])
    swap(arr[i + 1], arr[end])
    return (i + 1)


void Quicksort(arr,start,end)
  if (start < end)
      index <- Partition(arr, start, end)
      OMP start sections
            OMP section
                Quicksort(arr,start,index-1)
            OMP section
                Quicksort(arr,start,index-1)
```

Bubble sort:

Parallel Bubble sort algorithm, also known as odd-even bubble sort, and can be implemented in different ways. The main idea in parallel bubble sort is to compare all pairs in the input list in parallel, then, alternate between odd and even phases. The main benefit of this is faster computation. Implementation of parallelism depends on some factors. For example, if there are parts of code that can be implemented independently and executed irrespective of any order, then we can execute at the same time. However, some parts of the program may still require other parts of the program to run first.

When comparing, just like in normal bubble sort, we swap the elements, if the initial element is greater than the next element in comparison. Both phases occur one after the other and the algorithm stops once no swap occurs in both odd and even phase

Pseudocode of Parallel Bubble Sort:

Input:

A[size]; an integer array

Output:

```
int main (int argc, char *argv[])
        int SIZE <- 1<<8;
        int A[SIZE];
        Start for
           A[i]=rand()%SIZE;
        End for
        int N <- SIZE;
        int i<-0, j<-0;
        int first;
        double start,end;
        start=omp_get_wtime();
        Start for
                first <- i % 2;
                #pragma omp parallel for default(none),shared(A,first,N)
                Start for
                        if( A[ j ] > A[ j+1 ] )
                                swap( &A[ j ], &A[ j+1 ] );
                End for
        End for
      end=omp_get_wtime();

void swap(int *num1, int *num2)
        int temp <- *num1;
        *num1 <-  *num2;
        *num2 <- temp;
```

Counting sort:

Parallel Counting sort sorts the values over specific range. It counts the number of occurrences of each value and then calculates the number of values less than each value. Then it places the values in sorted order based on the count of the values. Because the algorithm uses only simple for loops, without recursion or subroutine calls, it is straightforward to analyze. The initialization of the count array, and the second for loop which performs a prefix sum on the count array, each iterate at most k + 1 times and therefore take O(k) time. The other two for loops, and the initialization of the output array, each take O(n) time. Therefore, the time for the whole algorithm is the sum of the times for these steps, O(n + k).

Pseudocode of Parallel Counting Sort:

Input:

a[rj: A list of sequence, start, end

Output:

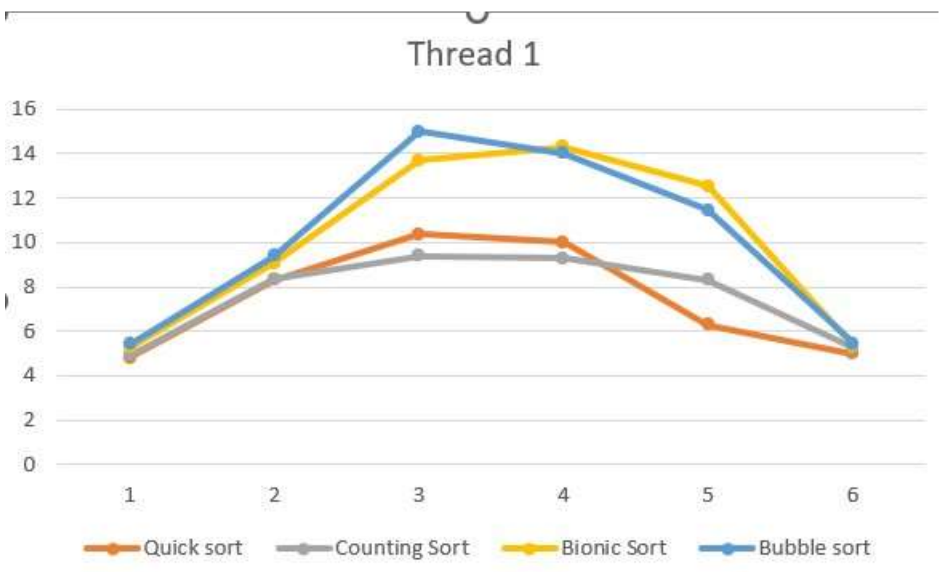
```
void Quicksort()
 initialize array[N]
 min <- array[0]
 max  <- array[0]
 For i <- 1 to N
    If array[i] < min
       min <- array[i]
   If array[i] > max
      max <- array[i]
Range <- max – min + 1
Initialize count[range+1]
  #omp parallel for
  For i <- 0 to N
    count[ array[i] - min ] <- count[ array[i] - min ] + 1
  end #omp for
initialize z <- 0
 for i <- min to max
   for j <- 0 to count[ i - min ]
       array[z++] <- i
```

Observation:

The graph shown are tabulated with CPU usage per unit time of executing the process with difference in thread number.

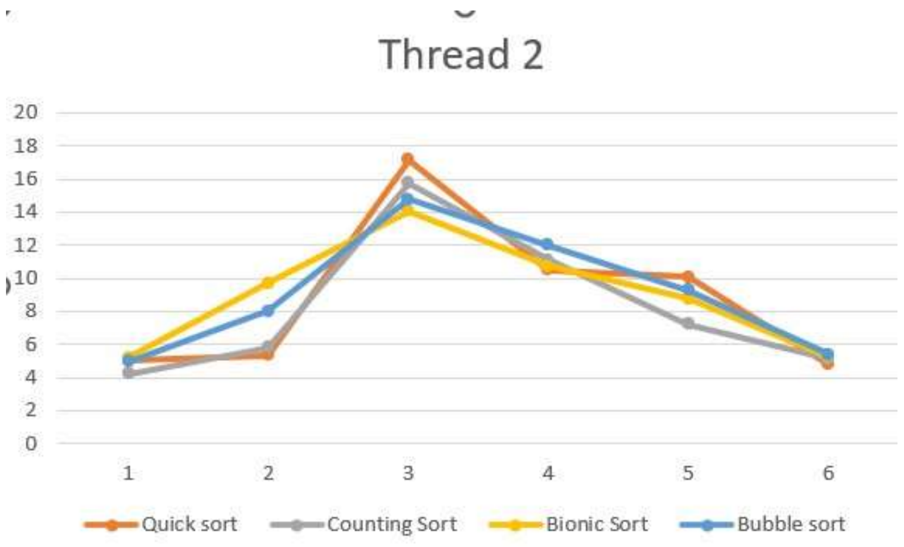
Thread 1:

As the CPU usage percentage increases quick sort experiences, a gradual increase but after a particular period of time there's a decrease in efficiency. Counting sort shows the least increase with CPU usage percentage and also shows a slow decrease after a certain period of time. Bitonic sort shows a drastic increase when compared to the other 2 and the decrease is also gradual after a certain period of time. Bubble sort shows the highest increase with CPU usage percentage and immediately shows a drastic increase after it reaches its peak.



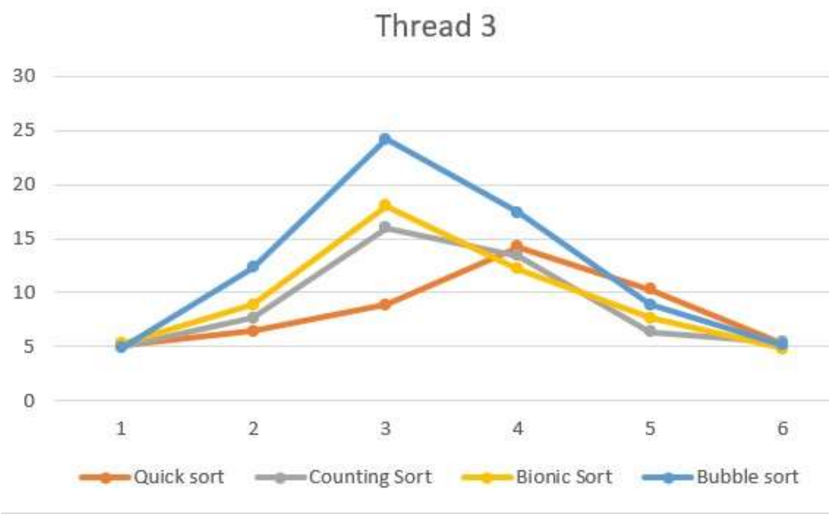
Thread 2:

Quick sort shows the highest peak followed by counting sort, bubble sort and Bitonic sort respectively. All the sorting algorithms experience a gradual decrease and this decrease is quite similar in each sort. The peak is also reached quicker when compared to thread 1.



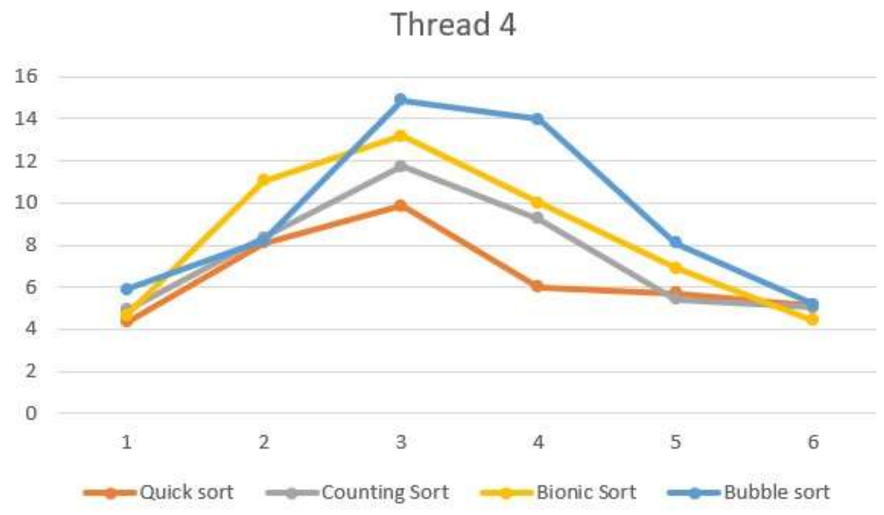
Thread 3:

Bubble sort shows the highest peak followed by Bitonic sort, counting sort and quick sort. The peak is experienced a bit later when compared to thread 2 and the regression is also quite slow when compared to thread 2. All the sorting algorithms also a reach a higher peak in thread 3 when compared to all the threads.



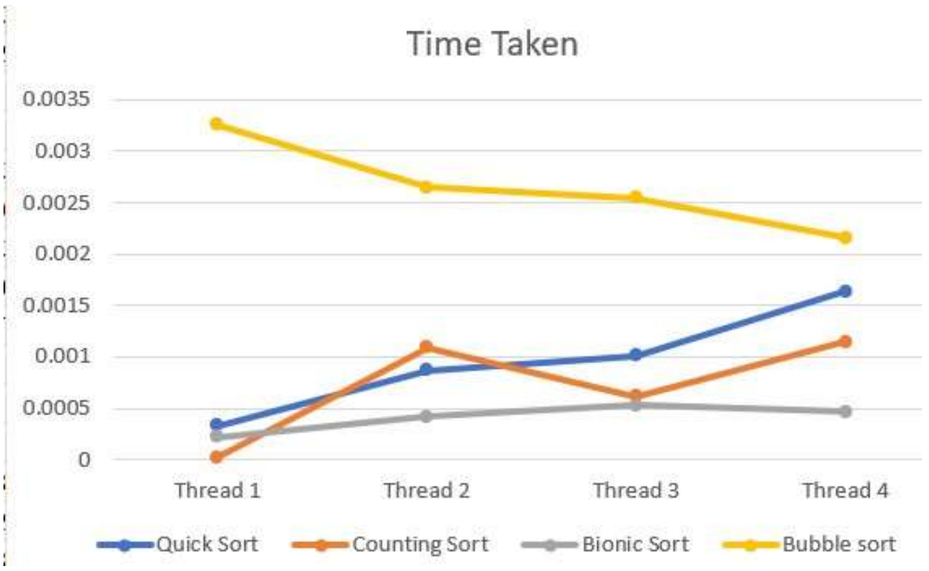
<u>Thread 4:</u>

Thread 4 follows the same pattern as thread 3. The peak reached here is lower than the ones achieved in thread 2 and 3. The regression is even slower when compared to thread 3 and it plateaus in the case of bubble sort.



RESULTS OF THE EXPERIMENT:

This experiment we implemented OM Parallel to find which sorting algorithm shows high performance for 1000 integers and tabulated with CPU usage and memory usage. GCC compiler is used to compile the program. The Bitonic sort shows high performance and the second algorithm which shows performance closer to Bitonic is counting. Quick sort executes faster but increase in thread increases the time taken to execute. Bubble sort shows the worst time take for parallel execution in the all the algorithms we did in the paper. **We conclude that Bitonic is best suited for parallel sorting in this paper.**



References:

[1] Blelloch, G., Leiserson, C., Maggs, B. et al. An Experimental Analysis of Parallel Sorting Algorithms . Theory Comput. Systems 31, 135–167 (1998)

[2] Yong Liu and Yan Yang, "Quick-merge sort algorithm based on Multi-core linux," Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC), 2013, pp. 1578-1583, doi: 10.1109/MEC.2013.6885313.

[3] Jing-mei Li and Jie Zhang, "The performance analysis and research of sorting algorithm based on OpenMP," 2011 International Conference on Multimedia Technology, 2011, pp. 3281-3284, doi: 10.1109/ICMT.2011.6001870.

[4] Aydin, Ahmet & Alaghband, Gita. (2013). Sequential & Parallel Hybrid Approach for Non-Recursive Most Significant Digit Radix Sort.

[5] RajasekharaBabu, M. & Khalid, Mohd & Soni, Sachin & ChowdaryBabu, Sunil & Mahesh,. (2011). Performance Analysis of counting sort algorithm using various parallel programming models. International journal of Computer Science and Information technologies, (IJCSIT). 2. 2284-2287.

[6] Yong Cheol Kim, Minsoo Jeon, Dongseung Kim Andrew Sohn. " Communication-Efficient Bitonic Sort on a Distributed Memory Parallel Computer*". Parallel and Distributed Systems, 2001. ICPADS 2001. Proceedings. Eighth International Conference of 2001, page no. 165- 170, 0-7695-1 153-8.