Comparative Study of Parallel Sorting Algorithms

Pushkar Sharma
Dept. of CSE
SRMSCET, Unnao
Uttar Pradesh, India
pushkar1291@gmail.com

Akhilesh Pratap Singh
Dept. of CSE
SRMSCET, Unnao
Uttar Pradesh, India
akhilesh.chauhan88@gmail.com

Dev Kant
Dept. of CSE
SRMSCET, Unnao
Uttar Pradesh, India
devkantg@gmail.com

***Abstract*— Sorting is a very common task in computing which is arranging elements in increasing or decreasing order. It is well known that the best we can do is sort n data elements in O(n log n) time. The question then arises, can we do better than this by employing parallel processing? It turns out that with an increase in cost we can sort n elements in O(1) time. However for this we will need a comparatively large number of processors which would be processing the elements simultaneously. In this paper we will study and compare the parallel sorting algorithms namely the Odd Even Transposition Sort, the Enumeration Sort, the Bitonic Sort and the Parallel Quick Sort.**

***Keywords*— running time; processors; cost, cost optimality; odd even transposition; enumeration; bitonic; parallel quick**

I. Introduction

Prior methods to sort elements made use of a single processor which implemented a sequential algorithm. With the advent of multiprocessor systems there began a rise of parallel algorithms which performed multiple steps of execution at the same time to give the results faster than those achieved from sequential processing. We study the parallel algorithms for sorting because though it is a common task, its need arises somewhere or the other in any application or an enterprise. In this paper we will study the parallel methods of sorting namely odd even transposition sort, enumeration sort, bitonic sort and parallel quick sort. After giving the description of these algorithms we provide the analysis of the same. Following this we perform a comparison between these algorithms and finally present the conclusion.

II. Terms and Descriptions

A. Running Time

It is the time taken by the algorithm to solve a problem on a parallel computer. If the various processors do not begin and end their computation simultaneously then the running time is equal to the time elapsed between the moment the first processor to begin computing starts and the moment the last processor to end computing, terminates.

B. Number of Processors

The number of processors a parallel algorithm requires to solve a problem is an important criteria in evaluating a parallel algorithm. This is because using a large number of processors to solve a problem is expensive due to its purchase and maintenance cost.

C. Cost

Cost of the parallel algorithm is measured by multiplying the number of processors and the running time.

$c(n)=t(n)*p(n)$

D. Cost Optimality

A parallel algorithm is cost optimal if the cost of the parallel algorithm is in the same complexity class as an optimal sequential algorithm, otherwise, the parallel algorithm is not cost optimal.

III. Parallel Algorithms for Sorting

We are now going to study the four widely used parallel algorithms for sorting.

A. Odd Even Transposition Sort

It works on a processor array model in which processing elements are organized in a 1 dimensional mesh. Let $A=(a_1,a_2,....a_n)$ be the unsorted sequence. Each of the n processors use two local variables; a - unique element of A, t - variable containing value retrieved from neighbouring processor. The algorithm performs n/2 iterations each with 2 phases. In the odd-even exchange value of a in all odd-numbered processors (except n-1) is compared with the value of a stored in successor processor. The values are exchanged or not such that the lower numbered processor contains the smaller value. In the even-odd exchange value of a in every even numbered processor is compared with the value of a in successor processor. The values are again exchanged or not such that the lower numbered processor contains the smaller value. After n/2 iterations the array is completely sorted.

```
procedure ODD EVEN TRANSPOSITION SORT(1D Mesh Processor Array)
    Parameter    n
    Global       i
    Local        a- Element to be sorted
                 t- Element taken from adjacent processor
    begin
    for i=1 to n/2 do
        for all $P_j$ where $0 \le j \le n-1$ do in parallel
            if j<n-1 and odd(j) then [Odd even exchange]
                t=successor(a)
                successor(a)=max(a,t)
                a=min(a,t)
            end if
            if even(j) then [Even odd exchange]
                t=successor(a)
                successor(a)=max(a,t)
                a=min(a,t)
            end if
        end for
    end for
end
```

Indices	0		1		2		3		4		5		6		7
Initially	V		W		U		S		T		R		Q		P
Odd-even	V		U	<	W		S	<	T		Q	<	R		P
Even-odd	U	<	V		S	<	W		Q	<	T		P	<	R
Odd-even	U		S	<	V		Q	<	W		P	<	T		R
Even-odd	S	<	U		Q	<	V		P	<	W		R	<	T
Odd-even	S		Q	<	U		P	<	V		R	<	W		T
Even-odd	Q	<	S		P	<	U		R	<	V		T	<	W
Odd-even	Q		P	<	S		R	<	U		T	<	V		W
Even-odd	P	<	Q		R	<	S		T	<	U		V	<	W

Fig. 1. Odd even transposition sort of eight values

After i iterations of outer for loop, no element can be farther than n-2i positions away from its final sorted position. Hence n/2 iterations are sufficient to sort the array giving t(n)=O(n). The number of processors involved are n giving p(n)=n. This results in cost, $c(n)=O(n^2)$. This results in it not being a cost optimal algorithm.

B. *Enumeration Sort*

In this method we try to sort the given array by finding the number of elements which are smaller than the element under consideration. Once we obtain this frequency, say p, we place the element at position p+1 in the sorted list. If there are n elements we would require n^2 CRCW processors. To determine the position of each element si in the sorted array we compute c_i - the number of elements smaller than s_i. If two elements s_i, s_j are equal, then if i>j, a_i is taken as larger number. Each processor P(i,j) compares elements s_i and s_j and writes either 0/1 in c_i. To avoid write conflict, the sum of values computed by all the processors is written in the particular memory location. After computing c_i, s_i is placed at position $1+c_i$ of sorted sequence. Shared memory contains 2 arrays S and C. Array S contains the input sequence; Count c_i is stored in array C. The sorted sequence is returned in S.

```
procedure ENUMERATION SORT(S)
    Step 1: for i=1 to n do in parallel
                for j=1 to n do in parallel
                    if($s_i$>$s_j$) or ($s_i$=$s_j$ and i>j) then
                        P(i,j) writes 1 in $c_i$
                    else
                        P(i,j) writes 0 in $c_i$
                    end if
                end for
            end for
    Step 2: for i=1 to n do in parallel
                P(i,1) stores si in position 1+$c_i$ of S
            end for
```

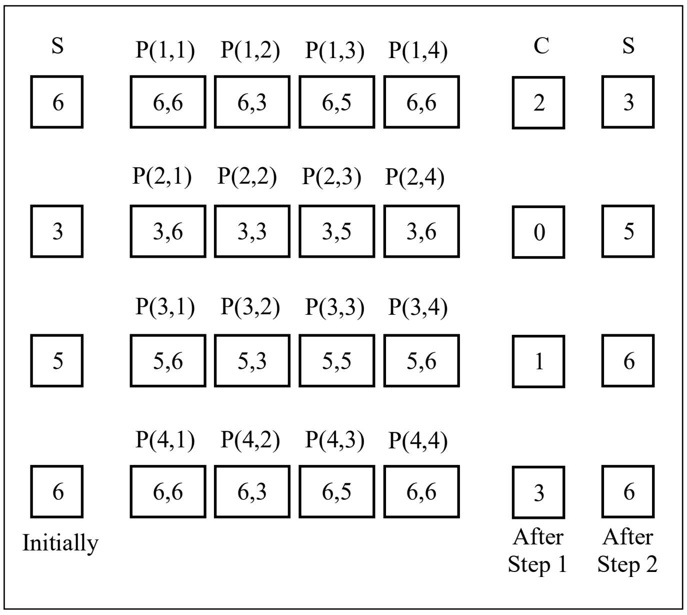
S
P(1,1) P(1,2) P(1,3) P(1,4)
C
S
6
6,6 6,3 6,5 6,6
2
3
P(2,1) P(2,2) P(2,3) P(2,4)
3
3,6 3,3 3,5 3,6
0
5
P(3,1) P(3,2) P(3,3) P(3,4)
5
5,6 5,3 5,5 5,6
1
6
P(4,1) P(4,2) P(4,3) P(4,4)
6
6,6 6,3 6,5 6,6
3
6
Initially
After Step 1
After Step 2

Fig. 2. Sorting sequence of 4 elements

Each of steps 1 and 2 consists of operations running in constant time. Thus t(n)=O(1). So, this algorithm sorts n elements in constant time at the expense of n^2 processors. Here $p(n)=n^2$, giving the cost as $c(n)=O(n^2)$. Since there exists a sequential algorithm for sorting in O(n log n) time this method of sorting is not cost optimal.

C. *Bitonic Sort*

First we discuss about a Bitonic sequence and then we move on to study how to sort such a given sequence.

1) Bitonic Sequence: A sequence is bitonic if

i) There exists an index i, with $0 \le i \le n-1$ such that a_0 through a_i is monotonically increasing and a_i through a_{n-1} is monotonically decreasing;

ii). There exists a cyclic shift of indices so that first condition is satisfied.

2) Compare Exchange Operation: The compare exchange operation is performed by a comparator which takes in two numbers and swaps them if necessary so that they are in proper order as shown in figure.

3) Sorting a Bitonic Sequence (Batcher's Bitonic Sort): A single compare exchange step can split a single bitonic sequence into 2 bitonic sequences. If number of elements, n is even then n/2 comparators are sufficient to transform the n values into 2 bitonic sequences of n/2 values. Comparisons are performed in such a way that first subsequence contains min(a0,an/2), min(a1,an/2+1), .. min(an/2-1,an-1) while second subsequence contains max(), max(), .. max(). No value in the first subsequence is greater than any value in second subsequence. This is illustrated in figure. If there are $n=2^k$ elements then k compare exchange steps are sufficient to sort the sequence.

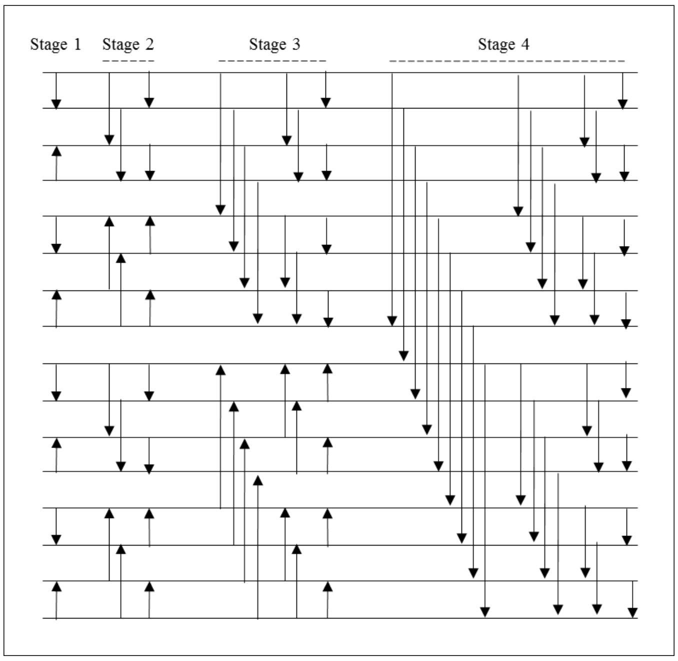

Fig. 3. Sorting network based on bitonic merge

4) Sorting a general sequence using Sorting network based on Bitonic Merge (Knuth's): Assume we have an array A[0..n-1] with n elements. To sort these elements we would need n/2 comparators. Comparators and the inputs to them are shown in figure. The arrows represent the compare exchanges. If there are $n=2^k$ elements, then there will be k stages.Each stage i will contain i steps for a total of k(k+1)/2 steps. In each step n/2 comparators will be used. At the end of all the steps the sequence will be sorted.

A list of n elements to be sorted can be viewed as a set of n unsorted sequence of length 1 or as n/2 Bitonic sequences of length 2. Thus, we sort any sequence of elements by successively merging more and more Bitonic sequences. Given $n=2^k$ unsorted elements, the total comparators used is $(n/2 * \text{Steps}) = (2^k / 2) * (k(k+1)/2) = 2^{k-2} k(k+1)$. The parallel execution of each level requires constant time, so the total running time is $k(k+1)/2 = \log n (\log n + 1) / 2 = (\log^2 n + \log n)/2 = O(\log^2 n)$. Thus the running time is $O(\log^2 n)$.

D. Parallel Quick Sort

Quick Sort works recursively by dividing an unsorted list of elements into smaller sub lists of elements by partitioning. In parallel quick sort, a number of identical processors are used. Elements are stored in an array in global memory. A stack in global memory stores the indices of sub arrays that are still unsorted. When a processor is without work, it tries to pop the indices for an unsorted sub array off the global stack. If successful, the processor partitions the sub array, based on a supposed median element, into 2 smaller arrays, containing elements less than or equal to the supposed median value or greater than the supposed median respectively. After the partitioning step, the processor pushes the indices for one sub array onto the global stack of unsorted sub arrays and repeats the partitioning process on the other sub array.

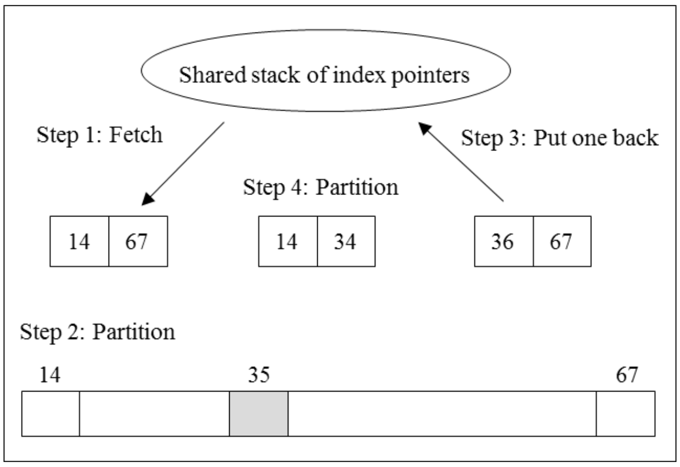

Fig. 4. Illustration of parallel quick sort for UMA multiprocessor.

Parallel Quick Sort works in 2 phases.

When the execution begins, there is only a single unsorted array. Except 1, all processors have to wait till the single processor partitions the array. This iteration requires n-1 time units to make n-1 comparisons.

For $p\geq2$, 2 processors can partition the two resulting sub arrays in (n-1)/2-1= (n-3)/2 time units, making n-3 comparisons. Similarly, for $p\geq4$, third iteration requires time at least [(n-1)/2-1]/2 -1= (n-7)/4 to make n-7 comparisons. So, for the first log p iterations, there are at least as many processors as partitions and time required by this phase is $T_1(n,p)=2(n+1)(1- 1/p)-\log p$

The number of comparisons performed is $C_1(n,p)=(n+1)\log p - 2(p-1)$

In the second phase, there are more sub arrays to be sorted than processors. All processors are active. If all processors perform an equal number of comparisons, then time required is simply the number of comparisons made divided by p.

$C_2(n,p)=T(n)-C_1(n,p)$

$T_2(n,p)=C_2(n,p)/p$

Hence the total running time of this parallel algorithm is T_1+T_2.

IV. Comparative Study of the Algorithms

Now that we have studied the various popular parallel algorithms, we now compare each of them in a tabular format.

	Odd Even Transposition	Enumerati on Sort	Bitonic Sort	Parallel Quick Sort
Time	n	1	$\sqrt{n}$	T_1+T_2
Processors	n	n^2	n	P^*
Cost	n^2	n^2	$n\sqrt{n}$	$P(T1+T2)^*$
Cost-Optimal	No	No	Yes	No

*Given in section D.

V. Conclusion

In this paper we have only discussed four out of the parallel sorting algorithms. However we cannot rule out the existence of many other parallel algorithms which also execute optimally. We should also bear in mind that to decrease the running time at the expense of a large number of processors incurs a greater cost so there is a need for the algorithm to be cost-optimal. It is thus for the same reason why cost-optimality is taken into account while analyzing the algorithms.

Acknowledgment

The authors would like to thank anonymous reviewers for their helpful comments and suggestions.

References

[1] D. Bitton, D. DeWitt, D.K. Hsiao, J. Menon, A Taxonomy of Parallel Sorting, ACM Computing Surveys, 16,3,pp. 287-318.

[2] Song, Y.D., Shirasi, B. A Parallel Exchange Sort Algorithm. South Methodist University, IEEE.

[3] Lucas, K.T., Jana, P.K.: An Efficient Parallel Sorting Algorithm on OTIS-Mesh of Trees. In: IEEE International Advance Computing Conference, India, pp. 175–180 (2009)

[4] Qureshi K., “A Practical Performance Comparison of Parallel Sorting Algorithms on Homogeneous Network of Workstations”, Department of Mathematics and Computer Science, Kuwait University, Kuwait.

[5] Makinde O.E., Adesina O.O., Aremu D.R., Agbo-Ajala O.O., “Performance evaluation of sorting techniques”, In press.

[6] Madhavi Desai, Viral Kapadiya, Performance Study of Efficient Quick Sort and Other Sorting Algorithms for Repeated Data, National Conference on Recent Trends in Engineering & Technology, 13-14 May 2011.

[7] M. J. Quinn, Parallel Programming in C with MPI and OpenMP, Tata McGraw Hill Publications, 2003, p. 338

[8] S. S. Skiena, The Algorithm Design Manual, Second Edition, Springer, 2008, p. 129.

[9] Ananth G. ,Anshul G.,George K. & Vipin K.(2007): ,Introduction to Parallel Computing, 2nd Ed.,Addison-Wesley

[10] M. Banikazemi, V. Moorthy, and D. K. Panda. Efficient Collective Communication on Heterogeneous Networks of Worksations. In Proc. Of the 27th International Conference on Parallel Processing, pages 460-467. IEEE Computer Society Press, 1998.

[11] U. Banerjee. Loop Parallelization. Kulwer Academic Publishers, 1994.

[12] M. Adler, J. W. Byers, and R. M. Karp. Parallel sorting with limited bandwidth. In Proceedings the Symposium on Parallel Algorithms and Architectures, July 1995.

[13] P. Brucker. Scheduling Algorithms. Spriger-Verlag, 2004.

[14] M. J. Quinn. Parallel Computing. Theory and Practice. Mc-Graw Hill, 1994.

[15] S. G. Akl. The Design and Analysis of Parallel Algorithms. Prentice Hall, 1989.

[16] H. S. Stone. Parallel processing with the perfect shuffle. IEEE Computer, C-20(2), February 1971.