Journal of Computer Science 4 (1): 66-71, 2008
ISSN 1549-3636
© 2008 Science Publications

Parallel Two-Dimensional Quicksort Algorithm (PTSA)

[1]Hamed Al Rjoub, [1]Ahmad Odat, [2]Abdullah Audat
[1]Department of Computer Sciences, Irbid National University, Irbid, Jordan
[2]Adaptive TechSoft, Amman, Jordan

Abstract: In this research, a Parallel Two-Dimensional Sorting Algorithm (PTSA) is presented that has better performance than the classical Quicksort, to sort a data vector of size n = r (rows) × c(columns). PTSA algorithm divides the input vector into n/r sub-vectors, which represents tow-dimensional vector of Slave Processor Elements (VPE), the maximum number of VPE for parallel sorting is equal to r×c, VPE just the read,and write operations. The number of Master Processors (MP) which do the sort operation is equal to c, The time needed for PTSA algorithm is reduced by θ (n/r log n/r) with respect to the time needed by Quicksort θ (n log n) to sort the same vector. Simulation results show that the efficiency of sorting using PTSA algorithm is increased and the complexity is reduced significantly compared with classical Quicksort.

Key words: Data mining, accelerate, sorting rows, sorting columns, proccessors, iterations

INTRODUCTION

Sorting is an integral component of most database management systems (DBMSs) and data stream management systems (DSMSs), its efficiency can influence drastically the overall system performance.

To speed up the performance of database system, parallelism is applied to the data administration operations.

We all know that Quicksort is one of the fastest algorithms for sorting, in spite of its slow running time in the worst case that is $\theta(n^2)$ on an input array of n numbers. However, its running time in the average case is θ(n log(n)). Given the importance of parallelism in improving the running time of Quicksort many researches have been done in this area. Previous results show that the parallel Quicksort outperform the sample sort algorithm. In[11], the experiments show that the speed of the parallel Quicksort is more than six units higher than the speed of sample sort for three processors of the Enterprise 1000. While, for one processor, parallel Quicksort achieved 15% faster execution times than the traditional Quicksort. This is due to its low memory requirement and that parallel Quicksort could sort data sets twice the size that of traditional Quicksort could under the same system memory restrictions.

In this research, we present a parallel two dimensional sorting algorithm that outperform traditional Quicksort and many of the previous parallel Quicksort algorithms in running time and in the number of comparisons. The idea is dividing the vector data into number of processors to sort data in parallel. Due to that, the transactions of comparison and data transfer is reduced and the complexity time to sort an array of size n is reduced to O(n/r (logn/r)). In addition, the complexity time of this algorithm depends on the number of processors, where many of the previous work in parallel Quicksort have higher complexity time and generally the complexity time is dependent on the number of processors used in parallel and in some cases is dependent on the number of partitions.

Bitonicsort[3,4,9] has running time of $O((n/p)\log^2 p)$ where p is the number of processors used and n is in the range of $\Omega(p)$. Mergesort[10], has running time of $O(\log^2 p/\log(p/n))$ where n = O(p). ColumnSort[1], has running time of O((nlogn)/p) where $n = \Omega(p^{1+\varepsilon})$ ε>0. Cubesort[6], its running time is $O((n/p)\log^2 p/\log(n/p))$ essentially $n = \Omega(p)$.

In[8], the cost of parallel Quicksort algorithm is measured given the pair (t,p) where t is the time needed and p is the number of processors that are both dependent on the input size. However, to sort n words all of length l over an alphabet Σ of size O(n) it requires only O(log(nl)/log(nl/p)×nl/p) times and using p<nl processors.

RELATED WORKS

An early advancement in parallel sorting comes in 1968 when Batcher discovered the elegant $\theta(\log^2 n)$ depth bitoni sorting networks[3].

Corresponding Author: Hamed Al Rjoub, Department of Computer Sciences, Irbid National University, Irbid, Jordan

Sorting is the most studied problem in computer science for two reasons: First, it is used as a substep in many applications. Second, it is a simple combinatorial problem with many interesting and diverse solutions. Parallel algorithms for sorting have been studied since at least the 1960's. Batcher bitonic sorting technique provides a parallel algorithm for sorting n numbers in $\theta(\log^2 n)$ time with n processers. Bitonic sort used parallelism to increase the chip speed by treating several bits in parallel at each step.

The idea on which bitonic sort depends is the sorting and merging operations; given two inputs A and B, the algorithm must define the two correspondent output in a manner that if (A<B) so, A = L that represents the smallest element and B = H that represents the highest element. If (B<A) then B = L and A = H.

This procedure can be applied to two input lists to obtain only one-sorted lists. The merging operation between the two lists is done in two stages: the first is the odd stage in which all the elements in both lists are compared with the odd index and arranged given H, L as described above. In the same manner, in the second stage (the even stage) all the elements in both lists are compared with even index. This means, given two lists a_1, a_2,.....a_s, b_1,b_2,.....b_t the algorithm must produce a sorted list C_1, C_2,... C_{t+s}. Figure.1 represent a simple description of how bitonic sort work.

The main advantage of bitonic sort is that it can be extended to compare 4 by 4 elements in each step. Using bitonic sort, 2^p words can be sorted in only 1/2 p (p+1) steps. In the literature, many parallel sorting

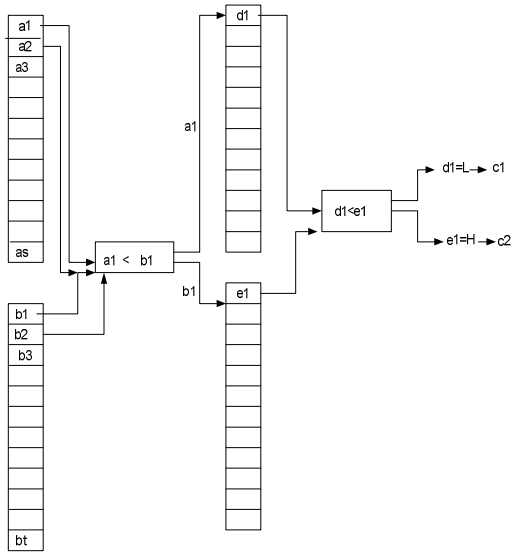

Fig. 1: Bitonic sort

algorithms have been proposed like radixsort and Quicksort[5] and a variant of Quicksort is called columnsort[1]. In[11], a parallel Quicksort algorithm is presented and implemented that works in stages. In the first stage, a parallel partition of data is done in which the array is divided in k blocks of fixed dimension B. A number of processors p is assigned to sort these blocks.

The algorithm works as follow: each processor sorts in parallel two blocks one from the head and the other from the tail of the array. This procedure is repeated until most of the blocks are sorted. In the second stage, it assigns one processor to sort the remaining unsorted blocks. The third stage is the process partition in which the array is divided into two partitions and the processors are divided into two groups, each group is assigned to a partition. In the Final stage, it uses Quicksort in each group of processors to sort its partition. If one processor finishes sorting its blocks it helps the other processors in the same group by a consecutive operation of push and pop.

The time complexity of the algorithm depends on the size of block B and on the number of processors P and does not depend on the distribution of keys. In[10], comparisons of three parallel sorting algorithms are: odd-even transition Sort[2], Parallel Rank Sort, Parallel Merge Sort is presented. The odd-even transposition algorithm[2], works as follow: first of all the data is divided into (n*p) partitions, where p is the number of processors. These partitions are distributed to all the processors. Each processor sorts sequentially its part. Then, the merging stage consists of two substages: odd stage and even stage. In the even stage, the even processors (i) communicate with the odd processors (i+1) to merge their data in a manner that higher data values are kept in the higher number of processors. The lower values are kept in the lower number of processors. The same procedure is done in the odd stage where the odd processors (i) communicate with the even processors (i-1) to merge their data in the same previous manner. The whole data will be sorted in at most p stages and the time complexity is $O(bn^2)$ where $b = 1/2p^2$. That means the time will be reduced to $1/p^2$. The second algorithm compared was Parallel Merge Sort, this algorithm is based on the Divide and Conquer Algorithm, but here we have the concept of slaves and master processors. The initial array is divided into (n/p) elements. Each processor denoted as slave sorts its part in parallel and then these sorted subarrays will be returned to the master processor that sorts the entire final array. The time complexity of this algorithm is O(n/p log(n/p)). The third algorithm compared and implemented in[10], is Parallel Rank Sort Algorithm that works as follow: the basic idea is to determine the rank

of the elements. The data is divided in different partitions, each slave processor is assigned a data of size (n/p) elements where n is the size of the input data and p is the number of processors. Each slave processor is responsible to rank its elements and return the ranks to the master processor, which in turn is responsible of constructing the whole sorted list. The time complexity is $O(n^2) = 1/p$.

Parallel Two-Dimensional Sorting Algorithm PTSA: Quicksort[7] is a sequential sorting algorithm that is widely believed to be the fastest comparison-based sequential sorting algorithm. It is a recursive algorithm that uses the Divide and Conquer method to sort all keys. The standard Quicksort first picks a key from the list, the pivot and finds its position in the list where the key should be placed. This is done by walking through the array of keys from one side to the other. When doing this, all other keys are swapped into two parts in the memory: first the keys less than or equal to the pivot are placed in one part of the pivot and the keys larger than the pivot are placed in the other part of the pivot.

PTSA Algorithm Description: The main idea of PTSA algorithm is to divide the input into number of rows and then to use Quicksort in parallel manner to sort theses rows and obtain the following:

- Sorted rows in parallel, where each row has its own processors
- Sorted columns in parallel, where each column has its own processors

PTSA passes through three stages: distribution data is the first stage, parallel sorting rows is the second stage and parallel sorting columns is the third stage.

Distribution data stage: In this stage, given an input array *A* to a matrix of processor Elements of size n, where, n = rows (r)×columns (c), A will be divided into a r×c, where r is an odd number greater than or equal to 3 and c is greater than or equal to r, where c = n/r. The first row is assigned to the first processor (Mp_0), the second row to the second processor (Mp_1) and so on to till the last row (Mp_{r-1}). Next, the first column is assigned to the first processor (Mp_0), second column (Mp_1) to the second processor and so on to till the last column (Mp_{c-1}).

Parallel sorting rows stage: In this stage, the processors MP_0, MP_1, MPr_{-1}, are assigned to the data rows respectively to sort them in parallel fashion.

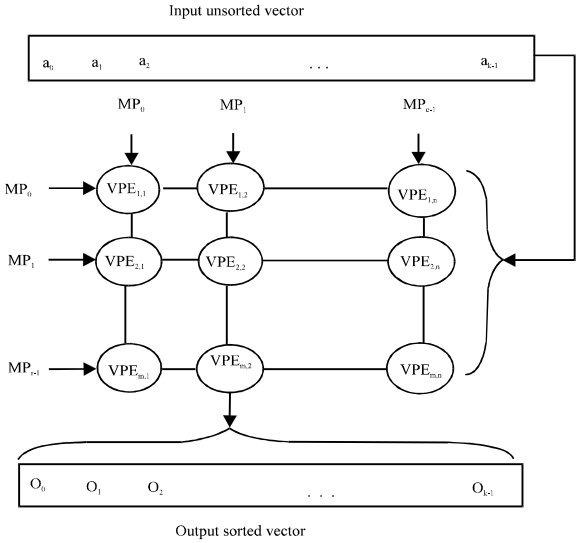

Fig. 2: Parallel two-dimensional sorting algorithm structure

Parallel sorting columns stage: In this stage, the processors MP_0,$MP_{1,...,}$ MPc_{-1}, are assigned respectively to the data columns to sort them in parallel fashion using quicksort algorithm.

At the end of stage1 and stage2 minimum and maximum values are obtained, where the minimum value is located at O_0 and the maximum value is located at O_{n-1}. The values are shifted in Mp_0 one step to the left and the values in Mp_{r-1} are shifted one step to the right. The above steps are repeated until it ends-up with one middle row. Finally, the middle row is sorted using quicksort algorithm. Figure. 2, illustrates the three stages mentioned above.

PTSA algorithm analysis: PTSA as a two dimensional sorting algorithms is based on Quicksort algorithm which is used to sort rows and columns in parallel.

However, it is known that Quicksort performs better when the input data is increased, but here in PTSA algorithm the additional time required by Quicksort to sort a number of data inputs (which is equal to n) is reduced to n/r. In each iteration tow values (maximum, minimum) are transferred at the same time, while in the Divide and Conquer the transfer is made once only either the min or the max. The algorithm analyses steps are as follows:

Step 0: It assumed that:

i. Data Vector (A) of size n

ii. Cl is a number of clock cycles required to send one data item from one master processor to another

Step 1: The required clock cycle (S_{r1}) to divide the data vector (DV) into rows and columns is computed and each row is sent to one of the row processor elements:

$$S_{r1} = (2n/r) * cl,$$

where, cl is a number of clock cycle required to send one data item from one processor to another.

Step 2: The required clock cycles (S_{r2}) to sort all rows in parallel is computed using quick sort algorithm:

$$S_{r2} = (n/r \log(n/r)) * j,$$

where, j is the number of clock cycles needed to process one iteration of traditional quicksort algorithm. Throughout this work, j is assumed to be equal to 10* cl, where cl = 1.

Step 3: The required clock cycles (S_{r3}) to distribute all sorted rows item to the processors in VPE is computed as:

$$S_{r3} = r * (c-1) * cl.$$

Step 4: The required clock cycles (S_{r4}) to sort in parallel r items using quicksort algorithm is compute an shifted as:

$$S_{r4} = (r \log r) * j.$$

Step 5: Compute the required clock cycles (Sr5) to take maximum and minimum items then shift:

$$S_{r5} = cl.$$

Step 6: Steps 4, 5 are repeated c/2 times.

Step 7: Steps 4, 5 and 6 are repeated r/2 times.

Step 8: The required clock cycles (S_{r8}) to sort the middle row processors using Quicksort algorithm is computed as: $S_{r8} = (n/r \log(n/r)) * j.$

Step 9: The required lock cycles (S_{r9}) to send in parallel middle row processors values to the suitable location is computed as:

$$S_{r9} = cl.$$

Step 10: The mathematical model to calculate the total number of clock cycles to sort input data using PTSA is:- $S_{tot} = \sum_{i=1}^{9} S_{ri}$.

Complexity Analysis: The time complexity of PTSA algorithm is calculated for the three stages together. The time needed by PTSA algorithm to sort an array of n elements denoted as T_{tot}. T_{Sor} refers to the time needed by the Quicksort to sort one subarray. T_{mer} is the time needed to put maximum and minimum items in their locations in the output vector. T_{par} is the time needed to distribute data to VPE of size r × c.

$T_{tot} = T_{Sor} + T_{mer} + T_{par.}$
$T_{tot} = (r*T(n/r)+c*T(r))+\theta((n-c)/2)+ \theta(1).$
T(n/r), T(r) is the time needed by Quicksort to sort an input data of size n/r rows, r columns, respectively
$T_{tot} = r*O(n/r\log n/r)+c*O(r\log r)+(n-c)/2.$
$T_{tot} \leq C(n/r\log n/r)+C(r\log r)+(n-c)/2$
$\leq Cn/r[\log n-\log r] + Cr\log r+(n-c)/2$
$\leq C[n/r(\log n-\log r) + r\log r]+(n-c)/2.$
$T_{tot} \leq O(n/r\log n/r + r\log r).$

RESULTS

Simulation results: The idea behind the work in this simulation is to compare the time needed for Quicksort (to sort an array of n items) with the time needed by PTSA (to sort the same array) by dividing it into r×c VPE.In the case of sorting an array of size n, f(n) becomes the count function. That is; f(n) gives the number of basic operation (clock cycles) done by the PTSA. Suppose that the time spent in each clock cycle is t, then the total time it would take to execute f(n) is tf(n). Clearly, the constant time t depends on the speed of the computer and therefore varies from computer to computer. However, f(n), the number of clock cycles, is the same for each computer. If we know how the function f(n) grows as the size of the problem grows,we can determine the efficiency of the PTSA with respect to Quicksort algorithm. We compare its results of the traditional Quicksort on the same stand-alone PC with Pentium III processor running at 800Mhz under windows XP platform using Microsoft Excel.

Table 1: The running time of Quicksort and PTSA algorithm for different input data size and r = 99

Size(n)	Running Time Quick sort(µs)	Running Time PTSA(µs)
1000000	199315.686	70116.104
2000000	418631.371	140642.209
3000000	645495.932	211319.291
4000000	877262.743	282094.417
5000000	1112674.833	352942.450
6000000	1350991.864	423848.581
7000000	1591724.644	494803.028
8000000	1834525.486	565798.835
9000000	2079134.421	636830.804
10000000	2325349.666	707894.900

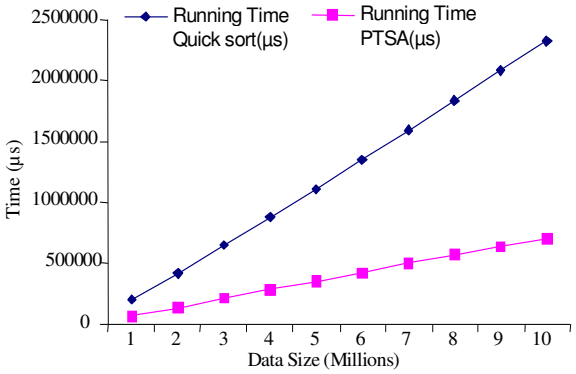

Fig. 3: The variation in running time in Quicksort and PTSA, for different input data size and r = 99

The dynamic data size and fixed number of rows: For different data size and fixed number of rows = 99, suppose that a computer can executes 1 billion operations per second, Table. 1 and Figure. 3 shows the time required to execute PTSA and the time required to execute traditional Quicksort.

The fixed data size and dynamic number of rows: For fixed data size = 10^7 and different number of rows, suppose that a computer can executes 1 billion operations per second, Table 2 and Figure 4 shows the time that the computer takes to execute PTSA and Quicksort algorithm.

The PTSA accelerate: Analyzing the results presented in Table. 3 it can be seen the accelerate (Ac = Quicksort time on one processor / PTSA time on c processors) is around (1.5-1.7).

Example 1: Figure. 5, shows an example of an array of size 9, At the end we obtain a sorted array S, The initial values for the counters are zero (i = 0, j = 0) :

Step 1: The vector is distributed into three rows and three columns (to the VPE).

Table 2: The running time of Quicksort and PTSA algorithm for fixed input data size = 10^7 and different number of vectors

Vectors(r)	Running Time Quick sort(µs)	Running Time PTSA(µs)
100	2325349.7	707894.9
200	2325349.7	790105.3
300	2325349.7	842908.3
400	2325349.7	881605.4
500	2325349.7	912103.5
600	2325349.7	937246.8
700	2325349.7	958617.5
800	2325349.7	977185.5
900	2325349.7	993588.1
1000	2325349.7	1008266.0

Table. 3: The Accelerate of parallelism for PTSA Algorithm

Size (n)	Running Time Quick sort (µs)	Running Time PTSA (µs)	Accelerate PTSA (Sp)
50	2.822	1.880	1.501
100	6.644	4.186	1.587
150	10.843	6.642	1.633
200	15.288	9.196	1.662
250	19.914	11.824	1.684
300	24.686	14.509	1.701
350	29.579	17.243	1.715
400	34.575	20.018	1.727
450	39.662	22.829	1.737
500	44.829	25.672	1.746

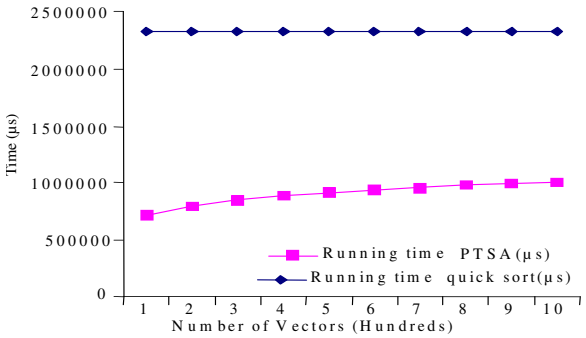

Fig. 4: The variation in running time in Quicksort and PTSA algorithm, for fixed size of data = 10^7 and different number of vectors

Step 2: The first row is sent to the first MP (p_0) and the second row is sent to the second MP (p_1) and the third row is sent to the third MP (p_2).

Step 3: Sorting is carried out in parallel in the same order for all rows.

Step 4: The first column is sent to the first MP column (p_0) and the second column is sent to the second MP column (p_1) and third column is sent to the third MP column (p_2)

Step 5: Sorting is carried out in parallel in the same order for all columns.

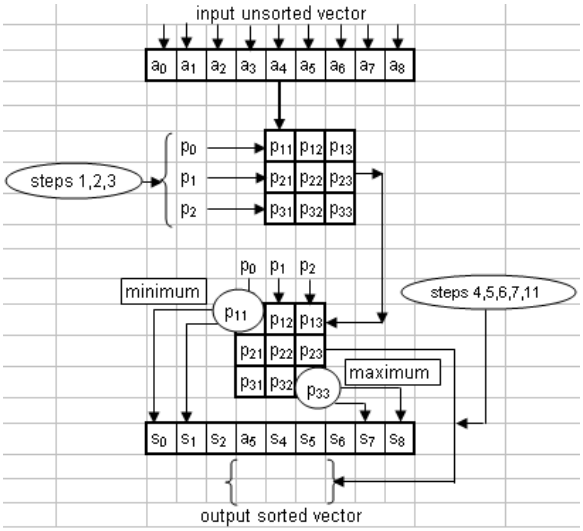

Fig. 5: Example 3×3 PTSA Algorithm

Step 6: In the location VPE (p_{11}) we get the minimum element and in location VPE (p_{33}) we get the maximum element for giving vector.

Step 7: In parallel put the value at VPE (p_{11}) in the first location (minimum) and the value at VPE (p_{33}) (maximum) in last location for output vector.

Step 8: The first row of processor elements is shifted one position to the left and third row of processor elements shifted one position to the right.

Step 9: i = i+1, j = j-1.

Step 10: Repeat step5, step 6, step 7, step 8 and step 9 (c-1) iterations.

Step 11: Sort the middle row and the result send it in parallel to the output vector (S) locations from s_i to s_J.

CONCLUSION

In this research a new algorithm is proposed for sorting an array of size n. The idea of this algorithm is to use parallelism to reduce the running time. We have obtained a scalable system, in which its performance is improved when the number of input data is increased. The reduction becomes significant and that is confirmed by our simulation results where the reduction in number of comparisons respectively to the traditional Quicksort is about 55-65%.

REFERENCES

1. Aggarwal, A. and D. Huang, 1988. Network complexity of sorting and graph problem and simulating CRCWPRAMs by nterconection Networks. Springer-Verlag, 319: 339-350.
2. Bitton, D., D. Dewitt, K. Hsiao and J. Menon, 1984. Ataxonomy of parallel sorting. ACM Comput. Surveys, 16. 3: 287-318.
3. Batcher, K.E., 1968.v Sorting networks and their applications. In Proceedings of the AFIPS SPring Joint Comput. Sci., 32: 307-314.
4. Baudet, G. and D. Stevenson, 1978. Optimal sorting algorithm for parallel computer. IEEE Transa. on Comput., C-27: 84-87.
5. Thomas H.C., Charles E. Leiserson, Ronald L. Rivest, 2000. Introduction to Algorithm. The MIT Press, Cambridge. Massachusetts, London-England. ISBN: 0-262-03141-8.
6. Cypher, R.E. and J.L. Sanz, 1988. Cubesort: An optimal sorting algorithm for feasile parallel computers. Springer-Verlag, 319: 456-464.
7. Hoare, C.R., 1962. Quicksort. Comput. J., 5 (1): 10-16.
8. Costas, S., 1986. Iliopouls algorithmic time parallel for partitioning. Technical Reports of the Department of Computer Science of Purdue University. West Lafayette. IN 47907: CSD TR#86-603.
9. Johnson, S.L., 1984. Combining parallel and sequential sorting on a boolean n-cube. In: Proceedings of the 1984 IEEE. International conference on Parallel Processing, 444-448.
10. Nassimi, D. and S. Sahni, 1982. Parallel permutation and sorting lgorithms and a new generalized connection networks. JACM. 29: 642-667.
11. Song, D. and A. Shirasi, 1988. Parallel exchange sort algorithm. South Methodist University, IEEE.