439

Int'l Conf. Parallel Proc 12/1984

Combining parallel and sequential sorting on a Boolean n-cube

S. Lennart Johnsson
Department of Computer Science
Yale University
New Haven 06520

Abstract - Three parallel algorithms for sorting M uniformly distributed elements on a Boolean n-cube of $N=2^n$ $N \leq M$ processors are presented. Two of the algorithms combine sequential sort with bitonic sort to accomplish a time complexity of $O(M\log M/N)$ for $N<<M$, and $O(\log^2 M)$ for $M \simeq N$. One algorithm sorts the elements cyclically, such that a processor holds sorted elements that are congruent modN. The other algorithm sorts subsequences of M/N consecutive elements into each processor. The third algorithm is a parallel bucket sort that sorts the elements into L buckets in time $O(M/N+L)$ if $N \leq L$ and time $O(M/N+L+\log N-\log L)$ for $N>L$.

I. Introduction

Our interest in sorting algorithms for Boolean n-cubes is motivated by the construction of a multiprocessor with that configuration [8]. Currently a 64 processor machine is in operation, and a 1024 processor machine is planned. Another machine featuring a Boolean n-cube configuration and a large number of processing elements, 10^4 or more, is the Connection Machine being constructed at MIT. In the computational model we use, each processor has its own local storage and control. In Flynn's terminology [3] it is a MIMD machine, with no global storage. A processor with its local storage is refered to as a node.

Batcher's bitonic sort [1], is a well known parallel sorting algorithm of time complexity $O(\log^2 M)$, for sorting M elements on the appropriate sorting network [1], [5]. Stone [9] observed that comparison-exchange operations in bitonic sort are made on elements that can be brought in proximity by shuffle operations, and presented an algorithm for bitonic sort on a perfect shuffle network. Nassimi and Sahni [7] describe an implementation of Batcher's bitonic sort on a mesh configured array. Adaptations of Batcher's odd-even merge to mesh configured arrays are given by Thompson and Kung [12], and Kumar and Hirschberg [6]. Sorting requires global communication and $O(M^{1/2})$ is a lower bound for sorting on processing elements configured as a mesh. The Boolean n-cube offers node connections that are a perfect fit for bitonic sort. Comparison-exchange operations are simply performed along the different coordinate directions of the cube.

The above references only address the case where the number of processing elements, N, equals the number of elements to be sorted. For N<M a naive mapping of bitonic sort on to the n-cube does not produce an efficient algorithm. Lower time complexity is achieved by combining good sequential sorting algorithms with bitonic sort. The bitonic sort is used for internode sorting, while sequential sort is used within a node. Baudet and Stevenson [2] outline one combination. We present two algorithms, one having the characteristics of the suggestion in [2]. The time complexity of both algorithms is $O(M\log M/N)$ for a cube small relative to M, and $O(\log^2 M)$ for a cube of a size comparable to M.

Two sorting orders are considered. In *cyclic order* node i stores all elements {j} of the sorted sequence such that $i=j\bmod N$. In *consecutive order* M/N successive elements of the sorted sequence are stored in each node, with successive sets of M/N elements being stored in nodes in order of increasing node address. If the total storage of the n-cube is viewed as a matrix with one column for each node, and one row for each element, then cyclic order implies sorting in row major order, and consecutive sort sorting in column major order.

The bucket sort described last completes the rank assignment of all elements in time $O(M/N+L)$ if $N \leq L$, and time $O(M/N+L+\log N-\log L)$ if $N>L$. The increased time complexity compared to Hirschberg's algorithm is due to the restricted communication in the cube.

II. Cyclic sort

In bitonic sort sorted subsequences of equal length are merged recursively. By storing one of the sequences in the half cube with highest address bit 0, and the other in the half cube with highest address bit 1, and with corresponding elements in corresponding nodes, the first step in bitonic merge is carried out by comparing elements stored in nodes that differ only in the highest address bit. Subsequent steps compare elements in nodes that differ in successivly lower order address bits. A mask determines whether sorting is performed in non-descending or non-ascending order, as shown in the algorithm below.

```
For i:=1,2,...,n do
  If i<n do
    nodes $(a_{n-1},..., a_{i+1}, a_i, a_{i-1},..., a_0)$, $a_i$=1, set mask=1.
    nodes $(a_{n-1},..., a_{i+1}, a_i, a_{i-1},..., a_0)$, $a_i$=0, set mask=0.
  end
  For j:=i-1,i-2,...,0 do
    nodes $(a_{n-1}, ..., a_{j+1}, 1, a_{j-1}, ..., a_0)$, send their elements to
    nodes $(a_{n-1}, ..., a_{j+1}, 0, a_{j-1}, ..., a_0)$, which compare local
    and recieved elements
    nodes with mask=0 keep the smaller element and
    nodes with mask=1 keeps the larger
    rejected elements are sent to $(a_{n-1}, ..., a_{j+1}, 1, a_{j-1}, ..., a_0)$
  end
end
```

This algorithm requires a time of $n(n+1)(2t_{s/r}+t_{ce})/2$. In sorting a sequence of 2^n numbers on 2^n nodes, only half of the nodes are used for comparison operations. 2^{n-1} nodes suffice to sort 2^n elements. Conversly, two sequences of 2^n elements each can be sorted in time $n(n+1)(4t_{s/r}+t_{ce})/2$ time on 2^n nodes. The storage need per processor remains the same.

Assume for simplicity that $M=2^{k+n}$. To create a sorted sequence of 2^{k+n} elements from two sorted subsequences of size 2^{k-1+n} by bitonic sort requires k+n steps. With cyclically stored sequences, the first k steps are local to each node. The result after those k steps is 2^k bitonic sequences ordered with respect to each other. Each sequence has one element per node. The last n steps are separately perfomed on each of those sequences.

Carrying out the first k local steps as bitonic sort results in an algorithm with poor performance. The operational complexity of bitonic sort is $O(M\log^2 M)$, compared to $O(M\log M)$ for a good sequential sort. The algorithm outlined below is a generalization of Batcher's 16-sorter constructed out of 4-sorters [1]. The first set of sorters consist of 2^n sorters, each merging two sorted sequences of length 2^{k-1}. Since the two sequences are stored cyclically the input to each sorter is bitonic. The number of sorters in the second set is 2^k, each sorting a bitonic sequence of size 2^n. Choosing the input to each of the second set of sorters as corresponding elements from the outputs of the sorters in the first set makes the inputs to the second set of sorters bitonic. Figure 1a shows two sorted subsequences, Figure 1b their storage in the n-cube, and Figure 1c the result of the k first steps of bitonic sort. Figure 2 shows the generalization of Batcher's 16-sorter.

Each of the first set of sorters is implemented within a node as a merge. Each of the second set of sorters is implemented as an internode bitonic merge. Two bitonic sequences can be merged concurrently. The outline of the algorithm for cyclic sort is as follows:

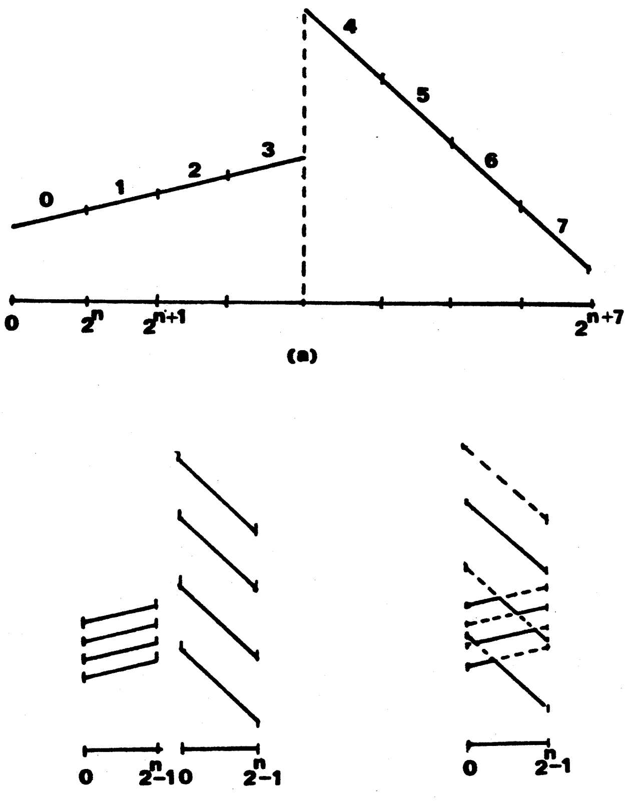

Figure 1: Cyclic storage of bitonic sequences

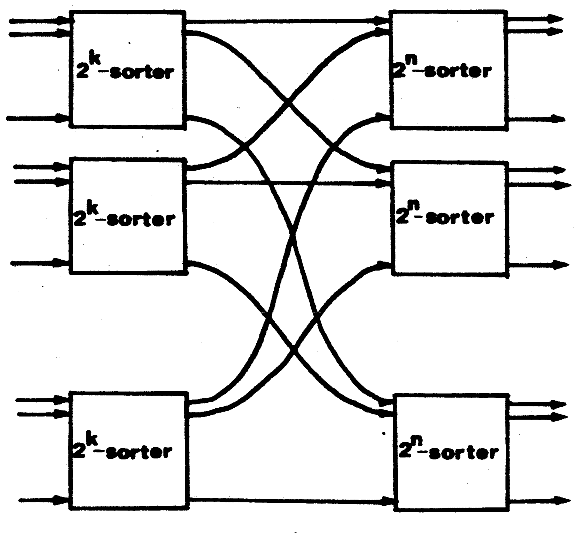

Figure 2: A network of 2^n 2^k-sorters followed by 2^k 2^n-sorter

1. Separately sort in cyclic order two sequences, each of length $2^{k-1}2^n$, one in non-descending order, the other in non-ascending order. Time $2T_{k-1}$.

2. Concurrently in all processors perform a merge-sort.

3. For $j:=1,2,3,\ldots,2^{k-1}$ perform bitonic merge of two sequences on a n-cube with one element per processor.

A mask is used to determine whether sorting shall be performed in non-descending or non-ascending order. The mask remains the same for all nodes and steps required to implement the merge of one pair of sequences ($k>0$). A mask bit is associated with each sequence.

The time for cyclic sort by this algorithm is given by the following first order linear recurrence

$T_k = 2\,T_{k-1} + (2^k-1)t_{ce} + 2^{k-1}n(4t_{s/r}+t_{ce})$

$T_1 = n(n+1)(4t_{s/r}+t_{ce})/2 + n(4t_{s/r}+t_{ce}) + t_{ce}$

or $T_k = 2^{k-1}(n(n+2k+1)(4t_{s/r}+t_{ce})/2 + 2(k-1)t_{ce}) + t_{ce}$

Hence, $T_k = O(M\log M/N)$ for $M>>N$ and $O(\log^2 M)$ for $M\simeq N$. The speed-up is linear for a small cube and gradually changes to $O(N/\log N)$, the speed-up for bitonic sort.

III. Consecutive sort

An algorithm for consecutive sort can be devised to be recursive on the number of elements in a node, as the cyclic sort described above. However, the following algorithm incorporates all elements in a node into sorted subsequences of increasing length by a recursion on the number of nodes. Such a recursion concurs with Baudet and Stevenson's [2] suggestion.

Bitonic sort is used for internode sort. The merge-split operation [2] is replaced by a comparison-exchange operation on subsequences. The initial local sort is carried out so that sequences to be merged in the first step of the bitonic internode sort are sorted in different order (non-descending and non-ascending). In the first step of the internode sort pairs of subsequences, each with M/N elements, are compared and the resulting two bitonic sequences sorted independently by separate processors. The comparions are shared between the two processors holding the two subsequences in order to minimize the time needed. The result after the local sort is a sorted sequence of 2M/N elements with M/N elements per processor. The process is repeated recursively until all nodes are included. The bitonic property of sequences sorted locally in each step of the internode sort is exploited by performing the local sort as a merge after the maximum (or minimum) is located by bisection.

1. All odd processors, i.e., $(a_{n-1},\ldots,a_1,1)$, set mask=1, and all even processors set mask=0.

2. Concurrently perform a local (merge) sort in all processors. Time $[M/N(\log(M/N)-1)+1]t_{ce}$.

3. For $i:=0,1,2,\ldots,n-1$ do steps 4 through 7

4. Processors $(a_{n-1},a_{n-2},\ldots,a_0)$, $a_{i+1}=1$ set mask=1. All other processors set mask=0.

5. for $j:=i,i-1,\ldots,0$ do step 6

6. Processors $(a_{n-1},a_{n-2},\ldots,a_0)$, $a_j=1$, send their last M/(2N) elements to processors $(a_{n-1},a_{n-2},\ldots,a_0)$, $a_j=0$, and processors $(a_{n-1},a_{n-2},\ldots,a_0)$, $a_j=0$, send their first M/(2N) elements to processors $(a_{n-1},a_{n-2},\ldots,a_0)$, $a_j=1$. Concurrently in all processors compare elements pairwise, one from each sequence. Processors with mask=0 and $a_j=0(1)$, keep the smaller (larger) element in each comparison and send the larger (smaller) to the processors $a_j=1(0)$. Processors with mask=1 and $a_j=0(1)$ keep the larger (smaller) elements and send the smaller (larger) ones. This step takes time $M/(2N)(4t_{s/r}+t_{ce})$ for each j.

7. Locate the maximum (or minimum) of the bitonic sequence in each node and perform a merge on sequences of length M/N. Time $(M/N-1+2\log(M/N))t_{ce}$

The time for sorting $(M/N)2^i$ elements on 2^i processors is given by the recurrence,

$T_i = T_{i-1} + (M/(2N))(4t_{s/r}+t_{ce})i + (M/N-1+2\log(M/N))t_{ce}$

$T_0 = [M/N(\log(M/N)-1)+1]t_{ce}$

Solving this recurrence yields for $i=n=\log N$

$T_n = M/(2N)(\log N(\log N+1)(4t_{s/r}+t_{ce})/2 + 2(\log M-1)t_{ce}) + \log N(2\log(M/N)-1)t_{ce}$

or $T_n = O(M/N(\log M+\log^2 N)+\log N(\log M-\log N))$. The speed-up is linear for a small cube and approaches $O(N/\log N)$ for $M\simeq N$.

IV. Bucket sort

Hirschberg [4] gives an algorithm for parallel bucket sort that performs rank assignment to M numbers sorted into L buckets using M processors and (L+2)M storage in $O(\log M+\log L)$ time. The algorithm is based on a computational model that allows an arbitrary number of processors to access in parallel any set of distinct storage locations. Our model only allows each processor to access any location in its own storage in unit time. In the following we consider the case where $K\geq 1$ elements ($M=K*N$) are stored in each node. The storage need is

2M+LN for the elements, the rank assignments, and the bucket counters. Each node has its own set of bucket counters. The time complexity is $O(L)$ for the global bucket count, accumulation, and rank assignment if $N \leq L$ and $O(L+\log N-\log L)$ for $N>L$. The initial bucket assignment and final rank assignment within a processor needs $O(M/N)$ time.

The global bucket count is made by concurrently carrying out several "folding" operations, each terminating in a different processor. For each folding operation the number of partial bucket counts being propagated is reduced by two. In the first global counting step the last L/2 local bucket counts can be sent from each even processor to each succeeding odd processor, and each odd processor can send the first L/2 buckets to the preceeding even processor. After the initial step global bucket counting can proceed concurrently for each half of the buckets in each half of the cube. In the final step L/N global bucket counts are contained in every processor, for $N \leq L$. The time for the global bucket count is $L(1-1/N)(2t_{s/r}+t_a)$, where t_a is the time for incrementing a bucket counter. For $N>L$ there is only one value that needs to be propagated in the last logN-logL steps, which renders the time complexity $(L-1)(2t_{s/r}+t_a)+(\log N-\log L)(t_{s/r}+t_a)$.

The global bucket count can be accumulated in the same storage locations as the local bucket count. After the initial step of the global bucket count half of the nodes have the local bucket counts in their counter locations, and half the nodes have the sum of their own local bucket counts and the bucket counts of the nodes that are unchanged. The process is repeated recursively until only 1 bucket remains, or the folding is completed. The local counts that are replaced by partial sums can be recovered by performing the summation process in reverse order, and querying processors containing a preceeding partial sum.

The global bucket counts are scattered throughout the cube. Accumulation for the assignment of ranks to buckets can be accomplished in 2logN steps by recursive doubling, [10], [11]. For $N \leq L$ there is one initial and one final local step. For $N>L$ there are logN-logL initial and final propagation steps. After the initial logN-logL steps the values for the accumulation are in a subcube of size L. The initial and final phase each requires time $(L/N-1)t_a$ for $N \leq L$, and $(\log N-\log L)t_{s/r}$ for $N>L$. The accumulation requires a time $2\log N(t_{s/r}+t_a)$ for $N \leq L$, $2\log L(t_{s/r}+t_a)$ for $N>L$. On completion of the accumulation the rank of the last element in each bucket is contained in the node that contained the global bucket count for that particular bucket. The accumulation can be made using the same storage locations used by the global bucket count.

The rank assignment to buckets in individual processors is obtained from the highest rank for the buckets and the partial accumulation of bucket counts contained in the counter locations in different nodes by performing the global bucket count in reverse order. In the first step the processors that have the highest rank for the buckets send the accumulation values they have to the processors from which they last received partial bucket counts. Those processors keep the highest bucket ranks, deduct their partial bucket accumulation, and send back the difference. The process is repeated until the highest rank for a bucket reaches the node that contains the local count for that particular bucket. The buckets in those nodes will be assigned the highest rank. The nodes in which the global bucket counts terminate are assigned the lowest rank for their particular buckets. The time needed for global rank assignment is $vL(1-1/N)(4t_{s/r}+t_a)$ for $N \leq L$, and $(L-1)(4t_{s/r}+t_a)+(\log N-\log L)(2t_{s/r}+t_a)$ for $N>L$ At the expense of additional storage one communication action instead of two would suffice in each step of the rank assignment process.

To complete the rank assignment a local assignment is made concurrently in all nodes.

Adding up the time needed in the various phases of the algorithm yields

For $N \leq L$

$$T = [M/N+L+\log N-1]2t_a + [L(1-1/N)3+\log N]2t_{s/r}$$

which for N=1 equals $[M+L-1]2t_a$, the time for a sequential algorithm.

For $N>L$

$$T = [M/N+L+\log N-1]2t_a + [6(L-1)+5\log N-3\log L]t_{s/r}$$

For few processors and a large number of elements compared to the number of buckets the algorithm offers linear speed-up. If the number of buckets is comparable to the number of elements to be sorted the speed-up is sublinear. For few buckets and a large number of processors, M/N small, the speed-up is $O(N/\log N)$.

V. Conclusions

The order behavior of the consecutive and the cyclic sort is the same. However, the number of communication actions and the number of comparison operations differ. The consecutive sort is always more efficient than the cyclic sort for more than eight elements per node. For fewer than eight elements, the relative costs of communication and comparison will determine which sorting algorithm is the more efficient. If the communication time $t_{s/r}$ is of the same order as the time for a comparison, t_{ce}, then the recursion on the number of nodes is always more efficient.

Rearranging a sorted sequence from one storage order to the other can be accomplished in logN steps. Successive steps perform exchanges of M/(2N) elements between nodes that differ in successive lower order address bits. In each step one processor exchanges the content of the lower (or upper) half of the locally stored elements with the content of the upper (or lower) half in the other processor, followed by a shuffle (one step left cyclic shift) on all local addresses. The time for rearrangment is $M\log N(t_{s/r}+ t_{sh})/N$, where $t_{s/r}$ is the time for sending or recieving one element between adjacent nodes, and t_{sh} is the time to perform a shuffle on local storage addresses.

The bucket sort presented for the n-cube offers linear speed-up of the element term, but is linear in the number of buckets. When each node has its own buckets, there is no improvement possible if all the buckets are nonempty in all the nodes. However, with sparsely populated buckets an improvement is possible. The algorithm also reaches a point of diminishing returns for $N>L$. For $t_{s/r}=0$ the optimum is $N=M$, for $t_{s/r}=t_a$ the optimum is $N=M/3$ and for $t_{s/r}=10t_a$ it is obtained at $N=M/25$.

The programming of the bucket sort is more complex than the other algorithms. In the algorithms employing bitonic sort the flow of control in a processor depends in each step only on a single address bit and the mask. In the bucket sort successively more bits are distinguishing the operations taken by a processor. This added complexity stems from the increasing number of folding operations being performed concurrently in the global bucket count (and the reverse operation for rank assignment)

Acknowledgement

The author is indebted to Pey-yun Peggy Li for many helpful discussions.

This work was supported in part by the Office of Naval Research under Contract N00014-84-K-0043

References

[1] Batcher, K.E.
Sorting Networks and Their Applications.
In *Spring Joint Computer Conference*, pages 307-314. IEEE, 1968.

[2] Baudet,G., Stevenson D.
Optimal Sorting Algorithms for Parallel Computers.
IEEE Trans. on Computers (1):84-87, 1978.

[3] Flynn M.J.
Very High-Speed Computing Systems.
Proc. IEEE (12):1901-1909, 1966.

[4] Hirschberg D.S.
Fast Parallel Sorting Algorithms.
Comm. of the ACM (8):657-661, 1978.

[5] Knuth. D.E.
The Art of Computer Programming, vol. 3.
Addison-Wesley, 1973.

[6] Kumar M., Hirschberg D.S.
An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and Its Application in Parallel Sorting Schemes.
IEEE Trans. on Computers (3):254-264, 1983.

[7] Nassimi, D., Sahni S.
Bitonic Sort on a Mesh-Connected Parallel Computer.
IEEE Trans. on Computers (1):2-7, 1979.

[8] Seitz C.L., Fox G.
The Homogeneous Machine.
Technical Report , Computer Science, California Institute of Technology, 1983.

[9] Stone H.S.
Parallel Processing with the Perfect Shuffle.
IEEE Trans. on Computers (2):153-161, 1971.

[10] Stone H.S.
An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations.
Journ. of the ACM (1):27-38, 1973.

[11] Stone H.S.
Parallel Tridiagonal Equation Solvers.
ACM Trans. Mathematical Software (4):289-307, 1975.

[12] Thompson, C.D., Kung H.T.
Sorting on a Mesh-Connected Parallel Computer.
Comm. ACM (4):263-270, 1977.