

# Balanced Parallel Sort on Hypercube Multiprocessors

Bülent Abali, Füsün Özgüner, *Member, IEEE*, and Abdulla Bataneh, *Member, IEEE*

**Abstract**—A parallel sorting algorithm for sorting  $n$  elements evenly distributed over  $2^d = p$  nodes of a  $d$ -dimensional hypercube is presented. The average running time of the algorithm is  $O((n \log n)/p + p \log^2 n)$ . The algorithm maintains a perfect load balance in the nodes by determining the  $(kn/p)$ th elements ( $k = 1, \dots, (p-1)$ ) of the final sorted list in advance. These  $p-1$  keys are used to partition the sorted sublists in each node to redistribute data to the nodes to be merged in parallel. The nodes finish the sort with an equal number of elements  $(n/p)$  regardless of the data distribution. A parallel selection algorithm for determining the balanced partition keys in  $O(p \log^2 n)$  time is presented. The speed of the sorting algorithm is further enhanced by the distance- $d$  communication capability of the iPSC/2 hypercube computer and a novel conflict-free routing algorithm. Experimental results on a 16-node hypercube computer show that the new sorting algorithm is competitive with the previous algorithms, and faster for skewed data distributions.

**Index Terms**—Hypercube, multiprocessing, parallel algorithms, selection, sorting.

## I. INTRODUCTION

**S**ORTING is one of the most used and fundamental of all computer operations. With increasing database sizes, parallelism must be exploited to obtain acceptable sorting times. Optimal sequential sorting algorithms that use binary comparisons are known to sort  $n$  elements in  $O(n \log n)$  time [1]. Therefore an optimal  $p$ -processor parallel sorting algorithm would sort  $n$  elements in  $O((n \log n)/p)$  time. In this paper, we present a load balanced parallel sorting algorithm, *balanced-sort*, that runs in  $O((n \log n)/p + p \log^2 n)$  average time for randomly distributed data on a hypercube multiprocessor.

A  $d$ -dimensional hypercube multiprocessor (Fig. 1) is an MIMD (multiple instruction multiple data) machine with  $2^d$  processing elements (nodes) connected to form a Boolean  $d$ -cube. Each processing element has its own memory, and processing elements communicate by exchanging messages. The delay incurred by interprocessor message communication is due to two components: the message setup time  $T_{SU}$  and the actual transfer time  $N \times T_B$ , where  $N$  is the number of bytes transferred and  $T_B$  is the one-byte transfer time. In most of the commercially available message passing multiprocessors  $T_{SU} \gg T_B$ .

The *balanced-sort* algorithm assumes that the  $n$  distinct elements to be sorted are initially distributed evenly over

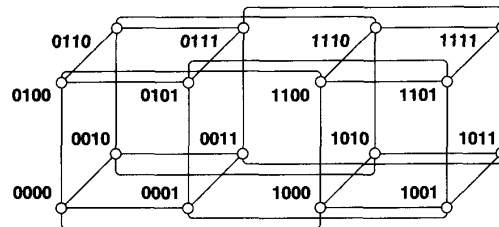


Fig. 1. A four-dimensional hypercube.

$2^d = p$  nodes of a  $d$ -dimensional hypercube ( $n \gg p$ ). The  $n$  elements are considered sorted when a global order is obtained such that for  $p-1 \geq i > j \geq 0$ , any element in node  $i$  is greater than any element in node  $j$ , and within each node  $n/p$  elements are sorted among themselves. In the *balanced-sort* algorithm, each node sorts its list of  $n/p$  elements in  $O(n/p \log(n/p))$  expected time by performing a quicksort. Each of these sorted sublists is then partitioned into  $p$  segments so that the partitions from different nodes can be merged in parallel. By determining the exact partition keys, the algorithm ensures that nodes are left with an equal number of elements  $(n/p)$  at the end of the sort, regardless of the data distribution. This is important for efficient memory utilization in a distributed memory multiprocessor. Furthermore, the exact partition keys provide perfect load balance during the merge phase. The sorting algorithms given in [2]–[5] select the partition keys either randomly or by sampling the elements that may distribute data unevenly across processors. For example, in the hyperquicksort algorithm [2] almost all of the  $n$  elements may end up being merged in one node instead of  $n/p$  in each node. A parallel selection algorithm referred to as the *fast-partition* algorithm is presented in Section IV that determines the  $p-1$  partition keys used in the *balanced-sort* algorithm in  $O(p \log^2 n)$  time. The *fast-partition* algorithm is designed to minimize the number of setups in hypercubes with coarse-grain communication.

A routing algorithm is presented in Section V that makes use of the Direct-Connect capability of the iPSC/2 hypercube to deliver elements to their destination node in just one communication step, thus reducing the communication overhead caused by store-and-forward schemes. In other algorithms, elements are stored and forwarded in the intermediate nodes  $\log p$  to  $\log p(1 + \log p)/2$  times [6], [2], [4], [3]. We show that the resulting routing algorithm is faster than the store-and-forward scheme for large values of  $n$ . However, the *balanced-sort* algorithm does not rely on the existence of the Direct-Connect capability and can be implemented on any hypercube by using the store-and-forward scheme. Another feature of the

Manuscript received August 29, 1989; revised July 23, 1990 and April 22, 1991.

B. Abali is with Bilkent University, 06533 Ankara, Turkey.

F. Özgüner is with the Department of Electrical Engineering, The Ohio State University, Columbus, OH 43210.

A. Bataneh is with Cray Research Inc., Eagan, MN 55121.

IEEE Log Number 9208489.

balanced-sort algorithm is the overlapping of interprocessor communication and computation by using asynchronous communications. In the hypercube model assumed here, nodes communicate using the *send* and *receive* primitives. These primitives are divided into two categories as synchronous send (*csend*) and receive (*crecv*), and asynchronous send (*isend*) and receive (*irecv*). Synchronous primitives block the calling process until the message is transmitted/received. On the other hand, asynchronous primitives allow a process to initiate the communication, and then continue with the computation, thus benefiting from the overlap of those two. We also assume that nodes can communicate to only 1 of their  $d$  neighbors at a time (1-port communication capability [4]).

The average running time of the balanced-sort algorithm is  $O(n/p \log n + p \log^2 n)$  for randomly distributed data. A comparison with hyperquicksort [2] on the iPSC/2 hypercube computer shows that the balanced-sort algorithm performs better for a wide range of  $n$  and  $p$  values.

The organization of the paper is as follows: Previous work on sorting is summarized in the next section. An overview of the balanced-sort algorithm is given in Section III. The algorithm for finding the balanced partition keys is described in Section IV. The routing algorithm is presented in Section V. Finally, experimental results on a 16 node Intel 386 based iPSC/2 hypercube multiprocessor and conclusions are presented in Sections VI–VII.

## II. BACKGROUND

Several parallel sorting algorithms for distributed memory hypercube multiprocessors were previously given in [6], [2], [4], [3], [7]. Johnsson's algorithm [6] is an adaptation of Batcher's [8] bitonic sort to hypercube computers. In this algorithm, each node sorts its list of size  $n/p$  using a sequential sorting algorithm in  $O(n/p \log(n/p))$  time. These lists are then exchanged among the nodes and merged  $d(d+1)/2$  times according to the bitonic sorting rule ( $d = \log p$ ), giving an overall sorting time of  $O(n/p \log(n/p) + n/p \log^2 p)$ . Wagar's [2] hyperquicksort algorithm is known to be one of the fastest sorting algorithms for hypercubes. In hyperquicksort, each node quicksorts its list of size  $n/p$ . Then, node 0 broadcasts its median element as the partition key. Nodes use the partition key to split their lists into two. The two subcubes separated by dimension  $d - 1$  exchange sublists so that the sublists that contain elements greater than the key are sent to the upper half of the hypercube along dimension  $d - 1$ , and the sublists that contain elements less than the key are sent to the lower half of the hypercube. The sublists are then merged by each node. The procedure is recursively repeated in the subcubes of the hypercube along dimensions  $d - 2, d - 3, \dots, 0$ . The sorting time for a uniform data distribution is  $O(n/p \log n + n/p \log p)$ . However, hyperquicksort does not always perform well. Randomly chosen partition keys do not necessarily split the elements evenly among the nodes. Thus, some nodes end up merging more than  $n/p$  elements, leaving the rest of the nodes idle [2]. The *samplesort* algorithm given by Fox *et al.* [3] has the same time complexity and tries to circumvent this load balancing problem by choosing a sample

of  $\ell$  keys from every node. This sample of total size  $\ell p$ , which is a representative of the distribution of  $n$  elements, is sorted and the  $(\ell i)$ th elements ( $i = 1, 2, \dots, p - 1$ ) in the sample are chosen as the partition keys. The probability of choosing good partition keys increases with large  $\ell$ . However, additional time is spent for sorting larger samples [3]. Seidel and George [4] describe several *parallel binsort* algorithms based on sampling of the elements. In the *min-max binsort*, nodes send their minimum and maximum elements to node 0, which then determines the global minimum and maximum elements in the hypercube to compute  $p - 1$  partition keys [4]. Parallel binsort algorithms of Seidel and George also assume that a hypercube node can communicate with its  $d$  neighbors simultaneously ( $d$ -port communication capability), which is reported to reduce the communication costs by a factor of at least  $d$  [4]. Won and Sahni [5] describe an improved binsorting algorithm that requires less memory than that of [4] due to the improvements in the sampling algorithm. Plaxton [7] describes sorting and selection algorithms for hypercubes which have better worst-case time complexities than previous algorithms. Plaxton's parallel quicksort runs in  $O((n \log n)/p + (n \log^{3/2} p)/p + \log^3 p \log(n/p))$  time. An  $O((n/p) \log \log p + \log^2 p \log(n/p))$  time parallel selection algorithm is used to determine the exact partition keys. Using the exact partition keys is an improvement over the previous sorting algorithms that choose the partition keys by sampling the elements. Theoretically, Plaxton's algorithms are more robust than the algorithms presented in this paper. Our contributions are new sorting, selection, and communication algorithms that have small constant factors associated with their time complexities and therefore are fast in practice.

## III. OVERVIEW OF THE BALANCED-SORT ALGORITHM

An overview of the balanced-sort algorithm will be given before the steps are described in greater detail in Sections IV–V. Initially,  $n$  distinct elements are distributed over  $2^d = p$  nodes of a hypercube with each node having  $n/p$  elements. The balanced-sort algorithm rearranges the  $n$  elements to obtain an ordered list  $L[1 \dots n]$  distributed over  $p$  nodes, such that any element in node  $i$  is greater than every element in node  $j$  whenever  $i > j$ , within each node elements are sorted, and each node is left with exactly  $n/p$  elements at the end of the sort. The major steps of the sorting algorithm are described below:

### Algorithm 1 Balanced-Sort

1. **Quicksort:** Each node independently quicksorts the  $n/p$  elements initially residing in its memory to form a sorted list  $A[0 \dots (n/p) - 1]$  in  $O(n/p \log(n/p))$  expected time [1], [9].
2. **Select Partition Keys:** The elements  $L[kn/p]$  ( $k = 1, \dots, p - 1$ ) of the final sorted list  $L[1 \dots n]$  are determined. A parallel selection algorithm that finds these  $p - 1$  partition keys in  $O(p \log^2 n)$  time is described in Section IV.
3. **Global Exchange:** Each node finds the insertion point of the  $p - 1$  partition keys in its list  $A[0 \dots (n/p) - 1]$ . (Key  $X$ 's insertion point is between  $A[r]$  and  $A[r + 1]$ ,

if  $A[r] \leq X < A[r+1]$ .) This partitions each sorted list  $A[0 \cdots (n/p) - 1]$  into  $p$  sorted segments  $A_\ell$  ( $\ell = 0, \dots, p-1$ ) such that for  $\ell > m$ , an element in segment  $A_\ell$  of any given node is greater than all the elements in segment  $A_m$  of any node, and the sum of the number of elements in the  $\ell$ th segments of the  $p$  nodes is  $n/p$ . Each node sends its segment between the insertion points of  $L[kn/p]$  and  $L[(k+1)n/p]$  to node  $k$  ( $k = 1, \dots, p-2$ ). The end points are treated similarly: each node sends the elements smaller than or equal to  $L[n/p]$  to node 0, and the elements greater than  $L[(p-1)n/p]$  to node  $p-1$ . Running time of the global exchange algorithm is between  $O(p \log p)$  and  $O(n + p \log p)$ , depending on the initial global ordering of data, as will be discussed in Section V and is overlapped with the computations in the binary tree merge step.

4. **Binary Tree Merge:** Each node  $k$  receives  $p-1$  sorted segments from other nodes, and has its segment  $k$ . Each node independently forms a single sorted list out of these  $p$  segments in  $O(n/p \log p)$  time by a binary tree merge, which completes the sort.

In step 1, the heapsort algorithm can be used to attain an  $O(n/p \log(n/p))$  worst-case time bound [1], however quicksort was implemented as it is faster in practice [9]. As indicated earlier, the partitioning step ensures that the nodes receive  $n/p$  elements each for the binary tree merge step. Thus, merge times will be equal in each node, and nodes will finish the sort exactly with  $n/p$  elements each. The partitioning step also guarantees that the segments to be merged on different nodes are disjoint so that no interprocessor communication is needed during binary tree merge.

#### IV. SELECTION OF THE PARTITION KEYS

In the selection of the partition keys, the elementary hypercube algorithms *exchange-add* and *transpose* are used. These algorithms and their variations originally appeared in several references including [10]–[14]. We include them here for the sake of completeness.

##### A. Elementary Hypercube Algorithms

The exchange-add algorithm finds the global sum of  $2^d = p$  numbers distributed over the nodes of a  $d$ -cube in  $O(\log p)$  time and leaves each node with a copy of the result. Each node  $z$  executes the following in exchange-add:

###### Algorithm 2 Exchange-Add

$z = (z_{d-1} \cdots z_0)$ : This node's id  
 $r$ : Initially, the partial sum residing in this node.  
 The global sum is returned in  $r$ .  
 for  $i = 0, \dots, d-1$   
   irecv partial sum into  $s$  from node  $(z_{d-1} \cdots \bar{z}_i \cdots z_0)$   
   along dimension  $i$   
   csend partial sum  $r$  to node  $(z_{d-1} \cdots \bar{z}_i \cdots z_0)$   
   along dimension  $i$   
   wait for irecv to complete  
    $r = r + s$   
 endfor

The transpose algorithm distributes  $p$  values in every node, each addressed to a different node, in  $O(p \log p)$  time. Values are represented in the form of tuples to identify their source and destination nodes. Let  $\langle val, dst, src \rangle$  denote the value  $val$  to be sent from node  $src$  to node  $dst$ . In node  $z$  ( $z = 0, 1, \dots, p-1$ ), initially there are  $p$  tuples  $\langle val_j^z, j, z \rangle$  ( $j = 0, 1, \dots, p-1$ ). Upon completion of the algorithm, node  $z$  receives the  $p$  tuples  $\langle val_z^j, z, j \rangle$  ( $j = 0, 1, \dots, p-1$ ) addressed to it from the other nodes.

Elements are transposed with the following algorithm:

###### Algorithm 3 Transpose

$z = (z_{d-1} \cdots z_0)$ : This node's id  
 $T[0 \cdots p-1]$ : List of  $p$  tuples  $\langle val_{dst}^z, dst, z \rangle$   
 residing in this node.  
 for  $i = 0, \dots, d-1$   
   split  $T$  into two lists  $B$  and  $B'$   
    $B$  contains the tuples whose  $dst$  fields  
   agree with  $z_{d-1} \cdots z_0$  in  $i$ th bit position,  
   and  $B'$  contains tuples that do not.  
   irecv list  $C$  from node  $(z_{d-1} \cdots \bar{z}_i \cdots z_0)$   
   along dimension  $i$   
   csend list  $B'$  to node  $(z_{d-1} \cdots \bar{z}_i \cdots z_0)$   
   along dimension  $i$   
   wait for irecv to complete  
    $T \leftarrow B \cup C$   
 endfor

The algorithm communicates along  $d$  different dimensions, and in each direction lists of size  $p/2$  are exchanged, resulting in an overall execution time of  $O(p \log p)$ .

##### B. Fast-Partition Algorithm

The *fast-partition* algorithm for finding the partition keys, namely  $L[kn/p]$  ( $k = 1, \dots, p-1$ ), is based on the following scheme: An element  $X$  is proposed as the partition key  $L[kn/p]$ . Each node  $i$  ( $i = 0, \dots, p-1$ ) determines the number of elements smaller than or equal to  $X$  (referred to as the *local rank*) in its sorted list  $A[0 \cdots (n/p) - 1]$ . Since in each node  $A[0 \cdots (n/p) - 1]$  is already sorted, the local rank of  $X$  can be determined in  $\log(n/p)$  comparisons by a binary search. Then, the  $p$  local ranks of  $X$  are summed by using the exchange-add algorithm, to find its global rank, i.e.,  $X$ 's position in the final sorted list  $L[1 \cdots n]$ . If  $X$ 's global rank is greater(smaller) than  $kn/p$ , a new candidate smaller(greater) than  $X$  is proposed as the partition key, and the procedure is iterated in this fashion until the key with the global rank  $kn/p$ , i.e.,  $L[kn/p]$  is found. The fast-partition algorithm amortizes the high setup cost of interprocessor communication over  $p-1$  partition keys ( $k = 1, \dots, p-1$ ) by processing them in one batch.

For the  $k$ th balanced partition key  $L[kn/p]$  ( $k = 1, \dots, p-1$ ), each node keeps two local variables  $\min[k]$  and  $\max[k]$  which are pointers to its sorted list  $A[0 \cdots (n/p) - 1]$ . The *local search space*, in each node, for the  $k$ th balanced partition key is between  $\min[k]$  and  $\max[k]$  such that  $A[\min[k]] < L[kn/p] < A[\max[k]]$ . The *global search space* for the  $k$ th partition key is the collection of its  $p$  local search spaces. Initially, the global search space for the  $k$ th partition key  $L[kn/p]$  consists of  $n$

elements (i.e., all the elements in the hypercube) and each of the  $p$  local search spaces consists of  $n/p$  elements.

In the first iteration, each node proposes  $A[\lfloor kn/p^2 \rfloor]$  as a candidate for the  $k$ th partition key. After the first iteration, each node  $i$  ( $i = 0, \dots, p - 1$ ) proposes the median element  $C^i[k]$  of its local search space as a candidate for the balanced partition key  $L[\lfloor kn/p \rfloor]$  ( $k = 1, \dots, p - 1$ ). The median of the  $p$  candidates  $C^0[k], C^1[k], \dots, C^{p-1}[k]$ , each coming from a different node, is selected as the final candidate and used during the local and global ranking phases of the iteration. For determining the medians, the *transpose* algorithm rearranges the  $p(p - 1)$  candidates in the hypercube such that all of the candidates associated with the partition key  $L[\lfloor kn/p \rfloor]$ , namely  $C^i[k]$  ( $i = 0, \dots, p - 1$ ), move to node  $k$ , in parallel for  $k = 1, \dots, p - 1$ , in  $O(p \log p)$  total time. Then, the medians of the  $p$  candidates for each of the  $(p - 1)$  partition keys are determined in parallel, with node  $k$  determining the median of the  $p$  candidates proposed for  $L[\lfloor kn/p \rfloor]$ . Since  $1 \leq k \leq p - 1$ , node 0 receives only *NIL* values, hence does not participate in the median selection. Fig. 2 illustrates this search procedure for only one of the partition keys ( $L[\lfloor n/p \rfloor]$ ). Initially, the algorithm determines that node 0's candidate  $X_1$  is the median candidate for the first partition key  $L[\lfloor n/p \rfloor]$ . Each node finds the local rank (insertion point) of  $X_1$  in its list and the global rank of  $X_1$  is found to be smaller than  $n/p$ . The new search space is shown in Fig. 2 below the insertion point of  $X_1$ . In iteration 2, each node proposes the median key in its current local search space as the new candidate. The median of these candidates,  $X_2$ , is selected and found to have a global rank greater than  $n/p$ . The search space is further reduced as shown in Fig. 2 and iterations continue until  $L[\lfloor n/p \rfloor]$  is found. In a manner similar to the sequential binary search, the global search space is approximately cut by half in each iteration by proposing a candidate from the middle of the global search space.

Each node  $i$  executes the following steps in the fast-partition algorithm:

**Algorithm 4 Fast-Partition**

1. **Initialize:** Let  $A[0 \dots (n/p) - 1]$  be the sorted list of  $n/p$  elements in node  $i$  ( $i = 0, \dots, p - 1$ ). Let the local variables  $\min[k]$  and  $\max[k]$  ( $k = 1, \dots, p - 1$ ) be the pointers for the sorted list  $A[0 \dots (n/p) - 1]$ . During the iterations, the local search space for the  $k$ -th balanced partition key  $L[\lfloor kn/p \rfloor]$  will always be between  $\min[k]$  and  $\max[k]$  such that  $A[\min[k]] < L[\lfloor kn/p \rfloor] < A[\max[k]]$ . Initialize  $\min[k] = -1$ ,  $\max[k] = n/p$  for  $k = 1, \dots, p - 1$ . Initially, propose  $A[\lfloor kn/p^2 \rfloor]$  as a candidate for the balanced partition key  $L[\lfloor kn/p \rfloor]$  for  $k = 1, \dots, p - 1$ . This step takes  $O(p)$  time.
2. **Transpose:** Each node  $i$  ( $i = 0, \dots, p - 1$ ) is now holding a candidate for  $L[\lfloor kn/p \rfloor]$  ( $k = 1, \dots, p - 1$ ). The  $p$  candidates associated with  $L[\lfloor kn/p \rfloor]$  and distributed over the  $p$  nodes are moved to node  $k$  ( $k = 1, \dots, p - 1$ ) using the transpose algorithm. Since  $k \geq 1$ , node 0 gets only *NIL* values. This step takes  $O(p \log p)$  time.
3. **Select Median of the Candidates:** Node  $i$  now holds the  $p$  candidates for the partition key  $L[\lfloor in/p \rfloor]$ . Node  $i$  determines the median candidate by sorting the  $p$  keys and taking their median in  $O(p \log p)$  time. Candidates

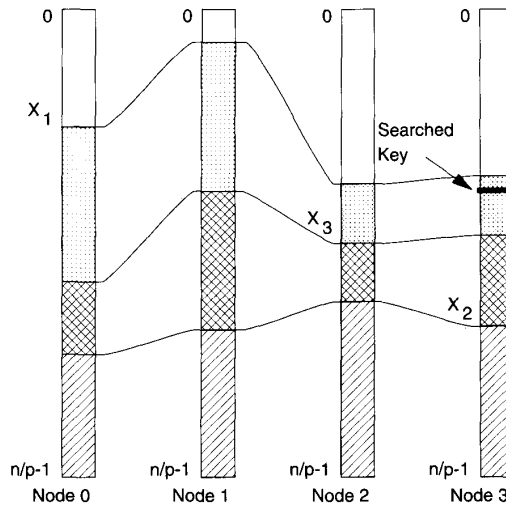


Fig. 2. Illustration of the search procedure for finding the partition key  $L[\lfloor kn/p \rfloor]$  for  $k = 1$ .

other than the median are discarded. Node 0 is idle at this step since it holds all the *NIL* values.

4. **Broadcast:** Each node  $i$  ( $i = 1, \dots, p - 1$ ) broadcasts its median key to the rest of the nodes  $0, 1, \dots, p - 1$ . The  $p$  medians are broadcast in parallel in  $O(p)$  time using an *Exchange-Add* like algorithm that has the arithmetic addition replaced with the set union operation (see also [11] for broadcast on hypercubes). Each node receives the broadcast keys and forms a local copy  $C[1 \dots p - 1]$  of the candidate list. ( $C[k]$  is the key received from node  $k$ . Thus,  $C[k]$  is the candidate for the balanced partition key  $L[\lfloor kn/p \rfloor]$ .)
5. **Local Rank Computation:** Each node determines the local rank  $R[k]$  of  $C[k]$  ( $k = 1, \dots, p - 1$ ) by a binary search in  $A[0 \dots (n/p) - 1]$ . This step takes  $O(p \log(n/p))$  time.
6. **Global Rank Computation:** The  $p$  local ranks of  $C[k]$  distributed over  $p$  nodes are summed using the exchange-add algorithm and stored in the local variable  $G[k]$  ( $k = 1, \dots, p - 1$ ), resulting in  $O(p \log p)$  overall time for this step.  $G[k]$  holds the global rank, i.e., the position of  $C[k]$  in the sorted output  $L[1 \dots n]$  of the *Balanced-Sort* algorithm. Each node has a copy of  $G[1 \dots p - 1]$ . If  $G[k] = kn/p$ , then the  $k$ th balanced partition key is found ( $L[\lfloor kn/p \rfloor] = C[k]$ ).
7. **Reduce the Search Space:** If  $G[k] > kn/p$ , it is known that  $C[k] > L[\lfloor kn/p \rfloor]$ . Therefore, each node decrements its  $\max[k]$  pointer to the smallest possible value such that  $C[k] \leq A[\max[k]]$  in its list  $A[0 \dots (n/p) - 1]$ . Likewise, if  $G[k] < kn/p$ , then each node increments its  $\min[k]$  pointer to the largest possible value such that  $C[k] \geq A[\min[k]]$  in its list  $A[0 \dots (n/p) - 1]$ .
8. **Propose New Candidates:** For  $k = 1, \dots, p - 1$ , each node proposes  $A[\lfloor (\max[k] + \min[k])/2 \rfloor]$  as the new candidate for  $L[\lfloor kn/p \rfloor]$ . If  $\max[k] = \min[k] + 1$ , the balanced partition key  $L[\lfloor kn/p \rfloor]$  cannot be in

this node, since there are no elements left between the two pointers  $\min[k]$  and  $\max[k]$ . In that case, the node proposes a dummy value *NIL* as the candidate instead of  $A[\lfloor(\max[k] + \min[k])/2\rfloor]$ . This ensures that an element which does not belong to the global search space for  $L[kn/p]$  is not proposed and thus, the median selection process at step 3 is not adversely affected. At step 3, the *NIL* values are discarded before a median is chosen. Iterations continue from Step 2 until all of the  $p - 1$  balanced partition keys are found.

By adding the dominant complexity terms,  $O(p \log p)$  in steps 2, 3, 6 and  $O(p \log(n/p))$  in step 5, the time complexity of each iteration is found to be  $O(p \log n)$ . An upper bound for the number of iterations can be established as follows: The candidate key  $C[k]$  for the balanced partition key  $L[kn/p]$  is the median element (i.e.,  $A[\lfloor(\max[k] + \min[k])/2\rfloor]$ ) of one of the  $p$  local search spaces. Therefore, once the global rank  $G[k]$  of  $C[k]$  is determined, the size of the local search space in the node that proposed  $C[k]$  will be reduced exactly by half. If the global rank  $G[k] < kn/p$ , it is known that proposed candidate  $C[k]$  is smaller than  $L[kn/p]$ . Therefore,  $L[kn/p]$  cannot be between  $A[\min[k]]$  and  $C[k]$ , and the pointer  $\min[k]$  is incremented to eliminate the elements between  $A[\min[k]]$  and  $C[k]$  from consideration, which eliminates one half of the elements between  $\min[k]$  and  $\max[k]$  (Step 7). Likewise, if the global rank  $G[k] > kn/p$ , the pointer  $\max[k]$  is updated to eliminate the other half of the elements in the local search space. In each iteration, the size of at least one local search space will be reduced by half as described above.

There are  $p$  local search spaces with each having  $n/p$  elements initially, and it takes  $\log(n/p)$  iterations to reduce the size of each local search space. Therefore, an upper bound for the number of iterations is  $p \log(n/p)$ . However, more than one local search space will be reduced at each iteration in general. On the average, Algorithm 4 will iterate  $\log n$  times: The candidate  $C[k]$ , as determined in Step 3, is the median of  $p$  candidates each of which is the median of a local search space. Thus,  $C[k]$  falls approximately in the middle of the global search space for  $L[kn/p]$ . This means that on the average, the size of the global search space for  $L[kn/p]$  will be halved in each iteration. Since the size of the global search space for  $L[kn/p]$  is initially  $n$ , the average iteration count will be  $\log n$ . Each iteration takes  $O(p \log n)$  time. Therefore, Algorithm 4 finds the  $p - 1$  partition keys in  $O(p \log^2 n)$  average time. Note that the communication setup cost per iteration is only  $(3 \log p)T_{SU}$  which is the sum of the setup times in steps 2, 4, and 6.  $T_{SU}$  is the message setup time.

Algorithm 4 has a property similar to that of the sequential binary search: the size of the search space decreases geometrically. In the first few iterations Algorithm 4 makes big jumps in the global search space and begins proposing candidates very close to the balanced partition keys. In practice, if a candidate  $C[k]$  has a global rank sufficiently close to  $kn/p$  such that the criterion  $\epsilon \geq |kn/p - G[k]|$  is satisfied, iterations can be terminated earlier, resulting in a faster partitioning algorithm with partitions of size  $n/p \pm 2\epsilon$  at worst.

As a further improvement, the upper bound for the number of iterations can be reduced to  $O(\log n)$  iterations by using

*weighted* medians in Step 3 of the algorithm. The weight of a candidate key is defined as the size of the corresponding local search space, i.e.,  $\max[k] - \min[k] - 1$ . Algorithm 4 is modified as follows: In Step 2, each node sends to the destination node the candidate key and its weight. Thus, each node  $k$  ( $k = 1, \dots, p - 1$ ) receives  $p$  candidate-weight pairs  $\{(C_k^0, W_k^0), (C_k^1, W_k^1), \dots, (C_k^{p-1}, W_k^{p-1})\}$ . In Step 3, each node  $k$  sorts the  $p$  candidate keys so that

$$C_k^{\ell_1} < C_k^{\ell_2} < \dots < C_k^{\ell_p}$$

where  $0 \leq \ell_j \leq p - 1$  and then determines  $m$  such that

$$\sum_{j=1}^{m-1} W_k^{\ell_j} < \frac{S}{2} \quad \text{and} \quad \sum_{j=1}^m W_k^{\ell_j} \geq \frac{S}{2} \quad (1)$$

where  $S = \sum_{j=0}^{p-1} W_k^j$  is the size of the global search space for this iteration. The *weighted* median  $C_k^{\ell_m}$  is selected as the final candidate for  $L[kn/p]$ , and the remaining  $p - 1$  candidates are discarded. Equation (1) guarantees that there are at least  $S/4$  elements smaller than or equal to  $C_k^{\ell_m}$ , and that there are at least  $S/4$  elements greater than  $C_k^{\ell_m}$  in the global search space. Thus, in each iteration at least  $1/4$ th of the elements in the global search space are eliminated, and therefore the algorithm terminates after  $O(\log n)$  iterations. Note that communicating the weights and computing  $m$  increase the time complexity of each iteration only by a small constant factor. The idea of using *weighted medians* for selection is due to Galil and Megiddo [15] and Frederickson and Johnson [16]. The procedure is explained in detail in Ibaraki and Katoh [17]. Our contribution here is the application of the procedure to all partition keys in parallel.

## V. GLOBAL EXCHANGE

Let  $A_i^\ell$  ( $\ell = 0, 1, \dots, p - 1$ ) denote the  $p$  sorted segments in node  $i$ , induced by the  $p - 1$  balanced partition keys. In the global exchange step of the balanced-sort algorithm, segments are exchanged among the nodes such that each node  $i$  sends its segment  $A_i^\ell$  to node  $\ell$ . A communication scheme similar to that of the hyperquicksort algorithm could easily be used to implement the global exchange [2]: Segments that contain the elements smaller (greater) than the  $p/2$ th partition key (i.e.,  $L[n/2]$ ) are sent to the lower (upper) half of the hypercube along dimension  $d - 1$ . The upper and lower subcubes repeat this procedure recursively along dimensions  $d - 2, d - 3, \dots, 0$  using the rest of the partition keys. However, this scheme results in up to  $\log p$  memory-to-memory copy operations for each element. In this section, a communication algorithm for reducing this overhead for large values of  $n$  is described. The algorithm makes use of a hardware feature of the iPSC/2 hypercube that is described below.

In the iPSC/2 hypercube, each node is equipped with a *direct connect module* (DCM), which allows nonneighboring nodes to communicate directly [18]. A DCM can be considered to be a  $(d + 1)$  input,  $(d + 1)$  output crossbar switch. The  $d$  input-output pairs of the DCM are connected to the  $d$  neighbors of the node through hypercube links. The remaining input-output pair is connected to the internal bus of the node,

hence to the local memory. A DCM can be set up so that a message coming from one link can be immediately directed to another link, thereby avoiding the store-and-forward overhead. The connection through the DCM's is a circuit-switched type connection. Measurements on iPSC/2 indicate that communication between two nonneighboring nodes is as fast as communication between adjacent nodes if all the links in the communication path between the two nodes are available. Therefore, the objective of our global exchange algorithm is to ensure that the communication paths between the nodes are available during the exchange of the sorted segments. The communication hardware uses the *e-cube* algorithm for routing messages [18], where the routing tag is obtained by taking the bit by bit logical exclusive-OR of the source and destination node addresses. The nonzero bit positions in the routing tag, read from right to left, give the hypercube coordinates as a message goes along. For example, if the routing tag is  $r = (r_4 r_3 r_2 r_1 r_0) = (01011)$ , the message travels along dimensions 0, 1, and 3 to arrive at its destination.

By making use of the DCM's and *e-cube* routing, the following algorithm delivers segments directly to their destinations with segments following disjoint paths. Hypercube nodes distributively execute Algorithm 5, where  $\oplus$  denotes an exclusive-OR operation:

*Algorithm 5 Global-Exchange*

```

z: this node's id
A0,...,p-1z: segments to be delivered

for k = 1, ..., p - 1
    irecv segment Az⊕kz⊕k from node z ⊕ k
    isend segment Azz to node z ⊕ k
    wait for irecv and isend to complete
    sync
endfor
    
```

The  $p - 1 = 7$  steps of the global-exchange on a 3-cube are shown in Fig. 3. Processors wait at the *sync* instruction until it is executed by all  $p$  of them to ensure that no processor gets ahead of the others and blocks the network links. In each step, each node  $z$  sends to the node numbered  $z \oplus k$ , which means that the routing tag is identical ( $z \oplus (z \oplus k) = k$ ) for all the segments being exchanged. The nonzero bit positions in  $k$  give the dimensions traversed by the segments. For example, for  $k = (101)$ , the links along the dimensions 0 and 2 are used by the segments as seen in Fig. 3. Since the value of  $k$  is the same in all of the nodes, every source node sends in direction 0, and the crossbars forward messages coming from direction 0 to direction 2. This ensures that no more than one segment is routed to the same link, thus segments follow disjoint paths.

The *sync* instruction is executed in  $O(\log p)$  time. To synchronize, each node sends and receives a dummy token along dimensions  $0, 1, \dots, d - 1$ . Thus, until *sync* is issued by all of the processors, none of them can proceed to the next step of the algorithm. When segment sizes are more or less equal ( $\approx n/p^2$ ), Algorithm 5 runs in  $O(n/p + p \log p)$  time since processors finish each step of Algorithm 5 at about the same time, and the *sync* operation does not delay them. However, when segment sizes are significantly different, such

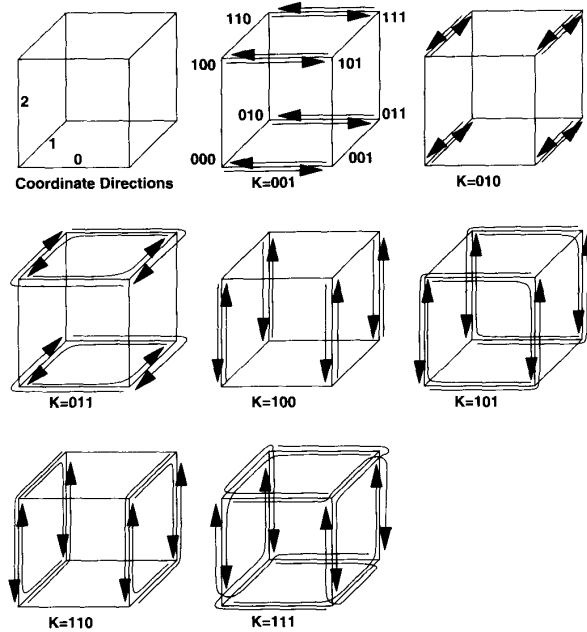


Fig. 3. The 7 communication steps of *Global-Exchange* on a 3-cube.

as in the case where each node has one segment of size  $n/p$  and  $p - 1$  segments of size 0, the exchange of segments may be serialized by the *sync* operation and therefore may take longer. An upper bound for the segment transfer time is  $O(n + p \log p)$  due to this serialization. However, this is a very pessimistic upper bound. Even when nodes have one segment of size  $\approx n/p$ , the segments are transferred in parallel in many cases. Furthermore, experimental results show that when segment sizes are significantly different, the performance is not affected significantly. This is explained by a combination of factors: Communication time is smaller than computation time, the global exchange step is overlapped with the merge step, and the binary tree merge is usually faster with such data distributions as described in Section VI.

While reducing  $\log p$  memory-to-memory copy operations to only 1, Algorithm 5 increases the number of communication steps from  $\log p$  to  $p - 1$ , since the  $p - 1$  segments in each node are individually delivered to their destinations. Thus, there is a tradeoff between the communication volume and the communication setup cost. For the case where all segment sizes are equal ( $n/p^2$ ) this tradeoff can be analyzed as follows: If the store-and-forward scheme is used as in hyperquicksort, each node will send half ( $p/2$ ) of its segments to the other subcube and keep the other half. This is performed  $\log p$  times until all of the segments reach their destination node. Therefore, the overall communication cost is

$$T_{comm} = \log p \left( T_{SU} + \frac{N}{2p} T_B \right) \quad (2)$$

where  $T_{SU}$  is the communication setup time,  $T_B$  is the one byte transfer time, and  $N$  is the amount of data being sorted

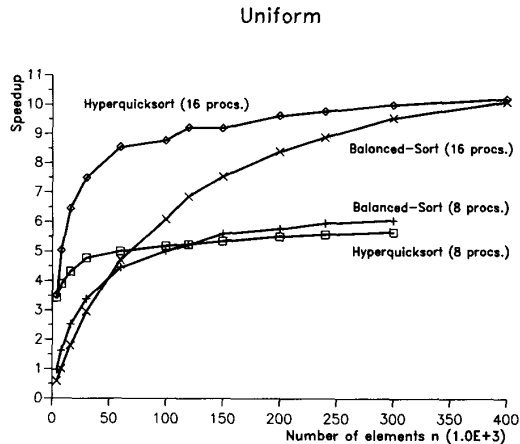


Fig. 4. Speedup of hyperquicksort and balanced-sort for the *UNIFORM* distribution.

in bytes ( $N = 4n$  for 4 byte elements). If Algorithm 5 is used instead of the store-and-forward scheme, each node sends its segments to  $p - 1$  other nodes directly. Therefore, the overall communication cost is

$$T_{\text{comm}} = (p - 1) \left( T_{SU} + \frac{N}{p^2} T_B \right). \quad (3)$$

Comparison of (2) and (3) indicates that Algorithm 5 takes less time than the store-and-forward scheme for large values of  $N$  and most values of  $p$ . Equating (2) and (3), and using the measured communication parameters of iPSC/2, the breakeven values of  $N$  for the two algorithms are found as 42 KBytes, 134 KBytes, and 432 Kbytes for  $p = 4, 8, 16$ , respectively ( $T_{SU} = 955 \mu\text{s}$  for messages longer than 100 bytes, and  $T_B = 0.366 \mu\text{s}$ ). Beyond those values of  $N$ , the communication time of Algorithm 5 is smaller than the store-and-forward scheme. Thus, Algorithm 5 is more suitable when the volume of communication is large and the setup time ( $T_{SU}$ ) is small.

The algorithm can be slightly improved by observing that the communication links used for  $k$  and  $\bar{k}$ , the ones complement of  $k$ , are also disjoint. For example, in Fig. 3 note that the links used for  $k = (101)$  and  $\bar{k} = (010)$  are disjoint. This allows two sets of segments to be exchanged between the execution of two *syncs*.

Another interesting feature of the balanced-sort algorithm is the ability to overlap communication and computation in the global exchange and binary tree merge steps using asynchronous communication primitives: as soon as node  $z$  receives the first segment  $A_z^t$ , it may begin merging the pair of segments  $A_z^t$  and  $A_z^z$ , while two more segments arrive at the node in parallel with the merge. Merging and exchanging of segment pairs continue in this pipelined fashion until all of the segments are exchanged. Note that Algorithm 5 provides conflict-free routing in omega [19], indirect binary n-cube [20], and generalized cube [21] multistage networks as well. Thus, balanced-sort can also take advantage of any of those networks.

TABLE I  
EXECUTION TIMES (ms) OF HYPERQUICKSORT AND  
BALANCED-SORT FOR THE *UNIFORM* DISTRIBUTION

$d$	$n(10^3)$	Hyper.	Blncd.	Q	P	GM
1	4	78	96	56	25	15
1	8	158	162	116	16	30
1	16	332	329	251	34	44
1	30	630	* 623	480	34	109
1	60	1280	1288	1037	36	215
2	4	51	101	28	54	19
2	8	96	151	57	63	31
2	16	194	246	118	68	60
2	30	380	410	234	68	108
2	60	766	789	503	78	208
2	100	1340	*1274	856	73	345
2	120	1613	*1532	1037	83	412
3	4	38	134	14	96	24
3	8	66	158	28	96	34
3	16	124	213	58	103	52
3	30	223	315	113	117	85
3	60	458	516	237	125	154
3	100	777	805	410	127	268
3	120	942	946	503	134	309
3	150	1182	*1129	628	127	374
3	200	1574	*1503	857	143	503
3	240	1900	*1778	1040	135	603
3	300	2396	*2235	1338	148	749
4	4	37	221	8	176	37
4	8	51	256	15	191	50
4	16	83	297	28	206	63
4	30	142	361	55	221	85
4	60	268	486	114	235	137
4	100	460	663	200	264	199
4	120	535	719	238	250	231
4	150	686	837	306	255	276
4	200	900	1032	411	266	355
4	240	1084	1192	504	268	420
4	300	1350	1418	634	267	517
4	400	1823	1831	872	287	672

$d$ : dimension of the hypercube.

\*: Indicates the faster balanced-sort cases.

Hyper: Hyperquicksort time.

Blncd: Balanced-Sort time.

Q and P: Quicksort and partitioning times of balanced-sort.

GM: Global exchange and merge time of balanced-sort.

## VI. IMPLEMENTATION RESULTS

The performance of hyperquicksort and the balanced-sort algorithm (Algorithm 1) which uses Algorithm 4 for partitioning were compared on a 16-node iPSC/2 hypercube. Won and Sahni [5] compared the performance of Wagar's hyperquicksort [2] extensively with bitonic sort of Johnsson [6], [8], min-max binsort of Seidel and George [4], and several versions of the parallel binsort of Won and Sahni [5] on the NCUBE/7 hypercube multiprocessor. Won and Sahni's results show that hyperquicksort is faster than the other algorithms in many cases. Therefore, we consider comparison with hyper-

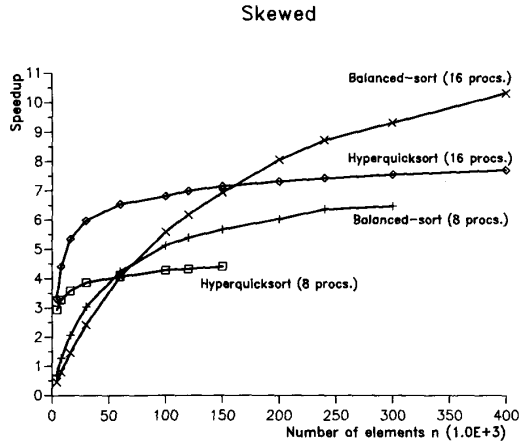


Fig. 5. Speedup of hyperquicksort and balanced-sort for the *SKEWED* distribution.

quicksort to be sufficient to demonstrate the performance of the balanced-sort algorithm.

The iPSC/2 hypercube used in evaluating the algorithms consists of 16 nodes each of which consists of a 16-MHz 386 microprocessor, a 64 KByte cache, 1 MByte of memory and a DCM. Communication bandwidth was measured as 2.73 MBytes/s. Communication setup time was measured as 536  $\mu$ s for short messages ( $\leq 100$  bytes), and 955  $\mu$ s for long messages ( $> 100$  bytes). Randomly generated 32-bit integers were sorted. The global exchange and the binary tree merge steps of the balanced-sort were implemented using asynchronous communication primitives to allow communication and computation overlap as described in Section V. However, the amount of overlap was not measured. To observe the effect of initial data distribution on the performance of the partitioning and global exchange algorithms, three different initial data distributions were used.

The *UNIFORM* distribution consists of randomly distributed elements over the hypercube such that the  $p - 1$  balanced partition keys  $L[kn/p]$  ( $k = 1, \dots, p - 1$ ) partition each list  $A[0 \dots (n/p) - 1]$  into  $p$  segments of almost equal size ( $\simeq n/p^2$ ). Hyperquicksort achieves its best performance with the *UNIFORM* distribution, since it is most likely that the nodes receive  $n/p$  elements each during the merge phase and have equal loads throughout the sort. Fig. 4 shows the speedup of balanced-sort and hyperquicksort as a function of  $n$ , for 8 and 16 node hypercubes, for the *UNIFORM* distribution. Table I shows the sort times for hyperquicksort and balanced-sort. For small  $n$ , balanced-sort performs significantly worse than hyperquicksort for all hypercube dimensions. The partitioning overhead dominates the overall time in balanced-sort. However, as  $n$  grows, the speedup of balanced-sort grows faster than hyperquicksort speedup and it is greater than hyperquicksort speedup for a few cases (indicated by \* in Table I). We attribute this result to the global exchange algorithm described in Section V which is implemented to overlap communication and computation, and avoid the store-and-forward overhead.

TABLE II  
EXECUTION TIMES (ms) OF HYPERQUICKSORT AND  
BALANCED-SORT FOR THE *SKEWED* DISTRIBUTION

$d$	$n(10^3)$	Hyper.	Blncd.	Q	P	GM
1	4	78	114	56	50	8
1	8	159	188	119	54	15
1	16	327	335	247	59	29
1	30	645	*613	496	64	53
1	60	1315	*1208	1034	70	104
2	4	54	128	27	88	13
2	8	102	175	56	98	21
2	16	207	261	119	106	36
2	30	397	413	235	116	62
2	60	803	*742	496	125	121
2	100	1372	*1204	873	134	197
2	120	**	1406	1035	135	236
3	4	44	185	13	149	23
3	8	78	201	27	149	25
3	16	149	258	57	164	37
3	30	275	349	111	178	60
3	60	562	*538	235	194	109
3	100	941	*785	402	208	175
3	120	1139	*914	497	209	208
3	150	1431	*1112	628	226	258
3	200	**	1437	873	226	338
3	240	**	1665	1036	225	404
3	300	**	2090	1348	240	502
4	4	39	279	8	238	33
4	8	58	317	15	265	37
4	16	100	367	28	292	47
4	30	178	437	54	321	62
4	60	350	561	111	353	97
4	100	592	721	201	377	143
4	120	705	797	236	381	180
4	150	882	910	299	410	201
4	200	1185	*1076	402	415	259
4	240	1427	*1216	497	415	304
4	300	1793	*1453	628	451	374
4	400	2413	*1805	873	441	491

$d$ : dimension of the hypercube.

\*: Indicates the faster balanced-sort cases.

\*\* : Hyperquicksort does not run due to insufficient memory.

Hyper: Hyperquicksort time.

Blncd: Balanced-Sort time.

Q and P: Quicksort and partitioning times of balanced-sort.

GM: Global exchange and merge time of balanced-sort.

In the *SKEWED* distribution, data are globally presorted in a way to increase the communication time of the global exchange step of balanced-sort; the distribution is such that in node  $i$  ( $i = 0, 1, \dots, p - 1$ ), segment  $A_i^\ell$  has a size of  $n/p$  if  $\ell = i + 1(\text{mod } p)$ , and has a size of 0 if  $\ell \neq i + 1(\text{mod } p)$ . Thus, all of the  $n/p$  elements in node  $i$  have to move to node  $i + 1(\text{mod } p)$  during the global exchange step. Fig. 5 and Table II show that for large  $n$ , balanced-sort is faster than hyperquicksort. For example, for  $n = 400\,000$  and  $p = 16$ , a speedup of 7.7 for hyperquicksort and 10.3 for balanced-sort is obtained. The skew in the initial data distribution



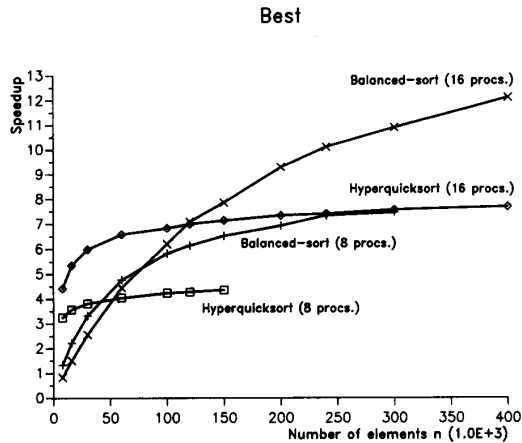


Fig. 6. Speedup of hyperquicksort and balanced-sort for the *BEST* distribution.

causes a load imbalance in the merge step of hyperquicksort; some nodes merged much more than  $n/p$  elements, while in the balanced-sort algorithm, each node merged exactly  $n/p$  elements.

The increase in the communication time due to the *SKEWED* distribution did not affect balanced-sort adversely. On the contrary, comparison of Tables I and II show that for the same values of  $n$  and  $p$ , balanced-sort is faster for the *SKEWED* distribution than the *UNIFORM* distribution for many cases. This result was partially because the interprocessor communication time is smaller than the other steps of balanced-sort, and partially because of the implementation of the two-way merge algorithm used in the binary tree merge. If one list is exhausted during the merge, the remainder of the other list is moved by a *block copy* operation which results in fewer comparisons. In the *SKEWED* distribution, since sizes of the segments being merged are significantly different, binary tree merge takes much less time than in the *UNIFORM* distribution. Thus, the anticipated increase in the communication time is nullified by a fast binary tree merge. This may be verified from the GM column of Tables I and II.

In the *BEST* distribution, data are globally presorted so that initially all elements in node  $i$  are greater than all elements in node  $j$ , if  $i > j$ , and within a node elements are randomly distributed. This is the ideal global data distribution for balanced-sort. Since data are globally ordered to begin with, no interprocessor communication takes place to exchange the segments during the global exchange step of balanced-sort. Fig. 6 and Table III show that for large  $n$  the balanced-sort algorithm performs better than hyperquicksort. For example, for  $n = 400\,000$  and  $p = 16$ , a speedup of 7.7 for hyperquicksort and 12.1 for balanced-sort is obtained. This difference is because balanced-sort did not incur a communication cost during the global exchange step, and merged fast due to unequal segment sizes, but primarily it is because hyperquicksort had a load imbalance. For example, for  $n = 400\text{K}$  and  $p = 16$ , the hyperquicksort sorts in 2413 ms

TABLE III  
EXECUTION TIMES (ms) OF HYPERQUICKSORT AND  
BALANCED-SORT FOR THE *BEST* DISTRIBUTION

$d$	$n(10^3)$	Hyper.	Blncd.	Q	P	GM
1	4	75	110	55	50	5
1	8	155	193	119	54	20
1	16	318	322	247	59	16
1	30	631	*589	496	64	29
1	60	1302	*1162	1034	69	59
2	4	55	123	27	88	8
2	8	107	164	56	96	12
2	16	216	243	118	106	19
2	30	414	*382	235	115	32
2	60	852	*681	496	124	61
2	100	1462	*1107	873	134	100
2	120	1723	*1288	1035	133	120
3	8	79	191	27	148	16
3	16	150	241	57	163	21
3	30	279	320	111	178	31
3	60	566	*481	235	194	52
3	100	952	*693	402	209	82
3	120	1155	*804	497	209	98
3	150	1449	*969	628	222	119
3	200	**	1251	872	223	156
3	240	**	1444	1035	224	185
3	300	**	1816	1348	240	228
4	4	38	276	8	237	31
4	8	58	307	14	265	28
4	16	100	355	28	294	33
4	30	178	415	53	323	39
4	60	348	514	112	349	53
4	100	591	651	201	377	73
4	120	705	*696	236	378	82
4	150	884	*803	298	408	97
4	200	1181	*933	403	406	124
4	240	1430	*1049	497	410	142
4	300	1790	*1236	629	436	171
4	400	2413	*1531	873	436	222

$d$ : dimension of the hypercube.

\*: Indicates the faster balanced-sort cases.

\*\* : Hyperquicksort does not run due to insufficient memory.

Hyper: Hyperquicksort time.

Blncd: Balanced-Sort time.

Q and P: Quicksort and partitioning times of balanced-sort.

GM: Global exchange and merge time of balanced-sort.

with the *BEST* distribution, while it takes only 1823 ms with the *UNIFORM* distribution which is the ideal distribution for hyperquicksort. This load imbalance in hyperquicksort occurs during its merge phase. For example, for the case of  $p = 8$  and  $n = 64\,000$  (not shown in the table), one node finished the sort with 15 000 elements and another node with 1000 elements using hyperquicksort, whereas every node finished the sort exactly with  $n/p = 8000$  elements using balanced-sort. Note that in Tables II and III the missing timing information for hyperquicksort (indicated by \*\*) is because some nodes did not have enough memory to complete the sort due to the uneven distribution of data among the nodes during the merge phase.

## VII. CONCLUSION

Results show that the balanced-sort algorithm is competitive with the other algorithms for randomly distributed data and faster for skewed data distributions that cause load imbalance in the other algorithms. Since each node processes exactly  $n/p$  elements, computational load is well distributed among the nodes which contributes to the speedup of the algorithm. Exact partitioning has the further advantage of most efficiently utilizing the distributed memory.

## ACKNOWLEDGMENT

We would like to thank the anonymous referees for their helpful suggestions.

## REFERENCES

- [1] D. E. Knuth, *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Reading, MA: Addison-Wesley, 1972.
- [2] B. Wagar, "Hyperquicksort," in *Hypercube Multiprocessors 1987*. Philadelphia, PA: SIAM, 1987, pp. 292-299.
- [3] G. Fox, M. Johnson, G. Lyzenga, S. O. J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors, Vol. I*. Englewood, Cliffs, NJ: Prentice-Hall, 1988.
- [4] S. R. Seidel and W. L. George, "Binsorting on hypercubes with d-port communication," in *Proc. Third Conf. Hypercube Concurrent Comput. and Appl.*, Jan. 1988, pp. 1455-1461.
- [5] Y. Won and S. Sahni, "A balanced bin sort for hypercube multicomputers," *J. Supercomput.*, vol. 2, pp. 435-448, 1988.
- [6] S. L. Johnsson, "Combining parallel and sequential sorting on a boolean n-cube," in *Proc. Int. Conf. Parallel Processing*, 1984, pp. 444-448.
- [7] C. G. Plaxton, "Load balancing, selection and sorting on the hypercube," in *Proc. 1989 ACM Symp. Parallel Algorithms and Architectures*, 1989, pp. 64-73.
- [8] K. E. Batchner, "Sorting networks and their applications," in *Proc. AFIPS 1968 SJCC*, 1968, pp. 307-314.
- [9] R. Sedgewick, "Implementing quicksort programs," *Commun. ACM*, vol. 21, pp. 847-856, Oct. 1978.
- [10] G. C. Fox and W. Furmanski, "Optimal communication algorithms on hypercube," Tech. Rep. CCCP-314, California. Inst. of Tech., Pasadena, CA, July 1986.
- [11] S. L. Johnsson and C.-T. Ho, "Spanning graphs for optimum broadcasting and personalized communication in hypercubes," *IEEE Trans. Comput.*, vol. 38, pp. 1249-1268, Sept. 1989.
- [12] Y. Saad and M. H. Schultz, "Data communication in hypercubes," Tech. Rep. YALEU/DCS/RR-428, Dept. Comput. Sci., Yale Univ., New Haven, CT, Oct. 1985.
- [13] Q. F. Stout and B. Wagar, "Passing messages in link-bound hypercubes," in *Hypercube Multiprocessors 1987*. Philadelphia, PA: SIAM, 1987.
- [14] C. Aykanat, F. Ozguner, F. Ercal, and P. Sadayappan, "Iterative algorithms for solution of large sparse systems of linear equations on hypercubes," *IEEE Trans. Comput.*, vol. 37, pp. 1554-1568, Dec. 1988.
- [15] Z. Galil and N. Megiddo, "A fast selection algorithm and the problem of optimum distribution effort," *J. ACM*, vol. 26, pp. 58-64, 1979.
- [16] G. N. Frederickson and D. B. Johnson, "The complexity of selection and ranking in  $X + Y$  and matrices with sorted columns," *J. Comput. and Syst. Sci.*, vol. 24, pp. 197-208, 1982.
- [17] T. Ibaraki and N. Katoh, *Resource Allocation Problems*. Cambridge, MA: M.I.T. Press, 1988.
- [18] S. F. Nugent, "The iPSC/2 direct-connect communications technology," in *Proc. Third Conf. Hypercube Concurrent Comput. and Appl.*, Jan. 1988, pp. 51-60.
- [19] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Comput.*, vol. C-24, pp. 1145-1155, Dec. 1975.
- [20] M. C. Pease, "The indirect binary n-cube microprocessor array," *IEEE Trans. Comput.*, vol. C-26, pp. 458-473, May 1977.
- [21] H. J. Siegel and R. J. McMillen, "The multistage cube: A versatile interconnection network," *IEEE Comput. Mag.*, pp. 65-76, Dec. 1981.



**Bülent Abali** received the B.S.E.E. degree from Middle East Technical University, Ankara, Turkey in 1983, and the M.S. and Ph.D. degrees in electrical engineering from the Ohio State University, Columbus, in 1985 and 1989, respectively.

Since 1989, he has been a research staff member at the IBM T. J. Watson Research Center, Yorktown Heights, NY.



**Füsun Özgüner** (S'74-M'75) received the M.S. degree in electrical engineering from the Istanbul Technical University in 1972, and the Ph.D. degree in electrical engineering from the University of Illinois, Urbana-Champaign, in 1975.

She was with the Design Automation group at the IBM T. J. Watson Research Center for one year and joined the faculty at the Department of Electrical Engineering, Istanbul Technical University in 1976. She spent the summers of 1977 and 1985 at the IBM T. J. Watson Research Center and was a visiting

Assistant Professor at the University of Toronto in 1980. Since January 1981 she has been with the Department of Electrical Engineering, The Ohio State University, where she is presently a Professor. Her research interests include parallel algorithms and architectures, fault-tolerant design and fault simulation techniques.



**Abdulla Bataineh** (S'90-M'92) received the B.S. degree in electrical engineering from Yarmouk University, Irbid, Jordan, in 1987, and the M.S. and Ph.D. degrees in electrical engineering from the Ohio State University, Columbus, in 1988 and 1991, respectively.

He spent the summers of 1989, 1990, and 1991 at Cray Research Inc., Eagan, MN. Currently, he is with Cray Research Inc., Eagan, MN. His research interests are in the area of parallel algorithms and architectures, logic simulation, and high-speed digital circuit and device simulation.