The Journal of Supercomputing 3, 41–61 (1989)

Hypercube-to-Host Sorting*

YOUNGJU WON† and SARTAJ SAHNI
Computer Science Department, University of Minnesota, Minneapolis, MN 55455

(Received January 1988; final version accepted November 1988.)

Abstract. This paper discusses sorting on a hypercube multicomputer, with the assumption that the data to be sorted is initially in the hypercube memory and the sorted data is to reside in the host memory. Three algorithms—heap-and-merge, cyclic merge, and embedded binary tree—are proposed. These are evaluated experimentally on an NCUBE/7 hypercube multicomputer. The cyclic merge algorithm is found to have the best performance.

Key words and phrases. Sorting, hypercube multicomputers

1. Introduction

Recently, several researchers have conducted empirical evaluations of different sorting methods on commercially available hypercube multicomputers. These evaluations have concentrated on an average rather than worst case performance. This empirical research on sorting has contributed significantly to the understanding of algorithm design and programming techniques that lead to efficient hypercube programs. We begin by reviewing some of this important work. Then we suggest additional sorting techniques that may be expected to work well under certain circumstances.

Seidel and Ziegler [1987] assume that the data to be sorted is initially in node 0 of the hypercube and the sorted data is to be in node 0 at the end. They present two bitonic sort algorithms and one quicksort algorithm. The two bitonic sort algorithms differ only in how the data within a processor is sorted. One uses a heap sort and the other a quicksort. The parallel quicksort algorithm begins with the data in node 0. The median of the data keys is found and used to split the data into two halves. The larger half is sent to the neighbor of processor zero along the dimension of the hypercube, d. The two halves are sorted independently using a $d - 1$ dimension hypercube for each. This is done recursively. When the hypercube dimension becomes 0, each processing element uses quicksort to sort the data it has. Following this, the processors send their sorted data back to processor 0.

Wagar [1987] develops a different version of parallel quicksort called hyperquicksort. The data to be sorted is initially in the host processor. The host distributes the data evenly over the $p = 2^d$ nodes of the hypercube. The data elements are split into two sets depending on whether they are less than or equal to a splitting key K or greater than K. The first set of elements is routed to one half of the hypercube and the second set, to the other half. The two half hypercubes sort their data in parallel and recursively. The splitting key K is found by a single processor in each subhypercube by examining only its own data.

*This research was supported, in part, by the National Science Foundation under grants DCR 84-20935 and MIP 86-17374.
†Dr. Youngju Won's current address is P.O. Box 77, Gongneung-Dong Nowo-Gu, Seoul, Korea 139-799.

Hyperquicksort can guarantee to complete the sort only in those cases when almost all of the data will fit into the local memory of a single hypercube processor. This occurs because although the initial distribution of data to the hypercube processors is even, following the first division into two subcubes, it may not be even. In fact, it is possible that all elements in processors 1, 2, ..., $p - 1$ are larger than the splitting key K and so must be accommodated in $(p/2)$ processors. At the next level, it may be necessary to accommodate all these elements in $(p/4)$ processors, and so on.

Another variant of parallel quicksort is proposed in [Felten et al. 1986] who have also developed bitonic sort and shell sort algorithms for hypercubes. Their experiments indicate that bitonic sort is the faster method when the number of elements to be sorted is close to the number of hypercube processors; quicksort and shell sort perform better when the number of elements is considerably larger than the number of processors.

Bin sort was also proposed in [Felten et al. 1986] but was not discussed in detail. Seidel and George [1987] have proposed this method for sorting on hypercubes with d–port communication. Such hypercubes permit near-simultaneous data transfer of up to d nearest neighbors. Experiments reported in [Seidel and George 1987] indicate that parallel bin sort on an FPS T-40 computer is slightly faster than hyperquicksort when $p \geq 8$ and $\frac{n}{p} \geq 512$ (n is the number of elements to be sorted). Won and Sahni [1988] have proposed an improved balanced bin sort.

In this paper, we restrict ourselves to sorting under the assumption that the data to be sorted is initially evenly distributed over the hypercube processor's local memories and the sorted data is to reside in the memory of the host. This case arises when sorting is the last operation and the results are to be output (say, to the disk) via the host or when the next step is better run on the host. The previously mentioned algorithms can be easily modified for this case. This modification simply deletes the initial host-to-hypercube data distribution step (if any) and adds a final step in which the sorted data are sent from the hypercube to the host. We shall explore the possibility of overlapping the data transfer from the hypercube with the sorting performed in the hypercube. We propose three algorithms to accomplish this overlapping: heap-and-merge, cyclic merge, and the embedded binary tree method. The algorithms are described in Sections 2 through 4 and empirically evaluated in Section 5. This evaluation is done on an NCUBE/7 hypercube multicomputer with 64 processors. The features of this hypercube that are important for algorithm design follow:

1. The hypercube nodes are attached to a host computer much the same way as a conventional peripheral device may be attached to a host.
2. Both the host processor and the node processors are capable of performing nonblocking reads from the host and/or from the nodes. As a result, it is possible for the host and the nodes to perform computations and reads in parallel.

2. Heap-and-Merge

In this scheme the host is used to merge sorted data from each of the hypercube nodes. The hypercube nodes first form a min-heap[1] [Horowitz and Sahni 1986] of their data. This allows for easy extraction of the smallest elements. Whenever the host requests data from

PROCEDURE *HeapAndMerge*;
{ host program }
Step 1: Establish 2 buffers, buffer i_a and buffer i_b, for each node i;
Step 2: Enable reading 2 buffers from each node by sending 2 enable signals to each node;
Step 3: [*p way merge*]
merge p buffers, one from each of the p node processors;
Step 4: **If** buffer i_a (buffer i_b) becomes empty during step 3 **then** complete the read of the next buffer from node i into buffer i_b (buffer i_a) and enable the reading of a new buffer from node i into i_a (i_b);
END *HeapAndMerge*;

PROCEDURE *HeapAndMerge*;
{ node program }
Step 1: Form heap structure;
Step 2: Sort next h smallest elements in the heap;
Step 3: **If** there is no enable signal received from the host **then** goto step 2;
Step 4: Send h elements to host consuming one enable signal received from the host;
Step 5: Repeat steps 2 - 4 until no more elements remain;
END *HeapAndMerge*;

Figure 1. Heap-and-merge sort.

a node, the node sends the next h smallest elements. The host uses a loser tree[2] [Horowitz and Sahni 1986] to perform a p way merge of the data from the p processors. In this way it is possible to substantially overlap the processing done by the nodes with the data transmission to the host. The algorithm is described more formally in Figure 1.

Notice that the host allocates two buffers to each processor. This allows the host to gather the next buffer load while the first is being merged. The static buffer allocation may be replaced by a dynamic one, as described in [Horowitz and Sahni 1986] for the case of an external disk sort. The dynamic scheme predicts which processor's buffer loads will be exhausted first and then gets the next buffer from this processor. While this dynamic allocation scheme is very useful in the case of external disk sorting, it is less useful in our environment since the node-to-host data transfer time is less than the time required by the host to consume a buffer load and the node processors can transfer data to the host in parallel. In the case of a disk sort, the reverse is true; the time to consume a buffer load is less than the disk-to-host transfer time and the buffer loads for different runs are transferred serially.

The choice of h affects the performance of heap-and-merge. When the host consumes the buffers uniformly, it takes $O(ph\log p)$ time to exhaust a buffer (Step 4). All p buffers exhaust at almost the same time. A node takes $O(h\log(n/p))$ time to generate a new buffer. The time required for the p nodes to send one buffer each to the host is $O(hp)$. Ideally, these three operations (buffer consumption by the host, buffer generation by the nodes, and buffer transmission from nodes to host) should take about the same time. From the analysis, however, we see that as n and p increase, the buffer consumption and generation times will dominate the transmission time. So, the best performance is obtained by matching

Table 1. Best h value.

Data size	Hypercube dimension					
	1	2	3	4	5	6
512K	—	—	—	—	1K	512
256K	—	—	—	1K	1K	512
128K	—	—	1K	1K	512	256
64K	—	1K	512	512	512	128
32K	2K	512	256	128	128	64
16K	1K	256	128	128	64	32
8K	512	128	128	128	64	32
4K	256	128	128	128	64	32
2K	256	128	64	64	32	16
1K	128	64	32	32	16	16

buffer consumption and generation times. This requires that $ph \log p \simeq ch \log(n/p) + e$ where c and e are constants. From this, we get

$$h \simeq \frac{e}{p \log p - c \log(n/p)}.$$

So as n increases, the optimal value of h increases and as p increases, the optimal h decreases. The optimal values of h may be determined experimentally. Table 1 gives these optimal h values for different values of n (number of data elements) and d (hypercube dimension). In determining these, we tried only values of h that are a power of two. As predicted by our analysis, the optimal h increases with n and decreases with an increase in $p = 2^d$. Since our experiments were limited to h, a power of two, the optimal h determined by these experiments is not a strictly increasing function of n nor a strictly decreasing function of $p = 2^d$.

It is important to note that the heap-and-merge algorithm is very space efficient. Each hypercube node needs memory only for its (n/p) elements.

3. Cyclic Merge Sort

While the heap-and-merge scheme of the preceding section has the virtue of overlapping communications (between the host and hypercube nodes) with internal hypercube and host processing, as p becomes large we expect the internal host processing to dominate the run time. This is evident from the following observations.

1. The total communication time is proportional to n.
2. The CPU time required by each node processor is proportional to $(n/p)\log(n/p)$.
3. The CPU time required by the host is proportional to $n \log p$.

So the heap-and-merge scheme is most effective when the host is quite a bit faster than an individual node. This is not the case in the NCUBE hypercube. The merging load on

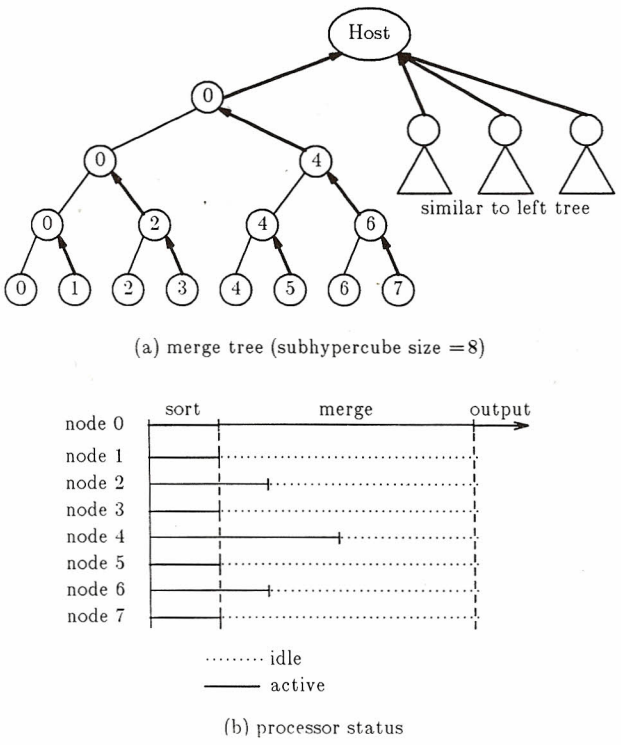


Figure 2. Collapsing merge within each subhypercube.

the host can be reduced by having the hypercube nodes do some of the merging. One possibility is to divide the hypercube into q subhypercubes of size (p/q) each. The processors of each subhypercube merge their data using a merge tree (see Figure 2a), and the root of the merge tree of each subhypercube sends its results to the host which performs a q-way merge. One difficulty with this scheme is that the node memory required is that for approximately $n/(2q)$ elements since the root node is involved in a merge at each level. To see this, note that for the highest level merge performed by node 0 (Figure 2a), node 0 begins with all the elements merged by nodes 0 through 3 and one buffer load from node 4. This is a total of $n/(2q) + b$ elements where b is the number of elements in a buffer load. Following this, merged buffers can be output to the host as they are formed and new buffer loads read from node 4 as needed. Since the read requirements from node 4 cannot outpace the output rate to the host, the memory requirements do not increase.

By having a node perform, concurrently, all the merges it is involved in, the memory requirements can be reduced to $O(n/p + b\log(p/q))$. In a concurrent merge, it is necessary for a node to retain only one buffer load from each of its right children. Thus, in Figure 2a, node 0 needs memory for a buffer load from nodes 4, 2, and 1 plus memory for its initial n/p data elements.

Figure 2b shows the processor status during a sort and merge for the case $q = 1$, which will be the only case considered further in this section. The diagram assumes that the sort and merge is divided into two phases: each processor sorts its data, and the data is merged using the tree shown in Figure 2a.

All processors are busy during the initial sort phase. Following this, processors 1, 3, 5, and 7 are required to send their data to processors 0, 2, 4, and 6, respectively, for merging. These odd-numbered processors remain idle for the entire duration of phase 2. Following the merge at this level, processors 2 and 6 become idle. Following the merge at the next level, processor 4 becomes idle. The processor idle time can be reduced by using several merge trees simultaneously. In fact, we propose using p merge trees in a p processor hypercube. Figure 3 shows these trees for the case $p = 4$. The bold arrows show the direction of data transfer. Each of the p trees has a different root. Our strategy is to have each root accumulate the elements in a particular range. Thus, if $b_0, b_1, \ldots, b_{p-1}$ are p distinct keys, then node i accumulates data elements in the range $(b_{i-1}, b_i]$[3] ($b_{-1} = -\infty$). We shall refer to each of these ranges as a bin. Processor i accumulates all elements with keys in the range $(b_{i-1}, b_i]$. That is, it accumulates elements in bin i, $0 \le i < p$. This is done by merging the bin i elements from the p processors, using the merge tree of which it is the root. So, when $p = 4$, processor 0 uses the first merge tree in Figure 3; processor 1 uses the second one; processor 2, the third; and processor 3, the fourth merge tree. The sorting algorithm calls for each node to repeatedly generate up to h elements for each of the p bins. The elements so generated for each bin are called a bucket. h is the bucket size. Let B_j^i be the bucket generated by processor i for bin j. Each processor routes its buckets to the next level of processors for the merge tree that is accumulating the respective bins. So, when $p = 4$, B_0^0, B_1^0, B_2^0, B_3^0 B_0^1, B_1^1, B_2^1, B_3^1 B_0^2, B_1^2, B_2^2, B_3^2 B_0^3, B_1^3, B_2^3, and B_3^3, are, respectively, routed to processors, 0, 1, 0, 1, 0, 1, 0, 1, 2, 3, 2, 3, 2, 3, 2, and 3. Processor 0 merges B_0^0 and B_0^1. It also merges B_2^0 and B_2^1. The result of the former is retained while the result of the latter is sent to processor 2. While processor 0 is doing this, processor 2 merges B_0^2 and B_0^3 and also B_2^2 and B_2^3. The result of the former merge is sent to processor 0 while the result of the latter is retained for later merging. Let $B_j^i \cup B_l^k$ denote the merge of buckets B_j^i and B_l^k. Then, when $p = 4$, the root level merge for processor 0 involves merging $B_0^0 \cup B_0^1$ and $B_0^2 \cup B_0^3$; for processor 1, it involves merging $B_1^0 \cup B_1^1$ and $B_1^2 \cup B_1^3$; for processor 2, it involves merging $B_2^0 \cup B_2^1$ and $B_2^2 \cup B_2^3$; and for processor 3, it involves merging $B_3^0 \cup B_3^1$ and $B_3^2 \cup B_3^3$. When the root level merges are complete, each processor has a sorted list of the elements in its bin. These sorted lists are then transmitted to the host. Following this, the host has an initial portion of each of the p bins. The entire process is repeated

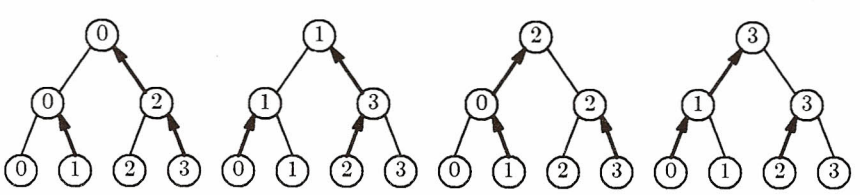


Figure 3. Four merge trees.

several times until all elements in the range b_{-1} to b_p have been merged. If x is the maximum number of elements initially in a bin range $(b_{i-1}, b_i]$ in any node, then $\lceil x/h \rceil$ repetitions are needed.

The p merge trees may be obtained in a fairly straightforward manner. Begin with one forest having p nodes $0, \ldots, p-1$ laid out left to right. Now transform this into two forests by adding a level with $(p/2)$ nodes. In one forest, these $(p/2)$ nodes are labeled using the left child indices; in the other, the right child indices are used. An edge is drawn between each parent and its two children. Each of these forests is transformed into two by adding $(p/4)$ nodes at the next level. In one of these, the nodes use the left child indices and in the other, the right child indices are used. This process is repeated until each forest is a tree. Note that when this construction is used, a parent is a hypercube neighbor of each of its children in the merge tree.

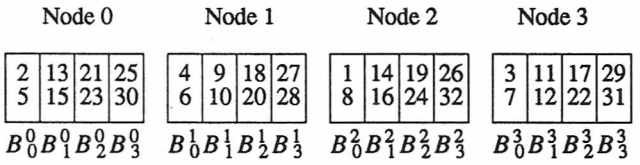
The sorting algorithm is given in Figure 4. An example for the case $p = 4$ is given in Figure 5. While we haven't specified how the bin boundaries are to be chosen, this is crucial to the success of the method. The boundaries should result in a near uniform distribution of the n elements into the p bins. Assume that for the example, the boundaries $(b_0, b_1, b_2, b_3) = (8, 16, 24, 32)$ are chosen. Also assume that the bucket size h is 2 (in practice, a much larger bucket size is used). Each node generates four buckets, one for each bin. The buckets are routed to the nodes that will merge them and the first level of merges performed. The merged buckets are then routed for the root level merging and finally to the host. Then the cycle is begun again. Figure 6 gives the processor status graph for cyclic merge.

To complete the description of cyclic merge, we need to describe how the p boundary values are determined. Ideally, these will not only partition the n elements into p equal size bins but will also partition the n/p elements in each node into p equal size bins. The

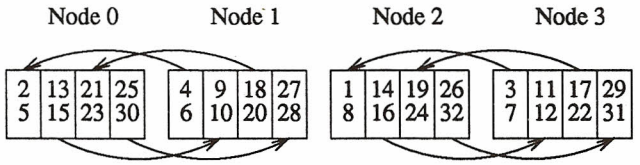
```
PROCEDURE CyclicMergeSort;
{ Bucket boundaries are determined by the host processor using random sampling over the node
    processors }
Step 1:  Form the next 2^d = p buckets in increasing order;
Step 2:  For i=1 to d-1 do
             Send buckets to processors that will perform the next level of merges for the
             individual bins;
             Receive buckets for this processor's next level merges;
             Perform these pairwise bucket merges to create buckets for the next level of
             merges;
         endfor;
Step 3:  Output merged list to host;
Step 4:  Repeat Steps 1 - 3 until no elements remain in the hypercube;
END CyclicMergeSort;
```

Figure 4. Node program for cyclic merge sort.

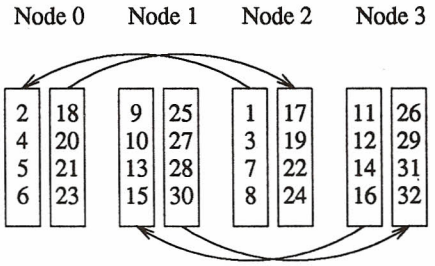
(1) Decide boundaries of buckets (8, 16, 24, 32).
(2) Generate 4 buckets from the heap.



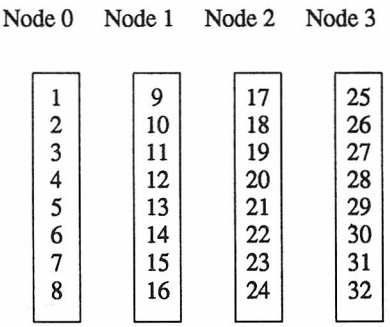
(3) Exchange and merge at level 1 of tree in Figure 3.



(4) Exchange and merge at level 2 of tree in Figure 3.



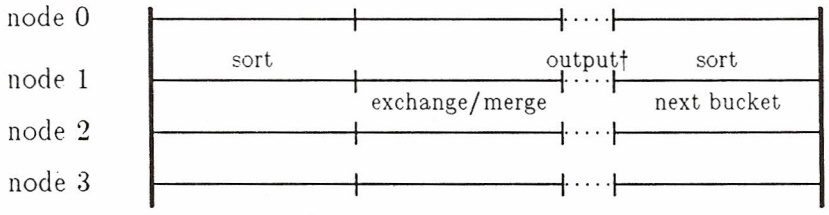
(5) Merge and ouput sorted list.



(6) Repeat (1) - (5) for next set of buckets.

Figure 5. Cyclic merge sort.

first condition ensures that each merge tree has the same load; the second ensures that no node becomes a bottleneck for bin transmission. (As remarked earlier, the number of times steps 1–3 of Figure 4 are repeated is a function of the maximum number of elements in any bin range in any node initially.) This ideal is seldom realizable. One way to approximate



† Time for moving bucket into buffer (not including time to transfer to host).

Figure 6. Status of nodes (cyclic merge sort).

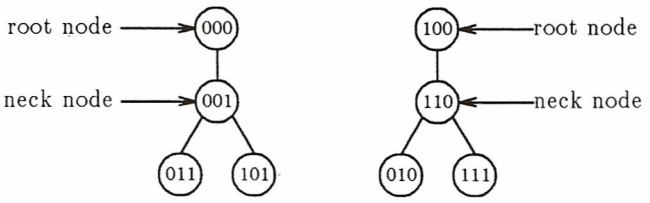
the ideal is to perform the sort in k stages. In the first stage, elements in the range $(-\infty, B_1]$ are merged using p bin boundaries; in the second, elements in the range $(B_1, B_2]$ are merged using p bin boundaries in this range; and so on, until the k-th stage, where elements in the range $(B_{k-1}, \infty]$ are merged using p bin boundaries. Using a k greater than 1 tends to balance the bin sizes in each stage. Using too large a k makes the per stage work too little and the overheads dominate the run time. The optimal k to use may be determined experimentally. For k stages, kp boundary points that partition the n elements into approximately equal parts are needed.

The boundaries in our experiments are computed by the host processor using random samples from each of the node processors. While the node processors are constructing their heaps, the host sorts the random samples and determines the boundary values that divide the sorted random samples into kp sublists of equal size. Thus the first p boundary values are used in the first stage of the sort; the next p values are used for the second stage, and so on. Preliminary experiments indicated that $k = 4$ generally resulted in the best performance. So we used this value of k in the final experiments. The size of the random sample list produced in each node was $\min\{p, n/p^2\}$ if $n/p > p$ and $n/4p$ if $n/p \leq p$.

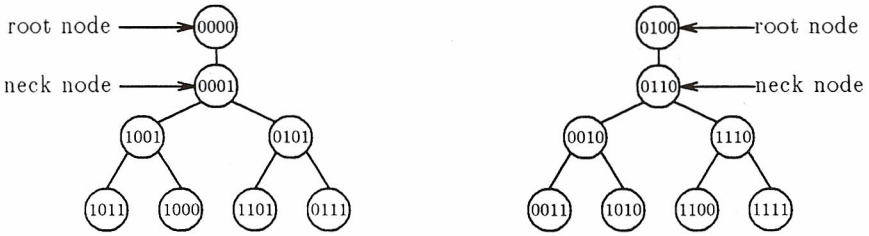
The above method to compute the bucket boundaries is very similar to that proposed by Felten et al. [1986] to compute splitting keys for parallel quicksort. While their computation was done on the hypercube, ours is done on the host since this allows us to overlap the computation of the boundaries with the heap formation being done in the nodes. We note that the cyclic merge method is reminiscent of the recursive doubling scheme used for histogramming [Siegel et al. 1981].

4. Embedded Binary Tree

An alternative to cyclic merge is to merge using a binary tree embedded in a hypercube. Such trees [Deshpande and Jenevein 1986] for the case of three- and four-dimensional subhypercubes are shown in Figure 7. Each intermediate and neck node performs a three-way merge of its own data and that received from its two children. Each root node performs a two-way merge since it has only one child. The $2q$ roots (two for each subhypercube) transmit their results to the host which performs a $2q$-way merge. The algorithm for the case $q = 1$ is given in Figure 8. The best q value to use for different hypercube dimensions



(a) Two binary trees in a three-dimensional subhypercube



(b) Two binary trees in a four-dimensional subhypercube

Figure 7. Embedded binary tree.

and sizes of data is shown in Table 2. By using embedded binary trees rather than merge trees, we eliminate the need for extra memory. Also, the hypercube merging is efficiently pipelined. The size of the bucket, denoted as b in Figure 8, was min $\{n/p^2, n/rp\}$ where r varies from 4 to 16, depending on the dimension of the hypercube and the size of data n. A small bucket size minimizes the idle time between the time a leaf node finishes its work to the time the root finishes. A small bucket size also increases the node-to-node communication time as each bucket transfer incurs a transmission startup penalty. Generally, a bucket size of n/p^2 was found to work well. In some cases, we found performance improvement by going to a smaller bucket size $n/(rp)$ for r between 4 and 16. This can be smaller than n/p^2 only when $p<16$. The smaller bucket size was useful for smaller p's as the increased number of buckets tended to offset the pipeline delay. (The embedded binary tree height is log$p$ while the number of buckets generated by leaves is $n/(bp)$; for $b = n/p^2$ the number of buckets is p, which isn't much bigger than log$p$ when p is small.) Also, when n is large, it is possible to use a bucket size smaller than n/p^2 and still have reasonably large buckets. We accomplish this by choosing $r>p$.

5. Experimental Results

First, we determined the best values of q to use for the embedded binary trees method. These are shown in Table 2 for different values of n (number of records to be sorted) and d (hypercube dimension). As can be seen, the best q values are small. In fact, the optimal

PROCEDURE *EmbeddedBinaryMerge*;
{ two ($p/2$) node binary trees are constructed at host }

Step 1: Receive information of parent-child relation from host;
NodeRole = {*RootNode*, *InterNode*, *LeafNode*};

Step 2: Each node constructs a heap;

Step 3: **Case** *NodeRole* **of**

`*LeafNode*´:

3.1a: Repeat Steps 3.2a - 3.3a until no element left in this node;

3.2a: **If** heap sort is not done **then** continue to sort next b elements;

3.3a: **If** there is a request from *parent* node **then** send the last b elements in the heap to *parent*;

`*RootNode*´:

3.1b: Repeat Steps 3.2b through 3.5b until no element is left at child and itself;

3.2b: Perform two-way merge between list received from the child (*necknode*) and its own;

3.3b: **If** it is required to generate next element in its own heap during Step 3.2b **then** generate next b elements from the heap;

3.4b: **If** none of the elements received from child during Step 3.2b are left **then** request next b elements from child; **If** next elements are already in buffer **then** continue Step 3.2b after reading b elements **else** continue to sort next b elements from its own heap if any.

3.5b: **If** buffer for output is full during Step 3.2b **then** wait until request arrives from host and send buffer to host;

`*InterNode*´:

Similar to Steps 3.1b through 3.5b except that this node performs a three-way merge of the lists from its two children and its own;

endcase;

END *EmbeddedBinaryMerge*;

Figure 8. Using embedded binary trees.

q is generally 1 or 2. For larger q values, the host merging becomes the bottleneck. As indicated in Section 3, for cyclic merge, best performance is expected when $4p$ bin boundaries are used. This requires us to run procedure *CyclicMergeSort* four times. Having determined the q values to use, we next experimented with the three methods: heap-and-merge, cyclic merge, and embedded binary tree. Run times for different values of n and d are given in Tables 3–5. In addition, run times using hyperquicksort and bitonic sort followed by a data transfer to the host are given in Tables 6 and 7. All times are in milliseconds and are the average for ten data sets. Figure 9 is a plot of the run time for $n = 16K$, and Figures 10–15 plot the run time data for different numbers of processors. For a one-dimensional hypercube, the heap-and-merge method is best. For two-dimensional hypercubes, the binary embedded

Table 2. Best q value for using embedded binary trees.

Data size	Hypercube dimension					
	1	2	3	4	5	6
1K	1	2	2	2	3	3
2K	1	2	2	2	2	2
4K	1	1	1	2	2	2
8K	1	1	1	1	2	2
16K	1	1	1	1	2	2
32K	—	1	1	1	2	2
64K	—	—	1	1	2	2
128K	—	—	—	1	2	2
256K	—	—	—	—	2	2
512K	—	—	—	—	—	2

Table 3. Heap-and-merge method.

Data size	Hypercube dimension					
	1	2	3	4	5	6
1K	86	53	42	37	27	25
2K	182	109	85	69	56	50
4K	394	247	154	136	101	92
8K	850	512	344	233	214	217
16K	1799	1033	753	482	404	421
32K	—	1954	1443	925	792	730
64K	—	—	3018	2141	1512	1251
128K	—	—	—	4024	3209	2110
256K	—	—	—	—	5819	4288
512K	—	—	—	—	—	8439

Table 4. Cyclic merge.

Data size	Hypercube dimension					
	1	2	3	4	5	6
1K	104	58	37	25	20	15
2K	214	116	74	51	42	39
4K	451	264	137	104	81	73
8K	953	529	317	172	157	136
16K	2181	1098	684	406	314	271
32K	—	2277	1376	815	602	481
64K	—	—	2792	1718	1195	822
128K	—	—	—	3591	2147	1515
256K	—	—	—	—	4314	3041
512K	—	—	—	—	—	4979

Table 5. Using embedded binary trees.

Data size	Hypercube dimension					
	1	2	3	4	5	6
1K	88	53	39	27	21	16
2K	182	106	73	54	45	40
4K	393	225	139	110	86	79
8K	862	515	333	182	162	139
16K	1794	1033	694	423	339	313
32K	—	1928	1379	833	654	542
64K	—	—	2886	1798	1275	888
128K	—	—	—	3695	2214	1621
256K	—	—	—	—	4475	3213
512K	—	—	—	—	—	6119

Table 6. Hyperquicksort and sequential output to host.

Data size	Hypercube dimension					
	1	2	3	4	5	6
1K	91	60	38	27	22	19
2K	199	121	78	56	45	41
4K	415	275	144	117	88	84
8K	883	538	343	197	166	151
16K	1869	1131	719	421	351	309
32K	—	2338	1474	914	684	497
64K	—	—	2911	1895	1307	847
128K	—	—	—	3733	2310	1647
256K	—	—	—	—	4715	3229
512K	—	—	—	—	—	6447

Table 7. Bitonic sort and sequential output to host.

Data size	Hypercube dimension					
	1	2	3	4	5	6
1K	94	61	44	36	27	23
2K	203	127	85	72	58	46
4K	430	278	178	143	107	96
8K	905	572	375	244	227	215
16K	1930	1252	791	497	433	389
32K	—	2424	1651	1029	795	626
64K	—	—	3322	2194	1511	1059
128K	—	—	—	4561	3187	2150
256K	—	—	—	—	5924	4211
512K	—	—	—	—	—	8314

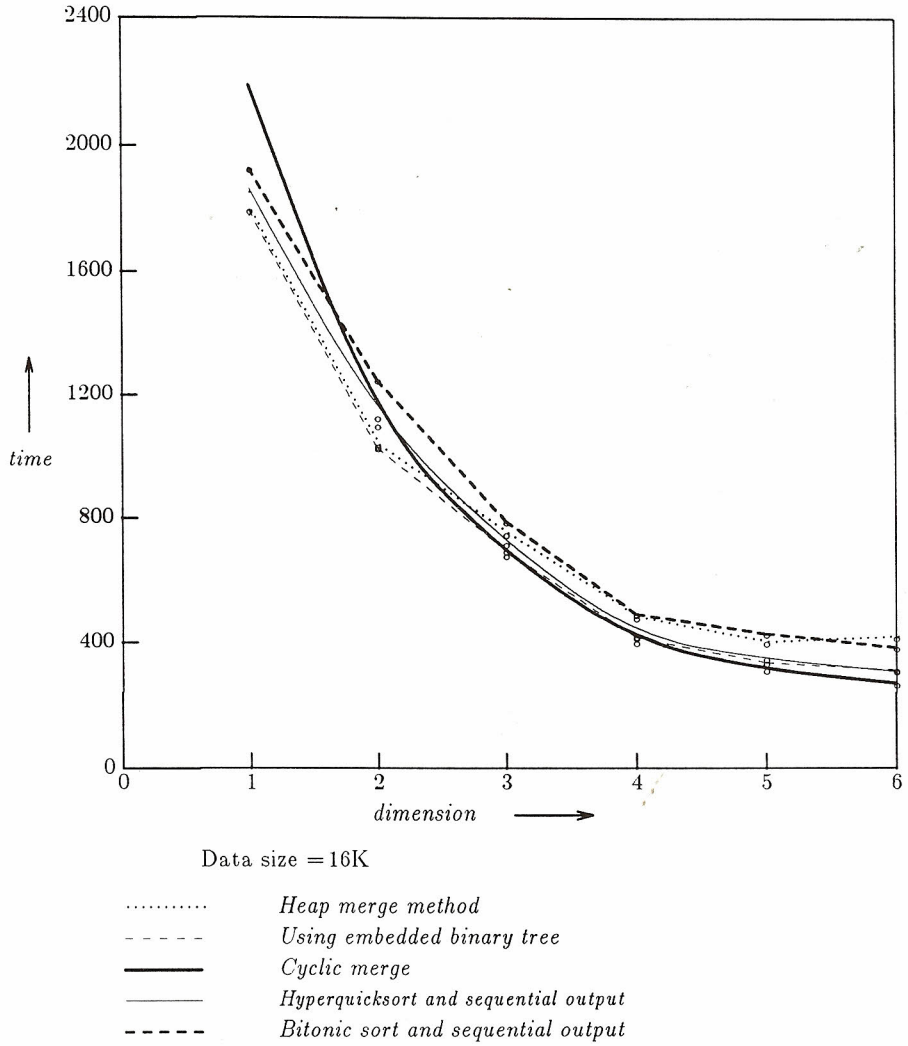


Figure 9. Performance of different methods.

tree method and the heap-and-merge method have the best run time. The cyclic merge scheme is best for larger hypercubes. Hyperquicksort takes 7 to 10% more time than does cyclic merge on hypercubes with $d \geq 3$. It takes about 20% more time than the embedded binary tree method when $d = 2$ and about 4% more time than heap-and-merge when $d = 1$. Bitonic sort was consistently slower than hyperquicksort. The times for bitonic sort and hyperquicksort were recomputed by us as those given in [Seidel and Zeigler 1987]; [Seidel and Wagar 1987] used different data sets and did not account for the time needed to transfer the sorted elements to the host.

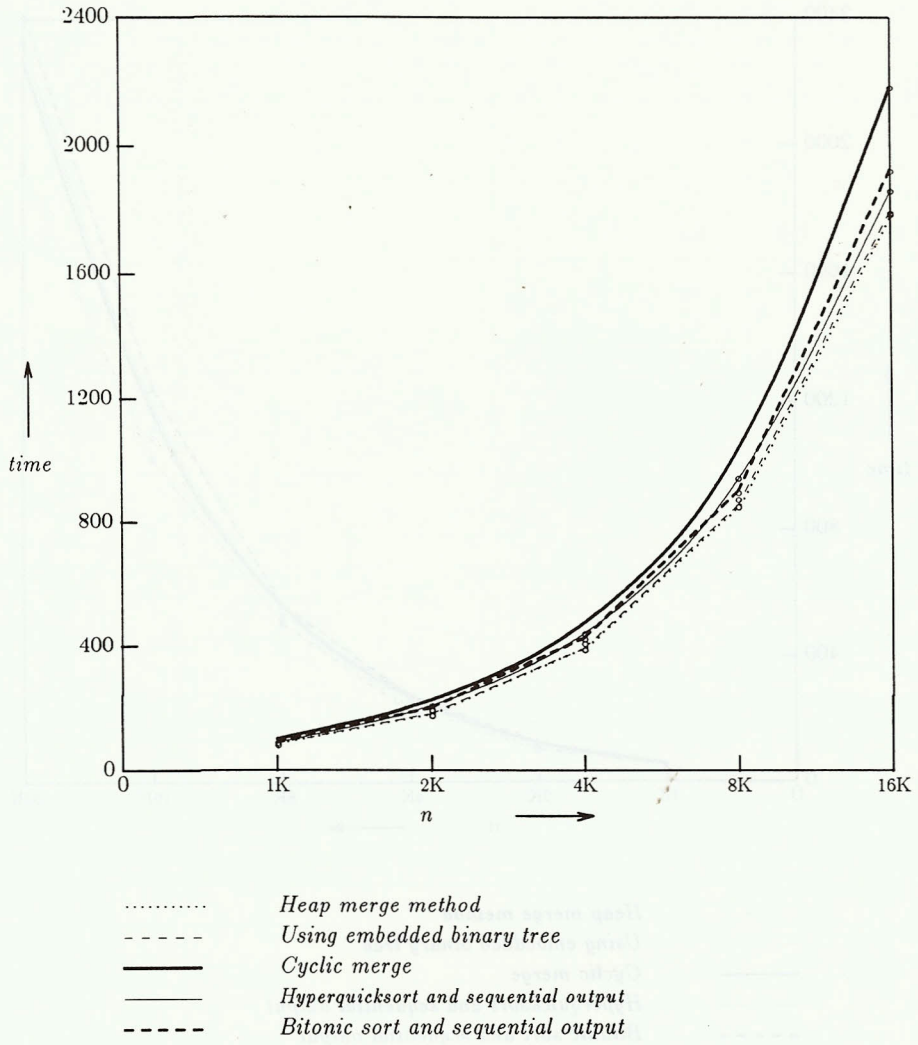


Figure 10. Performance of different methods when $p = 2$.

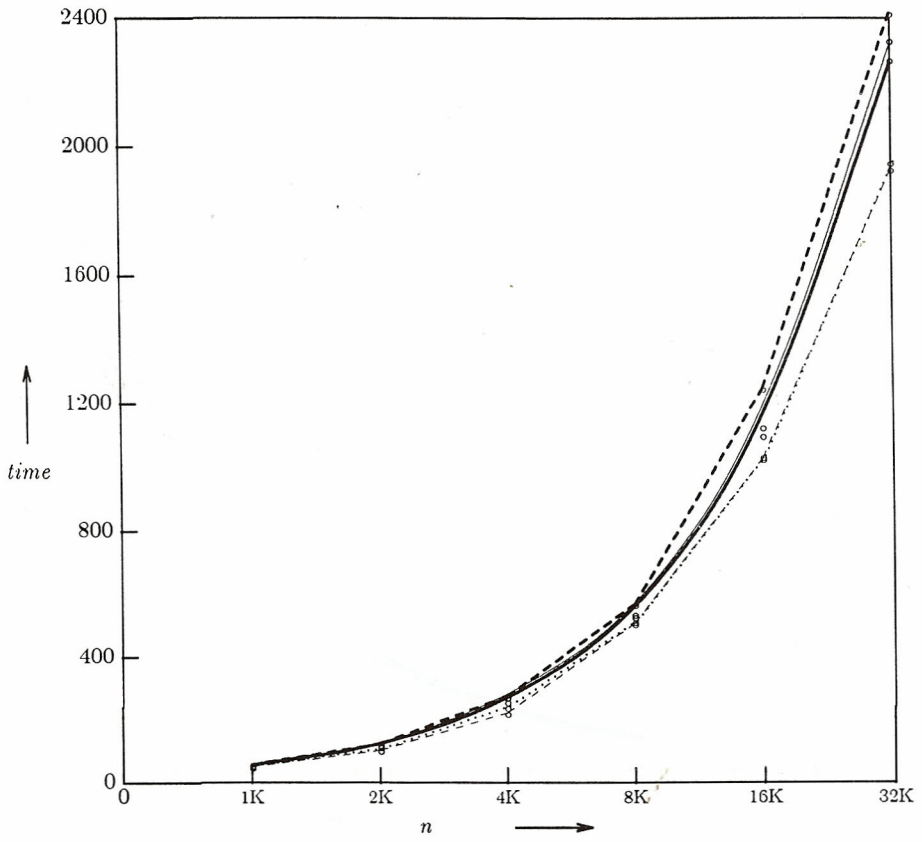


Figure 11. Performance of different methods when $p = 4$.

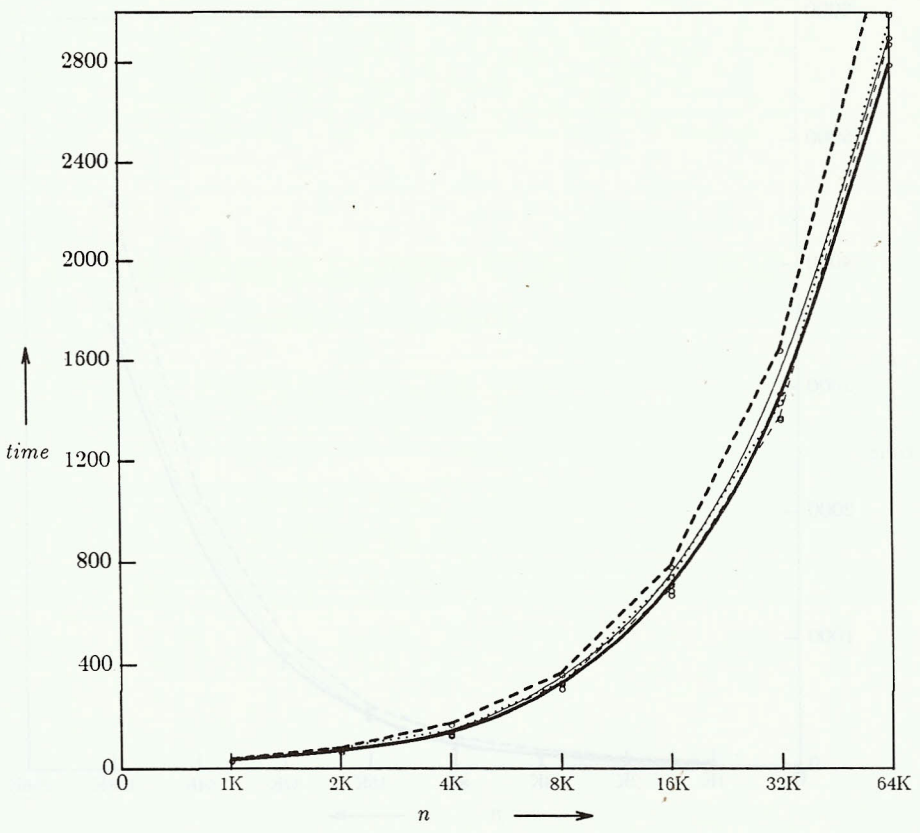


Figure 12. Performance of different methods when $p = 8$.

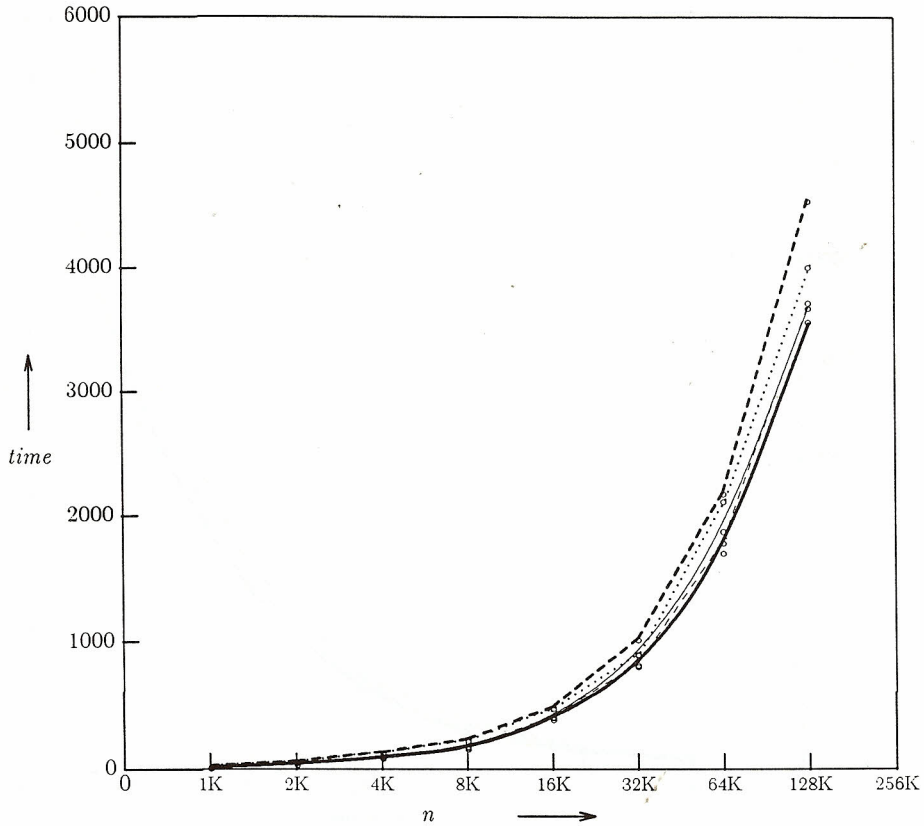


Figure 13. Performance of different methods when $p = 16$.

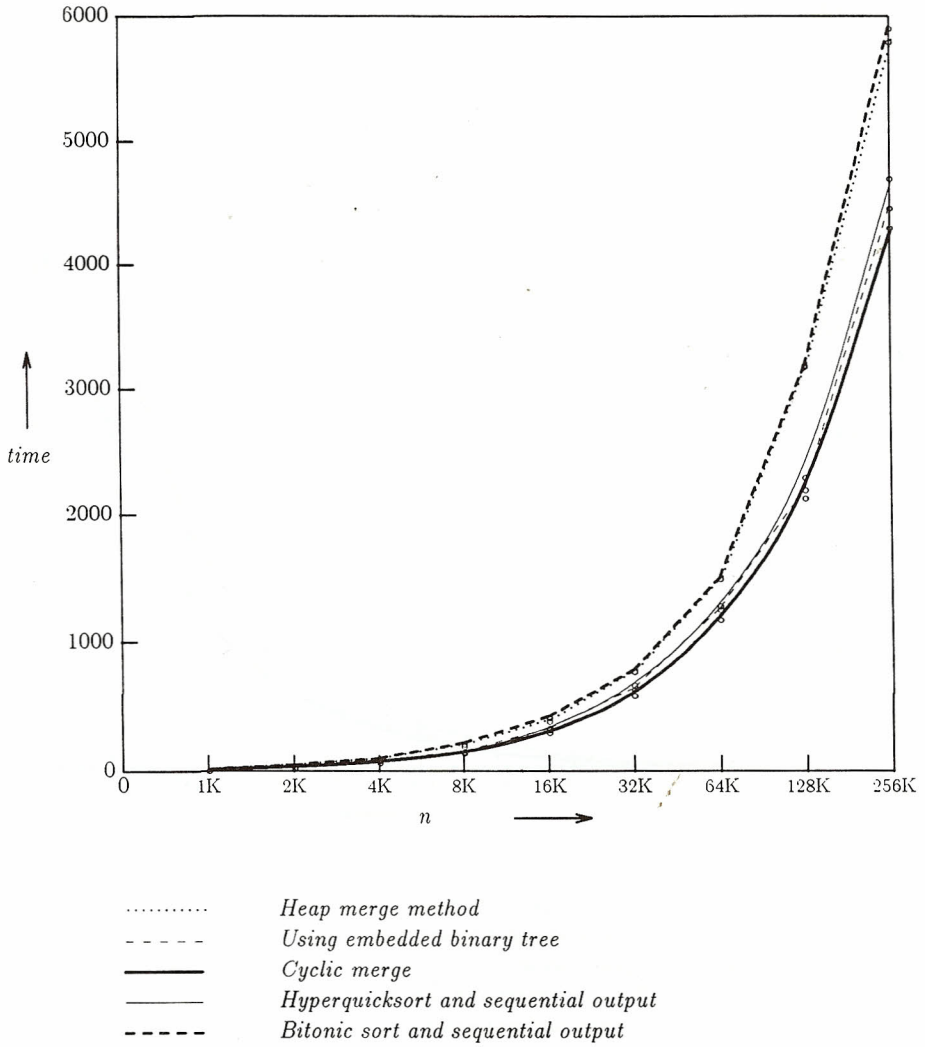


Figure 14. Performance of different methods when $p = 32$.

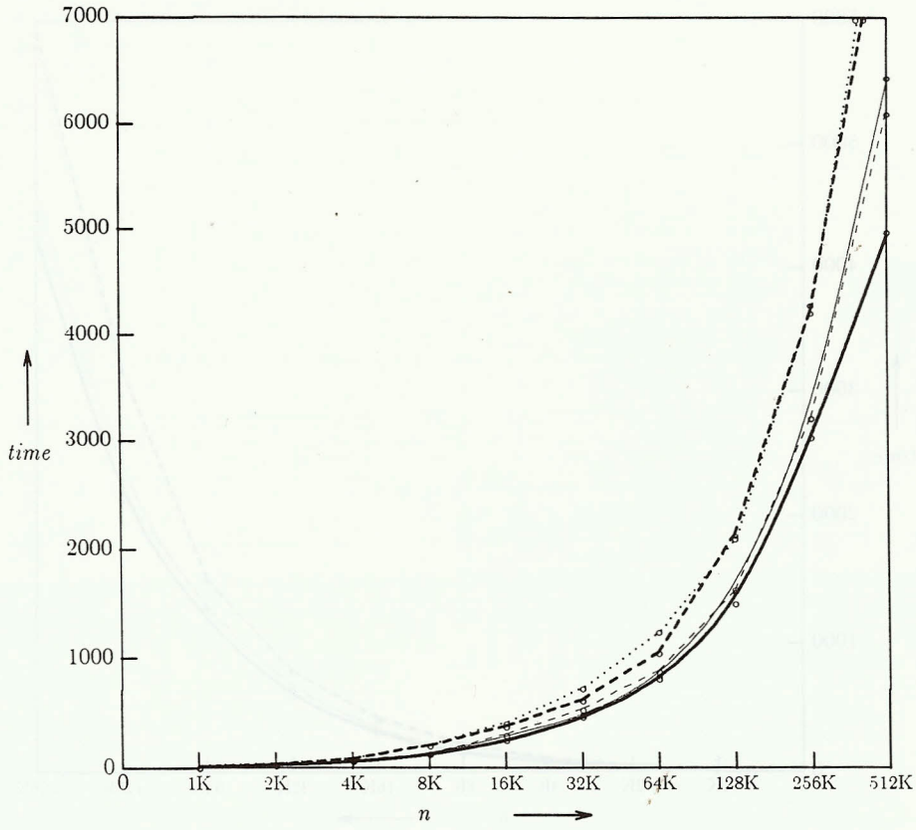


Figure 15. Performance of different methods when $p = 64$.

6. Conclusions

Three methods—heap-and-merge, cyclic merge, and the embedded binary tree method—have been proposed for hypercube-to-host sorting. Experimentation on the NCUBE hypercube shows that for small hypercubes ($d = 1$ and 2, or $p = 2$ and 4) heap-and-merge and the embedded binary tree method are very competitive and both are slightly superior to cyclic merge. For $d > 2$ ($p > 4$), cyclic merge outperforms the other two methods. Heap-and-merge is the slowest method when $p > 4$. This is because the host becomes a bottleneck.

References

Deshpande, S.R., and Jenevein, R.M. 1986. Scalability of a binary tree on a hypercube. In *Proceedings of the 1986 IEEE Intl. Conf. on Parallel Processing*, pp. 661–668.

Felten, E., Karlin, S., and Otto, S. 1986. Sorting on a hypercube. Caltech/JPL, Hm 244.

Horowitz, E., and Sahni, S. 1986. *Fundamentals of Data Structures in Pascal*. Computer Science Press.

Seidel, S.R., and Ziegler, L.R. 1987. Sorting on hypercubes. In *Proc. of the 2nd Conf. on Hypercube Multiprocessors* (Knoxville, Ken., Sept. 1986), SIAM.

Seidel, S.R., and George, W.L. 1987. A sorting algorithm for hypercubes with d-port communication. *Tech. Report*, Dept. of Mathematical and Computer Science, Michigan Technological University.

Siegel, J.S., Siegel, H.J., Kemmerer, F.C., Mueller, P.T., Smalley, H.E., and Smith, S.D. 1981. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers*, C-30, 12, (Dec.), pp. 934–947.

Wagar, B. 1987. Hyperquicksort—A fast sorting algorithm for hypercubes. In *Proc. of the 2nd Conf. on Hypercube Multiprocessors* (Knoxville, Ken., Sept. 1986), SIAM.

Won, Y., and Sahni, S. 1988. Balanced bin sort on hypercube multicomputers. *Tech. Report*, Department of Computer Science, University of Minnesota. (Dec.).

Notes

[1]A min-heap is a complete binary tree in which the value in each node is the smallest of the values in the subtree of which it is the root. Since it is a complete binary tree, a min-heap with m nodes has height $O(\log m)$. The minimum item can be extracted in this much time.

[2]A loser tree is a complete binary tree in which the leaf nodes represent players in a tournament. A tournament is played at each nonleaf node between the players who won the tournaments at its children nodes (in case these are also nonleaf nodes) or between the players at its children (in case these are leaf nodes). Each nonleaf node records the loser of the tournament played at this node. In our case, a smaller element wins a tournament against a larger element. Ties are broken arbitrarily. Since a loser tree with p players has p leaves and is a complete binary tree, its height is $O(\log p)$. Extracting the smallest element and inserting new elements take $O(\log p)$ time per operation.

[3]The interval $(a, b]$ consists of all elements with a value greater than a but less than or equal to b.