The Journal of Supercomputing, 2, 435–448 (1988)

A Balanced Bin Sort for Hypercube Multicomputers*

YOUNGJU WON† and SARTAJ SAHNI
University of Minnesota

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

Abstract. We propose a balanced bin sort for hypercube multicomputers. This sorting algorithm has an empirically measured expected run time that is greater than that of hyperquicksort but less than that of bitonic sort. Also, its space requirements are less than that of hyperquicksort but more than that of bitonic sort. So, it is useful in situations in which there is some excess memory but not enough to run hyperquicksort.

1. Introduction

Sorting is a fundamental operation that arises in many applications. While several researchers have studied sorting on hypothetical parallel computers, only a limited amount of work has been published on commercially available parallel computers. Most of the work using theoretical models has been for synchronous SIMD parallel computers. Further, this work has generally assumed the internode communication cost to be comparable to that of a basic arithmetic operation and has also assumed the availability of as many computation nodes as can be gainfully used. In particular, the number of nodes required by most of the proposed sorting algorithms for theoretical models increases as the number of elements to be sorted increases. Commercially available parallel computers, on the other hand, tend to be asynchronous MIMD computers; have a limited number of nodes; and have an internode communication cost that is significantly higher than that of a basic arithmetic operation.

Lakshmivarahan et al. [1984] reviews the sorting literature as it relates to abstract models of parallel computation. Cole [1987] develops an O(log n) merge sort algorithm to sort n elements on an n PE (processing element) computer. Felten et al. [1986], Seidel and Ziegler [1987], Seidel and George [1987], and Wagar [1987] consider sorting on existing hypercube computers.

Seidel and Ziegler [1987] assume that node 0 of the hypercube contains the data to be sorted initially and that the sorted data resides in this node at the end. They present two bitonic sort algorithms and one quicksort algorithm. The two bitonic sort

* 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, Seuol, Korea 139–799.

algorithms differ only in how the data within a processor are sorted. One uses a heap sort and the other uses a quicksort. The parallel quicksort algorithm begins with 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 independenly using a d–1 dimension hypercube. This is done recursively. When the hypercube dimension becomes 0, each PE uses quicksort to sort its data; then the processors send their sorted data back to processor 0.

Wagar [1987] develops a different version of parallel quicksort. This algorithm, called hyperquicksort, assumes that the data to be sorted are initially in the host processor. The host initially distributes the data evenly over the $p = 2^d$ nodes of the hypercube. Node 0 of the hypercube computes its median key K and broadcasts it to the remaining processors. The data elements are split into two according to whether they are less than or equal to the splitting key K or greater than it. 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.

While hyperquicksort has the least average running time of the proposed hypercube sort algorithms, it can guarantee completion of the sort only when almost all the data fits into the local memory of a single hypercube processor. This drawback exists because even though the initial distribution of data to the hypercube processors is even, the subsequent division into two subcubes may not be. 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, etc. Another shortcoming of hyperquicksort is that it does not leave the sorted elements evenly distributed across the p processors. This is a direct consequence of the first shortcoming and is significant only when the data are to be left in the hypercube for further processing. Note that bitonic sort does not suffer from either of these problems.

Another variant of parallel quicksort is proposed in [Felten et al. 1986]; however, we shall not discuss it here since its performance is slightly inferior to that of hyperquicksort. Felten et al. [1986] 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, and that quicksort and shell sort perform better than bitonic sort when the number of elements is considerably larger than the number of processors.

Bin sort was also proposed in [Felten et al. 1986] but 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 transfers to up to d nearest neighbors. The parallel bin sort algorithm of [Seidel and George 1987], called *min-max bin sort*, is reproduced in Figure 1. This method has the same drawbacks as hyperquicksort. Furthermore, as noted in [Seidel and George 1987], the expected storage requirements of parallel bin sort are higher than those of hyperquicksort. Experiments reported in [Seidel and George 1987] indicate that parallel bin sort on an

1. Each node finds its minimum and maximum keys.
2. Each node sends its minimum and maximum keys to node 0.
3. Node 0 determines the global minimum and maximum keys and uses them to compute $p-1$ splitting keys.
4. Node 0 broadcasts the $p-1$ splitting keys to all other nodes.
5. Each node uses the $p-1$ splitting keys it received in Step 4 to partition its subsequence into p bins of approximately equal length.
6. Each node i sends bin j to node j and receives bin i from node j, $(0 \leq i,j < p, i \neq j)$.
7. Each node quicksorts the subsequence it contains.

Figure 1. Parallel bin sort (reproduced from Seidel and George [1987]).

FPS T–40 computer is slightly faster than hyperquicksort when $p \geq 8$ and $n/p \geq 512$.

In this paper we propose modifications to min-max bin sort that improve its performance. The resulting sorting algorithm, called *balanced bin sort*, has better memory requirements. Experimental results obtained on an NCUBE hypercube indicate that min-max bin sort and hyperquicksort require up to 30% more memory per node than does balanced bin sort. Further, the run time of balanced bin sort is comparable to that of min-max bin sort and only slightly greater (approximately 20%) than that of hyperquicksort. Balanced bin sort is faster than bitonic sort but requires more memory.

2. Balanced Bin Sort

We consider some modifications of the parallel bin sort algorithm (Figure 1), which affect the manner in which the $p - 1$ bin splitting keys are computed and also the manner in which each node routes its bins to their destination processors. As in the case of Figure 1, we assume that the data to be sorted are initially evenly distributed over the p node processors.

To compute the $p - 1$ bin splitting keys, we use the algorithm shown in Figure 2. First, the nodes use quicksort to sort their elements independently. Next, each node selects a set of $p - 1$ splitting keys that will evenly divide its elements into p bins. This can be done in $O(p)$ time since the elements are already in sorted order. Next, the p splitting key lists of the p processors are merged pairwise using a standard binary tree scheme. See Figure 3 for the case $p = 8$. The merging is done by levels from the leaves to the root.

Each right child sends its current list to its parent (which is also its left sibling). The parent merges its current splitting list with the one received from its right child. During the merge, the odd position keys of the result are retained while those in even positions are discarded. The merge at each level can be done in $O(p)$ time. The total merge time is therefore $O(p \log p)$. The final splitting list at the root node 0 becomes

PROCEDURE *SplittingKeys;*

Step 1: Each node sorts its (n/p) elements using quicksort.

Step 2: Each node selects $p-1$ keys from its (n/p) elements. These correspond to the $(\frac{n}{p^2}) \times i$-th elements, $1 \leq i < p$, in the sorted list of step 1. These form the local splitting list S of the node processor.

Step 3: The local splitting lists of the p processors are merged using a binary merge tree. During each merge, two splitting lists of size $p-1$ are merged. Only the keys in the odd positions of the result are retained. Hence the merged list has exactly $p-1$ keys also.

Step 4: Node 0 broadcasts the final list of $p-1$ splitting keys to the remaining $p-1$ processors.

END *SplittingKeys;*

Figure 2. Algorithm to determine bin splitting keys.

the splitting list for all nodes. This list is broadcast to the remaining $p - 1$ nodes in $O(p\log p)$ time ($\log p$ transmissions of p elements each).

Once the nodes have received the common splitting key list, they partition their elements into p bins such that bin i contains all elements with keys in the range (k_i, k_{i+1}), $0 \leq i < p$. The splitting key list is $k_1, k_2, \ldots, k_{p-1}$ and we set $k_0 = -\infty$ and $k_p = \infty$. This partitioning into bins can be done in $\min\{p\log n/p, n/p\}$ time since the elements are in sorted order (cf. step 1 of Figure 2). Let the p bins in processor i be $b_0^i, b_1^i, \ldots, b_{p-1}^i$. The elements in bin b_j^i need to be routed to processor j, $0 \leq j < p$, $0 \leq i < p$. In Figure 1, this routing is accomplished using a direct route. Two alternatives to direct routing are one-way and two-way ring routing which are described below.

In a one-way ring route, the p processors of the hypercube are viewed as a ring with adjacent processors in the ring adjacent to the hypercube. To obtain the ring organiza-

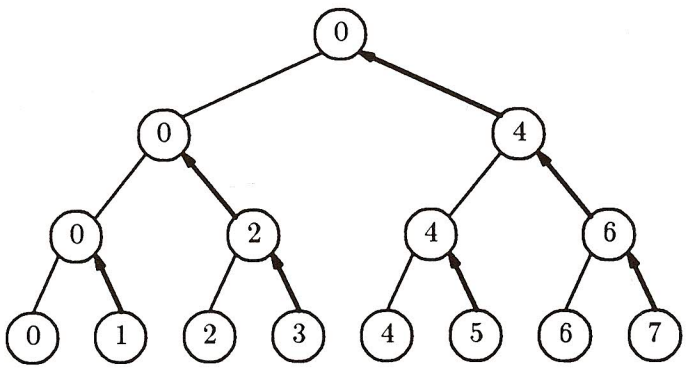

Figure 3. Merge tree for $p = 8$ processors (arrows show direction of data transfer in step 3 of the algorithm to determine bin splitting keys).

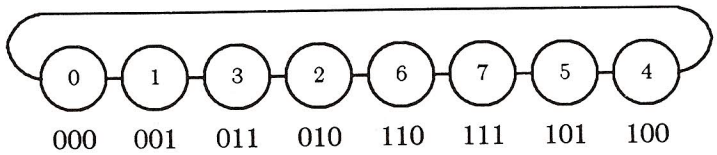

Figure 4. Ring organization for the case $p = 8$.

tion, the Gray code scheme of Chan and Saad [1986] may be used. Figure 4 shows such a ring for the case $p = 8$. The number inside a node is the processor index. The one-way ring algorithm to be run on each node is given in Figure 5. In this algorithm, each node performs three tasks simultaneously.

1. Receives data from the processor/node that is one unit clockwise from itself.
2. Sends data to the processor/node that is one unit counterclockwise from itself.
3. Merges data received in task 1 above with data in the bin having the same destination as that of the received data.

As an example, consider processor 2 in Figure 4. It initially sends the bin destined for processor 3 to processor 6. Simultaneously, it receives and merges the bin destined for processor 1 from processor 3. Thus, when processor 2 forwards the bin for processor 1 to processor 6, it really forwards the result of merging the bins of processors 3 and 2 that were destined for processor 1. In turn, when processor 6 forwards to processor 7 the bin destined for processor 1, it forwards the result of merging these bins from processors 3, 2, and 6. Likewise, when processor 2 forwards

PROCEDURE *OneWayRingRoute;*

Step 1: q := processor index.
r := index of processor $p-2$ units counterclockwise from q on the ring.
s := index of processor 1 unit counterclockwise from r.
pre := index of processor 1 unit clockwise from processor q.
next := index of processor 1 unit counterclockwise from q.

Step 2: Repeat, in parallel, steps 3, 4, and 5 until only the bin for processor q is left in processor q, Steps 3 and 4 have stopped and Step 5 is complete.

Step 3: Initiate a receive of a packet of the bin destined for processor r from processor *pre*; if there is no such packet then set r to the processor that is clockwise from the current r and repeat Step 3. When r = node clockwise from q, stop executing this step.

Step 4: If a request for a packet is received from processor *next* then send the next packet destined for processor s; if there is no such packet, send an end signal and set s to the node clockwise from the current s. When $s = q$, stop executing this step.

Step 5: Merge the packet (if any) received in step 3 with this processor's bin that is destined for the same processor.

END *OneWayRingRoute;*

Figure 5. One-way ring routing.

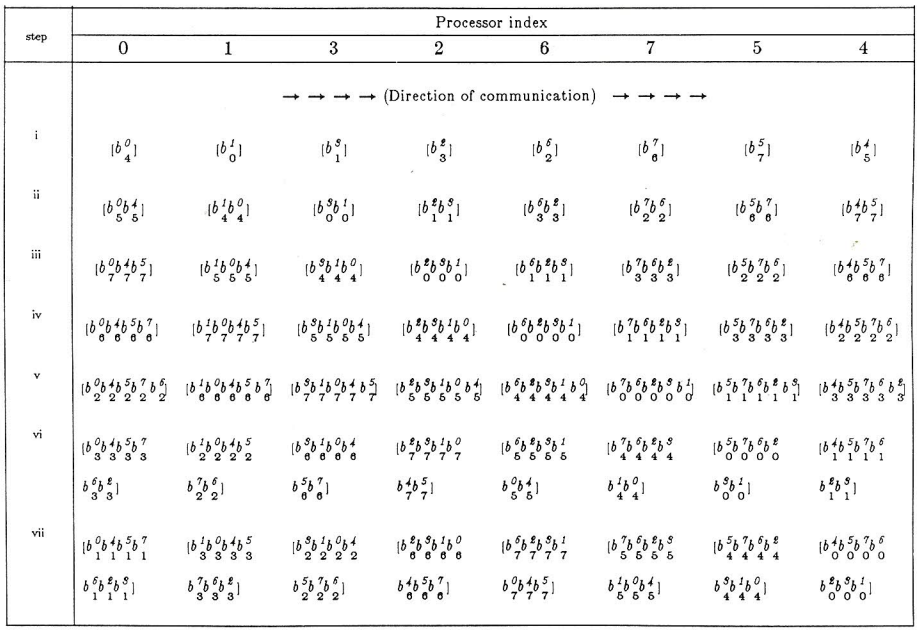

Figure 6. Data flow in one-way ring routing.

to processor 6 the bin destined for processor 0, it forwards the merging of the bins for this processor from processors 1, 3, and 2.

When the algorithm terminates, each processor has received and merged from its clockwise adjacent processor the merge of its bins from the remaining $p-1$ processors. Figure 6 shows the data flow assuming a synchronized model. b_j^i refers to the bin initially in processor i that is to be routed to processor j. The notation [ ] refers to the merge of the bins in brackets.

If we assume a uniform distribution of keys, then each b_j^i is expected to have $n/(p^2)$ elements. Let s denote the number of elements in a communication packet and let t_t be the time needed to transfer a packet of size s to an adjacent PE. The total communication/transfer time when one-way ring routing is used is

$$\sum_{i=1}^{p-1} \frac{in}{p^2 s} t_t = \frac{n}{2s}\left(1 - \frac{1}{p}\right) t_t.$$

This can be reduced by using a two-way ring routing as shown in Figure 7. The communication time for this is

$$\sum_{i=1}^{p/2} \frac{in}{p^2 s} t_t + \sum_{i=1}^{p/2-1} \frac{in}{p^2 s} t_t = \frac{n}{4s} t_t.$$

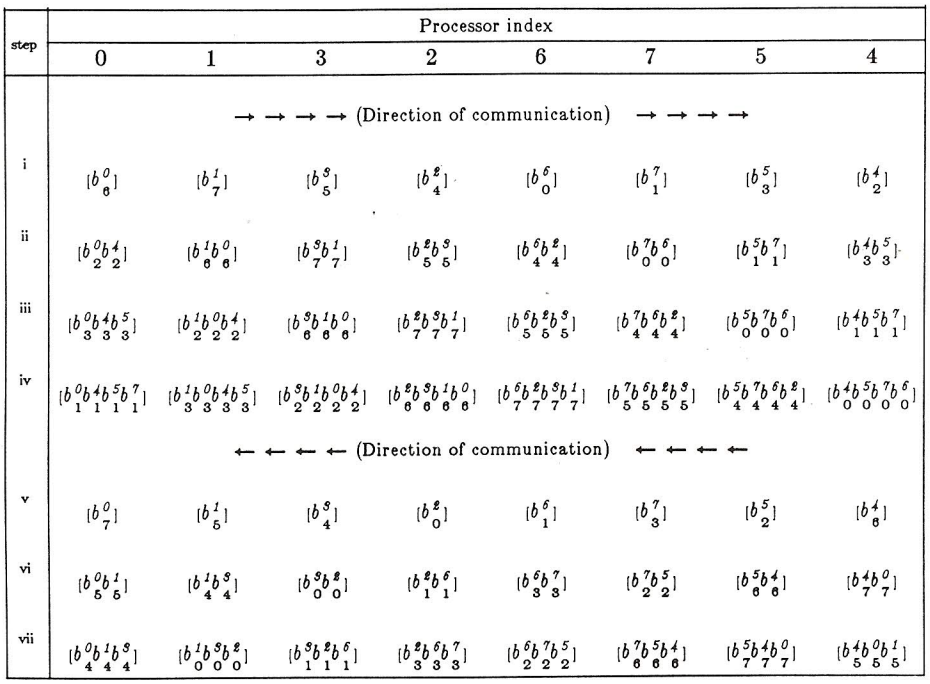

Figure 7. Data flow in two-way routing.

The communication time for direct routing depends on the router used by the hypercube and on the frequency with which routing collisions occur. Since no bin has to travel more than log$p$ hops, $n/(p^2s)\log p t_t$ is a lower bound on the transfer time. Another lower bound is obtained by considering the time required by a node to receive the p–1 bins being transmitted to it. Since a node has only log$p$ communication lines, it takes at least $n(p-1)/(p^2s\log p)t_t$ time to receive the p–1 bins. Hence, $n/(p^2s)\max\{\log p,(p-1)/\log p\}t_t$ is a lower bound on the communication time.

In synchronous hypercube computing models, routing problems such as our bin routing problem are often solved by using a recursive hypercube subdivision scheme. In this scheme, the hypercube is considered as composed of two subhypercubes. This is done by partitioning along one of the hypercube dimensions. Each node in a subhypercube sends its $p/2$ bins that are to be routed to nodes in the other subhypercube to its neighbour node in the other subhypercube (the neighbor is connected to it on the partitioning dimension). All nodes combine the received bins with their previous bins and then the two subhypercubes independently route the $p/2$ bins in each node to their destination nodes. When keys are uniformly distributed, the communication time for this scheme is $n\log p/(2ps)t_t$. This is greater than the lower bound for direct asynchronous routing.

While the modified bin sort retains the shortcomings of the original bin sort (that is, the sorted elements are not evenly distributed and the method may fail to complete

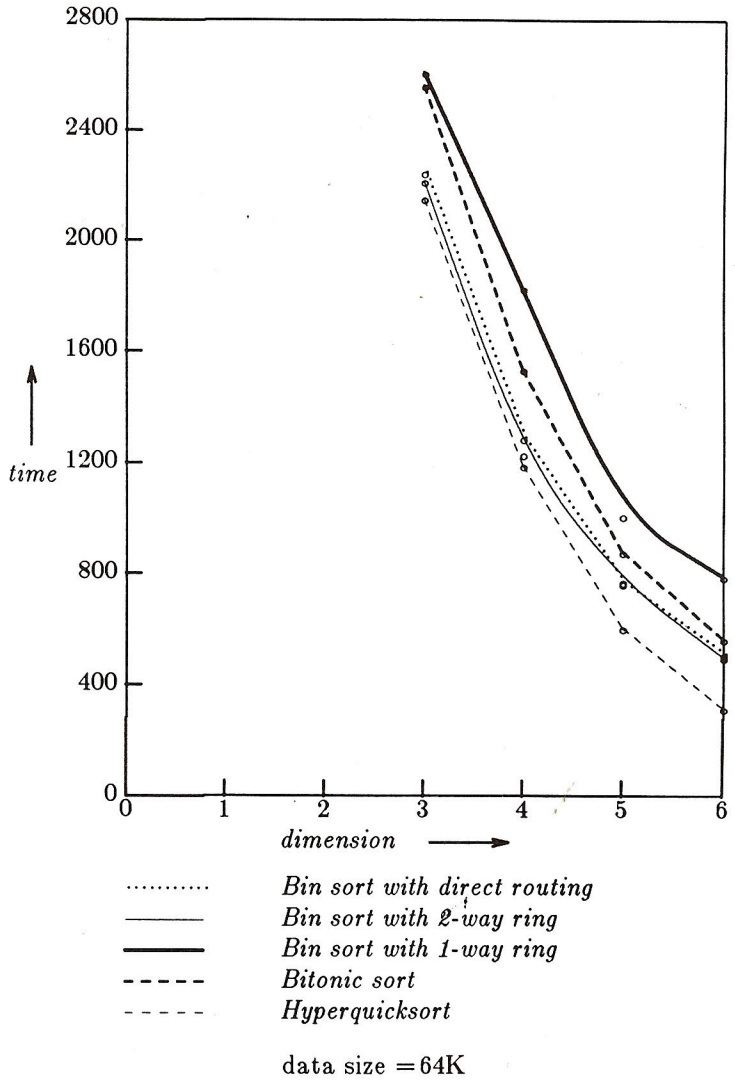

Figure 8. Performance of hypercube-to-hypercube methods.

the sort unless adequate memory is available in each node), their impact is less severe. To see this, consider the following analysis. Let $x = n/(p^2)$ be the number of elements that will fall into each bin if a node uses its p–1 splitting points (Step 2, Figure 2) to partition its (n/p) elements into p bins. If the $p-1$ splitting points that remain after the first level merge of step 3 are used on the $(2n/p)$ elements in the two nodes contributing the splitting points, then at most $3x$ elements can fall into one bin. To see this, let A and B denote the sorted sets of p–1 splitting points that are to be merged.

Let C denote the sorted set of $p-1$ splitting points that remain following the merge and elimination. The worst case arises when two adjacent splitting points of C are adjacent splitting points in A (or B). In this case, a splitting point of B (A) lies between these two points of C. There are x elements of A (B) between these two points and at most $2x$ elements of B (A) between them. When the $p-1$ splitting points computed at the root of the merge tree (see Figure 3) are used, at most $3^{\log_2 p}\ x$ elements can fall into a bin. So, to guarantee a successful sort, it is necessary for each processor to have enough memory to accommodate $3^{\log_2 p}(n/p^2) = (p^{\log_2 3 - 1})(n/p) \sim n/p^{0.415}$ elements. When a bitonic sort is used, space for only (n/p) elements is needed. When hyperquicksort is used, space for almost n elements is needed to guarantee successful completion. Hence, a balanced bin sort requires $p^{.585}$ times the space required by bitonic sort and $p^{-.415}$ times that required by hyperquicksort.

The worst case storage requirements of bin sort can be reduced at the expense of using more initial splitting keys and changing the strategy to compute global splitting keys. Suppose each node generates $ap - 1$ splitting keys, $a \geq 1$ that evenly split its data into ap bins. Then a total of $ap^2 - p$ keys are generated by the p processors. These may be transmitted to node 0 and sorted. Next, $p - 1$ of these $ap^2 - p$ keys are determined by selecting keys at positions i such that $i \bmod (ap) = 0$. The new keys have the property that if the entire n elements are partitioned into bins, then at most $(1 + 1/a)(n/p)$ elements can fall into a bin. To see this, note that when $ap - 1$ splitting keys are generated by each node, the number of elements in each nodes bin is $n/(ap^2)$. When all ap^2–p splitting keys are collapsed into $p-1$, we combine at most ap full node bins and at most p partial node bins. The total number of elements is therefore $(ap + p)n/(ap^2) = (1 + 1/a)(n/p)$. So if each processor has this memory capacity, the sort is guaranteed to succeed. Also, the degree of imbalance in the final data distribution is reduced. Note, however, that as a is increased, the time to generate the splitting keys increases because of the increased communication to node 0. This may be compensated for in the bin routing stage since bins will tend to be of a more uniform size.

3. Experimental Results

The expected performance of the candidate methods for hypercube-to-hypercube sorting was measured on an NCUBE/7 hypercube computer with 64 processors. We experimented with the following methods.

1. *Modified bitonic sort.* This is the algorithm of [Seidel and Ziegler 1987] that uses quicksort to do intranode sorting. This algorithm was modified as suggested in [Won 1987].
2. *Hyperquicksort.* This is described in [Wagar 1987].
3. *Min-max bin sort.* This is the algorithm given in Figure 1. Direct routing was used in all our experiments with this sort method.
4. *Balanced bin sort.* Three versions of bin sort were experimented with. In each, we

Table 1. Bitonic sort†.

data size	hypercube dimension					
	1	2	3	4	5	6
1K	80	42	24	14	12	13
2K	171	94	51	31	21	17
4K	362	207	113	68	45	39
8K	788	419	242	150	99	80
16K	1653	894	513	314	200	165
32K	–	2055	1247	727	416	284
64K	–	–	2561	1533	883	570
128K	–	–	–	3293	2012	1151
256K	–	–	–	–	4202	2417
512K	–	–	–	–	–	4915

† time unit is one *millisecond* for all tables in this paper

Table 2. Hyperquicksort.

data size	hypercube dimension					
	1	2	3	4	5	6
1K	71	42	21	17	14	11
2K	162	95	49	37	29	24
4K	331	211	113	66	43	35
8K	739	413	225	142	81	51
16K	1508	877	498	292	144	92
32K	–	1985	1051	571	308	162
64K	–	–	2155	1194	612	329
128K	–	–	–	2412	1224	681
256K	–	–	–	–	2523	1415
512K	–	–	–	–	–	2872

Table 3. Min-max bin sort with direct routing.

data size	hypercube dimension					
	1	2	3	4	5	6
1K	72	44	21	19	17	18
2K	161	92	52	44	33	24
4K	333	219	119	68	41	37
8K	746	422	238	140	81	55
16K	1521	887	496	292	145	94
32K	–	1995	1085	574	371	263
64K	–	–	2158	1266	754	485
128K	–	–	–	2654	1493	781
256K	–	–	–	–	2934	1661
512K	–	–	–	–	–	3198

Table 4. Balanced bin sort with direct routing.

data size	hypercube dimension					
	1	2	3	4	5	6
1 K	74	45	25	33	*	*
2 K	162	96	57	53	*	*
4 K	349	209	112	92	78	*
8 K	756	435	244	163	132	*
16 K	1573	891	509	323	219	310
32 K	–	2012	1134	595	394	407
64 K	–	–	2248	1294	770	513
128 K	–	–	–	2681	1515	795
256 K	–	–	–	–	2962	1691
512 K	–	–	–	–	–	3247

* not applicable

Table 5. Balanced bin sort with one-way ring routing.

data size	hypercube dimension					
	1	2	3	4	5	6
1 K	72	43	34	38	*	*
2 K	165	95	59	62	*	*
4 K	347	205	124	100	89	*
8 K	765	434	274	209	152	*
16 K	1588	915	604	434	309	277
32 K	–	2038	1267	815	493	396
64 K	–	–	2609	1835	1018	790
128 K	–	–	–	3908	2615	1422
256 K	–	–	–	–	5112	2915
512 K	–	–	–	–	–	5088

* not applicable

Table 6. Balanced bin sort with two-way ring routing.

data size	hypercube dimension					
	1	2	3	4	5	6
1 K	73	43	28	31	*	*
2 K	164	94	53	52	*	*
4 K	347	205	117	88	74	*
8 K	761	419	237	158	141	*
16 K	1582	875	501	319	214	195
32 K	–	1997	1101	578	383	341
64 K	–	–	2215	1233	779	502
128 K	–	–	–	2546	1499	802
256 K	–	–	–	–	2911	1618
512 K	–	–	–	–	–	3249

* not applicable

Table 7. Space required by balanced bin sort.

data size	hypercube dimension					
	1	2	3	4	5	6
1 K	108	114	119	121	*	*
2 K	109	114	115	120	*	*
4 K	110	115	122	125	119	*
8 K	108	119	118	121	129	*
16 K	111	110	121	119	128	131
32 K	–	116	126	129	133	135
64 K	–	–	129	124	136	140
128 K	–	–	–	130	134	134
256 K	–	–	–	–	135	139
512 K	–	–	–	–	–	138

* not applicable

Table 8. Space required by min-max bin sort.

data size	hypercube dimension					
	1	2	3	4	5	6
1 K	113	120	124	135	155	174
2 K	111	118	121	129	152	167
4 K	112	119	127	134	141	178
8 K	113	123	129	136	148	179
16 K	112	117	126	138	150	173
32 K	–	119	134	138	156	175
64 K	–	–	129	138	152	186
128 K	–	–	–	143	159	179
256 K	–	–	–	–	155	177
512 K	–	–	–	–	–	181

Table 9. Space required by hyperquicksort.

data size	hypercube dimension					
	1	2	3	4	5	6
1 K	115	122	128	140	168	188
2 K	112	124	129	134	171	201
4 K	109	121	135	141	155	192
8 K	114	125	138	148	158	186
16 K	116	120	137	146	161	182
32 K	–	126	140	148	166	192
64 K	–	–	135	149	159	188
128 K	–	–	–	150	167	196
256 K	–	–	–	–	176	195
512 K	–	–	–	–	–	187

modified the splitting key scheme to use $ap - 1$ keys with $a = 2$. The transfer packet size was min $\{512$ bytes, $n/2p^2\}$. The three bin sort algorithms differed in the routing scheme used: direct routing, one-way routing, and two-way routing.

The six algorithms were coded in FORTRAN. The average of the run times for ten data sets for each value of n (number of elements) and d (hypercube dimension) is reported in Tables 1 through 6. These times are plotted in Figure 8 for the case $n = 64K$. Of the three routing schemes studied for bin sort, direct routing and two-way ring routing are very competitive. Both are significantly superior to one-way ring routing. Balanced bin sort using direct or two-way ring routing has a run time that is comparable to that of min-max bin sort.

Tables 7, 8, and 9 give the space required by balanced bin sort, min-max bin sort, and hyperquicksort, respectively. This is given as a percentage with 100% being the space required to store the input data. Thus, to sort the ten 512 K instances on 64 processors, the three sort methods required 1.38, 1.81, and 1.87 times the space needed to hold the data assigned to each node initially. The space required by balanced bin sort is consistently less than that required by the other two sorting methods. Note, however, that bitonic sort requires no additional space; its space requirements are 100% for all n and d.

4. Conclusions

In this paper, we have proposed a method to select splitting keys for use in a parallel bin sort. While this takes more time than the schemes proposed earlier, the method results in bins of a more uniform size. This in turn reduces the subsequent bin routing time and the node memory requirements. The resulting bin sort scheme is called a balanced bin sort. In our experiments, balanced bin sort with direct routing took up to 5% more time than min-max bin sort with direct routing (for 64 K or more elements). However, min-max bin sort required up to 31% more space. Balanced bin sort is recommended over min-max bin sort for those situations when memory is limited.

Our experiments indicate that balanced bin sort with direct routing is faster than bitonic sort but slower than hyperquicksort. However, its memory requirements are less than those of hyperquicksort but more than those of bitonic sort. Balanced bin sort is therefore recommended over hyperquicksort and bitonic sort for those situations where there is enough memory to run it but not enough to run hyperquicksort.

References

Chan, T., and Saad, Y. 1986. Multigrid algorithms on the hypercube multiprocessor. *IEEE Transactions on Computers*, C-35 (Nov).

Cole, R. 1987. Parallel merge sort. Ultracomputer Note No. 115, Computer Science Tech. Rep. No. 278, Courant Institute of Mathematical Science, New York.

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.

Lakshmivarahan, S., Dhall, S., and Miller, L. 1984. Parallel sorting algorithms. *Advances in Computers*, 23: 295–354.

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

Seidel, S.R., and Ziegler, L.R. 1987. Sorting on hypercubes. *Hypercube Multiprocessors 1987, SIAM.*

Wagar, B. 1987. Hyperquicksort — A fast sorting algorithm for hypercubes. *Hypercube Multiprocessors 1987, SIAM.*

Won, Y. 1987. Parallel solutions for design automation problems. Ph.D. dissertation, Computer Science Dept., Univ. Minnesota.