

# A Balanced Bin Sort for Hypercube Multicomputers\*

YOUNGJU WON<sup>†</sup> 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.

<sup>†</sup> Dr Youngju Won's current address is P.O. Box 77, Gongneung-Dong Nowo-Gu, Seoul, 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 independently 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]).

FPST-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

