Parallel Computing 16 (1990) 273–278
North-Holland

Short communication

Parallel sorting with cooperating heaps in a linear array of processors *

Yen-Chun LIN ** and Ferng-Ching LIN +

** *Dept. of Electronic Engineering, National Taiwan Institute of Technology, P.O. Box 90-100, Taipei, Taiwan 10772, R.O.C.*
+ *Dept. of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan 10764, R.O.C.*

Received 25 January 1990
Revised 20 April/2 July 1990

Abstract. A parallel sorting algorithm using cooperating heaps in a linear array of processors is presented. It can sort a sequence whose length is much larger than the number of processors. Because the output begins one step after all the items have been input, sorting n items requires $2n+1$ steps. Two independent modifications of the algorithm are possible; one tries to reduce the number of processors used, and the other can sort more items on the same array.

Keywords. Heap, Linear array, Parallel sorting, Zero-time.

1 Introduction

Sorting is extremely important in information processing. Many parallel sorting algorithms have been developed to be implemented on processor arrays [1–3,6–10]. Parallel sorting on a linearly-connected array of processing elements (PEs), that takes I/O time into consideration, is of particular interest in this paper. A linear array has the merits of simple I/O and easy extendability. It is also easier to implement than other architectures such as mesh-connected, cube-connected, and tree-connected arrays.

Based on the up-down sorting described in [6], Miranker et al. devise a zero-time sorter and implement it as a linear array of VLSI PEs [9]. It is zero-time because it overlaps the sorting time with the I/O time and starts to output the results immediately after all the data have been input. This sorter is a special purpose device and must consist of a huge number of PEs to be of any practical use. Sorting n items on an array of p PEs, $p \geqslant n/2$, requires $2n$ steps. Akl and Schmeck present a parallel VLSI sorter especially for systems with serial I/O [2]. This hardware sorter also implements the up-down sorting of [6] and is claimed to be better than the sorter of [9], but still has some drawbacks. It requires a global communication line. To sort n data, it takes $\Theta(n)$ time on $n/2$ PEs.

* This research was partially supported by the National Science Council of R.O.C. under contract NSC-79-0408-E011-05.

All the previous zero-time sorters need to know the number of items to be sorted before sorting begins, and the maximum number of items is twice the number of PEs. In this paper we first present a parallel sorting algorithm that uses cooperating heaps. The algorithm can sort a much higher number of items than twice the number of PEs. It distributes data as evenly as possible to all PEs. Items to be sorted are fed into the array sequentially. One step after the last item has been input, the first PE begins its output phase to pump out the results. Other PEs begin the output phase one step later than their respective previous neighbors. Sorting n items on a p-PE array requires $2n + 1$ steps. If the computing time for one step is not larger than the I/O time of an item, the sorting time is almost covered by the I/O time. Furthermore, the input of a sequence of items can be overlapped with the output of another sequence.

Two independent modifications are possible; both can be applied simultaneously. The first is to fully utilize the storage locations in a PE before using the next PE. It requires fewer PEs without sacrificing the performance when the I/O transfer rate is slower than the processing speed. Then, the algorithm need not know the exact number of items to be sorted, which is only limited by the total number of storage locations in the whole array of PEs. The second is to make all the PEs start the output phase immediately after the last item has been input. Using the same linear array, this scheme can sort more items than the original one, but it requires either a set-up step or a global broadcasting line to synchronize the PEs. It takes $2n$ steps to sort n items.

2 A sorting algorithm

We consider sorting n items in ascending order on a linear array of p PEs as depicted in *Fig. 1*. Assume that each PE can manage a min heap of $k = \lceil n/p \rceil$ items in its local memory. A min heap is a complete binary tree in which the item of each non-leaf node is not larger than the items of its children [4]. Each PE is assumed to be able to do internal computation and communications with its neighbors concurrently. Though it can input from one neighbor and output to the other neighbor at the same time, it cannot input from and output to the same neighbor simultaneously.

Following are the heap-related operations to be used:

(a) INSERT: adds a new item to a heap. Initially, the new item is placed as the last in the heap. If it is smaller than its parent, exchange the positions of them. The exchange ends until the new item is not smaller than its parent or reaches the root.

(b) REBUILD: reconstructs a damaged heap, in which the root has been replaced by a larger item. The new item is checked to see if it is larger than the smaller of its children. If so, exchange the positions of the new root item and the smaller child. The exchange ends until the new item is not larger than either of its children or reaches a leaf.

(c) REMOVE: outputs the root and constructs the remaining items into a heap simultaneously. The reconstruction of the heap can first move the last item of the heap to the root and then perform the REBUILD operation.

The INSERT and the REBUILD operation each require $O(\log k)$ time for a heap of at most k items. Before the end of the INSERT operation, the two children of the root can be compared and possibly exchanged to make the second smallest item of the heap kept in the left

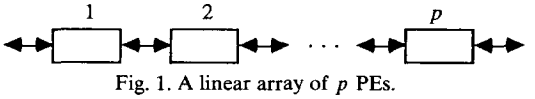

Fig. 1. A linear array of p PEs.

child of the root. The exchange is made only when the previous second smallest in the heap becomes the third smallest, and therefore the binary tree is still a min heap. This arrangement as part of the INSERT operation is assumed in this paper because it eases the description of our algorithms.

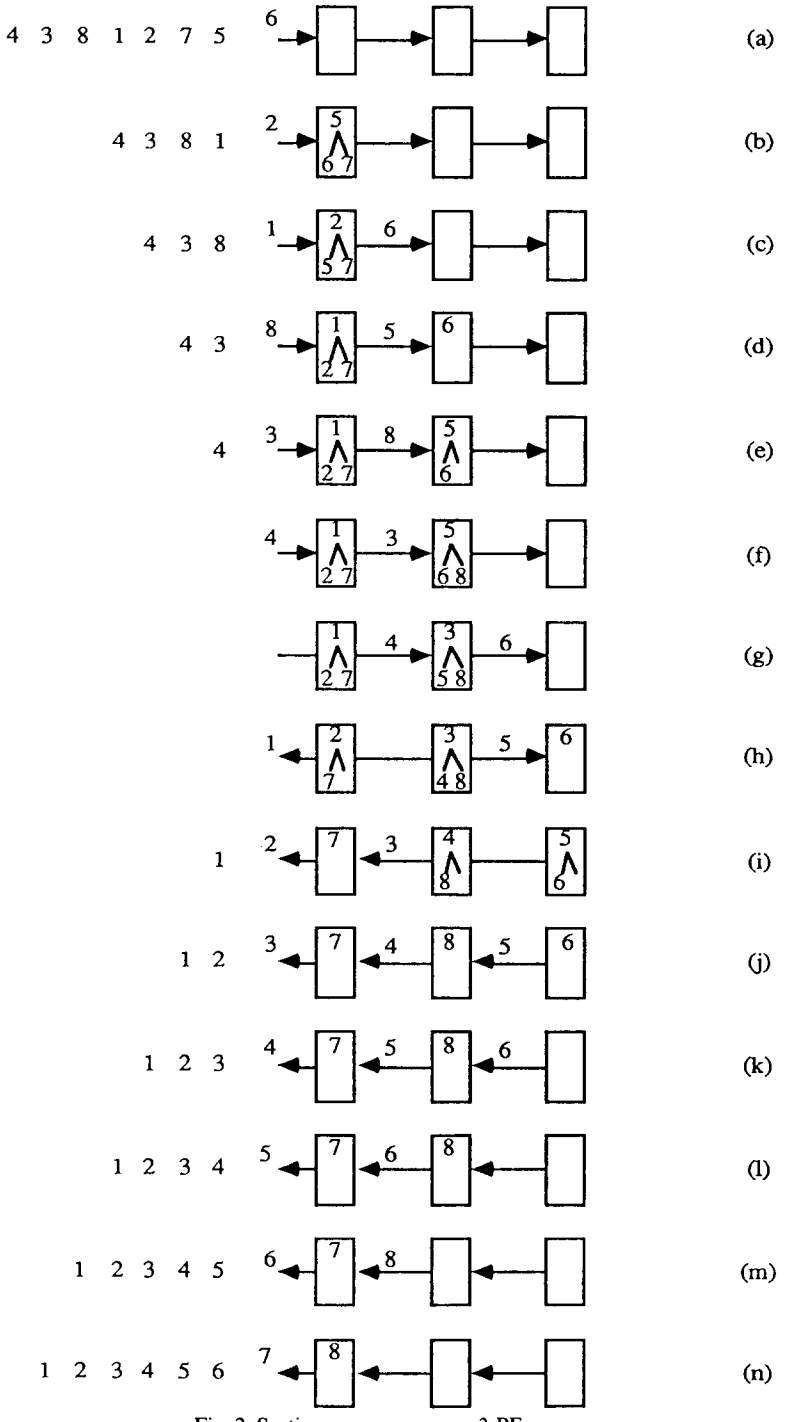

Fig. 2. Sorting a sequence on a 3-PE array.

Now we describe the operations of the PEs. After a PE has received the first input item from the left, it starts to build a heap by repeatedly performing the INSERT operation until it has created a heap of k items. Then, the PE will compare every input item with the first two smallest items of the heap. The largest of the three is forwarded to the right neighboring PE, while the smallest and the median are kept in the root and in the left child of the root, respectively. The PE still holds a min heap. The last input item is accompanied by an end-of-sequence flag to signal the end of the sequence. Therefore, after receiving an input item, a PE must check the flag to decide if it should start the output phase at the next step. The flag is passed, accompanying an item, to the right. Thus, every PE except the leftmost one starts its output phase one step later than its left neighbor.

In the output phase, the direction of data flow is reversed. At both the first and the second steps, the REMOVE operation is applied, i.e. each PE sends out its root item to the left and reconstructs the heap. Following that, a PE first compares the input from its right neighbor with the root. The smaller is output. If the root is output, the input becomes new root and the REBUILD operation is performed to maintain a min heap. When a PE receives no item from the right, it just performs the REMOVE operation to output the root.

To illustrate the algorithm, an example is shown in *Fig. 2*. An array of three PEs is used to sort the sequence (6, 5, 7, 2, 1, 8, 3, 4). *Fig. 2(a)* shows that the first item is being input. Three steps later, the first PE is receiving the fourth item and creating a min heap as depicted in *Fig. 2(b)*. *Fig. 2(g)* shows that the first PE is finishing its input phase. *Fig. 2(h)* shows that the first PE is outputting the smallest item, while the second PE is finishing its input phase. *Figs. 2(i)–2(n)* show the following step-by-step snapshots. The input phase and output phase operations of the PEs are better organized below.

Input phase

Stage 1: Receive an input and perform the INSERT operation until a heap of k items has been created. (As the rightmost PE receives no more than k items from the left, its stage 1 as well as input phase ends after the last input has been received.) The INSERT is overlapped with the receiving of next item.

Stage 2: Compare the input with the second smallest item of the heap and send the larger to the right neighbor. Compare the smaller with the root and possibly exchange them to maintain a min heap. The rightward output operation, the comparison with the root and exchange, and the receiving of next item are all done simultaneously. This stage ends after the last input has been received and processed; the last step performs comparisons and output operation, but does not receive any item.

Output phase

Stage 1: Perform the REMOVE operation twice to output the smallest and second smallest items and reconstruct the heap. The second REMOVE is overlapped with the receiving of an item from the right.

Stage 2: Send out the smaller of the input and the root. If the root is output, put the input at the root and REBUILD the heap. The output operation, the heap reconstruction, and the receiving of next item are all done in parallel. This stage ends after the last input has been processed; the last step does the same as previous steps, but does not receive any item. (As the rightmost PE receives no input from the right, it just performs the REMOVE operation.)

Stage 3: Perform the REMOVE operation until all items are output.

The number of items to be sorted by the previous algorithm is limited by p times the heap size in a PE, i.e. pk. Next we show that the algorithm really works correctly.

Theorem 1. *The algorithm can sort a sequence of n items in $2n + 1$ steps.*

Proof. During the input phase, in any PE, the items in the root and its left child are smaller than or equal to any other items in the PE as a result of the min heap construction. Furthermore, as every item sent to the right is larger than or equal to these two items, the two are smaller than or equal to any item in the PEs to the right.

During the output phase, the smallest and the second smallest items each PE has seen will be output at the beginning two steps, respectively. Thus, the leftmost PE will output the smallest and the second smallest of the sequence in proper order. Because the leftmost PE begins receiving an item from its right neighbor at its second output step, the third smallest item, if not already in the PE, will arrive at that step. If the item is already in the PE, it will be promoted to the root at the second step. In either case, the third smallest item will be selected to be output at the third step. In fact, the item being sent out of a PE is the smallest of all the items in that PE and all PEs to the right of it. Therefore, the ith smallest item will be output at the ith output step.

A sequence of n items requires n steps to input and another n steps to output. There is one step interval between the input and output of items. Therefore, a total of $2n + 1$ steps is required. □

Since there are at most k items in a heap, all the heap construction and reformation operations performed in the PEs take $O(\log k)$ time. Thus, each step takes $O(\log k)$ time. Because $2n + 1$ steps are required to sort n items, the algorithm takes $O(n \log k)$ time. The algorithm has a significant property: the sorting operations, except the last input step and some constant time comparison operations in most steps, can be overlapped with the I/O. When the $O(\log k)$ time required for a step is not larger than the time required to input or output an item, the algorithm takes only I/O time, some constant time in most steps, and $O(\log k)$ time to handle the last input. In this case, the algorithm takes $\Theta(n)$ time, which is optimal.

If two or more sequences are to be sorted, their sorting time can be overlapped. For example, in the output phase of the first sequence, when the pth PE contains no items of the first sequence, the second sequence can start entering the pth PE, which is regarded as the first PE by the second sequence.

3 Variants of the algorithm

Two independent modifications of the preceding algorithm are possible. They can be applied simultaneously, making a total of four variants. Suppose each PE can manage a heap of at most m items, where m could be larger than $\lceil n/p \rceil$. The first modification demands that, regardless of the length of a sequence, the first PE must have created a heap of m items before the second PE starts the construction, the second must have created a heap of m items before the third starts the construction, and so on. Thus, the algorithm tries to use as few PEs as possible. If input sequences are not necessarily available sequentially, it is possible to input and sort two sequences starting at both ends of the processor array in parallel.

The second modification is to make the PEs that are devoted to sorting a sequence begin the output phase at the same step. To be exact, all those PEs should start the output phase immediately after the first PE has received all the inputs, and therefore there is no intermission step between the input and the output of items. The use of flags to signal the end of a sequence is not required. However, either a set-up step or a global broadcasting line should be needed to inform the PEs when to start the output phase. The input phase of the PEs is simpler now, for it ends after the last input has been received; the last item is processed at the first output step. Moreover, the original stage 1 of the output phase is now eliminated. Note that all input items in the output phase are from the right except that the first input, which is received at the very

last step of the input phase, is from the left. One merit of this variation is that the array can sort more items. In addition to p heaps of size m each, the input buffer in each of the p PEs can also hold data, and therefore at most $mp + p$ items can be held and sorted.

4 Conclusion

A parallel algorithm that uses multiple heaps to sort a sequence of items has been presented. The output of sorted items begins one step after the input of them, and only $2n + 1$ steps are required to input, sort, and output n items. When the $O(\log m)$ time required to rebuild a damaged heap of size m is not larger than the time required to input or output an item, most of the sorting operations can be overlapped with the I/O operations. Therefore, the sorting and I/O time is $\Theta(n)$. In addition, the sorting of two sequences can be overlapped. For an array of p PEs, each with the ability of managing a heap of m items, the algorithm can sort a sequence of any length up to mp items.

Two independent modifications of the algorithm are also suggested. One modification tends to use fewer PEs to sort a sequence. The other makes all the PEs start their output phases simultaneously. It can sort up to $mp + p$ items and takes $2n$ steps to sort n items.

The use of a heap in each PE rather than any other data structure, such as a linked list, is crucial to prevent performance degradation. To keep a sorted sequence of length m with a linked list in each PE, it may take $O(m)$ time, rather than $O(\log m)$ time for a heap, at an input step. The computing time for each input step is likely to exceed the I/O time of an item, and consequently the sorting time may not be covered by the I/O time any more.

The presented algorithms can be programmed in the Occam language running on a linear array of transputers [5]. The number of connected transputers can be easily increased. Currently, the maximum memory size of each transputer is four megabytes, and it can be enlarged through advances in technology. Therefore, the maximum number of items that can be sorted on this system could be very huge.

References

[1] S.G. Akl, *Parallel Sorting Algorithms* (Academic Press, Orlando, FL, 1985).
[2] S.G. Akl and H. Schmeck, Systolic sorting in a sequential input/output environment, *Parallel Comput.* 3 (1986) 11–23.
[3] S.G. Akl, *The Design and Analysis of Parallel Algorithms* (Prentice-Hall, Englewood Cliffs, NJ, 1989).
[4] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms* (Computer Science Press, Rockville, MD, 1978).
[5] Inmos, *Occam 2 Reference Manual* (Prentice-Hall, Englewood Cliffs, NJ, 1988).
[6] D.T. Lee, H. Chang and C.K. Wong, An on-chip compare/steer bubble sorter, *IEEE Trans. Comput.* C-30 (1981) 396–405.
[7] C.E. Leiserson, Systolic priority queues, Tech. Rep. No. CMU-CS-79-115, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, April 1979.
[8] F.C. Lin and J.C. Shieh, Space and time complexities of balanced sorting on processor arrays, to appear in *J. Complexity*.
[9] G. Miranker, L. Tang and C.K. Wong, A 'zero-time' VLSI sorter, *IBM J. Res. Develop.* 27 (1983) 140–148.
[10] M.J. Quinn, *Designing Efficient Algorithms for Parallel Computers* (McGraw-Hill, New York, 1987).