Generation of Long Sorted Runs on the IBM SP2

Hung-Kuang Chen*† and Yen-Chun Lin*
*Dept. of Electronic Engineering, National Taiwan University of Science and Technology
P.O. Box 90-100, Taipei 106, Taiwan
†Dept. of Electronic Engineering, Lung-Hwa Institute of Technology
300 Wan-Shou Rd., Sec. 1, Kueishan, Taoyuan, Taiwan

Abstract—In the first phase of sorting a large file, sorted sequences, called runs, are generated. In the second phase, the runs are merged into a sorted file. The merge time can be much greater than the runs-generation time. Generating longer runs and thus fewer runs in the first phase may greatly reduce the merge time. In this paper, we present a parallel algorithm that can utilize the broadcast capability of the IBM SP2 to generate long runs. The new algorithm has been implemented in C and PVMe. Experimental results show that our algorithm generates longer runs than two earlier algorithms, and is more desirable.

Keywords: Broadcast, one-way linear array, sorted runs, sorting

1. Introduction

Sorting large files takes considerable processing time in most data processing systems [2]. Sorting a file that cannot be completely loaded into the main memory requires two phases. The first generates sorted sequences, called runs. The second merges the runs into a sorted file. The merge time can be much greater than the runs-generation time. If longer runs and thus fewer runs are generated in the first phase, the merge time can be greatly reduced because fewer merge passes over all the data are needed.

Internal sorting algorithms can generate runs of the same size as the main memory used for storing data. In contrast, the Replacement Selection algorithm can generate runs about two times longer for random inputs [4]. If the input file has been partially sorted or has some order, the Replacement Selection algorithm can generate even longer runs. The Replacement Selection algorithm has two other advantages [8]. First, when a memory of m items is used and every item in the input file has no more than m larger items before it, the Replacement Selection algorithm can sort the whole file in a single pass without merging. Second, data I/O and internal operations can be performed in parallel to reduce the runs-generation time to almost only data input or output time.

In this paper, we consider sorting items only in increasing order; sorting items in decreasing order can be performed similarly. A variant of the Replacement Selection algorithm presented by Knuth [4] is described in the following. Initially, a min-heap of a desirable size is constructed from beginning inputs. Subsequently, in each following step, the item in the root will first be output and replaced by an input item. If the input item is larger than its smaller child, it will be sifted down by exchanging it with its smaller child until it becomes a leaf or is not larger than its smaller child. However, if the input item is smaller than the latest output item, it is an item of the next run. When compared for exchange of locations, a next-run item is regarded as being larger than any item in the current run, and will be sifted down below current-run items.

Two parallel algorithms for execution on a linear array of processing elements (PEs) have been designed to generate long runs. The first algorithm, called Two-Way in this paper, runs on a two-way linear array, in which each PE can send and receive data to and from its neighboring PEs at the same time [5]. For random inputs, this algorithm can generate runs of length about two times of the total memory size in the linear array. Since I/O and internal computations can be performed in parallel, perfect overlapping can be achieved [5]. To achieve perfect overlapping, the computations are required to be completely overlapped with I/O, and input is mostly overlapped with output, reducing the elapse time to almost only input time. Specifically, perfect overlapping requires that both the time needed to input a block of data from the disk and the time needed to output a block of data to the disk be equal to or greater than the time taken to process the same amount of data.

The second algorithm, called One-Way, runs on a one-way linear array, where data can only be transferred in one direction [7]. For random inputs, the algorithm can generate runs almost twice the total memory size in the array. Compared with the first algorithm, the second has two merits. First, One-Way requires simpler hardware. Second, when perfect overlapping is achieved, the average run length obtained by

One-Way can be larger because it has less overhead and can use a larger heap in each PE.

In this paper, we propose a new algorithm, called Broadcast, for execution on the model depicted in Fig. 1. This model is the same as that used by Algorithm One-Way except that the last PE can broadcast data to other PEs. We have implemented the three algorithms on the IBM SP2 [1] with C and PVMe [3]. It should be noted that, in the original Two-Way, PEs communicate only a data item at a time [5]. For fair comparison with the other two algorithms, which transfer a packet of data items at a time, we have modified Two-Way to communicate a packet of items in each data transfer.

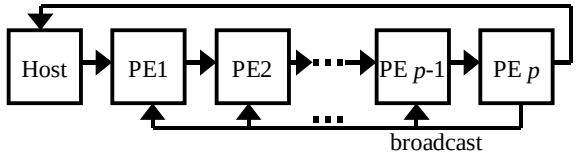

Fig. 1. A one-way linear array with broadcast capability.

2. The model

Our computation model has been depicted in Fig. 1. Assume that each of the first $p - 1$ PEs can compute, receive a packet from its left neighbor, send a packet to its right neighbor, and receive a value broadcast from the last PE simultaneously; the last PE can compute, receive a packet from its left neighbor, send a packet to the host, and broadcast a value to all the other PEs simultaneously. Each PE has a heap of m items, two buffers of k items for each inter-PE link to transfer a packet of k items, and a buffer for receiving or sending a broadcast value. The four buffers for the two inter-PE links of a PE are named as follows: the buffer holding the preceding input items is called the preceding input buffer, the buffer used to receive input items in the current step is called the input buffer, the buffer holding the items being output in the current step is called the output buffer, and the buffer used to hold the items for output in the next step is called the succeding output buffer.

The data transfer rate between the host and its neighboring PEs is assumed to be equal to that between neighboring PEs. The time taken to broadcast a value from the last PE to the other PEs is not larger than the data transfer time for a packet on any inter-PE link.

The host processor can input from and output to separate disks simultaneously. The time taken to read a block from the input disk is equal to the time taken to write a block to the output disk. To avoid possible confusion, in this paper, data transfer between the host and disks is called external I/O, and data transfer between the host and either of its neighboring PEs as well as that between neighboring PEs is called inter-PE transfer.

3. A new algorithm

3.1. Heap restructuring

The technique that maintains the heap property after the root has been replaced is reviewed here [6]. In each PE, a heap of size m can be easily implemented with the array data structure, e.g., H[1..m], where the parent of H[i] is H[$\lfloor i/2 \rfloor$], $2 \le i \le m$. In a PE, every item is either a current-run item or a next-run item. The first next-run item to reside in the heap is placed at H[m], the second is placed at H[m–1], and so on.

The items in the heap can be divided into two parts: the current-run part that contains only current-run items, and the next-run part that contains only next-run items. The current-run item with the largest index is called the trailing current-run (TCR) item. A pointer pointing to the TCR item, called the TCR pointer, can be used for heap management. If the pointer value is p, the items placed from H[1] to H[p] are current-run items; the items placed from H[p+1] to H[m] are next-run items. When the heap contains only current-run items, the TCR pointer value is m. If the heap contains only next-run items, the TCR pointer value is 0. If the root in a heap, called the root, has been replaced by a new item, the ADJUST operation can be used to maintain the heap property, as follows:

while the new item is larger than its
smaller child or only child **do**
exchange their positions

3.2. Overview

The operations of the PEs can be considered as executing an infinite loop. Each iteration of the loop is a step. In one step, each PE simultaneously receives a packet of k items from its left neighbor, sends out a packet of k items to its right neighbor, and performs computations on the data received in the preceding step to produce k items for output in next step. In addition, in the same step the last PE broadcasts the last item in the output buffer, called the latest output item, to the other $p - 1$ PEs, and the first $p - 1$ PEs receive the broadcast feedback.

To produce k output items in a step, a computation loop is performed for k iterations. In the ith iteration, the PEs check whether the ith item in the preceding input buffer, called the ith input item, is a current-run item, for $1 \le i \le k$. The last PE regards the first input item as a current-run item if the first item is not smaller than the latest output item. The $(i+1)$th input item is a current-run item if it is greater than or equal to the ith item in the output buffer, where $1 \le i < k$. For the first $p - 1$ PEs to know the latest output item, the last PE should broadcast the latest output item to the other PEs in each step. Thus, in the first $p - 1$ PEs, an item is a current-run item if it is not smaller than the broadcast feedback; otherwise, it is a next-run item. After the ith input item has been identified as either a current-run item or a next-run item, the smaller item of the ith input item and the root is moved to the succeeding output buffer as the ith output item. If the root is moved to the succeeding output buffer, the modified heap should be adjusted to retain the heap property.

3.3. Internal operations of the PEs

The internal operations of the p PEs can be described by the pseudocode given in Fig. 2. As illustrated in Fig. 2, a PE produces k output items in a step by executing a computation loop (lines 6-45) k times. An iteration of the computation loop can be divided into two portions. The first portion (lines 7-23) identifies an input item as a current-run item or next-run item. The second portion (lines 24-44) operates under one of four modes to decide an output item, and restructure the heap when necessary.

Note that, in the fourth mode of the last PE (lines 37-43), the PE outputs a next-run item, and a new run is being generated. All the next-run items in the array should then become current-run items. Thus, the last PE assigns m to its TCR pointer to indicate that all the items in the heap are current-run items. For the first $p - 1$ PEs to know whether a new run is being generated, the new feedback is compared with the last feedback (lines 2-4). If the new feedback is smaller, a new run is being generated, and the first $p - 1$ PE set the TCR pointer value to m to set all the next-run items in the heap as current-run items.

3.4. Initial and final steps

Initially, while the host is reading inputs from the disk, each PE should assign the smallest possible value to all the locations that store items, including the communication buffers, the location for the last feedback, and the heap. In this way, the first $(m + 2k)p + 2k$ output items will be the smallest possible value. On the other hand, after all the items of the input file have been sent to the array, $(m + 2k)p + 2k$ largest possible values are sent to the array, so that all the real data can be orderly extracted from the array.

```
/* PreIn = preceding input buffer, SucOut = succeeding
output buffer, Out= output buffer, NewFB = new feedback,
LastFB = last feedback, TCR = TCR pointer value, R = root
of the heap, ∞ = the largest possible value, CRI = current-
run item, NRI = next-run item */
1.   for the first p – 1 PEs do
2.     if NewFB< LastFB then
                              /* a new run is being generated*/
3.       TCR := m   /* set all items in the heap as CRIs*/
4.     end if
5.   end for
6.   for i := 1 to k do                    /*computation loop*/
7.     for the last PE do
8.       if i = 1 then
9.         if PreIn[i] < Out[k] then
10.          PreIn[i] is an NRI
11.        else PreIn[i] is a CRI end if
12.      else
13.        if PreIn[i] < SucOut[i-1] then
14.          PreIn[i] is an NRI
15.        else PreIn[i] is a CRI end if
16.      end if
17.    end for
18.    for the first p – 1 PEs do
19.      if PreIn[i] < LastFB then
20.        PreIn[i] is an NRI
21.      else PreIn[i] is a CRIs end if
22.    end for
23.    if PreIn[i] = ∞ then PreIn[i] is an NRI
24.    if PreIn[i] and R are CRIs then          /*Mode 1*/
25.      if PreIn[i] > R then
26.        SucOut[i] := R;  R := PreIn[i]
27.        ADJUST the current-run part
28.      else SucOut[i] := PreIn[i] end if
29.    else if PreIn[i] is a CRI and R is an NRI then
                                              /*Mode 2*/
30.      SucOut[i] := PreIn[i]
31.    else if PreIn[i] is an NRI and R is a CRI then
                                              /*Mode 3*/
32.      SucOut[i] := R; R := H[TCR]
33.      H[TCR] := PreIn[i]
34.      ADJUST the binary tree whose root is H[TCR]
35.      TCR := TCR – 1;  ADJUST the current-run part
36.    else if PreIn[i] and R are NRIs then     /*Mode 4*/
         /* output an NRI, a new run is being generated*/
37.      for the last PE do
38.        TCR := m /*set all items in the heap as CRIs*/
39.      end for
40.      if PreIn[i] > R then
41.        SucOut[i] := R;  R := PreIn[i]
42.        ADJUST the heap
43.      else SucOut[i] := PreIn[i]  end if
44.    end if
45.  end for
```

Fig. 2. Internal operations of a PE.

4. Experimental Results

In our experiments, eleven processors were used, one of which is the host processor. It should be noted that Two-Way uses more buffers and has more overhead than Broadcast and One-Way. For fair comparison, Broadcast and One-Way can use a larger heap in each PE. Specifically, if a packet of k items is transferred in each inter-PE transfer for the three algorithms, One-Way and Broadcast can construct a heap of m+4k items in each PE while Two-Way uses a heap of m items in each PE.

In our experiments, five different packet sizes were used (k = 40, 80, 120, 160, 200). For Algorithm Two-Way, a heap of 511 items was constructed in each PE; for the other two algorithms, a heap of 511 + 4k was built. The input file contains 10^6 integers uniformly distributed between 1 and 2×10^6. Because the performance of disk I/O is affected by other users sharing the SP2, we did not consider the disk I/O time in our experiments. All input items were read into the main memory of the host processor before the execution was started.

For each packet size, every algorithm is executed five times. To reduce the influence of other users sharing the SP2, only the least execution time of the five is used. The experimental results are summarized in Figs. 3 and 4. Fig. 3 shows the average run lengths generated, and Fig. 4 shows the execution times. From Fig. 3, Broadcast generates the longest runs for all the packet sizes used. From Fig. 4, One-Way takes the least execution time among the three algorithms for all the packet sizes used. Although Broadcast takes a little longer than One-Way, it can generate runs much longer than those generated by the others. Thus, Broadcast is preferable to the other two algorithms.

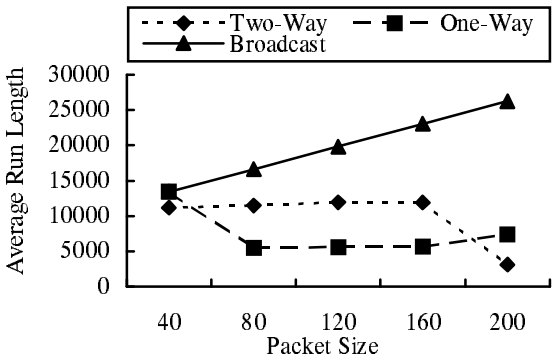

Fig. 3. Average run lengths.

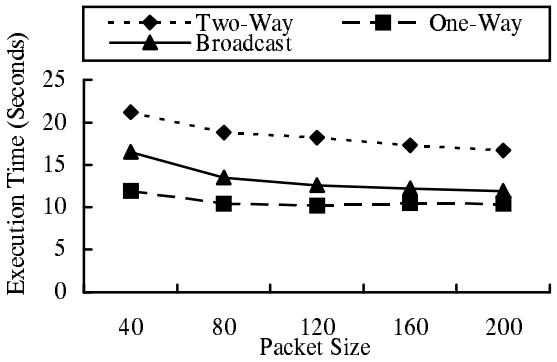

Fig. 4. Execution times.

Acknoledgements

This research was supported in part by National Science Council of the R.O.C. under contracts NCS 87-2213-E-011-005. We thank the National Center for High-Performance Computing for supporting our using the SP2.

References

[1] T. Agerwala, et al., SP2 system architecture, IBM Syst. J., vol. 34, 152-184, 1995.

[2] S.G. Akl, Parallel Sorting Algorithms. Orlando, FL: Academic Press, 1985.

[3] IBM, IBM AIX PVMe User's Guide and Subroutine Reference, Release 3.1, 3rd Ed., SH23-0019-02, Dec. 1994.

[4] D.E. Knuth, The Art of Computer Programming, vol. 3. Reading, MA: Addison-Wesley, 1973.

[5] Y.C. Lin, Perfectly overlapped generation of long runs for sorting large files, J. Parallel Distributed Comput., vol. 19, 136-142, Oct. 1993.

[6] Y.C. Lin and Y.H. Cheng, Fast generation of long sorted runs for sorting a large file, in Proc. Int. Conf. on Application Specific Array Processors, 1991, 445-456.

[7] Y.C. Lin and H.Y. Lai, Perfectly overlapped generation of long runs on a transputer array for sorting, Microprocessors Microsystems, vol. 20, 529-539, May 1997.

[8] R. Sedgewick, Algorithms. Reading, MA: Addison-Wesley, 1988.