JOURNAL OF ALGORITHMS 2, 88–102 (1981)

Finding the Maximum, Merging, and Sorting in a Parallel Computation Model

YOSSI SHILOACH

IBM Scientific Center, Technion, Haifa, Israel

AND

UZI VISHKIN*

Computer Science Department, Technion, Haifa, Israel

Received May 8, 1980

A model for synchronized parallel computation is described in which all p processors have access to a common memory. This model is used to solve the problems of finding the maximum, merging, and sorting by p processors. The main results are: 1. Finding the maximum of n elements ($1 < p \leq n$) within a depth of $O(n/p + \log\log p)$; (optimal for $p \leq n/\log\log n$). 2. Merging two sorted lists of length m and n ($m \leq n$) within a depth of $O(n/p + \log n)$ for $p \leq n$ (optimal for $p \leq n/\log n$), $O(\log m/\log p/n)$ for $p \geq n (= O(k)$ if $p = m^{1/k}n$, $k > 1$). 3. Sorting n elements within a depth of $O(n/p \log n + \log n \log p)$ for $p \leq n$, (optimal for $p \leq n/\log n$). $O(\log^2 n/\log p/n) + \log n)$ for $p \geq n$ $(= O(k \log n)$ if $p = n^{1+1/k}$, $k > 1$). The depth of $O(k \log n)$ for $p = n^{1+1/k}$ processors was also achieved by Hirschberg (*Comm. ACM* **21**, No. 8 (1978), 657–661) and Preparata (*IEEE Trans Computers* **C-27** (July 1978), 669–673). Our algorithm is substantially simpler. All the elementary operations including allocation of processors to their jobs are taken into account in deriving the depth complexity and not only comparisons.

1. INTRODUCTION

The algorithms given in this paper are all in the framework of the following model for parallel computation.

The model. There are p processors, each equipped with a small local memory and a processing unit capable of performing typical operations like reading, writing and arithmetic and Boolean operations. Each processor has an identification number i, $1 \leq i \leq p$.

All processors have access to and can write in a common, arbitrarily large, memory. A simultaneous writing of several processors in the same memory location is allowed only if they attempt to write the same thing. If

*This paper constitutes a part of the author's research towards his D.Sc. degree.

0196-6774/81/010088-15$02.00/0

several processors attempt to write different things in the same space at the same time, the algorithm is considered illegal. Thus, it will be the task of the correctness proof to show that it never happens. Simultaneous reading from the same memory location is also allowed.

The program is located in the common memory and written so that every processor knows what it should do in any time unit. There is a universal clock to the program that ticks every time unit and each processor can perform one and only one elementary operation between two ticks. A starting time will be assigned to some of the instructions. The execution of such an instruction must start exactly in the starting time assigned to it. This enables us to achieve synchronization whenever necessary. An instruction that lacks a starting time should be executed as soon as its predecessor is finished. The event of a processor reaching an instruction after its starting time will be considered illegal. The starting time is an integral part of the instruction and will usually be expressed as a function of the input parameters and the processor's serial number. The specified starting times will also be called *synchronization points*.

Besides the starting time, an instruction may contain an allocation part. This part will specify what should be done by each of the processors that reach this point in case that not all of them have to do the same.

EXAMPLE. Let n be the input length, i—the processor's number, E—a vector of length p. A typical instruction may look like:

$$[2ni]\ \textbf{if}\ i \leq n$$
$$\textbf{then}\ E(i) \leftarrow 5$$
$$\textbf{else}\ E(i) \leftarrow 6.$$

Processor i starts to perform this instruction in time $2ni$ and inserts 5 or 6 to $E(i)$ according to the i's value. For example, if $i = 4$ and $n = 7$ then processor 4 starts performing the instruction in time $t = 56$ and inserts 5 to $E(4)$.

The *depth* of an algorithm is the time elapsed between the starting of the first processor and the termination of the last one.

The literature is full of parallel algorithms. Still, there is no concensus about an exact model that will take care of all the overhead time and enable fair comparison of depth complexities of different algorithms on the same basis. We hope that our model will be proved adequate for this purpose.

The following works can be implemented directly in our model: [1–5, 8, 9, 13, 14, 17]. Works that can be implemented in models that are quite close to our's are [6, 12, 15, 16] and several works in the survey paper [7]. Reference [11] and most of its references seem to be quite far from our model. They assume no common memory and very limited communication

facilities among the processors. The communication network is determined in advance. Some of these references deal with unsynchronized algorithms.

In this paper we do the following:

1. We implement Valiant's algorithm [15] for finding the maximum in our model. The same depth of [15] is achieved (up to a constant factor) even though all the overhead time is taken into account. (In [15] only comparisons are counted. Valiant himself wonders whether his solution can be implemented within the same depth in a model that takes all the overhead into account.)

2. Two algorithms for merging are presented for the cases $p \leq n$ and $p \geq n$, respectively. Here m and n are the lengths of the lists ($m \leq n$) and p is the number of processors.

The first algorithm uses some ideas of Gavril [6], who counts only comparisons. Another crucial difference between Gavril's algorithm and ours is that his output is a linked list while ours is a ranking vector that enables recursive application of the algorithm. This recursive applicability is later used for sorting. These two differences yield an increase in the depth. Gavril's depth is $O((m/p)\log(n/m))$ while ours is $O(n/p + \log n)$. Note that $O(n/p)$ is optimal for a vector form of the output.

We have not found a way to implement Valiant's ideas in these algorithms within the same depth as his. The main problem seems to be the allocation problem. That is, to allocate p processors to p tasks in a constant time.

3. Two algorithms for sorting are presented for $p \geq n$ and $p \leq n$, respectively. Both algorithms use the corresponding merging algorithms.

Let $\mathrm{Max}_p(n)$, $\mathrm{Merge}_p(m,n)$, $\mathrm{Sort}_p(n)$ denote the best achievable depth by parallel algorithms in our model, for finding the maximum, merging, and sorting with p processors, respectively. The depth will be evaluated by the worst case criterion and will usually be expressed as a function of p, the number of processors, and the input size.

The following results are given in this paper:

1. Finding the maximum of n elements.

$$\mathrm{Max}_p(n) = O\left(\frac{n}{p} + \log\log p\right) \qquad \text{if } p \leq n,$$

(optimal for $p \leq n/\log\log n$)

$$= O\left(\log\log n - \log\log\left(\frac{p}{n} + 1\right)\right) \qquad \text{if } n \leq p \leq \binom{n}{2}, \text{ implying}$$

$$= O(\log k) \qquad \text{if } p = \lceil n^{1+1/k} \rceil (k > 1)$$

2. Merging two sorted lists of sizes m and n ($m \leq n$).

$$\begin{aligned}
\text{Merge}_p(m,n) &= O\left(\frac{n}{p} + \log n\right) && \text{if } p \leq n, \\
&\qquad (\text{optimal for } p \leq n/\log n) \\
&= O\left(\frac{\log m}{\log p/n}\right) && \text{if } p \geq n, \text{ and hence} \\
&= O(k) && \text{if } p = \lceil m^{1/k} n \rceil, k > 1.
\end{aligned}$$

3. Sorting n elements.

$$\begin{aligned}
\text{Sort}_p(n) &= O\left(\frac{n}{p}\log n + \log n \log p\right) && \text{if } p \leq n, \\
&\qquad (\text{optimal for } p \leq n/\log n) \\
&= O\left(\frac{\log^2 n}{\log p/n} + \log n\right) && \text{if } p \geq n, \text{ and hence} \\
&= O(k \log n) && \text{if } p = \lceil n^{1+1/k} \rceil, k > 1.
\end{aligned}$$

This paper extends the following known results:

1. Valiant's [15] maximum algorithm (the same depth is achieved while counting all the overheads).

2. Even [5] presents a sorting algorithm for $p \leq \log n$ processors that can nicely fit in our model. His solution is optimal in this range but has no apparent extension to a larger number of processors.

3. Batcher [2] constructs a sorting network that consists of $O(n \log^2 n)$ comparison units. This network can be regarded as a sorting algorithm in our model that uses $n/2$ processors and requires depth of $O(\log^2 n)$. When this algorithm is generalized to an arbitrary number $p \leq n/2$ of processors it yields depth of $O((n/p) \log^2 n)$ which is not optimal for any value of p.

4. Hirschberg [8] gives a sorting algorithm for $p = \lceil n^{1+1/k} \rceil$ within a depth of $O(k \log n)$. This algorithm does take care of overhead time and allocations. Preparata [12] achieves the same result (even without simultaneous reading from the same location—no "fetch conflicts"). However, both algorithms are more complicated than ours, which also does not use simultaneous writing in the same location.

The following algorithms are incomparable to ours.

1. Gavril's merging algorithm. As mentioned above, this algorithm counts only comparisons and uses a linked list as an output (while using a ranking vector as an input) and therefore cannot be applied recursively.

2. Valiant's merging algorithm. Only comparisons are counted and there is no apparent way to overcome the allocation problem. The depth of Valiant's algorithm is $O(\log\log m)$ for $p = \lceil \sqrt{mn} \rceil$. This depth cannot be achieved if $m = o(n)$ and one wants a ranking vector as an output and considers all the overhead time.

3. Preparata [12] gives a sorting algorithm for $p = n \log n$ processors and depth of $O(\log n)$. This algorithm is based upon Valiant's merging algorithm and therefore has the same difficulties. Preparata himself is aware of the allocation problem.

The following sections are devoted to: Section 2—Finding the Maximum, Section 3—Merging, Section 4—Sorting.

2. Finding the Maximum

2.1. Valiant's Algorithm and Its Implementation Problems

Our problem is to find the maximum among n elements $a_1, \ldots, a_n$ with p processors, $p \geq n$. The case $p < n$ is handled at the end of Section 2.

Since we use ideas of [15], we first give a brief description of Valiant's algorithm. It may be assumed that $a_i \neq a_j$ for all $i \neq j$. If $a_i = a_j$ and $i < j$, a_i is considered "greater" than a_j.

In the course of Valiant's algorithm, it is required to find a maximum among m elements using $\binom{m}{2}$ processors. In Valiant's "model" this is done within a depth of one comparison. Each processor compares a different pair a_i and a_j and since we have all the information, a maximal element can be found without further comparisons. The allocation problem of processors to pairs of elements and the overhead time involved in analyzing the results are not taken into account.

The iterative stage in Valiant's algorithm is as follows: Comparison are made only among elements that are potential "winners" (that have not yet lost in any comparison) and those that are still potential winners move to the next stage.

Each stage consists of the following steps:

(a) The input elements are partitioned into a minimal number l of sets $S_1, \ldots, S_l$ such that:

$$1.\ ||S_i| - |S_j|| \leq 1 \qquad \text{for all } 1 \leq i < j \leq l, \tag{2.1.1}$$

$$2.\ \sum_{i=1}^{l} \binom{|S_i|}{2} \leq p. \tag{2.1.2}$$

(b) To each set S_i we assign $\binom{|S_i|}{2}$ processors and find its maximal element, say a^i, within a depth of one comparison.

(c) If $l = 1$ we are done, else $a^1, \ldots, a^l$ will be the input for the next stage.

It is shown in [15] that if $p = n$, no more than $\log\log n + c$ stages will be required and that it is also a lower bound for this case. Except the allocation problem and the analysis of the comparisons results, one also faces the problem of calculating l while trying to implement Valiant's algorithm in constant depth per stage.

2.2. *The Algorithm in the Model*

Consider first the problem of calculating l. Let $rm(n,l) = n - l\lfloor n/l \rfloor$ denote the remainder of the division of n by l. Inequality (2.1.1) implies that the input set must be partitioned into $rm(n,l)$ subsets of size $\lfloor n/l \rfloor + 1$ and $l - rm(n,l)$ subsets of size $\lfloor n/l \rfloor$. Thus l should be the minimal number satisfying (see (2.1.2))

$$rm(n,l)\binom{\lfloor n/l \rfloor + 1}{2} + (l - rm(n,l))\binom{\lfloor n/l \rfloor}{2} \leq p. \qquad (2.2.1)$$

The value of l will be calculated in the first three instructions of the algorithm below.

Since the input set A, its size n and the number l vary from one stage to another, we express them as a function of s, the stage number. Thus, let $A[s,j]$ be an array in which $a(s,j)$ is the jth element in the input set for the $s+1$st stage.

Remark. The starting times whenever assigned to instructions are implicit since they strongly depend on the way and order in which certain operations are performed.

The Algorithm. (s—the stage number, i—the processor's number.)
Input: $n = n(s)$, $A = A(s) = (a(s-1,1), \ldots, a(s-1,n(s)))$.

1. $[t_1 = 0]$; **if** $i \leq n$
   then if $rm(n,i)\binom{\lfloor n/i \rfloor + 1}{2} + (i - rm(n,i))\binom{\lfloor n/i \rfloor}{2} \leq p$
   then $B(i) \leftarrow 1$
   else $B(i) \leftarrow 0$.

Comment. Processor i checks whether it is possible to partition A into i subsets such that (2.1.1) and (2.1.2) are satisfied. If so it inserts 1 to $B(i)$ and else 0. Thus the auxiliary vector B has the form $(0, \ldots, 0, 1, \ldots, 1)$. The location of the leftmost 1 is our desired l. Note that if $i > n$ Processor i

remains idle.

2. $[t_2]$ **if** $i = 1$
   then $C(1) \leftarrow B(1)$
   else if $i \leq n$
   then $C(i) \leftarrow B(i) - B(i-1)$.

Comment. The vector C contains 1 in the lth location and 0 elsewhere.

3. **if** $i \leq n$
   then if $C(i) = 1$
   then $(l =) l(s) \leftarrow i$.

In instructions 4, 5, 6 and 7 we allocate each processor to the pair of elements that it should compare. This is done in two steps. In instruction 6 each processor is allocated to a subset and in instruction 7 it is further allocated to a pair in this subset.

4. $[t_4] a_1 \leftarrow \lfloor n/l \rfloor$, $a_2 \leftarrow rm(n, l)$.
5. $b_1 \leftarrow \binom{a_1}{2}$, $b_2 \leftarrow \binom{a_1+1}{2}$.

Comment. The absence of an allocation part in instructions 4 and 5 means that they are executed by all the processors. They can actually be computed by one processor, however our model allows simultaneous writing of the same thing in the same location and therefore these instructions are valid in the model. Moreover, if just one processor executes these instructions, a synchronization point is required at instruction 6.

6. **if** $i \leq a_2 b_2$
   then $D(i) \leftarrow \lceil i/b_2 \rceil$, $E(i) \leftarrow rm(i, b_2) + 1$, $F(i) \leftarrow (D(i) - 1)b_2$
   else if $i \leq a_2 b_2 + (l - a_2)b_1$
   then $D(i) \leftarrow a_2 + \lceil (i - a_2 b_2)/b_1 \rceil$, $E(i) \leftarrow rm(i - a_2 b_2, b_1) + 1$,
   $F(i) \leftarrow a_2 b_2 +$
   $+ (D(i) - 1 - a_2)b_1$.

Comment. $D(i)$ is the serial number of the subset allocated to processor i and $E(i)$ is his serial number among the processors allocated to this subset. The auxiliary numbers $F(i)$ are used in instruction 8.

7. **if** $i \leq a_2 b_2$
   then call ALLOCATE$(b_2, E(i), (j_1(i), j_2(i)))$
   else if $i \leq a_2 b_2 + (l - a_2)b_1$
   then call ALLOCATE$(b_1, E(i), (j_1(i), j_2(i)))$.

Comment. For given integers $n, i, n > 0, 1 \leq i \leq \binom{n}{2}$

ALLOCATE$(n, i, (j_1, j_2))$ is a routine that assigns a pair (j_1, j_2) to i such that: (a) $1 \leq j_1 < j_2 \leq n$.

(b) If $i' \neq i$ then ALLOCATE assigns to i' a pair $(j_1', j_2') \neq (j_1, j_2)$.

ALLOCATE is a sequential algorithm.

ALLOCATE $(n, i, (j_1, j_2))$:

(a) $i_1 \leftarrow \lceil i/n \rceil$; $i_2 \leftarrow i - n(i_1 - 1)$

(b) if $i_1 < i_2$
then $j_1 \leftarrow i_1$; $j_2 \leftarrow i_2$
else $j_1 \leftarrow n - i_1$; $j_2 \leftarrow n + 1 - i_2$.

8. **if** $i \leq a_2 b_2 + (l - a_2)b_1$
then if $a(s-1, F(i) + j_1(i)) < a(s-1, F(i) + j_2(i))$
then $L_{s-1}(F(i) + j_1(i)) \leftarrow 1$
else $L_{s-1}(F(i) + j_2(i)) \leftarrow 1$.

Comment. The vector $L_{s-1}[r]$ $(r = 1, \ldots, n(s))$ contains only zeros initially. After the execution of instruction 8 $L_{s-1}(r) = 0$ if and only if $a(s-1, r)$ is the maximal element in its subset. This instruction is the only one in this algorithm where simultaneous writing in the same location cannot be avoided. Without it, a lower bound of $c \log n / \log \log p$ has been recently established for this problem.

9. $[t_9]$ **if** $i \geq n$
then if $L_{s-1}(i) = 0$
then if $i \leq (a_1 + 1)a_2$
then $a(s, \lceil i/(a_1 + 1) \rceil) \leftarrow a(s-1, i)$
else $a(s, a_2 + \lceil (i - (a_1 + 1)a_2)/a_1 \rceil) \leftarrow a(s-1, i)$.

Comment. This instruction gathers the winners of each subset into one vector $(a(s,1), \ldots, a(s, l(s)))$.

10. if $l(s) = 1$
then print "MAX = " $a(s,1)$ and stop.
else $n(s+1) \leftarrow l(s)$, $s \leftarrow s + 1$, initialize your clock and go to 1.

2.3. *The Depth*

In each stage, only a constant (independent of n or p) number of operations are performed. It follows from [15] that there are at most log log $n + 1$ stages. Thus, $\mathrm{Max}_n(n) = O(\log \log n)$.

COROLLARY 1 [15].

$$\mathrm{Max}_p(n) = O\left(\log \log n - \log \log \left(\frac{p}{n} + 1\right)\right) \qquad \text{for } n \leq p \leq \binom{n}{2}.$$

Proof. Follows from the same algorithm [15].

COROLLARY 2.

$$\mathrm{Max}_p(n) = O(\log k) \qquad \text{for } p = n^{1+1/k} \ (k > 1).$$

Proof. Follows directly from Corollary 1.

COROLLARY 3 [15].

$$\text{Max}_p(n) = O\left(\frac{n}{p} + \log\log p\right) \qquad \text{for } 1 < p \leq n.$$

Proof. We first partition the n elements into p disjoint sets of sizes $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$. Then each processor is allocated to a set and finds its maximal element sequentially. Finding the maximal element among the p winners is done by applying the parallel maximum algorithm described in Section 2.2.

COROLLARY 4.

The algorithm is optimal for $1 < p \leq \dfrac{n}{\log\log n}$.

Proof. By Corollary 3, $\text{Max}_p(n) = O(n/p)$ in this range.

THEOREM [15].

$$\text{Max}_p(n) \geq \log\log n - \log\log\left(\frac{p}{n} + 1\right) \qquad \textit{for } n \leq p \leq \binom{n}{2}.$$

Proof. Valiant shows that this depth is required even if only comparisons are counted.

3. MERGING

The algorithms in this section do not use simultaneous writing in the same location.

3.1. The Case $p \leq n$

The presentation of the following algorithm will be somewhat more informal than the previous one.

Input: Two sorted lists $X = (x_1, \ldots, x_m)$, $Y = (y_1, \ldots, y_n)$, $(x_1 \leq x_2 \leq \cdots \leq x_m, y_1 \leq y_2 \leq \cdots \leq y_n, m \leq n)$.

Output: A sorted list $Z = (z_1, \ldots, z_{m+n})$ resulting from the merging of X and Y.

We assume that no equality occurs between an element of X and an element of Y. If so happens, the X's element is considered as the smaller

one. Let us consider first an algorithm for the case $m = n = p$ as a warm-up. This algorithm has depth of $O(\log n)$.

1. Processor i finds by a binary search ($O(\log n)$ operations) the smallest y_j such that $x_i < y_j$ and then performs $z_{i+j-1} \leftarrow x_i$. If there is no such y_j it performs $z_{n+i} \leftarrow x_i$.

2. Processor i finds the smallest x_j such that $y_i < x_j$ and then feeds $z_{i+j-1} \leftarrow y_i$. If there is no such x_j it feeds $z_{m+i} \leftarrow y_i$.

The main algorithm solves the case $p \leq m \leq n$ within depth of $O(n/p + \log n)$. Some of the ideas here are due to Gavril [6].

Informal Description of the Algorithm

The algorithm consists of two stages. The first stage does some preparatory work that enables a smooth parallel processing in the second stage.

Stage 1.

1. We choose a set $X' \subseteq X$ of $p - 1$ elements (so called *distinguished* elements) that divide X into p intervals of about the same size. In the same way we choose a set $Y' \subseteq Y$ of $p - 1$ elements.

2. We merge X' and Y' into a vector A of length $2p - 2$. With each element in A we store its original set (X or Y) and its serial number in it.

Remark. A contains $2p - 2$ elements that divide Z into $2p - 1$ intervals. In the next stage each processor will merge and insert to Z the elements of two successive intervals (except Processor p that deals with only one interval). Note that the size of each interval does not exceed $2 \cdot (m + n)/p$ which is not the case if we divide Z into p intervals using elements of X' or Y' only. This bound is crucial in the depth evaluation later.

Stage 2.

3. Processor $i, 2 \leq i \leq p$, checks the origin of the $(2i - 2)$th element in A. If it belongs to $X(Y)$ it finds (by a binary search) the smallest element in $Y(X)$ that is greater than it. These two elements provide Processor i with a starting point for its merging. (The starting point for Processor 1 is x_1 and y_1.)

4. Processor $i, 2 \leq i \leq p - 1$, merges (and inserts to Z) all the elements that fall between the $(2i - 2)$th and the $2i$th elements of A. Processor 1 does the same for the elements that are smaller than the second element of A. Processor p merges the elements that are greater than the greatest element of A.

Detailed Description of the Algorithm

1. **if** $1 \le i \le p-1$
   then $x'_i \leftarrow x_{i\lfloor m/p \rfloor}, y'_i \leftarrow y_{i\lfloor n/p \rfloor}$.

Comment. $X' = (x'_1, \ldots, x'_{p-1})$, $Y' = (y'_1, \ldots, y'_{p-1})$.

2. **if** $1 \le i \le p-1$
   then a. Find by a binary search on Y' the smallest j such that $y'_j > x'_i$ and perform: $A(i+j-1, 1) \leftarrow x'_i, A(i+j-1, 2) \leftarrow i, A(i+j-1, 3) \leftarrow X$.
   If there is no such j—perform:
   $A(i+p-1, 1) \leftarrow x'_i, A(i+p-1, 2) \leftarrow i, A(i+p-1, 3) \leftarrow X$.
   b. Find by a binary search on X' the smallest j such that $x'_j > y'_i$ and perform:
   $A(i+j-1, 1) \leftarrow y'_i, A(i+j-1, 2) \leftarrow i, A(i+j-1, 3) \leftarrow Y$.
   If there is no such j—perform:
   $A(i+p-1, 1) \leftarrow y'_i, A(i+p-1, 2) \leftarrow i, A(i+p-1, 3) \leftarrow Y$.

Comment. A is a $(2p-2) \times 3$ array.

For $1 \le l \le 2p-2$:
- $A(l, 1)$ contains the value of the l-th element in the merging of X' and Y',
- $A(l, 2)$ contains its original index in X' or Y',
- $A(l, 3)$ contains its origin (X or Y).

3. **if** $2 \le i \le p$
   then if $A(2i-2, 3) = X$
   then find the smallest j such that $y_j > A(2i-2, 1)$,
   $Z_{\lfloor m/p \rfloor A(2i-2,2)+j-1} \leftarrow A(2i-2, 1), SY(i) \leftarrow j, SX(i) \leftarrow A(2i-2, 2), \lceil m/p \rceil\ TY(i) \leftarrow SY(i), TX(i) \leftarrow SX(i) + 1$

   else find the smallest j such that $x_j > A(2i-2, 1)$,
   $Z_{\lfloor n/p \rfloor A(2i-2,2)+j-1} \leftarrow A(2i-2, 1), SX(i) \leftarrow j, SY(i) \leftarrow A(2i-2, 2), \lceil n/p \rceil\ TX(i) \leftarrow SX(i), TY(i) \leftarrow SY(i) + 1$

   else$(i = 1) TX(1) \leftarrow 1, TY(1) \leftarrow 1$.

Comment. In TX and TY we store the indices of the elements that are going to be compared by Processor i. They are initialized in this instruction. SX and SY will be used to check the termination of the loops in instruction 4. In order to avoid undefined variables we set $x_{m+1} = y_{n+1} = \max(x_m, y_n) + 1$.

4. **if** $1 \le i \le p - 1$
   then while $TX(i) \ne SX(i+1)$ or $TY(i) \ne SY(i+1)$ do
   if $x_{TX(i)} > y_{TY(i)}$
   then $z_{TX(i)+TY(i)-1} \leftarrow y_{TY(i)}$, $TY(i) \leftarrow TY(i) + 1$
   else $z_{TX(i)+TY(i)-1} \leftarrow x_{TX(i)}$, $TX(i) \leftarrow TX(i) + 1$
   od
   else $(i = p)$ while $TX(p) \ne m + 1$ or $TY(p) \ne n + 1$ do
   if $x_{TX(p)} > y_{TY(p)}$
   then $z_{TX(p)+TY(p)-1} \leftarrow y_{TY(p)}$, $TY(p) \leftarrow TY(p) + 1$
   else $z_{TX(p)+TY(p)-1} \leftarrow x_{TX(p)}$, $TX(p) \leftarrow TX(p) + 1$
   od

Remark. Synchronization points are required in the beginning of instructions 3 and 4.

Depth. The depths of instructions 1, 2, 3 and 4 are $O(1)$, $O(\log p)$, $O(\log n)$ and $O(n/p)$, respectively. Hence the total depth of the algorithm is $O(n/p + \log n)$. A very slight modification of this algorithm solves the case $m < p \le n$ within the same depth.

The depth above is optimal for $p \le n/\log n$.

3.2. *The case* $p \ge n\ (\ge m)$

In the following, a merging algorithm for $p \ge n$ processors is presented. Its depth is shown to be $O((\log m)/(\log p/n))$. The particular case that $p = \lceil m^{1/k} n \rceil$ $(k > 1)$ yields depth of $O(k)$.

The algorithm will be described informally. The details can be easily filled in since no new techniques are used.

The Algorithm

(a) 1. Allocate $\lfloor p/n \rfloor$ processors to each $y \in Y$. (These processors will be used to rank y with respect to X in the following instructions.) Each such set of $\lfloor p/n \rfloor$ processors performs the following:

2. $\bar{X} \leftarrow X$.

3. while $|\bar{X}| > \lfloor p/n \rfloor$ do

a. Choose $\lfloor p/n \rfloor$ distinguished elements that divide $\bar{X}$ into $\lfloor p/n \rfloor + 1$ intervals of about the same size $I_1, \ldots, I_{\lfloor p/n \rfloor + 1}$.

b. Find j such that y falls within the range of I_j.

Comment. Since the number of processors allocated to y is equal to the number of distinguished elements, this j can be found in constant depth. (See the Maximum algorithm, Instructions 1, 2 and 3.)

c. $\overline{X} \leftarrow I_j$.

4. Rank y with respect to $\overline{X}$ and insert it to the appropriate location in Z.

Comment. Instruction 4 is executed like instruction 3b and takes constant depth since $|\overline{X}| \leq \lfloor p/n \rfloor$.

(b) We rank X's elements with respect to Y and insert them to Z in the same way.

The Depth

Each iteration of instruction 3 shrinks the interval within which y should be ranked by a factor of $\lfloor p/n \rfloor + 1 \geq p/n$. Thus, after at most $((\log m)/(\log p/n))$ iterations this interval becomes smaller than p/n and we switch to instruction 4. Thus the depth is $O((\log m)/(\log p/n))$. Ranking X's elements with respect to Y requires depth of $O((\log n)/(\log p/m)) \leq O((\log m)/(\log p/n))$.

4. Sorting

As in merging, two algorithms will be described for the cases $p \leq n$ and $p \geq n$, respectively. Both will be described informally since they just contain successive applications of the merging algorithms. Both are free of simultaneous writing in the same location.

4.1. The Case $p \leq n$

The Algorithm

1. Partition the input set X into p subsets, $X_1, \ldots, X_p$, of sizes $\lfloor n/p \rfloor$ and $\lceil n/p \rceil$ and allocate one processor to each subset.

2. Each processor sorts its subset sequentially.

3. $X_1^1 \leftarrow X_1, \ldots, X_p^1 \leftarrow X_p$; $P_1^1 \leftarrow \{\text{Processor } 1\}, \ldots, P_p^1 \leftarrow \{\text{Processor } p\}$; $s \leftarrow 1$, $q \leftarrow p$.

4. While $q > 1$ do

for $1 \leq t \leq \lfloor q/2 \rfloor$ do

$P_t^{s+1} \leftarrow P_{2t-1}^s \cup P_{2t}^s$

Merge X_{2t-1}^s and X_{2t}^s into X_t^{s+1} using the set P_t^{s+1} of processors

od

if q is odd
then $P^{s+1}_{\lceil q/2 \rceil} \leftarrow P^s_q$; $X^{s+1}_{\lceil q/2 \rceil} \leftarrow X^s_q$
$s \leftarrow s + 1$; $q \leftarrow \lceil q/2 \rceil$
od

Comment. Instruction 4 applies the sorting by merging technique.

The Depth

Step 1: $O(1)$,
Step 2: $O((n/p)\log(n/p))$,
Step 3: $O(1)$,
Step 4: The "while" loop is executed $\lceil \log p \rceil$ times. In each merge operation the ratio between the number of elements and the number of processors involved is bounded by $\lceil n/p \rceil$. Thus, the depth of each iteration is $O(n/p + \log n)$.
The total depth amounts to:

$$O\left(\frac{n}{p}\log\frac{n}{p} + \log p\left(\frac{n}{p} + \log n\right)\right) = O\left(\frac{n}{p}\log n + \log p \log n\right).$$

This depth is optimal for $p \leq n/\log n$.

4.2. *The Case* $p \geq n$

This case is essentially the parallel version of the sequential "sorting by merging" algorithm. Since this algorithm is well known, we describe it recursively.

The Algorithm

1. Divide n into two sets of $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ elements and allocate $\lfloor p/2 \rfloor$ processors to the first set and $\lceil p/2 \rceil$ to the second.
2. Sort each set with the processors allocated to it.
3. Merge the sorted sets.

Depth. In order to obtain a smooth evaluation of the depth we assume that only $p = 2^{\lfloor \log p \rfloor}$ processors are employed and that the input size is $n' = 2^{\lfloor \log n \rfloor}$. The actual depth of the algorithm will obviously not exceed the computed depth which satisfies the following recurrence formula:

$$\mathrm{Sort}_{p'}(n') \leq \mathrm{Sort}_{p'/2}\left(\frac{n'}{2}\right) + \mathrm{Merge}_{p'}\left(\frac{n'}{2}, \frac{n'}{2}\right) + C \leq \mathrm{Sort}_{p'/2}\left(\frac{n'}{2}\right) + O\left(\frac{\log n'}{\log p'/n'}\right) + C.$$

(The last inequality follows from the fact that $p' \geq n'/2$.)

One can easily verify now that

$$\mathrm{Sort}_p(n) \le \mathrm{Sort}_{p'}(n') = O\left(\frac{\log^2 n'}{\log p'/n'} + \log n'\right) = O\left(\frac{\log^2 n}{\log p/n} + \log n\right).$$

Note that if $p = \lceil n^{1+1/k} \rceil$ then $\mathrm{Sort}_p(n) = O(k \log n)$.The last bound for the case $p = \lceil n^{1+1/k} \rceil$ has been achieved by Hirschberg [8] and Preparata [12] by much more complicated algorithms.

ACKNOWLEDGMENTS

We wish to thank Dr. M. Rodeh for his suggestions that helped us to simplify the first merging algorithm and Professor S. Even for stimulating discussions.

REFERENCES

1. D. A. ALTON AND D. M. ECKSTEIN, Parallel breadth-first search of p sparse graphs, Proc. of Humboldt State University Conference on Graph Theory, Combinatorics and Computing (Phillis Clinn, Ed.), Utilitas Mathematica, University of Manitoba, Winnipeg, in press.
2. K. E. BATCHER, Sorting networks and their applications, *Proc. AFIPS Spring Joint Computer Conf.*, **32** (1968), 307–314.
3. D. M. ECKSTEIN, "Parallel Processing Using Depth-First Search and Breadth-First Search," Ph.D. thesis, Dept. of Computer Science, University of Iowa, Iowa City, Iowa, 1977.
4. D. M. ECKSTEIN AND D. A. ALTON, Parallel searching of nonsparse graphs, *SIAM J. Comput.*, in press.
5. S. EVEN, Parallelism in tape-sorting, *Comm. ACM* **17**, No. 4 (1974), 202–204.
6. F. GAVRIL, Merging with parallel processors, *Comm. ACM* **18**, No. 10 (1975), 588–591.
7. D. HELLER, A survey of parallel algorithms in numerical linear algebra, *SIAM Rev.* **20**, No. 4 (1978), 740–777.
8. D. S. HIRSCHBERG, Fast parallel sorting algorithms, *Comm. ACM* **21**, No. 8 (1978), 657–661.
9. D. S. HIRSCHBERG, A. K. CHANDRA, AND D. V. SARWATE, Computing connected components on parallel computers, *Comm. ACM* **22**, No. 8 (1979), 461–464.
10. D. E. KNUTH, "*The Art of Computer Programming*," Vol. 3, Addison–Wesley, Reading, Mass., 1973.
11. H. T. KUNG, The structure of parallel algorithms, *in* "Advances in Computers," Vol. 19, Academic Press, New York, in press.
12. F. P. PREPARATA, New parallel-sorting schemes, *IEEE Trans. Computers* **C-27** (July 1978), 669–673.
13. E. REGHBATI (ARJOMANDI) AND D. G. CORNEIL, Parallel computations in graph theory, *SIAM J. Comput.* **7**, No. 2 (1978), 230–237.
14. C. SAVAGE, "Parallel Algorithms for Graph Theoretic Problems," Ph.D. thesis, Univ. of Illinois, Urbana, Ill., 1977.
15. L. G. VALIANT, Parallelism in comparison problems, *SIAM J. Comput.* **4**, No. 3 (1975), 348–355.
16. S. WINOGRAD, On the parallel evaluation of certain arithmetic expressions, *J.Assoc. Comput. Mach.* **22**, No. 4 (1975), 477–492.
17. J. C. WYLLIE, "The Complexity of Parallel Computations", TR 79-387, Dept. of Computer Science, Cornell Univ. Ithaca, N. Y., 1979.