Searching, Merging, and Sorting in Parallel Computation

CLYDE P. KRUSKAL, MEMBER, IEEE

Abstract—**We study the number of comparison steps required for searching, merging, and sorting with P processors. We present a merging algorithm that is optimal up to a constant factor when merging two lists of equal size (independent of the number of processors); as a special case, with N processors it merges two lists, each of size N, in 1.893 lg lg N + 4 comparison steps. We use the merging algorithm to obtain a sorting algorithm that, in particular, sorts N values with N processors in 1.893 lg N lg lg N/lg lg lg N (plus lower order terms) comparison steps. The algorithms can be implemented on a shared memory machine that allows concurrent reads from the same location with constant overhead at each comparison step.**

Index Terms—**Comparison problems, computational complexity, merging, parallel computation, searching, sorting.**

INTRODUCTION

COMPARISON problems such as merging and sorting are of fundamental importance in computer science, and much effort has been devoted to finding efficient algorithms for solving these problems on sequential processors. Recently, a similar effort has been devoted to solving these problems on parallel processors [1]–[6], [8]. In this paper, we present parallel algorithms for searching, merging, and sorting that have good worst case performances.

Initially, our results are for the "comparison tree model" of parallel computation. This model was formalized by Borodin and Hopcroft [2] and is equivalent to the parallel computation model of Valiant [8]. At each step, the P processors can compare P (not necessarily disjoint) pairs elements; the result of each individual comparison will either be $<$, $=$, or $>$. The comparison tree model idealizes P-processor parallelism by a 3^P-ary tree where each node is labeled by a set of P comparisons, each child of a node represents one of the 3^P possible outcomes, and each leaf of the tree represents a solution to the problem being solved. The number of comparison steps required by an algorithm is the height of its comparison tree. Under this model, Valiant [8] presented algorithms that were faster than any previously known for merging and sorting.

More realistic models of parallel computation are shared-memory machines. One particular model of shared-memory machine is the CREW P-RAM which in a single cycle allows the processors to perform concurrent reads from the same location, but not concurrent writes (concurrent read exclusive write parallel random access machine; the notation is due to Snir [7]). Borodin and Hopcroft showed that Valiant's merging and sorting algorithms can, in fact, be implemented on a CREW P-RAM.

In this paper, we first present a searching algorithm that is optimal. We then present a merging algorithm that is optimal up to a constant factor when merging two lists of equal size (independent of the number of processors)[1]; as a special case, with N processors, it merges two lists, each of size N in 1.893 lg lg N + 4 comparison steps. Finally, we use the merging algorithm to obtain a sorting algorithm that, in particular, sorts N elements with N processors in 1.893 lg N lg lg N/lg lg lg N (plus lower order terms) comparison steps. All of our algorithms can be implemented on a CREW P-RAM with constant overhead at each comparison step.

We define $\text{Search}_P(N)$ to be the number of comparison steps required by P processors to search a sorted list of N elements for some specified value, $\text{Merge}_P(M, N)$ to be the number of comparison steps required by P processors to merge two sorted lists of sizes M and N, and $\text{Sort}_P(N)$ to be the number of comparison steps required by P processors to sort N elements. Throughout this paper, we write lg x for $\log_2 x$ and ln x for the natural logarithm $\log_e x$.

SEARCHING AND MERGING

In this section, we present algorithms for searching and merging. The general outline of the section closely follows Valiant [8, Sect. 2], and several of the algorithms are improvements on Valiant's.

The following theorem generalizes the sequential algorithm for searching a sorted list to the parallel case. The algorithm is optimal.

Theorem 1: $\text{Search}_P(N) = \lceil \log (N + 1)/\log (P + 1) \rceil$.

Proof: We show by induction that in k comparison steps, we can search a sorted list of size $(P + 1)^k - 1$. The formula certainly holds for $k = 0$. Assume that it holds for $k - 1$. Then to search a list of size $(P + 1)^k - 1$, we can compare the element being searched for to the elements in the sorted list subscripted by $i \cdot (P + 1)^{k-1}$ for $i = 1, 2, \cdots$. There are no more than P such elements [since $(P + 1)(P + 1)^{k-1} = (P +$

Manuscript received April 23, 1982; revised December 10, 1982. This work was supported in part by the National Science Foundation under Grant NSF-MCS-81-05896. A summary of this paper was presented at the 1982 International Conference on Parallel Processing.

The author is with the Department of Computer Science, University of Illinois, Urbana, IL 61801.

[1] As we discuss later, it turns out that Valiant's algorithm also shares this property.

$1)^k > (P+1)^k - 1]$. Thus, the comparisons can be performed in one step, and the problem is reduced to searching a list of size $(P+1)^{k-1} - 1$. In general, to search a list of size N in k comparison steps, we need

$$(P+1)^k - 1 \geq N$$

or

$$k \geq \frac{\log (N+1)}{\log (P+1)}$$

or

$$k = \left\lceil \frac{\log (N+1)}{\log (P+1)} \right\rceil .$$

We now show that the bound is tight. Given a sorted list of N elements, during the first step any algorithm can examine only P elements. Some segment of unexamined elements must have length at least

$$\left\lceil \frac{N-P}{P+1} \right\rceil \geq \frac{N-P}{P+1} = \frac{N+1}{P+1} - 1.$$

By induction, after the kth step, the problem must have size at least $(N+1)/(P+1)^k - 1$. Thus, the number of steps required by any algorithm is at least the minimum k for which

$$\frac{N+1}{(P+1)^k} - 1 \leq 0$$

or

$$k \geq \frac{\log (N+1)}{\log (P+1)}$$

or

$$k = \left\lceil \frac{\log (N+1)}{\log (P+1)} \right\rceil . \qquad \square$$

Corollary 1: $\text{Merge}_P(1, N) = \lceil \log (N+1)/\log (P+1) \rceil$.

Corollary 2: For $1 \leq M \leq P$ and $M \leq N$,

$$\text{Merge}_P(M, N) \leq \left\lceil \frac{\log (N+1)}{\log (\lfloor P/M \rfloor + 1)} \right\rceil .$$

Proof: Assign $\lfloor P/M \rfloor$ processors to each element in the smaller list and merge as in Corollary 1. $\square$

Corollary 3: For $1 \leq P \leq M \leq N$,

$$\text{Merge}_P(M, N) \leq \lceil M/P \rceil \lceil \lg (N+1) \rceil .$$

Proof: Assign $\lceil M/P \rceil$ elements in the smaller list to each processor and merge as in Corollary 1. $\square$

Theorem 2: For $P = \lfloor M^{1-1/k} N^{1/k} \rfloor$, integer $k \geq 2$, and $2 \leq M \leq N$,

$$\text{Merge}_P(M, N) \leq k \left\lceil \frac{\lg \lg M}{\lg k} + 1 \right\rceil$$

$$\leq \frac{k}{\lg k} \lg \lg M + k + 1.$$

Proof: We proceed inductively by showing how, given $\lfloor M^{1-1/k} N^{1/k} \rfloor$ processors, we can in k comparison steps reduce the problem of merging two lists of length M and N to the problem of merging a number of pairs of lists where each pair's shorter list has length less than $M^{1/k}$. The pairs of lists are so created that we can distribute the $\lfloor M^{1-1/k} N^{1/k} \rfloor$ processors among them in such a way as to ensure that for each pair, there will be enough processors allocated to satisfy the induction hypothesis.

Let $X = (x_1, \cdots, x_M)$ and $Y = (y_1, \cdots, y_N)$ be the two sorted lists. The following algorithm will merge them.

a) Mark the elements of X that are subscripted by $i \cdot \lceil M^{1/k} \rceil$ for $i = 1, 2, \cdots$ (thereby decomposing X into disjoint segments to be treated independently in the next induction step). There are at most $\lfloor M^{1-1/k} \rfloor$ of these (since $(\lfloor M^{1-1/k} \rfloor + 1)\lceil M^{1/k} \rceil \geq (\lfloor M^{1-1/k} \rfloor + 1) M^{1/k} > M^{1-1/k} M^{1/k} = M$).

b) Assign $\lfloor N^{1/k} \rfloor$ processors to each marked element in X. This requires (at most)

$$\lfloor M^{1-1/k} \rfloor \lfloor N^{1/k} \rfloor \leq \lfloor M^{1-1/k} N^{1/k} \rfloor = P$$

processors.

c) Independently merge each marked element of X into its proper location in Y. Let I be the integer satisfying $(I-1)^k \leq N < I^k$. By Corollary 1, the number of comparisons required is

$$\left\lceil \frac{\log (N+1)}{\log (\lfloor N^{1/k} \rfloor + 1)} \right\rceil \leq \left\lceil \frac{\log ((I^k - 1) + 1)}{\log (\lfloor I - 1 \rfloor + 1)} \right\rceil = k.$$

On completion of a), b), and c), we have identified where each marked element of X belongs in Y; this defines disjoint segments Y_i. There remain to be merged the disjoint pairs of sublists $(X_1, Y_1), (X_2, Y_2), \cdots$, where each X_i is a segment of X [as defined in a)] and each Y_i is a segment of Y. Furthermore, $\Sigma |X_i| < M$ and $\Sigma |Y_i| = N$. The number of processors needed to enable us to apply the algorithm recursively is

$$\begin{aligned} &\Sigma \lfloor |X_i|^{1-1/k} |Y_i|^{1/k} \rfloor \\ &\leq \lfloor \Sigma (|X_i|^{1-1/k} |Y_i|^{1/k}) \rfloor \\ &\leq \lfloor (\Sigma |X_i|)^{1-1/k} (\Sigma |Y_i|)^{1/k} \rfloor \text{ by Holder's inequality} \\ &\leq \lfloor M^{1-1/k} N^{1/k} \rfloor = P. \end{aligned}$$

Thus, there are enough processors to enable us to apply the merging algorithm recursively to every pair of sublists (X_i, Y_i). Furthermore, the length of each segment X_i of X is bounded by the kth root of the length of X. Thus, after the ith iteration of this algorithm, each segment of X has length no more than

$$\lambda_i = \lceil \lambda_{i-1}{}^{1/k} \rceil - 1$$

where $\lambda_0 = M$. Solving $\lambda_i < \lambda_{i-1}{}^{1/k}$ gives $\lambda_i < M^{1/k^i}$. The merging process terminates locally whenever a pair of sublists with a null component is produced. Thus, merging must be at most one iteration from complete when $\lambda_i = 1$ or

$$M^{1/k^i} \leq 2$$

or

$$i \geq \frac{\lg \lg M}{\lg k}.$$

At this point, one more iteration of the algorithm will complete the merge. Thus, $\text{Merge}_P(M, N) \leq k\lceil \lg \lg M/\lg k + 1\rceil$. □

The above theorem is a generalization of Theorem 3 of Valiant [8]. Valiant's result is the special case of $k = 2$, whereas the formula is minimized when $k = 3$.

Corollary 4: For $P = \lfloor M^{2/3}N^{1/3}\rfloor$ and $2 \leq M \leq N$,

$$\begin{aligned}\text{Merge}_P(M, N) &\leq 3\left\lceil \frac{\lg \lg M}{\lg 3} + 1\right\rceil \\ &\leq \frac{3}{\lg 3}\lg \lg M + 4 \\ &\cong 1.893 \lg \lg M + 4.\end{aligned}$$

The following corollary is very similar to Valiant's Corollary 5. However, besides having the obviously better constant factor for $k = 3$, the algorithm is slightly more natural and has a smaller constant term.

Corollary 5: For $P = \lfloor rM^{1-1/k}N^{1/k}\rfloor$ and $2 \leq r \leq M \leq N$,

$$\text{Merge}_P(M, N) \leq \frac{k}{\lg k}(\lg \lg M - \lg \lg r) + k.$$

Proof: We use the same argument as above, except that at step a), the elements in X marked are those subscripted by $i \cdot \lceil M^{1-1/k}/r\rceil$. Steps b) and c) still require no more than k comparison steps. Now $\lambda_i < (\lambda_{i-1}{}^{1/k}/r)$, from which the result follows. □

Corollary 6: For $2 \leq P \leq M + N$,

$$\text{Merge}_P(M, N) \leq \frac{M+N}{P} + \lg\left(\frac{M+N}{P}\right) + \frac{3}{\lg 3}\lg \lg P + 6.$$

Proof: Once again, let $X = (x_1, \cdots, x_M)$ and $Y = (y_1, \cdots, y_N)$ be the two sorted lists. In each list, mark the elements subscripted by $i \cdot \lceil (M + N)/P\rceil$, thereby creating segments of length $\lceil (M + N)/P\rceil - 1$ and possibly a remainder segment of smaller (positive) length. There are at most P marked elements in total, hence *a fortiori* in each list, so by Corollary 4, the marked elements can be merged in $(3/\lg 3) \lg \lg P + 4$ steps. This determines to which segment in the other list each element belongs. Assigning one processor to each marked element, it can be determined within $\lceil \lg\lceil (M + N)/P\rceil\rceil \leq \lg((M + N)/P) + 2$ comparison steps exactly where each marked element belongs in the other list, which further refines each list into subsegments delimited by the positions of all the marked elements. Thus, the problem is reduced to merging a number of pairs subsegments $(X_1, Y_1), (X_2, Y_2), \cdots$, where each subsegment is entirely contained within just one segment of X or Y.

There are at most $P + 1$ segments in all (since $P + 2$ segments would require having at least P segments of size $\lceil (M + N)/P\rceil$ and two remainder segments of positive size). If there are P or fewer segments, assign one processor to each segment. Otherwise, there are $P + 1$ segments, but the sum of the sizes of the remainder segments of X and Y is at most $(M + N)/P$; in this case, assign just one processor to the two remainder segments of X and Y and one processor to each of the other segments. For every pair of subsegments (X_i, Y_i), assign $|X_i|$ comparisons of the merge to the processor assigned to the segment containing the subsegment X_i, and do likewise for the processor assigned to the segment containing the subsegment Y_i. Thus, each pair of subsegments is assigned to (at most) two processors, and as noted by Valiant, the pair can be merged with no loss of efficiency. In this way, each pair of subsegments is merged, while no processor does more than $(M + N)/P$ comparisons. □

Corollaries 5 and 6 together define a merging algorithm that, for $M = N$ and all P, is optimal up to a constant factor[2]; this optimality is a consequence of the lower bound for merging given in Borodin and Hopcroft [2] and the fact that no parallel algorithm can be more than P times faster than its sequential counterpart.

All of these algorithms can be implemented on a CREW P-RAM in time equal in order to their number of comparison steps—see Borodin and Hopcroft [2].

Sorting

The merging algorithm of Corollary 6 allows us to obtain fast sorting algorithms by using an idea of Preparata [5]. In general, the algorithms are enumeration sorts, i.e., the rank of an element is determined by counting the number of smaller elements.

We will derive heuristically a formula to be used rigorously later. Under the temporary assumption that all variables are continuous, we have the following algorithm for parallel sorting. If $N = 1$ the list is sorted, while if $P = 1$ sort in $\text{Sort}_1(N)$ comparison steps using the best sequential sorting algorithm [it is well known that $\text{Sort}_1(N) = N \lg N - O(N)$]. Otherwise, apply the following procedure.

1) For some function G of N and P, split the processors into G groups with $P/G \geq 1$ processors in each, and split the elements into G groups with $N/G \geq 1$ elements in each.
2) Recursively sort each group independently in parallel.
3) Merge every sorted list with every other sorted list.
4) Sort the entire list by taking the rank of each element to be the sum of its ranks in each merged list in which it appears less $G - 2$ times its rank in its own list.

Noting that step 4) requires $\binom{G}{2} = G(G - 1)/2$ independent merges, we are led to the following recurrence relation for the time $S_P(N)$ this algorithm takes to sort N elements with $P > 1$ processors:

$$\begin{aligned}S_P(N) &= S_{P/G}\left(\frac{N}{G}\right) + \text{Merge}_{2P/G(G-1)}\left(\frac{N}{G}, \frac{N}{G}\right) \\ &\leq S_{P/G}\left(\frac{N}{G}\right) + \frac{N(G-1)}{P} + \lg\left(\frac{N(G-1)}{P}\right) \\ &\quad + \frac{3}{\lg 3}\lg \lg\left(\frac{2P}{G(G-1)}\right) + O(1).\end{aligned}$$

[2] Although Valiant only claims $\text{Merge}_P(M, N) \leq M + N/P + \lg(MN \log P/P) + 2 \lg \lg P + O(1)$, a careful analysis of his algorithm yields $\text{Merge}_P(M, N) \leq M + N/P + \lg(MN/P^2) + 2 \lg \lg P + O(1)$. For $M = N$, this latter upper bound is also optimal up to a constant factor, but not quite as good as our Corollary 6.

Let M be the minimum of P and N. Then the recursion stops when the product of the G's equals M. Let N_i, P_i, and G_i be the number of elements, processors, and groups at the ith stage of the recursion, and let r be the depth of recursion. Then $\prod_{i \leqslant r} G_i = M$, $N_i/P_i = N/P$, and $P_i = P/\prod_{j<i} G_j$. Thus, not counting the sequential sorting (after the final recursive call), the above algorithm requires

$$\sum_{i=1}^{r} \left[\frac{N(G_i - 1)}{P} + \lg\left(\frac{N(G_i - 1)}{P}\right) + \frac{3}{\lg 3} \lg\lg \left(\frac{2P}{(G_i - 1)\prod_{j \leqslant i} G_j} \right) \right]$$

comparisons. If the G_i are roughly equal, then except for the last few terms of the sum, $(G_i - 1)\prod_{j \leqslant i} G_j$ will be much less than P; thus, the third term in this formula can be approximated by $(3/\lg 3) \lg\lg P$. Furthermore, using this approximation, for fixed r the formula is minimized when the G_i are all equal, as can be seen by a variational argument respecting the product condition on the G_i. (This also shows that the assumption that the G_i are roughly equal is self consistent.) Thus, r satisfies $G^r = M$ or $r = \lg M/\lg G$. Now, the formula is approximately equal to

$$\frac{\lg M}{\lg G} \cdot \left[\frac{N(G-1)}{P} + \lg\left(\frac{N(G-1)}{P}\right) + \frac{3}{\lg 3} \lg\lg P + O(1) \right].$$

Minimizing this with respect to r, or equivalently G, by setting the derivative to 0 yields

$$G \cong \max\left(\frac{3}{\ln 3} \frac{P \lg\lg P}{N \lg G}, 2 \right). \qquad (*)$$

Actually, $S_P(N)$ happens not to be especially sensitive to the choice of G, so an approximation is justified here.

Now that we have found a formula for the optimum G, we proceed with an algorithm that works for integers. From now on, we assume that P, N, and G are all integers.

First consider the case $P \leqslant N$ and temporarily assume that $P = (G-1)G^I$ for some integer I. The algorithm proceeds as follows.

1) Partition the elements as evenly as possible into P groups and sort each group independently. This requires at most $\lceil N/P \rceil \lg \lceil N/P \rceil$ comparisons.

2) Partition the sorted lists into groups of lists, each group containing $G-1$ lists. Within each group, merge every list with every other list. Within each group, there are $\binom{G-1}{2} = (G-1)(G-2)/2$ pairs of lists to merge in total and $G-1$ processors in total. Thus, each processor must merge $(G-2)/2$ lists. (If G is odd, then each processor will have half a merge to share with some other processor, but as noted earlier, two processors together can accomplish one merge with no loss of efficiency.) Hence, all the merges can be accomplished in $(G-2)\lceil N/P \rceil$ comparisons.

3) For $i = 2, 3, \cdots$, partition the sorted lists into groups of lists, each group containing G lists, and within each group merge every list with every other list. At iteration i, there are within each group $G(G-1)/2$ pairs of lists to merge and $(G-1)G^{i-1}$ processors. So there are $2G^{i-2}$ processors for each merge. Since there are $(G-1)G^{i-2}\lceil N/P \rceil$ elements in each list, by Corollary 6 the number of comparisons required for each merge is

$$(G-1)\left\lceil \frac{N}{P} \right\rceil + \lg\left((G-1)\left\lceil \frac{N}{P} \right\rceil \right) + \frac{3}{\lg 3} \lg\lg P + 6.$$

If P is not of the form $(G-1)G^I$, there will be a fragmentary list left over at each iteration. This will add $O(GN/P)$ comparisons altogether through all of the iterations.

The loop in step 3) is iterated at most $\lceil \lg P/\lg G \rceil$ times. Thus, the number of comparisons in the entire algorithm is bounded by

$$\begin{aligned} &\left\lceil \frac{N}{P} \right\rceil \lg \left\lceil \frac{N}{P} \right\rceil + \left\lceil \frac{\lg P}{\lg G} \right\rceil \cdot \left[(G-1)\left\lceil \frac{N}{P} \right\rceil \right. \\ &\qquad \left. + \lg\left((G-1)\left\lceil \frac{N}{P} \right\rceil \right) + \frac{3}{\lg 3} \lg\lg P + 6 \right] + O\left(\frac{GN}{P} \right) \\ &\quad = \frac{N}{P} \lg \frac{N}{P} + \frac{\lg P}{\lg G} \cdot \left[(G-1)\frac{N}{P} + \lg\left((G-1)\frac{N}{P} \right) \right. \\ &\qquad \left. + \frac{3}{\lg 3} \lg\lg P \right] + O\left(\frac{GN}{P} + \frac{G \lg P}{\lg G} \right). \end{aligned}$$

If P divides N, the $O(G \lg P/\lg G)$ term can be deleted. Thus, when $N = P$, $G \cong (3/\ln 3)(\lg\lg N/\lg\lg\lg N)$ by equation (*), so

$$\begin{aligned} \mathrm{Sort}_N(N) &\leqslant \frac{3}{\lg 3} \frac{\lg N \lg\lg N}{\lg\lg\lg N - \lg\lg\lg\lg N + \lg\frac{3}{\ln 3} + O\left(\frac{\lg\lg\lg N}{\lg\lg N}\right)} \\ &\quad \cdot \left[1 + \frac{\lg e}{\lg\lg\lg N} + \frac{\lg 3}{3} \frac{\lg\lg\lg N}{\lg\lg N} \right. \\ &\quad \left. - \frac{\lg 3}{3} \frac{\lg\lg\lg\lg N}{\lg\lg N} + O\left(\frac{1}{\lg\lg N} \right) \right] \\ &\cong 1.893 \frac{\lg N \lg\lg N}{\lg\lg\lg N} \cdot \left[1 + \frac{\lg\lg\lg\lg N}{\lg\lg\lg N} + \frac{0.0066}{\lg\lg\lg N} \right. \\ &\quad \left. + \left(\frac{\lg\lg\lg\lg N}{\lg\lg\lg N} \right)^2 - O\left(\frac{\lg\lg\lg\lg N}{(\lg\lg\lg N)^2} \right) \right]. \end{aligned}$$

For $G = 2$, which is the optimal choice for G when $N \geq (3/2 \ln 3) P \lg \lg P$,

$$\text{Sort}_P(N) \leq \frac{N \lg N}{P} + \frac{3}{\lg 3} \lg P \lg \lg P + O\left(\frac{N}{P} + \lg P \lg \frac{2N}{P}\right).$$

Note that for $G = 2$, the algorithm is a pure comparison sort, i.e., it is no longer an enumeration sort.

We now consider the case when $P > N$. The algorithm is extremely similar to the above algorithm for $P \leq N$. Temporarily assume that $N = (G-1)G^I$ for some integer I. (The algorithm assumes that $G > P/N$.)

1) Partition the processors as evenly as possible into N groups, and assign one group of processors to each element to be sorted. Thus, each element has at least $\lfloor P/N \rfloor$ processors assigned to it.

2) Partition the elements into groups of size $G-1$. Within each group, compare every element to every other element, and thereby sort each group independently. There are within each group at least $(G-1)\lfloor P/N \rfloor$ processors allocated to perform $\binom{G-1}{2} = ((G-1)(G-2)/2)$ comparisons. Thus, each processor must perform at most $\lfloor G - 2/2\lfloor P/N \rfloor \rfloor$ comparisons.

3) For $i = 2, 3, \cdots$, partition the sorted lists into groups of lists, each group containing G lists, and within each group, merge every list with every other list. At iteration i, there are within each group $G(G-1)/2$ pairs of lists to merge and $(G-1)G^{i-1}\lfloor P/N \rfloor$ processors in total. So there are $2G^{i-2}\lfloor P/N \rfloor$ processors for each merge. Since there are $(G-1)G^{i-2}$ elements in each list, by Corollary 6 the number of comparisons required for each merge is

$$\frac{G-1}{\lfloor P/N \rfloor} + \lg\left(\frac{G-1}{\lfloor P/N \rfloor}\right) + \frac{3}{\lg 3} \lg \lg \frac{P^2}{N} + 6.$$

Once again, if P is not of the form $(G-1)G^I$, there will be a fragmentary list left over at each iteration. This will add $O(GN/P)$ comparisons altogether through all of the iterations.

The loop in step 3) is iterated at most $\lceil \lg N / \lg G \rceil$ times. Thus, the number of comparisons in the entire algorithm is bounded by

$$\left\lceil \frac{\lg N}{\lg G} \right\rceil \left(\frac{G-1}{\lfloor P/N \rfloor} + \lg \frac{G-1}{\lfloor P/N \rfloor} + \frac{3}{\lg 3} \lg \lg \frac{P^2}{N} + 6 \right) + O\left(\frac{GN}{P}\right).$$

As a special case, if $P = N(\lg N)^{1/k}$ and $1 \leq k = o(\lg \lg N / \lg \lg \lg N)$, then $G \cong (3k/\ln 3)(\lg N)^{1/k}$ by equation (*) and

$$\text{Sort}_P(N) = \frac{3k \lg N}{\lg 3} \cdot \left[1 - \frac{k \lg k}{\lg \lg N} + O\left(\frac{k}{\lg \lg N} \cdot \left(1 + \frac{k(\lg k)^2}{\lg \lg N} \right) \right) \right].$$

This is an improvement on Hirschberg [4] who showed that $N^{1+1/k}$ processors can sort in $O(k \lg N)$ time, and a generalization of Preparata [5] who showed the $N \lg N$ processors can sort in $O(\lg N)$ time.

It remains only to show that the algorithms can be implemented on a CREW P-RAM. As noted in the previous section, the merges can be implemented in the time order required. What remains to be determined is the time required to find the rank of each element. First consider the algorithm for $N \geq P$. At each iteration of step 3), there are enough processors in each group so that each processor can be assigned (at most) $\lceil N/P \rceil$ elements. Since each element participates in (at most) $G-1$ merges, a single processor can determine the rank of one element in $O(G)$ time by performing a sequential sum. Thus, each processor can determine the ranks of all the elements assigned to it in $O(GN/P)$ time.

In the algorithm for $P > N$, at each iteration of step 3), there are enough processors so that each element can be assigned $\lfloor P/N \rfloor$ processors. Therefore, using the standard parallel summing algorithm, the rank of each element can be determined in $O(G/\lfloor P/N \rfloor + \lg \lfloor P/N \rfloor$ time. Substituting $(3/\ln 3)(P \lg \lg P/N \lg G)$ for G gives $O(\lg \lg P / \lg G + \lg (P/N))$ for the time to determine the rank of each element. If $P = O(N \lg N)$, the order of this will be at most equal to the number of comparison steps.

REFERENCES

[1] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Comput. Conf.*, vol. 32, Apr. 1968, pp. 307–314.

[2] A. Borodin and J. E. Hopcroft, "Routing, merging and sorting on parallel models of computation," in *Proc. ACM 14th Annu. Symp. Theory of Comput.*, 1982, pp. 338–344.

[3] F. Gavril, "Merging with parallel processors," *Commun. Ass. Comput. Mach.*, pp. 588–591, Oct. 1975.

[4] D. S. Hirschberg, "Fast parallel sorting algorithms," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 657–661, 1978.

[5] F. P. Preparata, "New parallel-sorting schemes," *IEEE Trans. Comput.*, vol. C-27, pp. 669–673, 1978.

[6] Y. Shiloach and U. Vishkin, "Finding the maximum, merging, and sorting in a parallel computation model," *J. Algorithms*, pp. 88–102, Mar. 1981.

[7] M. Snir, "On parallel search," presented at the Ottawa Conf. Distributed Comput., Aug. 1982.

[8] L. G. Valiant, "Parallelism in comparison problems," *SIAM J. Comput.*, vol. 4, pp. 348–355, 1975.

Clyde P. Kruskal (M'83) was born on May 25, 1954. He received the A.B. degree in mathematics and computer science from Brandeis University, Waltham, MA, in 1976, and the M.S. and Ph.D. degrees in computer science from New York University, New York, NY, in 1978 and 1981, respectively.

Currently, he is an Assistant Professor in Computer Science at the University of Illinois, Urbana. His research interests include the analysis of sequential and parallel algorithms and the design of parallel computers.

Dr. Kruskal is a member of the Association for Computing Machinery and the IEEE Computer Society.