SIAM J. COMPUT.
Vol. 4, No. 3, September 1975

PARALLELISM IN COMPARISON PROBLEMS*

LESLIE G. VALIANT†

Abstract. The worst-case time complexity of algorithms for multiprocessor computers with binary comparisons as the basic operations is investigated. It is shown that for the problems of finding the maximum, sorting, and merging a pair of sorted lists, if n, the size of the input set, is not less than k, the number of processors, speedups of at least $O(k/\log\log k)$ can be achieved with respect to comparison operations. The algorithm for finding the maximum is shown to be optimal for all values of k and n.

Key words. parallel algorithms, comparison problems, sorting merging, tournaments, complexity

Introduction. We investigate the worst-case time complexity of parallel binary-comparison algorithms for the classical problems of merging, sorting, and finding the maximum. We do this for a model that in several senses can be regarded as embodying the intrinsic difficulty of solving these problems on a multiprocessor computer. Any lower bound on the time complexity of a task for this model will necessarily also be a bound for any other model of parallelism that has binary comparisons as the basic operations. Furthermore the best constructive upper bounds will correspond to the fastest algorithms for independent processor machines whenever the time taken to perform a comparison dominates all the overheads.

For each problem the input consists of a set of elements on which there is a linear ordering. The ordering relationship between any pair of elements can be discovered by performing a comparison operation on them. In our model there are k processors available, and therefore k comparisons can be performed simultaneously. The processors are synchronized so that within each time interval each of them completes a comparison. At the end of the interval the algorithm decides, by inspecting the ordering relationships that have already been established, which k (not necessarily disjoint) pairs of elements are to be compared during the next interval, and assigns processors to them. The computation terminates when the relationships that have been discovered are sufficient to specify the solution to the given problem.

The time complexity of each problem will be expressed as a function of the number of processors, and of the size of the input set. The function will give the number of time intervals taken for a worst-case input by the comparison algorithm that requires the least time in the worst case. Thus we define $\max_k(n)$ to be this measure of complexity for the problem of finding the maximum of n elements on a k processor machine. $\mathrm{Sort}_k(n)$ is defined analogously for putting n elements in order, and $\mathrm{Merge}_k(m, n)$ for merging two sorted lists of length m, n respectively.

The phenomena we exhibit for the three problems share certain qualitative features. For a given size of input set, the more processors we have available, the shorter the computation time. However, the price paid for increased speed is increased total number comparisons. Intuitively, we can say that the larger k is, the larger the number of comparisons that at each step we have to choose on the

* Received by the editors February 5, 1974.

† Centre for Computer Studies, University of Leeds, Leeds LS2 9JT, England. This research was carried out at Carnegie-Mellon University, Pittsburgh, Pennsylvania.

basis of fixed previous information, and consequently the lower the "average quality" of the choices made. For any given task P, we can conveniently measure this phenomenon by the "speedup factor" P_1/P_k, where P_i is the worst-case time complexity on P on i processors. The success of parallelization can then be judged by observing how close this speedup factor is to k.

That there are mathematically degenerate extreme cases has been observed before. All the problems can be solved in unit time if there are enough processors for every element to be compared with every other simultaneously. The speedup then, however, is rather small ($\sim\sqrt{k}$). At the other extreme, as the input set becomes very large in relation to k, then, as observed by Borodin and Munro [5], optimal speedups can be approached. Furthermore, such speedups can be attained by algorithms that use the processors largely independently (as in Corollaries 3, 6, 7, 9 below) and that are therefore efficient even on machines for which interprocessor communication is relatively expensive.

Here, however, we shall focus especially on the intermediate cases. As the fastest parallel algorithms previously studied for the case $k = n = m$ are those that can be realized on sorting networks (Batcher [2], Knuth [6]), it will be of interest to compare the results for these with our analysis. Thus, to find the maximum of k elements on k processors can be done, and requires $\lceil \log_2 k \rceil$ steps on a network. It is natural to ask whether better utilization of the available processors can be made if the network restriction is removed. For merging two lists of k elements on k processors again $O(\log_2 k)$ time is necessary and can be achieved. In this case, it has, furthermore, been proved (by R. W. Floyd [6]) that $O(k \log_2 k)$ comparisons are necessary, and hence that, under the network constraint, near optimal use of the k processors is being made. The question is whether the $\log_2 k$ bound represents the intrinsic complexity of the merging problem or is a consequence only of the extra constraints.

Even if the network restriction is relaxed to allow arbitrary disjoint comparisons, it is easy to see that the $\log_2 k$ lower bound remains for both problems. What our results show is that for the more general model, this barrier no longer exists. We note, however, that the overheads implied by our algorithms may grow as $\log k$, i.e., faster than the bounds we shall derive. Thus although we may validly ignore overheads for any fixed value of k, it will not be meaningful to do so asymptotically.

1. The maximum. We give a worst-case analysis of the problem of finding the maximum of n elements using k processors. We consider the case of $k = n$ first, and then show how solutions to all the others can be derived. The theorems are stated in the form of asymptotic inequalities. However, it will be apparent that the analysis itself is complete in the sense that given any k and n, a provably optimal algorithm can be developed using the observations made in the proofs. Although, for simplicity, we shall not explicitly consider the possibility of two elements being equal, our arguments apply to that case as well, as long as just one of the maximal elements is being sought.

THEOREM 1. *For* $k = n > 1$,

$$\max_k(n) \geqq \log\log n - \text{const}.$$

Proof. Consider the execution of an arbitrary comparison algorithm for finding the maximum of n elements. Let C_i be the set of all elements that up to time i have not been shown to be smaller than any other. Call these the *candidates* at time i, and denote their number by c_i.

To prove the theorem, we show that given $k = n$ and c_i, the value of c_{i+1} can be bounded from below. The result is then deduced by induction on i.

Suppose that in the next time interval in every comparison between a non-candidate and a candidate the candidate turns out to be the larger. Then the results of these comparisons will clearly not contribute to any reduction in the candidate population at all. Clearly, this will also be true for any comparisons that involve only noncandidates. Therefore, in this worst case the only comparisons that do contribute to reducing the number of candidates are those among the candidates themselves.

To obtain the bound, we show that if n comparisons are made on c_i elements, then there must be a sufficiently large subset of these elements in which no pair has been compared directly. In the worst case, it is possible that the elements in this subset happen each to be larger than each of the elements outside this subset. In that case, they will clearly all still be candidates at time $i + 1$.

The inductive step can be reduced to a graph theoretic formulation if we identify elements with nodes and comparisons with arcs in the obvious way. We call a subset of the nodes of a graph *stable* if no pair from it is connected by an arc. We can then express the relationship we require as follows:

$$c_{i+1} \geqq \min \{\max \{h | G \text{ contains a stable set of size } h\} | \\ G \text{ is a graph with } c_i \text{ nodes and } k \text{ arcs}\}.$$

As a corollary to Turan's theorem, it can be shown [3], that

$$c_{i+1} \geqq \frac{c_i^2}{2k + c_i}.$$

By solving this inequality, we get that, if $c_0 = n = k$, then for some constant, c_i will exceed unity as long as

$$i < \log \log n - \text{const}.$$

The result follows. □

COROLLARY 1. *If* $4 \leqq 2n \leqq k \leqq n(n-1)/2$, *then*

$$\max_k (n) \geqq \log \log n - \log \log (k/n) - \text{const}.$$

Proof. Solving the same inequality as above, i.e.,

$$c_{i+1} \geqq \frac{c_i^2}{2k + c_i}$$

with $c_0 = n$ gives the claimed solution. □

As we shall now indicate, not only are the known bounds on stability achievable, but the extremal graphs are such that comparison algorithms based on them do reduce the candidate population at an optimal rate.

THEOREM 2. *For* $k = n > 1$,

$$\max_k(n) \leqq \log\log n + \text{const.}$$

Proof. It is known [3], [7] that any graph with p nodes that has no stable set larger than x has at least as many arcs as the graph $G_{p,x}$. $G_{p,x}$ is defined to be the graph with p nodes that consists of x disjoint cliques of which $p - x(q-1)$ have q nodes and the remaining $xq - p$ have $q - 1$ nodes, where $q = \lceil p/x \rceil$. It is easily shown that such a graph has $(q-1)(2p - xq)/2$ nodes altogether.

In our parallel algorithm we shall at time i perform comparisons as dictated by some such graph with $p = c_i$. Clearly, c_{i+1} will equal x, since to each clique there will correspond exactly one candidate at time $i+1$. To minimize c_{i+1} we shall have to use from among the graphs

$$\{G_{c_i,x} | x = 1, 2, \cdots ; G_{c_i,x} \text{ has fewer than } k \text{ arcs}\}$$

the one with the smallest index x. We therefore have that

$$c_{i+1} = \min\{x | (\lceil c_i/x \rceil - 1) \cdot (2c_i - x \cdot \lceil c_i/x \rceil)/2 \leqq k\}.$$

This relation gives the inequality

$$c_{i+1} \leqq \frac{c_i^2}{k \cdot \text{const.}}.$$

Solving for $c_0 = n = k$ gives that for some constant, $c_i = 1$ for some $i \leqq \log\log n + \text{const.}$ The result follows.

From the considerations in the proof of Theorem 1, it is immediate that if $G_{c_i,x}$ is chosen at each step so as to minimize x, the implied algorithm is indeed optimal. □

COROLLARY 2. *For* $4 \leqq 2n \leqq k \leqq n(n-1)/2$,

$$\max_k(n) \leqq \log\log n - \log\log(k/n) + \text{const.}$$ □

The remaining case, that of $k < n$, can be dealt with by the following observations. Clearly with just k comparisons we can reduce c_i by at most k at each step. However, as long as $c_i \geqq 2k$, we can achieve this optimal reduction by having k disjoint pairs from c_i compared. Furthermore, once c_i is less than $2k$, the algorithm of the previous theorem can take over. We therefore conclude the following.

COROLLARY 3. *For* $1 < k < n$,

$$n/k + \log\log k - \text{const.} < \max_k(n) < n/k + \log\log k + \text{const.}$$ □

For each case we have arrived at upper and lower bounds that differ only by additive constants. Furthermore, the method of deriving a provably optimal algorithm for any given values of k and n is implicit in our analysis. We conclude by mentioning that for the special case of $k = n$, we can state the exact result explicitly as follows.

COROLLARY 4. *The sequence* $s_1, s_2 \cdots$ *with the property that* $s_i = \max\{y | \max_y(y) = i\}$ *is defined by*

$$s_i = 3 \quad \textit{and} \quad s_{i+1} = (2s_i + 1)s_i.$$

For some real number K, $s_i = \lfloor K^{2^i}/2 \rfloor$.

Proof. By induction on i. The given solution of the recurrence follows from the analysis of [1]. □

2. Merging. We now give an algorithm for merging that is considerably faster than any corresponding algorithm previously known.

THEOREM 3. *For* $k = \lfloor \sqrt{mn} \rfloor$ *and* $1 < n \leqq m$,

$$\text{Merge}_k(n, m) \leqq 2 \log \log n + \text{const}.$$

Proof. We proceed inductively, by showing how, given $\lfloor \sqrt{mn} \rfloor$ processors, we can, in two time intervals, reduce the problem of merging two lists of length n, m, respectively, to one of merging a number of pairs of lists, the shorter of each of which has length less than $\sqrt{n}$. The pairs of lists are so created that we can distribute the $\lfloor \sqrt{mn} \rfloor$ processors amongst them at the next stage in such a way as to ensure that for each pair there will be enough processors allocated to satisfy the inductive assumption.

Consider the following algorithm for the sorted lists $X = (x_1, x_2, \cdots, x_n)$, $Y = (y_1, y_2, \cdots, y_m)$.

(a) Mark the elements of X that are subscripted by $i \cdot \lceil \sqrt{n} \rceil$ and those of Y subscripted by $i \cdot \lceil \sqrt{m} \rceil$ for $i = 1, 2, \cdots$. There are at most $\lfloor \sqrt{n} \rfloor$ and $\lfloor \sqrt{m} \rfloor$ of these, respectively. The sublists between successive marked elements and after the last marked element in each list we call *segments.*

(b) Compare each marked element of X with each marked element of Y. This requires no more than $\lfloor \sqrt{nm} \rfloor$ comparisons and can be done in unit time.

(c) The comparisons of (b) will decide for each marked element the segment of the other list into which it needs to be inserted. Now compare each marked element of X with every element of the segment of Y that has thus been found for it. This requires at most

$$\lfloor \sqrt{n} \rfloor \cdot (\lceil \sqrt{m} \rceil - 1) < \lfloor \sqrt{nm} \rfloor$$

comparisons altogether and can also be done in unit time.

On the completion of (a), (b) and (c) we have identified where each of the marked elements of X belongs in Y. Thus 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 and, therefore, of length $|X_i| \leqq \lfloor \sqrt{n} \rfloor$. Furthermore, $\sum |X_i| < n$ and $\sum |Y_i| < m$ since the sublists are disjoint. But by Cauchy's inequality,

$$\sum \sqrt{(|X_i| \cdot |Y_i|)} \leqq \sqrt{(\sum |X_i| \cdot \sum |Y_i|)}.$$

It follows that

$$\sum \lfloor \sqrt{(|X_i| \cdot |Y_i|)} \rfloor \leqq \sum \sqrt{(|X_i| \cdot |Y_i|)} \leqq \lfloor \sqrt{mn} \rfloor.$$

There are therefore enough processors altogether that we can assign $\lfloor \sqrt{(|X_i| \cdot |Y_i|)} \rfloor$ to merge (X_i, Y_i) for each i simultaneously.

We have therefore established that the inductive process of successively splitting a pair of lists into a set of pairs of sublists can continue with the given number of processors. Furthermore, the length of the shorter component of each sublist pair is inductively bounded by the square root of the shorter component of the list pair. Thus at time $2i$, each pair of lists produced has a component of

length no more than λ_i where

$$\lambda_i = \lfloor \sqrt{\lambda_{i-1}} \rfloor,$$

and $\lambda_0 = n$. Solving $\lambda_i \leqq \sqrt{\lambda_{i-1}}$ gives $\lambda_i \leqq n^{1/2^i}$. The merging process clearly terminates locally whenever a pair of sublists with a null component is produced. Thus merging must be complete before $\lambda_i = 0$. This gives that

$$\text{Merge}_k(n, m) \leqq 2\lceil \log \log n + \text{const.} \rceil$$

The additive constant can be shown to be less than unity if logarithms to the base 2 are taken. □

COROLLARY 5. *For* $k = \lfloor r\sqrt{nm} \rfloor$ *where* $1 < n \leqq m$ *and* $r \geqq 2$,

$$\text{Merge}_k(n, m) \leqq 2(\log \log n - \log \log r) + \text{const.}$$

Proof. We use the same algorithm as above, except that at step (a) the objects marked are those subscripted by $i \cdot \lceil \sqrt{(n/r)} \rceil$ in X and by $i \cdot \lceil \sqrt{(m/r)} \rceil$ in Y for $i = 1, 2, \cdots$. It is easily verified that steps (b) and (c) then each require no more than k comparisons, and can thus be done in unit time. Now $\lambda_i < \sqrt{(\lambda_{i-1}/r)}$, from which the result follows. □

COROLLARY 6. *For* $1 < k \leqq n \leqq m$,

$$\text{Merge}_k(n, m) \leqq (n + m)/k + \log(mn \log k/k) + \text{const.}$$

Proof. Mark $k - 1$ elements in each list so as to induce k segments of about uniform size (i.e., n/k and m/k) in each one. Merge the two lists of marked elements as in the above theorem. Insert each of the $2(k - 1)$ marked elements into the segment to which it belongs in the other list. This can be done in time $\log(mn/k)$. This leaves $2k$ pairs of disjoint sublists to be merged, in which no pair contains more than $(n + m)/k$ elements. It only remains to schedule how this merging is to be done on the k processors in time $(n + m)/k$ (as opposed to time $2(n + m)/k$).

The first observation is that the problem of merging a given pair of lists by the standard sequential algorithm (Knuth [6, p. 160]) can be split arbitrarily into two independent subproblems with no loss of efficiency. If the two lists have x elements altogether, then for any y we can divide the task into processes that take y and $x - y - 1$ steps, respectively. The two processes simply execute the first y and $x - y - 1$ steps, respectively, of the standard merging algorithm, but start from different ends of the lists.

With this freedom to break up the merging of a pair arbitrarily, we can schedule the whole task optimally as follows. We symbolically assign the ith processor jointly to the ith *segments* of the two lists. These segments have $(m + n)/k$ elements between them. To any sublist pair which has say z elements in common with this pair of segments, we assign z steps of the ith processor. Then clearly, we are assigning no more than $(m + n)/k$ steps altogether to each processor. Furthermore, since, by construction, each sublist is totally contained in some segment, each sublist pair will be assigned to at most two processors. With this scheduling, we can therefore carry out the remainder of the computation optimally. □

This last corollary is an improvement on one described in [5] (and attributed to Kirkpatrick) for the case $k \ll n = m$. Asymptotically, a speedup of k is clearly

achieved, since it is known [6] that the merging of two lists of length n requires $2n - 1$ comparisons in the worst case.

The method suggested in [5] is essentially that described in the first paragraph of the proof above, with the suggestion that the number of elements initially marked in each list be not $k - 1$ but some function of n, such as $\log n$. Even with naive scheduling (i.e., whenever a processor becomes free supply it with an unmerged sublist pair) a speedup of k can be achieved asymptotically in this way. Although this is less efficient than our algorithm, the idea can be used to show that even in the general case of $n \leqq m$, optimal speedup is achievable in various asymptotic senses, such as the following.

COROLLARY 7. *If* $m = \alpha n$ *where* $\alpha \geqq 1$, *then*

$$\text{Merge}_1(n, m)/\text{Merge}_k(n, m) \to k \quad \text{as } n \to \infty.$$

Proof. Execute the first paragraph of the proof of Corollary 6 but with $\log n$ elements marked in each list. This requires $o(n)$ comparisons and time. Clearly the total number of comparisons required to merge the sublist pairs produced is no more than $\text{Merge}_1(n, m)$. Even with the naive scheduling indicated above, if optimal sequential merging algorithms are used for the sublist pairs, the total time taken is no more than $\text{Merge}_1(n, m)/k + o(n)$. Since $\text{Merge}_1(n, m) > n$ the result follows. □

Note that the asymptotic behavior of $\text{Merge}_1(n, m)$ itself is at present unknown [6].

3. Sorting. The well-known information theoretic argument gives that the sorting of n elements requires, in the worst case, $n \log n - O(n)$ comparisons. This immediately gives the following lower bound for sorting on n processors:

$$\text{Sort}_n(n) \geqq \log n - \text{const}.$$

We now derive a corresponding upper bound.

THEOREM 4.

$$\text{Sort}_{n/2}(n) \leqq 2 \log n \log \log n + O(\log n).$$

Proof. We show that the binary-merge sorting algorithm requires only this time if merging is done fast, as in Theorem 3.

We first consider the case $n = 2^j$ for some j. We assume inductively that after the ith stage, we have 2^{j-i} disjoint sorted lists each of length 2^i. By assigning 2^i processors to each such pair and using the fast merging algorithm, we clearly arrive at the inductive assumption of the following stage after time $2 \log i + \text{const}$. But sorting of the whole list will be complete when $i = j$. The total time needed is therefore no more than

$$\sum_{i=1}^{\log n} (2 \log i + \text{const}.) \leqq 2 \log n \log \log n + O(\log n).$$

In the general case, when n is not a power of two, there may be a fragmentary sorted list left over at each stage. However, the above argument applies in that case as well. □

COROLLARY 8. *For* $4 \leqq 2n < k \leqq n(n-1)/2$,

$$\text{Sort}_k(n) \leqq 2(\log n - \log(k/n))(\log\log n - \log\log(k/n) + \text{const.}).$$

Proof. With k processors we can split the input into sets of size $\lceil k/n \rceil$ and sort each such set completely in one step. We then need $\log n - \log(k/n)$ stages of merging in the manner of Corollary 5. □

COROLLARY 9. *For* $1 < k < n$,

$$\text{Sort}_k(n) \leqq (n \log n)/k + 2 \log k \cdot \log(n \log k/k).$$

Proof. As in [5], we split the input into k equal sets and sort each of these sequentially in time $(n/k)\log(n/k)$. We then successively merge pairs of these, in $\log k$ stages, using the algorithm of Corollary 6. At each stage, there will clearly be twice as many processors available per merge as at the previous one, and if we always use these, then the time taken for each stage will be about n/k. □

4. Conclusion. We have shown that for the most basic model of parallelism for comparison problems, algorithms for merging, sorting and finding the maximum exist that are much more efficient than any previously known. We suggest our model and analysis as part of the theoretical background against which parallelism for these problems can be studied and in appropriate instances exploited. In practice, to derive good algorithms suitable for a specific multiprocessor machine, additional considerations have also to be taken into account. In particular, the tradeoffs between optimizing the sequencing of the comparisons (which is what our analysis attempts), and minimizing the overheads (e.g., inter-processor communication), have to be weighed.

Of the many further questions implied, theoretically the most tantalizing is perhaps that of parallelism in the problem of finding the median. Since this can be done in linear time sequentially [4], but cannot be solved in less than time $\sim \log\log n$ on n processors (by implication, Theorem 1), it follows that for the case $k = n$, $O(k/\log\log k)$ is an upper bound on the attainable speedup. Since we have shown that for merging, sorting, and finding the maximum, a speedup of that order is attainable, any substantial lowering of this upper bound for the median, which we conjecture to be possible, would put this problem in a class of its own. It would confirm that near optimal sequential algorithms for the median problem need to be "more carefully sequenced" than those for any of the others, and would go some way to explaining why they have proved more difficult to find. By examining parallelism, we may in this way gain deeper insights into specific computational problems than is offered by sequential analyses alone.

REFERENCES

[1] A. V. AHO AND N. J. A. SLOANE, *Some doubly exponential sequences*, Fibonacci Quart., 2 (1973), no. 4, pp. 429–437.

[2] K. E. BATCHER, *Sorting networks and their applications*, Proc. AFIPS Spring Joint Computer Conf., 32 (1968), pp. 307–314.

[3] C. BERGE, *Graphs and Hypergraphs*, North-Holland, London, 1973.

[4] M. BLUM, R. W. FLOYD, V. PRATT, R. L. RIVEST, AND R. E. TARJAN, *Time bounds for selection*, J. Comput. System Sci., 7 (1973), pp. 448–461.

[5] A. B. BORODIN AND I. MUNRO, Notes on "Efficient and Optimal Algorithms", 1972.

[6] D. E. KNUTH, *The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading, Mass., 1973.

[7] P. TURAN, *On the theory of graphs*, Colloq. Math., 3 (1954), pp. 19–34.