THE INTRINSIC COMPLEXITY OF PARALLELISM IN COMPARISON PROBLEMS

Leslie G. Valiant

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

January 1974

ABSTRACT

The worst-case time complexity of algorithms for multi-processor 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/loglogk) 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.

INTRODUCTION

We investigate the worst case 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 multi-processor 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, in the limit that the time taken to perform a comparison is large in relation to 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, each one of which can do a comparison independently. 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 arbitrarily inspecting all 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 them to the processors. The computation terminates when sufficient relationships have been discovered to establish 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 maximum number of time intervals taken by the comparison algorithm that solves the problem the fastest in the worst case. Thus we define $\mathrm{Max}_k(n)$ 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.

DISCUSSION

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 employment of processor time. 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 basis of fixed previous information, and consequently the lower the "average quality" of the choices made. For any given task P, we can conveniently express this phenomenon as a speedup factor P_1/P_k, where P_i is the worst case time complexity of P on i processors. The success of parallelization can then be judged by comparing this speedup with 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 [3], optimal speedups can be approached. Furthermore, good speedups can be attained by algorithms that use the processors largely independently, 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 [1], Knuth [5]), it will be of interest to compare the results for these with our analysis.

Thus, to find the maximum of n elements on n processors can be done, and requires $\lceil \log_2 n \rceil$ 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 n elements on n processors again $O(\log_2 n)$ time is necessary and can be achieved. In this case it has, furthermore, been proved (by R. W. Floyd [5]) that $O(n \log_2 n)$ comparisons are necessary altogether and hence that, under the network constraint, near optimal use of the processors is being made. The question is whether the $\log_2 n$ 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 $\log_2 n$ lower bound remains for both problems. Our results will show, however, that if the disjointness condition as well is removed, then we can break through this barrier.

THE MAXIMUM

We now 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 later show how this yields solutions to all the others. The theorems are stated in the form of asymptotic bounds. 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 refer to 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. $\mathrm{Max}_n(n) \geq \log\log n - \mathrm{const.}$[*]

PROOF. Consider the execution of an arbitrary n processor comparison algorithm for finding the maximum of n elements. Let C_i be the set of elements that after time i have not yet been shown to be less than any other element. Call these the candidates at time i, and denote their number by c_i.

Clearly at any one time, there is nothing to be gained from making comparisons that involve elements that are no longer candidates. For, in any such comparison, each non-candidate could equally be replaced by the candidate that has been found to be larger than it. Furthermore it is easy to see that at each step the only information that can be used advantageously is the identity of the candidates. All other information about the relationships that have been previously discovered is redundant.

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

[*] Throughout log x will be taken to have value 0 for $x \leq 1$, and to be to the base 2.

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 of this subset happen each to be larger than each of the elements outside the subset. In that case they will clearly all still be candidates at time i+1.

Thus the inductive step is reduced to the following graph theoretic formuation:

$$c_{i+1} \geq \min\{\max\{h \mid G \text{ contains an } h\text{-clique}\} \mid G \text{ is the complement of a graph with } c_i \text{ nodes and } \leq n \text{ arcs}\}.$$

To obtain the bound we show that the "best" case (i.e. with the smallest maximal clique in the complement) can be achieved by graphs (called <u>optimal</u> graphs) of a certain simple form. We do this by proving that there is a strategy which, given any graph with c_i nodes and $\leq$ n arcs, extracts from its complement a clique of size $\geq og(c_i,n)$. That this is the best bound follows from the observation that the associated optimal graph has no cliques larger than this in its complement (c.f. [6]).

Consider the following strategy for picking a large clique from the complement graph: Pick a node with the fewest arcs incident to it, and erase all its neighbours (and arcs incident to them) from the graph. Repeat this procedure until the graph vanishes.

Suppose that at one step the node picked has x neighbors left. Then the number of arcs that are erased at that step is at least $x(x+1)/2$, this extreme case being realized only when the selected node together with its neighbours form a clique that is disconnected from the rest of the graph. Clearly this is also the best case, for if more arcs were erased and fewer left, the maximal clique in the complement of the remaining graph would be made no smaller. Since this argument holds at each step, it follows that a clique can be selected from the complement with $og(c_i,n)$ nodes, where this denotes the smallest number of disjoint cliques that can be formed from c_i arcs and no more than n nodes. Clearly such a union of disjoint cliques, by definition, achieves this lower bound.

We now observe that if in such a best graph two cliques occur of size x,y respectively where $x-y > 1$, then the arcs involved in these could be redistributed to form two cliques of sizes x-1, and y+1, (with some arcs left over). This new graph would clearly still be a best one. By repeating this procedure we can derive a graph composed of the union of disjoint cliques all of size z or z-1, for some z. The result we call an <u>optimal graph</u>.

To obtain c_{i+1} it only remains to determine the number of disjoint cliques in such an optimal graph. We can easily obtain a lower bound for this from the two approximations

$$c_{i+1} \cdot z \geq c_i \text{ and } c_{i+1} \cdot (z-1)(z-2)/2 \leq n,$$

which signify the constraints on the number of nodes and arcs respectively. Eliminating z, and observing that $c_i \leq n$, gives that for some constant greater than two,

$$c_{i+1} \geq \frac{c_i^2}{\text{const.}n} \quad .$$

If $c_0 = n$, by solving this inequality we get that, for some other constant, $c_i > 1$ as long as $i < \text{loglog } n - \text{const.}$

Returning to the terminology of the comparison problem, we conclude that for any algorithm there is a worst case input for which the solution will not be found before time loglog n - const. □

COROLLARY 1. When $k \geq n$,

$$\text{Max}_k(n) \geq \text{loglog } n - \text{loglog}(k/n) - \text{const.}$$

PROOF. The argument above now gives the inequality

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

For $c_0 = n$ this gives the claimed solution. □

THEOREM 2. $\text{Max}_n(n) \leq \text{loglog } n + \text{const.}$

PROOF. Using the same terminology as in the previous theorem we now observe that an algorithm for finding the maximum can be developed using the above described optimal graphs.

At step i+1, the algorithm calls on the optimal graph with c_i nodes that has the largest number of arcs no more than n. Sets of z and z-1 elements are allocated accordingly to the processors, so that within each set every element will be compared with every other at the next step. For each set, the element that is found to be larger than all the others will be the only one to be a candidate at the next step.

The constraints on the total number of nodes and arcs now give the inequalities

$$c_{i+1} \cdot (z-1) \leq c_i \quad \text{and} \quad c_{i+1} \cdot z(z-1)/2 \geq n \, .$$

from which the relation

$$c_{i+1} \leq \frac{c_i^2}{n.\text{const.}} \qquad (c_i > 1)$$

can be deduced. Solving for $c_0 = n$ gives that $c_i = 1$ for some $i \leq \text{loglog } n + \text{const.}$

COROLLARY 2. For $k \geq n$,

$$\text{Max}_k(n) \leq \text{loglog } n - \text{loglog}(k/n) + \text{const.}$$

PROOF. The algorithm is essentially the same except that it starts with $c_0 = n$, which may be already smaller than k. □

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 $c_i \geq 2k$, we can achieve this reduction by having k disjoint pairs from C_i compared at each time interval. This therefore gives an algorithm that reduces c_i optimally at each step until it becomes less than 2k. Since the algorithm of Theorem 2 can then take over, we conclude the following.

COROLLARY 3. For $k < n$, upper and lower bounds for $\text{Max}_k(n)$ can be obtained, both of the form

$$n/k + \text{loglog } k + \text{const.} \qquad □$$

For each case we have therefore 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 the analysis. We would need to construct the optimal graphs that are to be used at the successive steps of the algorithm. This can be done by examining the constraints used above to obtain the main inequalities and gives that

$$c_{i+1} = \min\{x \mid \lfloor c_i/x \rfloor . (c_i - x + c_i \bmod x)/2 \le k\}, \text{and}$$
$$z = \lceil c_i/c_{i+1} \rceil .$$

For the special case of $k = n$ we can expre ss the exact result that is implied as follows.

COROLLARY 4. The sequence $s_0, s_1, \ldots$ such that s_i is the largest integer s.t.

$$\mathrm{Max}_{s_i}(s_i) = i$$

is defined by

$$s_{i+1} = (2s_i+1)s_i$$

where $s_0 = 1$. There is a real number K such that for $i \ge 1$,

$$s_i = \lfloor K^{2^i}/2 \rfloor .$$

PROOF. The case when cliques of equal size are produced at each step, with no arcs left over, is given by $s_i = t_i(t_i-1)/2$ where

$$t_i = 2 \prod_{j=0}^{i-1} t_j + 1$$

and $t_0 = 1$. D. E. Knuth has pointed out to the author that this reduces to the given recurrence, and that the form of the solution of the latter can be explicitly expressed, as shown, using the analysis of Aho and Sloane [7]. □

MERGING

We now give an algorithm for merging that is much faster than the corresponding ones previously known.

THEOREM 3. For $k = \lfloor\sqrt{(mn)}\rfloor$ and $n \le m$,

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

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 $X = (x_1,x_2,\dots x_n)$, $Y = (y_1,y_2,\dots y_m)$.

(a) Mark the elements of X that are subscripted by $i\lceil\sqrt{n}\rceil$ and those of Y subscripted by $i\lceil\sqrt{m}\rceil$ for $i = 1,2,\dots$. 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 merged. 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

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

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),... where each X_i is a segment of X and therefore of length $|X_i| \le \lfloor\sqrt{n}\rfloor$. Furthermore $\Sigma|X_i| < n$ and $\Sigma|Y_i| < m$ since the sublists are disjoint. But by Cauchy's inequality [4],

$$\Sigma\sqrt{(|X_i|.|Y_i|)} \le \sqrt{(\Sigma|X_i|.\Sigma|Y_i|)}.$$

It follows that

$$\Sigma\lfloor\sqrt{(|X_i|.|Y_i|)}\rfloor \le \Sigma\sqrt{(|X_i|.|Y_i|)} \le \lfloor\sqrt{(mn)}\rfloor.$$

There are therefore enough processors altogether that we can assign $\lfloor\sqrt{(|X_i|.|Y_i|)}\rfloor$ 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 λ_i, where

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

and $\lambda_0 = n$. Solving $\lambda_i \le \sqrt{\lambda_{i-1}}$ gives $\lambda_i < n^{1/2^i}$. 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

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

where the constant is less than unity if the logarithms are to the base 2.

COROLLARY 5. For $k = \lfloor r\sqrt{nm}\rfloor$ where $n \le m$ and $r \ge 1$,

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

PROOF. We use the same algorithm as above, except that at step (a) the objects marked are those subscripted by $i\lceil\sqrt{(n/r)}\rceil$ in X and by $i\lceil\sqrt{(m/r)}\rceil$ in Y for $i = 1,2,\ldots$. 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 $k \le n \le m$,

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

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. If done independently on separate processors, this will require time log(m/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 [5], 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 list.

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 <u>segments</u> 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. $\square$

This last corollary is an improvement on one described in [3] (and attributed to Kirkpatrick) for the case $k \ll n = m$. Asymptotically a speedup of k is clearly achieved, since it is known [5] that the merging of two lists of length n requires $2n-1$ comparisons in the worst case. In [3] the method suggested for alleviating the scheduling problem is that of initially marking not k but some function of n (say $\sqrt{n}$) elements. Though this will be slower than our algorithm, asymptotically it still has optimal speedup. Furthermore it enables one to deduce that even in the general case of $m \neq n$, optimal asymptotic speedup can be theoretically attained if use is made of optimal sequential merging algorithms yet unknown.

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:

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

We now derive a corresponding upper bound.

THEOREM 4. $\mathrm{Sort}_{n/2}(n) \leq 2\log n \log\log n + O(\log n)$.

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

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

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 clearly applies in that case as well. □

COROLLARY 6. For $k \geq n$,

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

PROOF. With k processors we can split the input into sets of size $[k/n]$ 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 7. For $k < n$,

$$\mathrm{Sort}_k(n) \le (n \log n + O(n))/k.$$

PROOF. As in [3] 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. □

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. Our analysis of complexity bounds for this model we suggest as part of the theoretical background against which parallelism for these problems can be studied and exploited. In practice, to derive good algorithms suitable for a specific multiprocessor machine, additional considerations have, of course, 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 [2], but cannot be solved in less than time loglog 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 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.

REFERENCES

1. BATCHER, K. E., Sorting networks and their applications, 1968 Proc. AFIPS SJCC, 32, 307-314.

2. BLUM, M., FLOYD, R. W., PRATT, V., RIVEST, R. L., and TARJAN, R. E., Time bounds for selection, JCSS 7, 448-461 (1973).

3. BORODIN, A. B., and MUNRO, I., Notes on "Efficient and Optimal Algorithms," (1972).

4. CAUCHY, A. L., Cours d'analyse de L'Ecole Royale Polytechnique, 1re partie, Analyse algébrique, Note II, Paris 1821. (Oeuvres complètes, IIe série, III).

5. KNUTH, D. W., The art of computer programming, vol. 3, Addison-Wesley (1973).

6. TURAN, P., On the theory of graphs, Colloq. Math., 3, 19-34 (1954).

7. AHO, A.V. and SLOANE, N.J.A., Some doubly exponential sequences, Fibonacci Quarterly, II:4, 429-437 (1973).