Applying Parallel Computation Algorithms in the Design of Serial Algorithms

NIMROD MEGIDDO

Tel Aviv University, Tel Aviv, Israel

Abstract. The goal of this paper is to point out that analyses of parallelism in computational problems have practical implications even when multiprocessor machines are not available. This is true because, in many cases, a good parallel algorithm for one problem may turn out to be useful for designing an efficient serial algorithm for another problem A unified framework for cases like this is presented. Particular cases, which are discussed in this paper, provide motivation for examining parallelism in sorting, selection, minimum-spanning-tree, shortest route, max-flow, and matrix multiplication problems, as well as in scheduling and locational problems.

Categories and Subject Descriptors: F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*relations among models*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; *geometrical problems and computations*; *routing and layout*; *sequencing and scheduling*; *sorting and searching*; G.2.1 [**Discrete Mathematics**]. Combinatorics—*combinatorial algorithms*; G 2 2 [**Discrete Mathematics**]: Graph Theory—*network problems, path and circuit problems*; *trees*

General Terms· Algorithms, Theory

Additional Key Words and Phrases: Parallel algorithms, parametric computation, spanning tree, scheduling, min-ratio cycle, algorithms on trees, max-flow-related problems

1. *Introduction*

Numerous models have been proposed for parallel computation, and it is not clear yet which of them will eventually be supported by technology. Thus some of the models may at present be criticized as being unrealistic, since issues like synchronization and memory conflicts still have to be settled. In this paper, however, we provide a new motivation for studying such models. An early model of parallel computation, considered by Valiant [37], now seems unrealistic since it only counts comparisons as time-consuming operations and also assumes synchronization. Valiant does obtain very nice theoretical results with respect to sorting, merging, and maximum finding in parallel. On the other hand, in this paper we show that the search for good parallel algorithms of Valiant's type is justified on *practical* grounds

Parts of this work were done while the author was Visiting Scientist of the National Research Institute for Mathematical Sciences, CSIR, Pretoria, South Africa and at Northwestern and Carnegie-Mellon Universities.

This work was partially supported by the National Science Foundation under grants ECS7909724, ECS8121741, SOC7905900 and by a grant from Control Data.

Author's present address: Department of Computer Science, Stanford University, Stanford, CA 94305.

regardless of the state of technology for parallel computation. This is because, in many cases, one may desire to have a good parallel algorithm for one problem in order to design a good serial algorithm for another problem. In this paper we describe a fairly large class of problems in which this phenomenon occurs.

The general idea is as follows. Suppose that a certain problem A is solved in T time units on a P-processor machine. In a serial implementation this amounts to TP time. Within the scope of serial computation we only wish to minimize the product TP. The common examination of parallelism in problem A amounts, essentially, to minimizing T, subject to keeping P at a reasonably low level. Now, suppose that an algorithm for problem A could somehow be applied to designing an algorithm for another problem B, so that a factor like $T \log P$ dominates the (serial) time bound for problem B, provided P is not too large. This would certainly justify an examination of parallelism in problem A, since minimizing $T \log P$, subject to P's not being too large, is very close to minimizing T. In a study of parallelism, P in many cases depends on the input length n, so that $n^{\epsilon} \leq P \leq n^{k}$ for some $\epsilon > 0$ and an integer k. In other words, we have $\log P \sim \log n$ in many cases.

In this paper we demonstrate situations like the one mentioned above. We start here with an abstract example. A more concrete example is considered in the next section. Suppose that $F(\lambda)$ is a monotone function of the real variable λ and problem A is to evaluate F at a given λ. Suppose that problem B is to solve the equation $F(\lambda) = 0$. Assume that the variable λ is involved throughout the evaluation of F only in additions, comparisons, and multiplications by constants. Then, as we demonstrate in the subsequent sections, in order to design a good serial algorithm for solving the equation $F(\lambda) = 0$, it is desirable to have a good parallel algorithm for evaluating F.

Particular cases of the general problem $F(\lambda) = 0$, which are discussed later, are minimax or maximin problems and ratio optimization problems. Another important type of problem is finding the maximum or minimum value of a parameter λ for which a certain property holds. We discuss examples of this kind as well.

In the next section we describe a relatively simple example which is aimed at explaining the general idea rather than presenting an optimal algorithm. Yet the subsequent sections present cases in which the method does improve upon existing algorithms and also motivates a more thorough study of parallelism. We claim that almost every combinatorial problem in which numbers are involved should be examined from the parallelism point of view, since this would probably lead to good algorithms for related problems.

The idea suggested in [23] has been used for solving problems taken from different fields (such as networks, scheduling, location, geometry, and statistics) [6, 7, 15–17, 21, 26–29]. Some of these papers already utilize the "parallel processing" principle as suggested in the preliminary version of the present paper [25]. We expect, on the one hand, additional applications to be discovered soon and, on the other hand, further improvements in the efficiency of the algorithms as a result of developments in the lively field of parallel computation.

2. *A Preliminary Example*

We start with a relatively simple example that demonstrates the general method.

Let $f_i(\lambda) = a_i + \lambda b_i$, $i = 1, \ldots, n$, be pairwise distinct functions of λ with all b_i's positive. For every λ, let $F(\lambda)$ denote the median of the set $\{f_1(\lambda), \ldots, f_n(\lambda)\}$. Obviously $F(\lambda)$ is a piecewise-linear, monotone-increasing function with $O(n^2)$ breakpoints. Given λ, $F(\lambda)$ can be evaluated in $O(n)$ comparisons [1] (once the $f_i(\lambda)$'s have been computed). Consider the equation $F(\lambda) = 0$. One possible way of solving this equation is first to identify the intersection points λ_{ij}, where $a_i + \lambda_{ij}b_i = a_j + \lambda_{ij}b_j$ $(i \neq j)$. Every breakpoint of F is equal to one of the λ_{ij}'s. We can thus search the set of λ_{ij}'s for two values λ^1, λ^2 such that $F(\lambda^1) \leq 0 \leq F(\lambda^2)$ and such that there is no λ_{ij} in the open interval (λ^1, λ^2). Once λ^1 and λ^2 have been found, we can solve the equation in one step since F is linear over $[\lambda^1, \lambda^2]$. The search for such an interval requires $O(\log n)$ F-evaluations and (if we do not wish to presort the set of λ_{ij}'s) finding medians of subsets of the set of λ_{ij}'s whose cardinalities are n, $n/2$, $n/4, \ldots$. Thus, such an algorithm runs in $O(n^2)$ time, which is dominated by the evaluation of all the intersection points λ_{ij}.

An alternative approach suggested by the author in a previous paper [23] is as follows. Let us start the evaluation of $F(\lambda)$ with λ not specified. Denote the solution of the equation $F(\lambda) = 0$ (which is, of course, not known at this point) by λ^*. The outcome of the first comparison, performed by the algorithm for evaluating F, depends, of course, on the numerical value of λ. Specifically, if our median-finding algorithm for evaluating F starts with comparing $f_1(\lambda)$ and $f_2(\lambda)$, then the intersection point λ_{ij} of these two lines is a critical value for this comparison; namely, for $\lambda \geq \lambda_{12}$, $f_1(\lambda) \geq f_2(\lambda)$, whereas for $\lambda \leq \lambda_{12}$, $f_1(\lambda) \leq f_2(\lambda)$, or vice versa. Thus, we may find λ_{12}, evaluate $F(\lambda_{12})$, and then decide whether $\lambda^* \geq \lambda_{12}$ or $\lambda^* \leq \lambda_{12}$ according to the sign of $F(\lambda_{12})$. We can then proceed with the evaluation of $F(\lambda)$, where λ is still not specified but is now restricted either to $(-\infty, \lambda_{12}]$ or to $[\lambda_{12}, \infty)$. The same idea is repeatedly applied at the following points of comparisons. In general, when we need to compare f_i with f_j and λ is currently restricted to an interval $[\lambda', \lambda'']$, if λ_{ij} does not lie in the interval, then the outcome of the comparison is uniform over the interval; otherwise, by evaluating $F(\lambda_{ij})$, we can restrict our interval to either $[\lambda', \lambda_{ij}]$ or $[\lambda_{ij}, \lambda'']$. Since $O(n)$ such comparisons have to be performed and each may require an evaluation of F (which amounts to one median-finding), it follows that such an algorithm runs in $O(n^2)$ time.

At this point we are ready to see the role of parallelism. In the latter algorithm we were running a "master" median-finding algorithm where at every comparison point we had to test the sign of $F(\lambda)$ at a certain critical value of λ. Now, suppose that instead of using the linear-time median-finding algorithm [1], we employ a parallel sorting algorithm such as Valiant's [37] or Preparata's [32]. Let P denote the number of processors and let $T(n, P)$ denote the number of comparison steps required on a P-processor machine. Preparata's scheme uses $P = n \log n$ and achieves $T(n, P) = O(\log n)$. A suitable special case of Valiant's scheme uses $P = n$ and achieves $T(n, P) = O(\log n \log\log n)$. We, of course, simulate these algorithms serially; that is, we employ one "processor" at a time, according to some fixed permutation, letting each perform one step in every cycle. Thus, all potential difficulties related to synchronization and memory conflicts are not present in our analysis. Recall that we are trying to sort the set $\{f_1(\lambda^*), \ldots, f_n(\lambda^*)\}$, where λ^* is not known; however, for any λ it can be decided in $O(n)$ time whether $\lambda^* \geq \lambda$ or $\lambda^* \leq \lambda$. When two functions f_i, f_j have to be compared, the equation $f_i(\lambda) = f_j(\lambda)$ is solved and the solution λ_{ij} is declared a critical value. After one step of the "multiprocessor" we are provided with P such critical values. The crucial point is that these values are produced independently of each other since the processors work "in parallel." Thus, the kth processor does not have to know the outcome of the comparison for which processor $k - 1$ is responsible, and, therefore, the critical value produced by processor $k - 1$ does not have to be tested before processor k produces its own critical value. We start the testing only when all the P processors have produced critical values. Less informally, let these critical values be $\lambda_1 \leq \cdots \leq \lambda_P$; we do not assume that these values are sorted. It can be seen that within $O(\log P)$ F-evaluations, we can identify an interval $[\lambda_i, \lambda_{i+1}]$ ($i = 0, 1, \ldots, P$; assuming $\lambda_0 = -\infty$, $\lambda_{P+1} = \infty$) such that $\lambda_i \leq \lambda^* \leq \lambda_{i+1}$. This is done by a binary search in the set of critical values, namely, testing the median of the set and then one of the quartiles, etc. Once the parameter λ is restricted to the intersection of this interval with the previous interval of λ, we can proceed since the outcomes of all the comparisons so far are known for λ's in the new interval. This repeats $T(n, P)$ times until we arrive at an interval over which our functions do not intersect and the median function crosses the zero level. Then the equation can be solved directly.

During each such "step" we evaluate P critical values and then perform a binary search that requires $O(P)$ time for median-findings in subsets of the set of critical values, as well as $O(\log P)$ F-evaluations. These evaluations require $O(n \log P)$ time if the linear-time median-finding algorithm is used to test a fixed value of λ. There is also some additional overhead time which is, however, dominated by $n \log P$. The proof of this is left to the reader. We remark that similar arguments regarding the overheads have to be made for any parallel algorithm in Valiant's comparison model. We should also remark that recently Borodin and Hopcroft [4] showed that results similar to Valiant's and Preparata's sorting algorithms hold in parallel models of computation that take all overheads into account. Thus the total amount of serial time required is $O((P + n \log P)T(n, P))$. Preparata's scheme thus yields an $O(n(\log n)^2)$ bound, while Valiant's yields $O(n(\log n)^2 \log\log n)$, both bounds being superior to the previous $O(n^2)$.

We note that our simple problem can be solved directly by finding the median of the set of values of λ at which some function crosses the zero level, that is, the set $\{-a_1/b_1, \ldots, -a_n/b_n\}$. However, we present this example just as a simple demonstration of a general principle. In the subsequent sections we sketch some more complicated examples where our method leads to the best known bounds.

3. *Spanning Tree Problems*

Consider a graph with edge-weights w_e that are themselves linear functions of a real variable: $w_e = w_e(\lambda) = a_e + \lambda b_e$. We assume that the b_e's are either all nonnegative or all nonpositive. Let $F(\lambda)$ denote the weight of the minimum spanning tree relative to the weights $w_e(\lambda)$. The minimum-ratio spanning-tree problem [5] can be formulated as that of solving the equation $F(\lambda) = 0$. Another related problem involving a parametrized spanning tree can be described as follows. Suppose that a spanning tree has to be constructed and the construction takes place in two time periods. Let a_e denote the cost of work to be done on arc e during the first period (if the arc e is indeed selected for the spanning tree). Let b_e denote the *estimated* cost of work to be done on arc e during the second period. All the costs of the type b_e are subject to an increase by a currently unknown factor of λ. Given a budget B, we wish to decide which spanning tree to construct so as to allow for a maximum cost increase that would still leave the total construction cost not greater than B.

The author [23] gave an algorithm for this problem which runs in $O(E(\log V)^2 \log\log V)$ time, where E and V are the numbers of edges and vertices, respectively, in the graph. That algorithm exploited the relatively high degree of parallelism involved in Sollin's algorithm [3] for the minimum-spanning-tree problem. There, the application of Sollin's algorithm was not exactly the same as suggested in the present paper. Since Sollin's algorithm runs in $O(V \log V)$ time on a V-processor machine, it follows that it can solve the equation $F(\lambda) = 0$ in $O((V + T_F \log V) V \log V)$ time, where T_F is the amount of time required for a single minimum-spanning-tree computation. There are several ways [8, 38] to achieve $T_F = O(E \log\log V)$ so that we obtain a bound of $O(EV(\log V)^2 \log\log V)$ for our problem. This is inferior to the previous bound.

We now describe an even simpler approach, which is also based on parallel computation, yielding the $O(E(\log V)^2 \log\log V)$ time bound. It is well known that the minimum-spanning-tree solution depends only on the linear order which is induced on the set of edges by the weights. As a function of the variable λ, this linear order changes only at values of λ where at least two edges have equal weights. There are $O(E^2)$ such values of λ. By finding all of them in advance (as suggested by Chandrasekaran [5]), we may search for λ^* by employing a minimum-spanning-tree algorithm $O(\log V)$ times. This implies a time bound of $O(E^2)$. Now, suppose that we start with a parallel sorting algorithm, repeatedly restricting the interval containing λ^*, until the edges are sorted by the $w_e(\lambda^*)$'s. Specifically, apply Preparata's sorting scheme with $E \log V$ processors to the set of $w_e(\lambda)$'s. The process works in $O(\log V)$ phases. During a single phase, $O(E \log V)$ critical values are produced and then a binary search, consisting of $O(\log V)$ minimum-spanning-tree evaluations, is performed. This implies a time bound of $O((E \log V + (E \log\log V)\log V)\log V))$, which is simply $O(E(\log V)^2 \log\log V)$. Given the correct permutation of the edges, we can find λ^* as well as the corresponding tree.

4. *A Scheduling Problem*

In the present section we describe another application of parallel sorting algorithms. Consider a one-machine scheduling problem with n tasks. Denote the processing times by $t_1, \ldots, t_n$, the finishing times (depending on the schedule) by $f_1, \ldots, f_n$, and the weighted mean-flow time by $W = \sum c_i f_i$, where $c_1, \ldots, c_n$ are the deferral costs [9]. Suppose we seek to schedule a maximal fixed intermission between every two consecutive tasks, subject to the condition that the weighted mean-flow time will not be greater than a given bound b.

If the length of the intermission is λ, then the weighted mean-flow time is minimized by processing the tasks in order of increasing magnitude of $(t_i + \lambda)/c_i$. Let $F(\lambda)$ denote the minimum value when the length of the intermission is λ. The function F is piecewise linear monotone increasing and we have to solve $F(\lambda) = b$. The breakpoints of F are at values of λ where $(t_i + \lambda)/c_i = (t_j + \lambda)/c_j$ for some $i \neq j$.

One possible way of solving the problem, as in the case of the spanning tree, is to evaluate all the intersection points in $O(n^2)$ time and then to search for λ^* by testing $O(\log n)$ values of λ. Each test requires sorting of n ratios and hence takes $O(n \log n)$ time.

An improvement upon the easily achieved $O(n^2)$ bound is obtained by applying Preparata's sorting scheme. In this case we have $O(\log n)$ phases. During each phase we evaluate $n \log n$ critical values of λ and then search for λ^*. The search involves $O(\log n)$ F-evaluations, each requiring $O(n \log n)$ time. Thus, the problem is solvable in $O((n \log n + (n \log n)\log n)\log n) = O(n(\log n)^3)$ time.

E. Lawler [21] has applied the general method presented in this paper to several other scheduling problems.

5. *Cost-Effective Resource Allocation*

In this section we mention one more application of parallel sorting. A more detailed discussion of the example of this section is given elsewhere [24].

The problem is to maximize[1] $(\sum f_i(x_i))/(\sum g_i(x_i))$ subject to $\sum x_i \leq k$ and the x_i's

[1] Summation in the present section is over $i = 1, \ldots, n$

being nonnegative integers. The f_i's are concave and nonnegative, while the g_i's are convex and positive. For solving this problem we look at the regular resource allocation problem; that is, maximize $\sum u_i(x_i)$ subject to the same constraints, where the u_i's are concave. Defining $F(\lambda)$ to be the maximum of the regular problem with $u_i(x_i) = f_i(x_i) - \lambda g_i(x_i)$, $i = 1, \ldots, n$, we need to solve the equation $F(\lambda) = 0$. The regular problem has been studied for a long time and fast algorithms, based on selection in a set with presorted subsets, have been given by Frederickson and Johnson [12] and Galil and Megiddo [14]. For solving the cost-effective problem, one desires to have a good parallel algorithm for the main selection routine of the regular resource allocation problem. On the basis of Valiant's algorithms and Frederickson and Johnson's work, the author [24] has given an algorithm which runs in $O(n(\log n)^2(\log k)^2 \log\log n)$ time. In this case, an interesting point is that *the number of "processors" may vary throughout*. In particular, when k is very large, it becomes beneficial to use large numbers of processors, even like $P = n(n-1)/2$, which enable "sorting" in one time unit. Thus, an even smaller bound may be obtained when k is relatively large. Also, the $\log\log n$ factor may be eliminated if Preparata's scheme is applied. Incidentally, the bound is sublinear in the input length, which is $O(nk)$, if the functions f_i, g_i are given in tabular form. The algorithm actually uses only a small fraction of the function values, which may be supplied interactively or computed by another routine.

6. *Minimum Ratio Cycle*

In this section we describe an application of parallel algorithms for the all-pair shortest paths problem.

The minimum-ratio-cycle problem is to find a cycle in a network with edge-costs and edge-times, such that the ratio of the total cost to the total time of edges on the cycle is minimized. This problem has been considered by several authors [10, 20, 23], and the best previously known bound was [23] $O(EV^2 \log V)$. This was achieved as follows. Define $F(\lambda)$ to be the length of the shortest cycle relative to the distance function that assigns to every edge a length that equals the cost of the edge less λ times the time of the edge. Then, solve the equation $F(\lambda) = 0$. Essentially, this calls for a *negative-cycle detector*, that is, an algorithm for deciding whether a network contains a cycle of negative length. An algorithm by Karp [18], which runs in $O(EV)$ time, was applied by the author [23], and some part of its parallelism was exploited to yield the $O(EV^2 \log V)$ bound.

In this paper we improve the latter bound in two different ways by further exploiting parallelism. In the following procedures we use algorithms for the renowned all-pair shortest paths problem as negative-cycle detectors. In other words, we rely on the obvious observation [20] that a network contains a negative cycle if and only if there is a vertex i such that the distance from i to itself is negative.

Let π_{ij}^k denote the length of the shortest of all paths from i to j with no more than k intermediate vertices. The existence of a negative cycle is thus equivalent to the existence of a negative diagonal entry in the matrix $\pi^{V-1} = (\pi_{ij}^{V-1})$. The analogy between matrix multiplication and the computation of π_{ij}^k is well known [1]. The numbers π_{ij}^0 are of course given. The recursive relation $\pi_{ij}^{2k+1} = \min_r\{\pi_{ir}^k + \pi_{rj}^k\}$ can be used for finding all the values π_{ij}^k where k is of the form $k = 2^s - 1$. It thus enables us to detect a negative cycle in $O(\log V)$ steps which are analogous to matrix squaring. This procedure has a high degree of parallelism which is exploited later.

When the edge-lengths are themselves linear functions of λ, the quantities π_{ij}^k are piecewise-linear functions of λ. However, throughout the computation, λ always belongs to an interval (which is repeatedly updated) over which those $\pi_{ij}^k(\lambda)$'s currently under consideration are linear. We now describe two different approaches to applying the algorithm implicit in the above-mentioned recursive formula for π_{ij}^{2k+1}.

Approach 1: V^2 processors. We associate a processor with each ordered pair of vertices (i, j). For a fixed k, all the π_{ij}^{2k+1} values are determined "simultaneously." The processor associated with the pair (i, j) determines all the breakpoints of the minimum function $\pi_{ij}^{2k+1}(\lambda) = \min_r\{\pi_{ir}^k(\lambda) + \pi_{rj}^k(\lambda)\}$ over the current interval. We now rely on the fact that, given V linear functions $f_r(\lambda)$, $r = 1, \ldots, V$, the breakpoints of the minimum function $f(\lambda) = \min_r f_r(\lambda)$ can all be found in $O(V \log V)$ time. A detailed algorithm is given in [23, Appendix] and is based on sorting the functions by their slopes. The processor associated with (i, j) can therefore find all the breakpoints of $\pi_{ij}^{2k+1}(\lambda)$ in $O(V \log V)$ time, since all the $\pi^k(\lambda)$'s are linear over the current interval. Thus, the total serial time for this step is $O(V^3 \log V)$. At most $O(V^3)$ breakpoints are produced during this step and now a binary search takes place, which amounts to $O(\log V)$ negative-cycle detections. A single detection required $O(EV)$ time [18] and hence the search runs in $O(EV \log V)$ time. Since the whole process consists of $O(\log V)$ steps like this, it follows that the algorithm runs in $O(V^3(\log V)^2)$ time.

Approach 2: $V^2(V - 2) + V$ processors. In this case we associate a processor with each triple (i, j, r) such that $r \neq i, j$. The fundamental iteration $\pi_{ij}^{2k+1} = \min_r\{\pi_{ir}^k + \pi_{rj}^k\}$ is, again, carried out simultaneously for all (i, j). However, here the minimum corresponding to each (i, j) is determined by a parallel algorithm with the $V - 2$ processors associated with the triples (i, j, r), $r \neq i, j$, in case $i \neq j$ ($V - 1$ processors if $i = j$). Valiant's parallel algorithm for finding the minimum performs $O(\log \log V)$ comparisons. Recently Shiloach and Vishkin [33] developed an algorithm which finds the minimum in $O(\log \log V)$ time, taking all overheads into account. Thus, a single step (i.e., the computation of all π_{ij}^{2k+1} for a fixed k) runs in $O(\log \log V)$ stages. During a typical stage, each processor makes one comparison, that is, it produces the critical value of λ with respect to that comparison. Then a binary search takes place in the set of critical values. The binary search requires $O(\log V)$ negative-cycle detections. Thus, a single stage takes $O(V^3 + EV \log V)$ time and, since we have $O(\log V)$ steps, each consisting of $O(\log \log V)$ stages, the overall time bound with this approach is $O((V^3 + EV \log V)\log V \log \log V)$. One of the referees of this paper has suggested the following improvement. Allocate $V/\log \log V$ processors for each pair so that the minimum can be found in $O(\log \log V)$ parallel time. This yields a serial time bound of $O(V^3 \log V + EV \log^2 V \log \log V)$.

We should also note that under an infinite precision multiplication model Yuval [39] finds the all-pair shortest distances in $O(V^{2.81})$ time. Thus, under this model a negative cycle can be detected in $O(V^{2.81})$ time, so that the second approach leads to an $O(V^3 \log V \log \log V)$ algorithm.

7. *Algorithms on Trees*

Many problems that are NP-hard on general graphs turn out to be polynomially solvable on trees. Among these are most of the location-theoretic problems. However, in most of the cases the tree algorithms do not have a high degree of parallelism. In particular, if an algorithm works from the leaves of the tree toward the root, or vice versa, then its obvious parallel version runs in time which is at least proportional to the radius of the tree. The radius may of course be linear in the size of the tree, while it is conceivable that logarithmic-time parallel algorithms exist. As a representative case we discuss here the max-min tree k-partitioning problem by Perl and Schach [31]. Another problem which has been successfully attacked by the same method is the continuous p-center problem on a tree [29].

The max-min tree k-partitioning problem is formulated as follows. Given a tree T with n edges and a nonnegative weight associated with each vertex, delete k edges of T so as to maximize the weight of the lightest of all the resulting subtrees (formed by the remaining edges). To follow the general framework of the present paper, define $F(\lambda)$ to be the maximal number of edges that can be deleted, so that the weight of every subtree is at least λ. We are interested in finding the maximal λ such that $F(\lambda) \geq k$, which is essentially solving the equation $F(\lambda) = k$. As pointed out by Perl and Schach, it takes $O(n)$ time to evaluate F at a given λ. Specifically, the following rules establish such an algorithm. (i) If a leaf has a weight greater than or equal to λ, then its linking edge should be deleted. (ii) If the "sons" of a certain vertex v are all leaves with weight less than λ, then replace v and its sons by a new vertex v' whose weight equals the total weight of v together with the sons. It is straightforward to establish an n-processor parallel version of this algorithm which runs in $O(\mathrm{rd}(T))$ time, where $\mathrm{rd}(T)$ is the radius of T. A processor is linked with each leaf. A typical step is as follows. Each processor provides the weight of its respective leaf as a critical value of λ. Also, if v is a vertex all of whose sons are leaves, then the total weight of these sons is considered a critical value of λ at this point. It takes $O(\log n)$ time for $O(n)$ processors to establish $O(n)$ such critical values of λ during a single step. Then a binary search for λ^* is performed over the set of critical values, which requires $O(\log n)$ F-evaluations. This repeats $O(\mathrm{rd}(T))$ times and hence we have an algorithm which runs in $O(n \log n \,\mathrm{rd}(T))$ time. This algorithm, which is a straightforward adaptation of our general method, is already better than Perl and Schach's $O(k^2\,\mathrm{rd}(T) + kn)$ for certain values of k. We later, however, provide an even more efficient algorithm. Also, note that by a more careful analysis of our previous algorithm, a bound of $O(n \log(n/\mathrm{rd}(T))\mathrm{rd}(t))$ can be proved. This is because if n_i is the number of "processors" required during step i ($i = 1, \ldots, \mathrm{rd}(T)$), then $\sum n_i = n$ and the algorithm actually runs in $O(n \sum \log n_i)$ time.

As pointed out earlier, the key to improving the bound even further is by developing a parallel algorithm for evaluating $F(\lambda)$ in $o(n)$ time. We shall describe an $O(\log n)$ parallel algorithm for the special case of partitioning a sequence of numbers (i.e., a *linear tree*). This algorithm will then be employed to establish an $O((\log n)^2)$ algorithm for general trees.

Let $(a_1, \ldots, a_n)$ be a sequence of real numbers, and let $F(\lambda)$ denote the maximal number of subsequences in a partition $(a_1, \ldots, a_{i_1})$, $(a_{i_1+1}, \ldots, a_{i_2})$, $\ldots, (a_{i_r+1}, \ldots, a_n)$, where each subsequence total is at least λ. If all the partial sums $A_i = a_1 + \cdots + a_i$ ($i = 1, \ldots, n$) are known, then it takes $O(\min(n, k \log n))$ time to decide (for a given λ) whether or not $F(\lambda) \geq k + 1$. Thus, even without exploiting parallelism in the computation of $F(\lambda)$, we may solve the max–min problem in $O(\min(n^2, n + k^2(\log n)^2)$ time in the following way. First, compute all the partial sums. Then, run that algorithm for $F(\lambda)$, testing every critical value encountered.

A good parallel algorithm for evaluating F is as follows. First, evaluate all the A_i's, recursively, in $O(\log n)$ time. Specifically, once we obtain A_i, $i = 1, \ldots, [n/2]$ and $A_i - A_{[n/2]}$ for $i = [n/2] + 1, \ldots, n$, then it takes just one step to find all the A_i. The second phase consists of finding, for every i ($i = 0, 1, \ldots, n - 1$), an index $j(i)$ such that $A_{j(i)-1} - A_i < \lambda \leq A_{j(i)} - A_i$ ($A_0 = 0$; if $A_n - A_i < \lambda$, then define $j(i) = \infty$). This is easily carried out in $O(\log n)$ time by letting each processor independently perform a binary search for $j(i)$ of a different i. Finally, we compute for every i, $i = 0, 1,$

$\ldots, n-1$, two indices $s(i)$, $k(i)$, which are defined as follows. If $a_{i+1} + \cdots + a_n < \lambda$, then let $s(i) = i$ and $k(i) = 0$; otherwise, let $s(i) = s(j(i))$ and $k(i) = k(j(i)) + 1$. Intuitively, $k(i)$ is the maximal number of subsequences in a partition of $(a_{i+1}, \ldots, a_n)$ such that every subsequence total is at least λ, while $s(i)$ is the minimal index such that $(a_{i+1}, \ldots, a_{s(i)})$ can also be partitioned into $k(i)$ subsequences subject to the same constraint. We are interested only in $k(0)$, but all other values are required for an efficient parallel computation. Let $s'(i)$ and $k'(i)$ denote the same type of indices, determined relative to the sequence $(a_1, \ldots, a_{[n/2]})$. Recursively, first find $s'(i)$ and $k'(i)$ for $i = 0, 1, \ldots, [n/2]$ and $s(i)$ and $k(i)$ for $i = [n/2], \ldots, n$. Now, for $i = 0, 1, \ldots, [n/2]$, set $s(i) = s(j(s'(i)))$ and $k(i) = k'(i) + 1 + k(j(s'(i)))$ if $j(s'(i)) \neq \infty$, and set $s(i) = s'(i)$ and $k(i) = k'(i)$ if $j(s'(i)) = \infty$. The inductive step requires constant time on $[n/2]$ processors, and hence the entire computation takes only $O(\log n)$ time.

The resulting time bound for max-min sequence partitioning is as follows. We run in $O(\log n)$ steps, each of which requires $O(\log n)$ tests. The cost of a test is $O(\min(n, k \log n))$ and we have n processors. Thus our bound is $O((n + \min(n, k \log n)\log n)\log n)$. On the other hand, the "correct" λ^* is of the form $A_j - A_i$. Since selection in a set of the form $X - X$ can be carried out in $O(n \log n)$ time [12], it follows that a direct binary search in the set of $(A_j - A_i)$'s yields an $O(n(\log n)^2)$ bound. In fact, the special structure of this set implies that an $O(n \log n)$ algorithm can be constructed following the ideas of Frederickson and Johnson [13]. These, however, do not seem to generalize to general trees. This is because the sets A_i generalize to total weights of subtrees and hence the number of potential values for λ grows exponentially.

We now consider the general case of a tree. We shall develop an $O((\log n)^2)$ parallel algorithm for partitioning a tree into a maximal number of components so that every component weighs at least λ. Let the tree be rooted at an arbitrary vertex u. First, consider the recursive function $g(v)$, defined on the vertices. If v is a leaf, then let $g(v) = w_v$ if $w_v < \lambda$ and let $g(v) = 0$ if $w_v \geq \lambda$. If v is not a leaf and $v_1, \ldots, v_s$ are the sons of v, then let $g(v) = w_v + \sum g(v_i)$ if the right-hand side is less than λ; otherwise, let $g(v) = 0$. Intuitively, if $g(v) = 0$, then the edge linking v to its father is deleted (except for one case discussed later); otherwise, $g(v)$ is the "residue" of weight which v passes to its father. At the root u, if $g(u) = 0$, then the deleted edges partition the tree properly; otherwise, the root is currently contained in a component whose total weight is less than λ, while all other components weigh at least λ. In the latter case we restore one of the previously deleted edges so that the component containing the root unites with another component. Our algorithm will compute g at all vertices in $O((\log n)^2)$ parallel time. The algorithm can easily keep track of the deleted edges.

Our algorithm utilizes the notion of "centroid decomposition" of a tree [13, 30]. The centroid is a vertex v_0 which minimizes the size of the maximal subtree in the forest generated when v_0 is deleted from the tree. Obviously, the size of the maximal subtree at the centroid is less than or equal to $n/2$. Now, let the path from the centroid v_0 to the root u be $v_0, v_1, \ldots, v_r = u$. Consider all the subtrees $T_1, \ldots, T_t$ which are rooted at sons of vertices along this path but not containing any v_j. Thus, a typical subtree consists of a vertex v_j', which is a son of v_j $(0 \leq j \leq r;\ v_j' \neq v_{j-1}$ if $j > 0)$, together with all the vertices v such that v_j' lies on the path from the root u to v. We allocate to the tree T_i as many processors as there are vertices in T_i. Recursively, we evaluate the function g over all the subtrees $T_1, \ldots, T_t$ in parallel. Then, the following is evaluated in parallel for all j, $j = 0, 1, \ldots, r$. Let a_j equal w_{v_j} plus the sum of g-values taken over the sons of v_j (except for v_{j-1} in case $j \geq 0$). Obviously, all the a_j's are found in $O(\log n)$ parallel time. The problem is now reduced to evaluating the function g along the sequence $a_0, a_1, \ldots, a_r$. This can be carried out in $O(\log r)$ time, as indicated earlier in the present section. If $H(n)$ is the amount of parallel time that this procedure requires on a tree with n vertices, then $H(n) \leq H(n/2) + c \log n$ so that $H(n) = O((\log n)^2)$.

When the latter parallel algorithm is applied to designing a serial algorithm for the max–min k-partitioning problem on a tree, we obtain a bound of $O((n + n \log n)(\log n)^2) = O(n(\log n)^3)$. Analogous ideas can be applied to the min–max problem [2].

8. *Max-Flow-Related Problems*

The renowned max-flow problem is presently solvable in $O(V^3)$ time by Karzanov's algorithm [19] or $O(EV \log V)$ time by Sleator's algorithm [36]. The best parallel algorithm known for max-flow is that of Shiloach and Vishkin, which runs in $O(V^2 \log V)$ time if V processors are employed. This is not very satisfactory for our needs in this paper. The main difficulty with the good max-flow algorithms is that they all build upon Dinic's [11] basic idea which leads to $O(V)$ phases. In fact, all the improvements upon Dinic's work regard the solution of a single phase. Thus, we do not expect a parallel max-flow algorithm to run in $o(V)$ time.

A good parallel algorithm for max-flow (even with as many as EV processors) would be helpful for the following kind of problems. Suppose that c_{ij} is the capacity of the edge (i, j) but it is desirable not to use an edge at its full capacity. Less informally, suppose that every edge (i, j) has some safety level s_{ij} such that the ratio f_{ij}/s_{ij} (where f_{ij} is the flow through (i, j)) is sought to be minimized. A closely related problem which can also be solved by the same methods is that of maximizing the minimal ratio. Consider the problem of sending a certain amount of flow v (or, more generally, any flow with lower bounds [20]) through the network so that the maximal ratio f_{ij}/s_{ij} is minimized. This can be solved by defining $F(\lambda)$ to be the value of the maximum flow, subject to additional capacity constraints of the form $f_{ij} \leq \lambda s_{ij}$. We then solve the equation $F(\lambda) = v$.

For a demonstration of the method in this case, consider Dinic's [11] algorithm for max-flow. Like most of the other efficient algorithms for max-flow, Dinic's algorithm works in $O(V)$ phases, where during a single phase a maximal (rather than maximum) flow is sought through a "layered" network with modified capacities. In the parameterized version these capacities become nonnegative linear functions of λ over a certain interval. In Dinic's algorithm we repeatedly look for paths through this layered network. The bottleneck of a path, that is, the minimum of (modified) capacities along the path, determines the amount by which the flow may be increased. This minimum is in fact a piecewise-linear function of λ. Thus, breakpoints of this minimum function are critical values of λ. Recall that, in general, a critical value of λ arises at a point where the algorithm has to compare two linear functions of λ. Now, an augmenting path can be found by $V + 2E$ processors in $O(\log V)$ time [34]. The bottleneck is found in $O(\log \log V)$ time. Each phase of Dinic's algorithm finds an augmenting path at most E times. A parallel algorithm for max-flow which is suitable for our needs can now be devised to run in $O(EV \log V)$ time. This, however, does not justify utilizing parallelism, since Sleator's serial algorithm runs in the same time. On the other hand, Shiloach and Vishkin's parallel max-flow algorithm [35] runs in $O(V^2 \log V)$ time if V processors are employed. Using their algorithm, we run in $O(V^2 \log V)$ phases, where in each phase we run $O(\log V)$ regular max-flow problems. Since the max-flow problem can be solved in $O(V \min(E \log V, V^2))$ time, it follows that the corresponding parametric problem can be solved in $O(V^3 \log^2 V \cdot \min(E \log V, V^2))$ time. This readily applies to improving the bound obtained by Gusfield in [16], where the parametric method [23] was applied without exploiting parallelism in max-flow for solving a problem of program module distribution.

Another related problem was presented at the recent Israeli–South African Symposium on Operations Research (February 1981) by Dr. Eiselt of Concordia University. Given is a directed graph together with weights w_i associated with the vertices. The weight is interpreted as the number of customers located at vertex i. We have to select a vertex u at which a facility will be established. We then route the customers to the facility, minimizing the maximal number of customers using any single edge. Our method in the present paper is also applicable to this problem and does improve significantly the previously known bounds. We note that in this case one F-evaluation amounts to solving n max-flow problems (corresponding to the n potential locations of the facility). Thus, one obvious aspect of parallelism that should be exploited is the fact that these n problems can be solved by n machines in parallel. This is, of course, independent of the issue of parallelism within the max-flow problem itself.

A good parallel algorithm for the max-flow problem should also be helpful for solving problems related to the min-cost flow problem, like minimizing certain edge-flows subject to a budget constraint together with a total flow requirement (or lower bounds). This, of course, suggests further research toward a parallel algorithm for the min-cost flow problem.

9. *"Second-Order" Applications*

The importance of parallelism may be "doubled" when we imagine problems in which the basic principle presented in this paper is applied more than once to the same problem. Consider, for example, the following parametric variant of the minimum-ratio-cycle problem. Suppose that the edge-costs are themselves increasing linear functions of a parameter λ, with an interpretation similar to that discussed in the section on spanning trees. Now, suppose that we wish to find the maximum λ such that the minimum cost-to-time ratio of a cycle is less than or equal to a given bound b. Here, we define $F(\lambda)$ to be the minimum ratio relative to the cost evaluated at λ. We would therefore like to have a good *parallel* algorithm for finding minimum-ratio cycles.

A parallel version of the algorithm of the kind presented here would work as follows. Suppose that we can detect a negative cycle on a P-processor machine in $T(n, P)$ time. The evaluation of $F(\lambda)$ on a P-processor machine will work as follows. There will be $T(n, P)$ stages. During each stage, P critical values (of the parameter used for the minimum-ratio computation) will be produced in one time unit. Then, a binary search will take $O(\log P)$ detections,[2] that is, $O(T(n, P)\log P)$ time. Thus $F(\lambda)$ is evaluated in $O((T(n, P))^2\log P)$ parallel time. The solution of $F(\lambda) = b$ will be as follows. The procedure will run in $O((T(n, P))^2\log P)$ stages. During a stage, P critical values of λ will be produced in P time units (serially). Then the search requires $O(\log P)$ F-evaluations. A single evaluation takes $O((P + T_S\log P)T(n, P))$ time, where T_S is the serial time bound for detecting negative cycles. Thus, altogether, our algorithm runs in $O((P + (P + T_S\log P)T(n, P)\log P)(T(n, P))^2\log P)$ time. Assuming $P = O(T_S\log P)$, this reduces to $O(T_S(\log P)^3(T(n, P))^3)$, where the significance of reducing $T(n, P)$ is amplified considerably. Typical values in our example here would be $T_S = O(n^3)$, $P = O(n^3)$, and $T(n, P) = O(\log n\log\log n)$ so that we obtain a bound of $O(n^3(\log n)^6(\log\log n)^3)$. Without exploiting any parallelism the algorithm would only have an $O(n^{12})$ bound.

[2] In general, it may sometimes be beneficial to perform detections in parallel, as we show later.

A modification which yields a further improvement is as follows. Employ $P = n^3 \log n$ processors. Thus, there will be $O(n^3 \log n)$ critical values produced. Now, instead of detecting negative cycles at one value at a time, let us select $\log n$ values which are equally spaced in the set of critical values. By allocating n^3 processors to each of the selected values, we can detect negative cycles at all of them in $O(\log n \log\log n)$ parallel time. The binary search takes $O(\log\log n)$ time (but no detections) and results in reducing the set of remaining critical values to a size of $O(n^3)$. We then, again, select $\log n$ values and repeat the same idea. The search thus takes $O((\log(n^3 \log n))/\log\log n) = O(\log n/\log\log n)$ steps, each requiring $O(\log n \log\log n)$ time. Thus, the entire search takes $O((\log n)^2)$ time, which means that a minimum-ratio cycle is found in $O((\log n)^3)$ parallel time. The solution of $F(\lambda) = b$ is as follows. There will be $O((\log n)^3)$ stages. During each stage, $O(\log n)$ values will be tested, requiring $O(n^3(\log n)^2)$ time if we use approach 1 for finding minimum-ratio cycles. Thus, the algorithm runs in $O(n^3(\log n)^6)$ time.

The idea of using the basic technique more than once in the same problem has also been used by the author very successfully for solving the weighted Euclidean 1-center problem [26]. In that problem we are given n points (a_i, b_i), $i = 1, \ldots, n$ in the plane, together with positive weights w_i and we seek a point (x, y) so as to minimize $\max\{w_i \cdot [(x - a_i)^2 + (y - b_i)^2]^{1/2} : i = 1, \ldots, n\}$. By viewing the variables x, y themselves as parameters playing the role of λ, as throughout the present paper, a "second-order" application of our basic idea was obtained. The resulting bound was $O(n(\log n)^3(\log\log n)^2)$, a significant improvement over the previously known bound of $O(n^3)$ (see [26]).

10. *Conclusion*

The application of parallel algorithms to serial computation does not necessarily have to follow exactly the general scheme developed in this paper. It has been shown throughout that, very often, deviations from the general principle result in further improvements. Nevertheless, all these secondary improvements are themselves due to parallelism in one way or another and hence conform with the general spirit of what we have been trying to indicate. We prefer not to formalize our claims in the form of a general theory, since it is likely that the basic idea may be applicable in cases that do not presently seem to conform with our general framework.

The results in this paper motivate further research in parallelism in combinatorial algorithms. It turns out that there are interesting problems with any number of processors. Sometimes it may even be useful to have a number of processors which is much larger than the serial time complexity of the problem. Also, it may happen that a variable number of processors becomes useful when applied in serial computation, as indicated in this paper. Particularly stimulating are the so-called second-order applications. When we have several parameters involved, there is an interesting variety of ways to apply parallelism, as shown in Section 9.

ACKNOWLEDGMENTS. Discussions with and comments made by G. Lev, Z. Galil, N. Pippenger, M. Sniedovich, A. Tamir, and E. Zemel are gratefully acknowledged. A contribution of one of the referees to the section on minimum-ratio cycles is also gratefully acknowledged.

REFERENCES

1. AHO, A V , HOPCROFT, J E., AND ULLMAN, J D. *The Design and Analysis of Computer Algorithms* Addison-Wesley, Reading, Mass., 1976
2. BECKER, R.I., PERL, Y , AND SCHACH, S R A shifting algorithm for min-max tree partitioning. *J. ACM 29*, 1 (Jan 1982), 58-67.