Discrete Applied Mathematics 14 (1986) 135-156
North-Holland

AN INTRODUCTION TO PARALLELISM IN COMBINATORIAL OPTIMIZATION

G.A.P. KINDERVATER and J.K. LENSTRA
Centre for Mathematics and Computer Science, Amsterdam, The Netherlands

Received February 1985
Revised 17 July 1985

This is a tutorial introduction to the literature on parallel computers and algorithms that is relevant for combinatorial optimization. We briefly discuss theoretical as well as realistic machine models and the complexity theory for parallel computations. Some examples of polylog parallel algorithms and log space completeness results for $\mathcal{P}$ are given, and the use of parallelism in enumerative methods is reviewed.

Keywords. Parallel computer, computational complexity, polylog parallel algorithm, sorting, shortest paths, scheduling, log space completeness for $\mathcal{P}$, linear programming, dynamic programming, knapsack, branch and bound, traveling salesman.

Parallel computing is receiving a rapidly increasing amount of attention. In theory, a collection of processors that operate in parallel can achieve substantial speedups. In practice, technological developments are leading to the actual construction of such devices at low cost. Given the inherent limitations of traditional sequential computers, these prospects appear to be very stimulating for researchers interested in the design and analysis of combinatorial algorithms.

In this paper, we attempt to give a tutorial introduction to the literature on parallel computers and algorithms that is relevant for the area of combinatorial optimization. For a more complete survey which is reasonably up to date until July 1983, we refer to our annotated bibliography [Kindervater & Lenstra 1985].

The organization of the paper is as follows.

Section 1 is concerned with *machine models* designed for parallel computations. Theoretical as well as practical models are described. While in many theoretical models the processors communicate through a common memory without delay, in more realistic models the communication is achieved through a specific interconnection network. Such networks are illustrated on the problems of matrix multiplication, determining a transitive closure, and finding a minimum spanning tree. In later sections, we will restrict ourselves to theoretical models, which can usually be simulated fairly efficiently by models with a specific interconnection network.

Section 2 deals with the *complexity theory* for parallel computations. Given the basic distinction between *membership of* $\mathcal{P}$ and *completeness for* $\mathcal{NP}$ in sequential computations, we consider the speedups possible due to the introduction of

parallelism. Within the class $\mathcal{P}$, this leads to a distinction between 'very easy' problems, which are solvable in *polylogarithmic parallel time*, and the 'not so easy' ones, which are *log space complete for* $\mathcal{P}$.

Section 3 gives examples of *polylog parallel algorithms* for elementary problems like finding the maximum and sorting, for finding shortest paths, and for two problems from scheduling theory.

Section 4 discusses the *log space completeness for* $\mathcal{P}$ of the linear programming problem and the maximum network flow problem.

Section 5 reviews the use of parallelism in *enumerative methods* for $\mathcal{NP}$-hard problems, such as dynamic programming for the knapsack problem and branch and bound for the traveling salesman problem.

The reader will not fail to observe that the algorithms presented in this paper do not rely on the sophisticated refinements for sequential algorithms developed in the past two decades but go back to the simple and explicit basic principles of combinatorial computing. In that sense (and recent, more advanced achievements notwithstanding), parallelism in combinatorial optimization is still in its infancy and holds many promises for a further development in the near future.

1. Machine models

Many architectures for parallel computations have been proposed in the literature. Some of these machines actually exist or are being built. Other models are useful for the theoretical design and analysis of parallel algorithms, while their realization is not feasible due to physical limitations.

The most widely used classification of parallel computers is due to [Flynn 1966]. Flynn distinguishes four classes of machines (cf. Fig. 1).

(1) SISD (*single instruction stream, single data stream*). One instruction is performed at a time, on one set of data. This class contains the traditional sequential computers.

(2) SIMD (*single instruction stream, multiple data stream*). One type of instruction is performed at a time, possibly on different data. An enable/disable mask selects the processing elements that are allowed to perform the operation on their data. The ICL/DAP (Distributed Array Processor) belongs to this class.

(3) MISD (*multiple instruction stream, single data stream*). Different instructions on the same data can be performed at a time. This class has received very little attention so far.

(4) MIMD (*multiple instruction stream, multiple data stream*). Different instructions on different data can be performed at a time. There are two types of MIMD computers: the processors of a *synchronized* MIMD machine perform each successive set of instructions simultaneously; the processors of an *asynchronous* MIMD machine run independently and wait only if information from other processors is needed. The Denelcor/HEP (Heterogeneous Element Processor) is an example of an asynchronous MIMD machine.

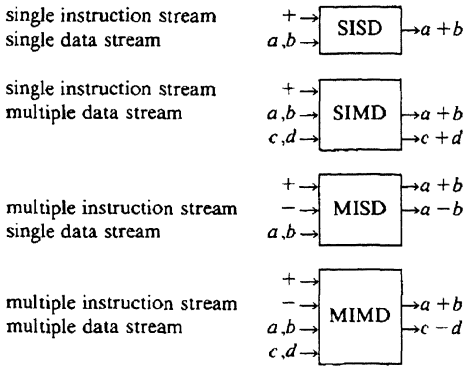


Fig. 1. The classification of Flynn.

If one considers the many types of algorithms that are suitable for execution on parallel computers, then both ends of the spectrum can be characterized in a way that resembles the above distinction between the two types of MIMD machines. *Systolic* algorithms lead to highly synchronized computations, where the processing elements act rhythmically on regular streams of data passing through the (SIMD or synchronized MIMD) machine. Typical examples are the matrix multiplication algorithm introduced later in this section and the dynamic programming recursions in Section 5. *Distributed* algorithms lead to asynchronous processes, in which the processors perform their own local computations and communicate by sending messages every now and then. Branch and bound (see Section 5) lends itself to this approach.

Flynn's classification is not concerned with the way in which information is transmitted between the processors. This is dealt with by Schwartz [Schwartz 1980], who distinguishes between paracomputers and ultracomputers.

In a *paracomputer*, the processors have simultaneous access to a *shared memory*, which allows for communication between any two processors in constant time. A further distinction is based on the way in which shared memory computers handle *read* and *write conflicts*, which occur when several processors try to read from or to write into the same memory location at the same time. Paracomputers are of great theoretical interest, but current technology prohibits their realization.

In an *ultracomputer*, the processors communicate through a fixed *interconnection network*. Such a network can be viewed as a graph with vertices corresponding to processors and (undirected) edges or (directed) arcs to interconnections. Two parameters of the graph are important in this context: the maximum vertex degree d_1, which should be bounded by a constant on grounds of practical feasibility, and the maximum path length d_2 (the 'diameter'), which should grow at most logarithmically in the number p of processors to ensure fast communication.

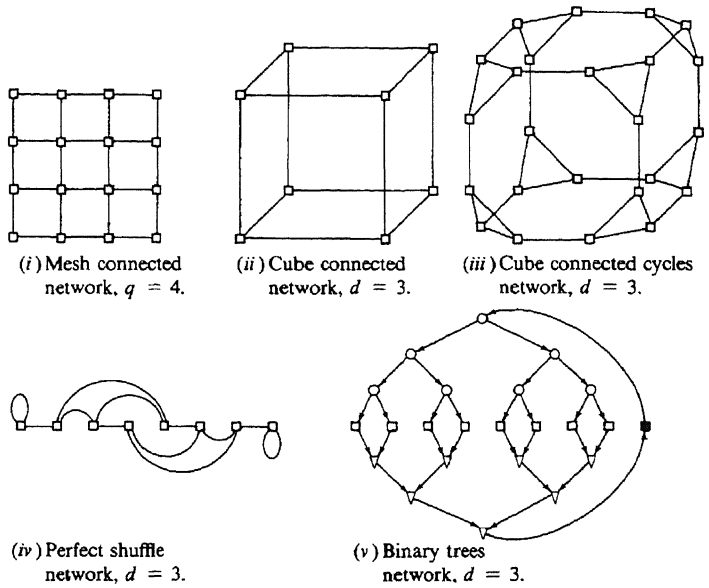


Fig. 2. Five interconnection networks.

Of the many interconnection networks that have been proposed, five are briefly described below. They are illustrated in Fig. 2.

(i) *Two-dimensional mesh connected* network [Unger 1958]. Each processor is identified with an ordered pair (i, j) $(i, j = 1, \ldots, q)$, and processor (i, j) is connected to processors $(i \pm 1, j)$ and $(i, j \pm 1)$, provided they exist. Note that $d_1 = 4$ and $d_2 = 2(q-1) = \Theta(\sqrt{p})$.

(ii) *Cube connected* network [Squire & Palais 1963]. This can be seen as a d-dimensional hypercube with 2^d processors at the vertices and interconnections along the edges. Note that $d_1 = d_2 = d = \log p$. (All logarithms in this paper have base 2.)

(iii) *Cube connected cycles* network [Preparata & Vuillemin 1981]. This is a cube connected network with each of the 2^d processors replaced by a cyclicly connected set of d processors; each of them has two cycle connections and one edge connection. This yields $d_1 = 3$ and $d_2 = \Theta(\log p)$.

(iv) *Perfect shuffle* network [Stone 1971]. There are $p = 2^d$ processors with interconnections $(i, 2i-1)$, $(i+p/2, 2i)$, $(2i-1, 2i)$ for $i = 1, \ldots, p/2$. The first two types of interconnections imitate a perfect shuffle of a deck of cards. Here, $d_1 = 3$ and $d_2 = 2d - 1 = \Theta(\log p)$.

(v) *Binary trees* network [Bentley & Kung 1979]. There are $p = 3 \cdot 2^d - 2$ processors, interconnected by two binary trees with common leaves. The 2^d processors corresponding to these leaves perform the actual computations. The other $2^d - 1$ processors in the first tree (an out-tree) send the data down to their descendants, and those in the second tree (an in-tree) combine the results from their ancestors.

An additional 'master processor' controls the network by providing the input for one root and receiving the output from the other. Note that $d_1 = 3$ and $d_2 = \Theta(\log p)$.

All these networks can simulate each other quite efficiently; see [Siegel 1977, 1979] for details. Still, it appears that the cube connected cycles and perfect shuffle networks are reasonably versatile, while the mesh connected and binary trees networks have been designed for more restricted types of computations. Their suitability for their limited purpose will be demonstrated on some examples below.

The quality of the parallelization of an algorithm will be judged on the resulting *speedup*, which is the running time of the best sequential implementation of the algorithm divided by the running time of the parallel implementation using p processors, and the *processor utilization*, which is the speedup divided by p. The best one can hope to achieve is a speedup of p and a processor utilization of 1. Note that these concepts are defined here relative to a given algorithm, irrespective of the possible existence of more efficient sequential algorithms for the problem at hand.

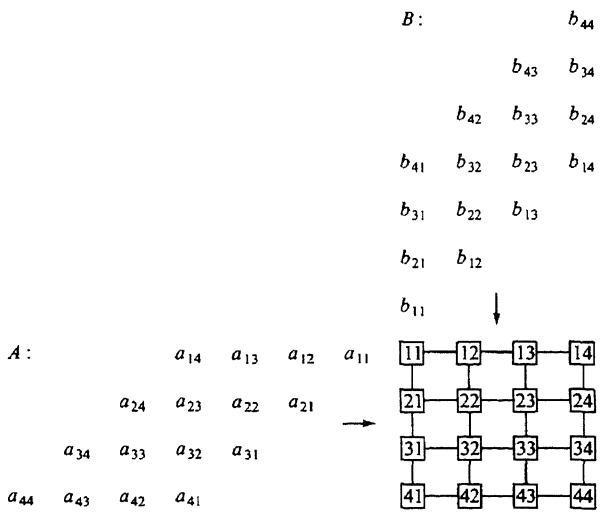


Fig. 3. Matrix multiplication on a mesh connected network.

Example 1. *Matrix multiplication.* Two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$ can be multiplied in $O(n)$ time on an $n \times n$ mesh connected network. The basic idea is the use of the skewed input scheme illustrated in Fig. 3. At each step of the computation, matrix A makes one step to the right, matrix B goes one step down, and each processing element (i, j) multiplies its current values a_{ik} and b_{kj} and adds the result into its accumulator (which starts at 0). It is easily verified that after $2n - 1$ stages processor (i, j) contains the required value $\sum_k a_{ik} b_{kj}$ and that the procedure is best possible in terms of speedup and processor utilization. This is a typical example of a systolic algorithm performed on an SIMD machine and suitable for VLSI implementation.

Example 2. *Transitive closure* [Guibas, Kung & Thompson 1979]. The transitive closure of a directed graph G has an arc (i, j) if and only if G has a path from i to j. If G has n vertices, the algorithm from Example 1 can be applied to find the transitive closure in $O(n)$ time using n^2 mesh connected processors. Starting with A given by the adjacency matrix of G (i.e., $a_{ij}=1$ if G has an arc (i, j) and $a_{ij}=0$ otherwise) and $B=A$, one executes the matrix multiplication algorithm *three times*, with the modifications that addition is replaced by maximization and that any element a_{ij} or b_{ij} that passes through processor (i, j) is updated with the value of the accumulator. A correctness proof of this procedure can be found in the above reference.

Example 3. *Membership testing.* Given a set S of n elements and an element e, one can test whether $e \in S$ in $O(\log n)$ time on a binary trees network with $d=\lceil \log n \rceil$. Denote the processors corresponding to the common leaves by P_i $(i=1, \ldots, 2^d)$ and suppose that P_i stores the ith element e_i of S $(i \leq n)$. It takes d steps for the processors in the top tree to send e down, one step for the P_i's to check whether $e_i=e$, and d steps for the processors in the bottom tree to compute the disjunction of the results.

As an extension, one can test the membership of S for m elements $e^{(1)}, \ldots, e^{(m)}$ in $O(m+\log n)$ time by *pipelining* the flow of information through the network. As soon as $e^{(1)}$ leaves the first processor, $e^{(2)}$ is sent to it; and, in general, at each step all data are going down one level.

By asking the processors in the bottom tree to do a bit more than computing logical disjunctions, one can use the same model to *find the minimum* of n elements and to *compute the rank* of a given element in $O(\log n)$ time. We leave details to the reader.

Example 4. *Minimum spanning tree* [Bentley 1980]. Given a complete undirected graph G with vertex set $\{1, \ldots, n\}$ and a length c_{ij} for each edge $\{i, j\}$, a spanning tree of G of minimum total length can be found in $O(n^2)$ time by an algorithm from [Prim 1957; Dijkstra 1959]. The algorithm is based on the following principle. Let $T(V)$ be the collexion of edges in a minimum spanning tree of the subgraph of G induced by the subset V of vertices. If $i^* \notin V$ and $j^* \in V$ are such that $c_{i^*j^*}=\min_{i \notin V, j \in V}\{c_{ij}\}$, then $T(V \cup \{i^*\})=T(V) \cup \{\{i^*, j^*\}\}$.

The algorithm starts with $T(\{1\})=\emptyset$. At each iteration, a minimum spanning tree on a certain vertex set V with edge set $T(V)$ has been constructed and, for each $i \notin V$, a 'closest tree vertex' $j_i \in V$ and a corresponding distance l_i are known, i.e., $l_i=c_{ij_i}=\min_{j \in V}\{c_{ij}\}$. One selects an $i^* \notin V$ for which $l_{i^*}=\min_{i \notin V}\{l_i\}$, adds i^* to V and $\{i^*, j_{i^*}\}$ to $T(V)$, and updates the values j_i and l_i for the remaining vertices $i \notin V$. There are $n-1$ iterations, each requiring $O(n)$ time.

It is not hard to implement the algorithm on a binary trees network with $d=\lceil \log n \rceil$. The master processor stores the set T of spanning tree edges. Processor P_i keeps track of j_i and l_i and is able to compute any c_{ij} in constant time. Each

command that is sent down the tree is executed only by those P_i's that are turned on.

We initialize by setting $T=\emptyset$ and, for $i=2,\ldots,n$, turning on P_i and setting $j_i=1$ and $l_i=c_{i1}$. In each of the $n-1$ iterations, we first apply the minimum-finding procedure to determine i^* and add $\{i^*, j_{i^*}\}$ to T; we next send i^* down in order to turn off P_{i^*} forever (since now $i^* \in V$) and to turn off each P_i with $l_i \leq c_{ii^*}$ temporarily for the rest of this iteration (since no update is necessary); and we finally instruct all remaining P_i's to set $j_i=i^*$ and $l_i=c_{ii^*}$.

Since each iteration takes $O(\log n)$ time, this parallel version of the algorithm has a running time of $O(n \log n)$ using $O(n)$ processors and hence a processor utilization of only $O(1/\log n)$. We cannot improve on this by pipelining the loop, since each iteration needs information from the previous one. However, we can use a smaller network with $d=\lceil \log(n/\log n) \rceil$, in which each P_i takes care of $\lceil \log n \rceil$ vertices and performs all computations for them sequentially. This modified algorithm still runs in $O(n \log n)$ time, but now using $O(n/\log n)$ processors with a processor utilization of $O(1)$.

In the remaining sections, we will restrict ourselves to the paracomputer model, which lends itself better to complexity considerations and to the explanation of parallel algorithms. The implementation of such algorithms on a specific ultracomputer model is usually straightforward.

2. Complexity theory

The purpose of this section is to present an informal introduction to those concepts from the complexity theory for parallel computing that may have some impact on the theory of combinatorial optimization. The interested reader is referred to [Cook 1981] for a more thorough exposition and to [Johnson 1983, Section 2] for a very readable review (on which this section is largely based).

Central to this area is a hypothesis known as the *parallel computation thesis* [Chandra, Kozen & Stockmeyer 1981; Goldschlager 1982]: *time bounded parallel machines are polynomially related to space bounded sequential machines.* That is, for any function T of the problem size n, the class of problems solvable by a machine with unbounded parallelism in *time* $T(n)^{O(1)}$ (i.e., polynomial in $T(n)$) is equal to the class of problems solvable by a sequential machine in *space* $T(n)^{O(1)}$. This thesis is a *theorem* for several 'reasonable' parallel machine models and several 'well-behaved' time bounds; see [Van Emde Boas 1985] for a survey.

The parallel computation thesis holds, for example, in the case that the machine model is a PRAM (Parallel Random Access Machine) and $T(n)=n^{O(1)}$ (i.e., a polynomial function of problem size). The PRAM is a synchronized machine with an unbounded number of processors and a shared memory, which allows simultaneous reads from the same memory location but disallows simultaneous

writes into the same memory location. The computation starts with one processor activated; at any step, an active processor can do a standard operation or activate another processor; and the computation stops when the initial processor halts.

According to the parallel computation thesis, the class of problems solvable by a PRAM in polynomial time is equal to $\mathcal{P}$SPACE, the class of problems solvable by a sequential machine in polynomial space. In view of the apparent difficulty of many problems in $\mathcal{P}$SPACE (such as the $\mathcal{P}$SPACE-complete and $\mathcal{NP}$-complete ones), the PRAM is an extremely powerful model. It is of interest to see how it affects the complexity of the problems in $\mathcal{P}$, which are solvable by a sequential machine in polynomial time.

It turns out that many problems in $\mathcal{P}$ can be solved in *polylog parallel time* $(\log n)^{O(1)}$, i.e., in time that is polynomially bounded in the logarithm of the problem size n. Some examples are given in Section 3; other, more complicated, examples are finding a maximum flow in a planar graph [Johnson & Venkatesan 1982] and linear programming with a fixed number of variables [Megiddo 1982]. By the parallel computation thesis, these problems would form the class POLYLOGSPACE of problems solvable in polylog sequential space. They can be considered to be among the *easiest* problems in $\mathcal{P}$, in the sense that the influence of problem size on solution time has been limited to a minimum. No single processor needs to have detailed knowledge of the entire problem instance. (It should be noted here that a further reduction to sublogarithmic solution time is generally impossible. One reason for this is that a PRAM needs $O(\log n)$ time to activate n processors; a similar reason is that in any realistic model of parallelism a constant upper bound on the maximum 'fan out' d_1 implies a logarithmic lower bound on the minimum 'communication time' d_2.)

On the other hand, $\mathcal{P}$ contains problems that are unlikely to admit solution in polylog parallel time. These are the problems that have been shown to be *log space complete for* $\mathcal{P}$, i.e., that belong to $\mathcal{P}$ and to which any other problem in $\mathcal{P}$ is reducible by a transformation using logarithmic work space. Examples will be discussed in Section 4; they include general linear programming and finding a maximum flow in an arbitrary graph. If any such problem would belong to POLYLOGSPACE, then it would follow that $\mathcal{P} \subseteq$ POLYLOGSPACE, which is not believed to be true. Hence, their solution in polylog sequential space or, equivalently, polylog parallel time is not expected either. Any solution method for these *hardest* problems in $\mathcal{P}$ is likely to require superlogarithmic time and is, loosely speaking, probably 'inherently sequential' in nature.

We have thus arrived at a distinction within $\mathcal{P}$ between the 'very easy' problems, which can be solved in polylog parallel time, and the 'not so easy' ones, for which a dramatic speedup due to parallelism is unlikely.

The picture of the PRAM model as sketched above is in need of some qualification. The model is theoretically very useful, but its unbounded parallelism is hardly realistic. The reader will have no difficulty in verifying that a PRAM is able to activate a superpolynomial number of processors in subpolynomial time. If a

polynomial time bound is considered reasonable, then certainly a polynomial bound on the number of processors should be imposed. It is a trivial observation, however, that the class of problems solvable if both bounds are respected is simply equal to $\mathcal{P}$. Within this more reasonable model, hard problems remain as hard as they were without parallelism.

Discussions along these lines have led to the consideration of *simultaneous resource bounds* and to the definition of new complexity classes. For example, *Nick (Pippenger)'s Class* $\mathcal{NC}$ contains all problems solvable in polylog parallel time on a polynomial number of processors, and *Steve (Cook)'s Class* $\mathcal{SC}$ contains all problems solvable in polynomial sequential time and polylog space. Some sort of extended parallel computation thesis might suggest that $\mathcal{NC} = \mathcal{SC}$. This is a major unresolved issue in complexity theory, and outside the scope of this introduction. We refer to [Johnson 1983, Section 2] for further details and more references.

3. Polylog parallel algorithms

We will now describe polylog parallel algorithms for six problems. Examples 5, 6 and 7 deal with basic operations on a set of numbers, Example 8 discusses the shortest paths problem, and Examples 9 and 10 are concerned with the scheduling of a set of jobs on identical parallel machines. Other problems that are solvable in polylog parallel time have been mentioned in Section 2 and will return in Section 4.

The algorithms will be designed to run on an SIMD machine with a shared memory. Simultaneous reads are permitted and simultaneous writes are prohibited; the former assumption is not essential but simplifies the exposition. We note that the polylog parallel algorithms referred to in this paper require a polynomial number of processors, so that the problems in question belong to $\mathcal{NC}$.

In the PIDGIN ALGOL procedures in this section, we write

par $[a \le i \le z]$ s_i

to denote that the statements s_i are to be executed in parallel for all values of the index i in the given range.

Example 5. *Maximum finding.* Given n numbers, one wishes to find their maximum. We assume, for convenience, that $n = 2^m$ for some integer m and that the numbers are given by $a_n, a_{n+1}, \ldots, a_{2n-1}$. Consider the following procedure:

for $l \leftarrow m-1$ **downto** 0 **do**
  par $[2^l \le j \le 2^{l+1}-1]$ $a_j \leftarrow \max\{a_{2j}, a_{2j+1}\}$.

The computation is illustrated by means of a *binary tree* in Fig. 4. At step l, the values corresponding to the nodes at level l of the tree are calculated. At the end, a_1 is equal to the desired maximum.

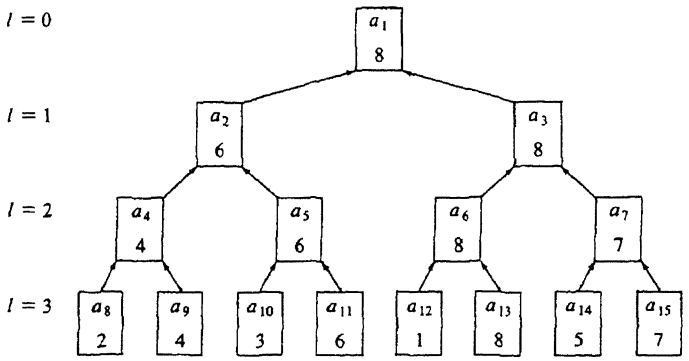


Fig. 4. Maximum finding: an instance with $n=8$.

The algorithm requires $O(\log n)$ time and $n/2$ processors. We can improve on this by applying a device similar to the one used in the last paragraph of Example 4: each processor has $\log n$ data assigned to it and computes their maximum sequentially, before the above procedure is executed. The resulting algorithm still runs in $O(\log n)$ time, but now using only $\lceil n/\log n \rceil$ processors with a processor utilization of $O(1)$.

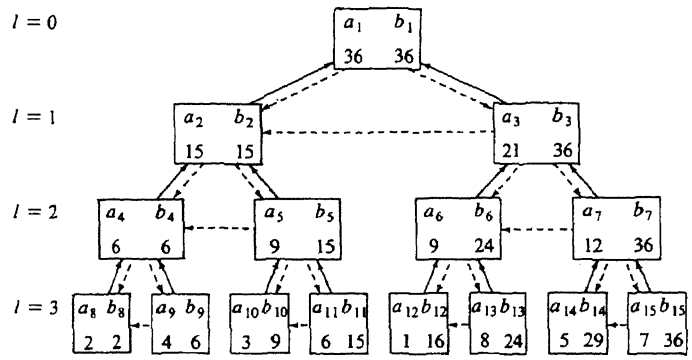


Fig. 5. Partial sums: an instance with $n=8$.

Example 6. *Partial sums* [Dekel & Sahni 1983a]. Given n numbers $a_n, a_{n+1}, \ldots, a_{2n-1}$ with $n=2^m$, one wishes to find the partial sums $a_n + \cdots + a_{n+j}$ for $j=0, \ldots, n-1$. Consider the following procedure:

for $l \leftarrow m-1$ **downto** 0 **do**
  par $[2^l \le j \le 2^{l+1}-1]$ $a_j \leftarrow a_{2j} + a_{2j+1}$;
$b_1 \leftarrow a_1$;
for $l \leftarrow 1$ **to** m **do**
  par $[2^l \le j \le 2^{l+1}-1]$ $b_j \leftarrow$ **if** j odd **then** $b_{(j-1)/2}$ **else** $b_{j/2} - a_{j+1}$.

The computation is illustrated in Fig. 5. In the first phase, represented by the solid

arrows, the sum of the a_j's is calculated in the same way as their maximum was calculated in Example 5. Note that the a-value corresponding to a non-leaf node is set equal to the sum of all a-values corresponding to the leaves descending from that node. In the second phase, represented by the dotted arrows, each parent node sends a b-value (starting with $b_1 = a_1$) to its children: the right child receives the same value, the left one receives that value minus the a-value of his brother. The b-value of a certain node is therefore equal to the sum of all a-values of the nodes of the same generation, except those with a higher index. This implies, in particular, that at the end we have $b_{n+j} = a_n + \cdots + a_{n+j}$ for $j = 0, \ldots, n-1$.

The algorithm requires $O(\log n)$ time and n processors. As before, this can be improved to $O(\log n)$ time and $O(n/\log n)$ processors.

Example 7. *Sorting* [Muller & Preparata 1975]. Given n numbers $a_1, \ldots, a_n$, one wishes to renumber them such that $a_1 \leq \cdots \leq a_n$. We assume, for simplicity, that $a_i \neq a_j$ if $i \neq j$. Consider the following procedure:

par $[1 \leq i, j \leq n]$ $\varrho_{ij} \leftarrow$ **if** $a_i \leq a_j$ **then** 1 **else** 0;
par $[1 \leq j \leq n]$ $\pi_j \leftarrow \text{sum}\{\varrho_{ij} \mid 1 \leq i \leq n\}$;
par $[1 \leq j \leq n]$ $a_{\pi_j} \leftarrow a_j$.

The algorithm is based on *enumeration sort*: the position π_j in which a_j should be placed is calculated by counting the a_i's that are no greater than a_j. There are three phases:

(i) computation of the relative ranks ϱ_{ij}: n^2 processors, $O(1)$ time - or $\lceil n^2/\log n \rceil$ processors, $O(\log n)$ time;

(ii) computation of the positions π_j: $n\lceil n/\log n \rceil$ processors, $O(\log n)$ time (by application of the first phase of the algorithm of Example 6);

(iii) permutation: n processors, $O(1)$ time.

The algorithm requires $O(\log n)$ time and $O(n^2/\log n)$ processors. Simultaneous reads occur in the first phase, but there is a way to avoid them within the same time and processor bounds. As sequential enumeration sort takes $O(n^2)$ time, the processor utilization is $O(1)$.

Example 8. *Shortest paths* [Dekel, Nassimi & Sahni 1981]. Given a complete directed graph with vertex set $\{1, \ldots, n\}$ and a length c_{ij} for each arc (i, j), one wishes to find the shortest path lengths between all pairs of vertices. In [Lawler 1976] an algorithm is given which requires $O(n^3 \log n)$ time. It is based on matrix multiplication. Let $d_{ij}^{(l)}$ denote the length of a shortest path from vertex i to vertex j, containing no more than l arcs. Since a path from vertex i to vertex j consisting of at most $2l$ arcs can be split into two paths of no more than l arcs each, we have that $d_{ij}^{(2l)} = \min_{k \in \{1, \ldots, n\}} \{d_{ik}^{(l)} + d_{kj}^{(l)}\}$. Taking into account that a shortest path contains at most $n-1$ arcs, we obtain the following algorithm:

par $[1 \le i, j \le n]$ $d_{ij}^{(1)} \leftarrow c_{ij}$;
for $m \leftarrow 1$ **to** $\lceil \log n \rceil$ **do**
  $l \leftarrow 2^m$,
  par $[1 \le i, j \le n]$ $d_{ij}^{(l)} \leftarrow \min\{d_{ik}^{(l/2)} + d_{kj}^{(l/2)} \mid 1 \le k \le n\}$.

Application of the routine of Example 5 with maximization replaced by minimization yields an algorithm which requires $O(\log^2 n)$ time and $O(n^3/\log n)$ processors, with a processor utilization of $O(1)$.

Example 9. *Preemptive scheduling* [Dekel & Sahni 1983b]. Given m machines M_i $(i = 1, \ldots, m)$ and n jobs J_j, each with a processing time p_j $(j = 1, \ldots, n)$, one wishes to find a preemptive schedule of minimum length. A preemptive schedule assigns to each J_j a number of triples (M_i, s, t), where $1 \le i \le m$ and $0 \le s \le t$, indicating that J_j is to be processed by M_i from time s to time t. A preemptive schedule is feasible if the processing intervals on M_i are nonoverlapping for all i, and the processing intervals of J_j are nonoverlapping and have total length p_j for all j. It is optimal if the maximum completion time of the jobs is minimum.

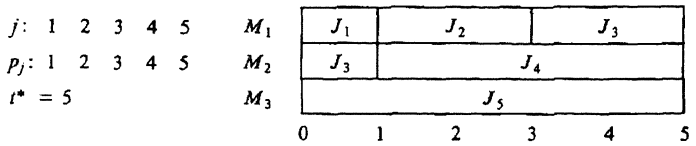


Fig. 6. Preemptive scheduling: an instance with $m = 3$ and $n = 5$.

An optimal schedule can be found in $O(n)$ time by the classical *wrap around rule* from [McNaughton 1959]. The algorithm first computes a value t^* which is an obvious lower bound on the minimum schedule length. It then constructs a schedule of length t^* by considering the jobs in an arbitrary order and scheduling them in the m periods $(0, t^*)$, carrying over the part of a job that does not fit at the end of the period on M_i to the beginning of the period on M_{i+1}. More formally:

$t^* \leftarrow \max\{\max\{p_j \mid 1 \le j \le n\}, \text{sum}\{p_j \mid 1 \le j \le n\}/m\}$;
$s \leftarrow 0$; $i \leftarrow 1$;
for $j \leftarrow 1$ **to** n **do**
  if $s + p_j \le t^*$
  then assign $(M_i, s, s + p_j)$ to J_j,
    $s \leftarrow s + p_j$
  else assign (M_i, s, t^*) and $(M_{i+1}, 0, p_j - (t^* - s))$ to J_j,
    $s \leftarrow p_j - (t^* - s)$, $i \leftarrow i + 1$.

An example is given in Fig. 6. There are two global parameters that are updated se-

quentially as the job index j increases: the starting time s and the machine index i of J_j. We can calculate all starting times and machine indices simultaneously in logarithmic time, using the parallel procedures for finding the maximum and the partial sums from Examples 5 and 6 as subroutines:

$t^* \leftarrow \max\{\max\{p_j \mid 1 \le j \le n\}, \text{sum}\{p_j \mid 1 \le j \le n\}/m\}$;
par $[1 \le j \le n]$ $q_j \leftarrow \text{sum}\{p_k \mid 1 \le k \le j-1\}$;
par $[1 \le j \le n]$
  $s_j \leftarrow q_j \bmod t^*$, $i_j \leftarrow \lfloor q_j/t^* \rfloor + 1$,
  if $s_j + p_j \le t^*$
  then assign $(M_{i_j}, s_j, s_j + p_j)$ to J_j
  else assign (M_{i_j}, s_j, t^*) and $(M_{i_j+1}, 0, p_j - (t^* - s_j))$ to J_j.

This algorithm can be implemented to require $O(\log n)$ time and $O(n/\log n)$ processors with a processor utilization of $O(1)$.

Example 10. *Scheduling fixed jobs* [Dekel & Sahni 1983b]. Given n jobs J_j, each with a starting time s_j and a completion time t_j $(j = 1, \ldots, n)$, one wishes to find a schedule on a minimum number of machines. A schedule assigns to each J_j a machine M_i. It is feasible if the processing intervals (s_j, t_j) on M_i are nonoverlapping for all i; it is optimal if the number of machines that process jobs is minimum. The problem is also known as the *channel assignment* problem: n wires are to be laid out between given points in a minimum number of parallel channels, each of which can carry at most one wire at any point.

An optimal schedule can be found in $O(n \log n)$ time by the following simple rule. First, order the jobs according to nondecreasing starting times. Next, schedule each successive job on a machine, giving priority to a machine that has completed another job before. It is not hard to see that, at the end, the number of machines to which jobs have been assigned is equal to the maximum number of jobs that require simultaneous processing. This implies optimality of the resulting schedule.

For a polylog parallel implementation, we need a more detailed sequential description of the algorithm [Gupta, Lee & Leung 1979]. We introduce an array u of length $2n$ containing all starting and completion times in nondecreasing order; the informal notation '$u_k \sim s_j$' ('$u_k \sim t_j$') will serve to indicate that the kth element of u corresponds to the starting (completion) time of J_j. We also use a stack S of idle machines; on top of S is always the machine that has most recently completed a job, if such a machine exists.

sort $(s_1, t_1, \ldots, s_n, t_n)$ in nondecreasing order in $(u_1, \ldots, u_{2n})$ whereby,
  if $t_j = s_k$ for some j & k, t_j precedes s_k;
$S \leftarrow$ stack of n machines;
for $k \leftarrow 1$ **to** $2n$ **do**
  if $u_k \sim s_j$ **then** take machine from top of S and assign it to J_j,
  if $u_k \sim t_j$ **then** put machine assigned to J_j on top of S.

Fig. 7 illustrates the algorithm as well as its parallelization, which is described below. There are four phases.

(i) First, we calculate the number σ_j of machines that are busy directly after the start of J_j and the number τ_j of machines that are busy directly before the completion of J_j, for $j=1,\ldots,n$:

sort $(s_1, t_1, \ldots, s_n, t_n)$ in nondecreasing order in $(u_1, \ldots, u_{2n})$ whereby,
  if $t_j = s_k$ for some j & k, t_j precedes s_k;
par $[1 \le k \le 2n]$ $\alpha_k \leftarrow$ **if** $u_k \sim s_j$ **then** 1 **else** -1;
par $[1 \le k \le 2n]$ $\beta_k \leftarrow \text{sum}\{\alpha_l \mid 1 \le l \le k\}$;
par $[1 \le k \le 2n]$
  if $u_k \sim s_j$ **then** $\sigma_j \leftarrow \beta_k$,
  if $u_k \sim t_j$ **then** $\tau_j \leftarrow \beta_k + 1$.

Note that the number of machines we need is equal to $\max_j \{\sigma_j\}$.

(ii) For each J_j, we determine its *immediate* predecessor $J_{\pi(j)}$ on the same machine (if it exists). The stacking mechanism implies that this must be, among the J_k satisfying $\tau_k = \sigma_j$, the one that is completed last before the start of J_j; if no such job exists, then it is convenient to take J_j as its own predecessor:

j:	1	2	3	4	5		k:	1	2	3	4	5	6	7	8	9	10
s_j:	0	1	3	4	7	→	u_k:	0	1	2	3	4	5	6	7	8	9
t_j:	2	8	5	6	9		↓										
							α_k:	1	1	−1	1	1	−1	−1	1	−1	−1
σ_j:	1	2	2	3	2	←	β_k:	1	2	1	2	3	2	1	2	1	0
τ_j:	2	2	3	2	1												
↓																	
$\pi(j)$:	1	2	1	4	4	→											

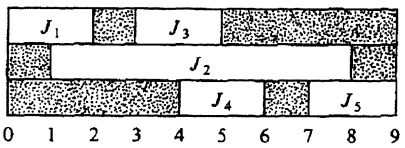


Fig. 7. Scheduling fixed jobs: an instance with $n=5$.

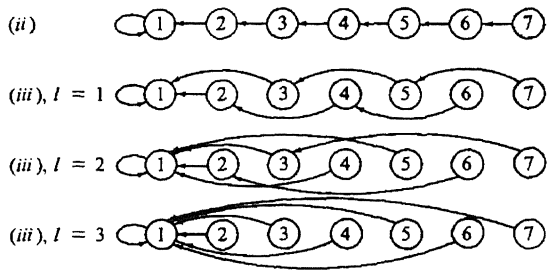


Fig. 8. Scheduling fixed jobs: finding the first preceding job on the same machine.

par $[1 \leq j \leq n]$
find k such that $\tau_k = \sigma_j$ & $t_k = \max\{t_l \mid t_l \leq s_j, \tau_l = \sigma_j\}$,
$\pi(j) \leftarrow$ **if** k exists **then** k **else** j.

(iii) For each J_j, we now turn $J_{\pi(j)}$ into its *first* predecessor on the same machine. This is done by simultaneously collapsing the chains formed by the arcs $(j, \pi(j))$ in a logarithmic number of steps (cf. Fig. 8):

for $l \leftarrow 1$ **to** $\lceil \log n \rceil$ **do par** $[1 \leq j \leq n]$ $\pi(j) \leftarrow \pi(\pi(j))$.

(iv) Finally, we use the $\pi(j)$'s to perform the actual machine assignments:

par $[1 \leq j \leq n]$ assign $M_{\sigma_{\pi(j)}}$ to J_j.

Using the maximum, partial sums and sorting routines from Examples 5, 6 and 7, we can implement this algorithm to require $O(\log n)$ time and $O(n^2/\log n)$ processors.

4. Log space completeness for $\mathcal{P}$

The first log space complete problem in $\mathcal{P}$ was identified by Cook [Cook 1974]. It involves the *solvability of a path system* and is proved log space complete by a 'master reduction' in the same spirit as Cook's $\mathcal{NP}$-completeness proof for the *satisfiability* problem. We will not define the *path* problem here and prefer to start from a different point.

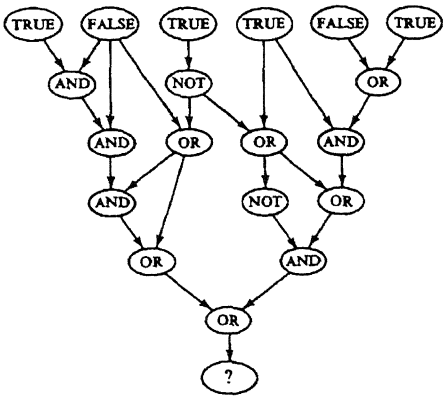


Fig. 9. A logical circuit.

Example 11. *Circuit value* [Ladner 1975; Goldschlager 1977]. Given a logical circuit consisting of input gates, AND gates, OR gates, NOT gates, and a single output gate, and given a truth value for each input, is the output TRUE of FALSE? Cf. Fig. 9.

The circuit value problem is trivially in $\mathcal{P}$. Ladner indicated how to simulate any

polynomial time deterministic Turing machine by a combinatorial circuit with only AND and NOT gates in logarithmic work space. It follows that the problem is log space complete for $\mathcal{P}$.

Goldschlager extended this result to the cases of *monotone* circuits, which have only AND and OR gates, and *planar* circuits, which have a cross free planar embedding, by giving log space transformations from the circuit value problem.

Example 12. *Linear programming* [Dobkin, Lipton & Reiss 1979; Valiant 1982]. Given a finite system of linear equations and inequalities in real variables, does it have a feasible solution?

Linear programming is known to be in $\mathcal{P}$ [Khachian 1979]. Dobkin, Lipton & Reiss established log space completeness for $\mathcal{P}$ of the problem by giving a log space transformation from the *unit resolution* problem, a variant of the *satisfiability* problem, that was already known to be log space complete for $\mathcal{P}$. Valiant gave a more straightforward transformation, starting from the *circuit value* problem.

The idea is to associate a variable x_j with the jth gate, such that $x_j = 1$ if the gate produces the value TRUE and $x_j = 0$ otherwise. More explicitly,

if gate j is	then we introduce the equations and inequalities
· an input gate with value TRUE,	· $x_j = 1$,
· an input gate with value FALSE,	· $x_j = 0$,
· an AND gate with inputs from gates h and i,	· $x_j \leq x_h, x_j \leq x_i, x_j \geq 0, x_j \geq x_h + x_i - 1$,
· a NOT gate with input from gate i,	· $x_j = 1 - x_i$,
· the output gate with input from gate i,	· $x_j = x_i, x_j = 1$.

OR gates may be excluded. We leave it to the reader to verify that each feasible solution is a 0–1 vector, that there exists a feasible solution if and only if the circuit value is TRUE, and that the transformation requires logarithmic work space.

Simple refinements of this transformation show that linear programming remains log space complete for $\mathcal{P}$ if all coefficients are equal to -1, 0 or 1, and each row and column of the constraint matrix contains at most three entries.

Example 13. *Maximum flow* [Goldschlager, Shaw & Staples 1982]. Given a directed graph with specified source and sink vertices and with capacities on the arcs, and given a value v, does the graph have a flow from source to sink of value at least v?

The maximum flow problem belongs to $\mathcal{P}$ [Edmonds & Karp 1972]. It was shown to be log space complete for $\mathcal{P}$ by a transformation from the monotone circuit value problem. The transformation simulates the implications of boolean inputs through a circuit with n AND and OR gates by integer flows through a network with the gates and an additional source and sink as vertices and with arc capacities of $O(2^n)$.

We conclude this section by mentioning two related results of a more positive nature.

(i) The maximum flow problem is solvable in polylog parallel time in the case of *planar* graphs, due to the relation of this case to the shortest path problem [Johnson & Venkatesan 1982].

(ii) The problem is solvable in *random* polylog parallel time in the case of *unit* capacities and in the more general case that the capacities are encoded in *unary*. This follows, through standard transformations [Lawler 1976], from the recent result that the maximum cardinality matching problem is in $\mathcal{RNC}$, the class of problems solvable by a randomized algorithm in polylog time on a polynomial number of processors [Karp, Upfal & Wigderson 1985]. The complexity of the maximum cardinality matching problem with respect to deterministic parallel computations is an open question, even for bipartite graphs.

5. Enumerative methods

The optimal solution to $\mathcal{NP}$-hard problems is usually found by some form of implicit enumeration of the set of all feasible solutions. In this section we will consider the parallelization of the two main types of enumerative methods: *dynamic programming* and *branch and bound*. We have already seen that, from a worst case point of view, intractability and superpolynomiality are unlikely to disappear in any reasonable machine model for parallel computations. In a more practical sense, parallelism has much to offer to extend the range in which enumerative techniques succeed in solving problem instances to optimality. Little work has been done in this direction, but we feel that the design and analysis of parallel enumerative methods is an important and promising research area.

Dynamic programming algorithms for combinatorial problems typically perform a regular sequence of many highly similar and quite simple instructions. Hence, they seem to be suitable for implementation in a systolic fashion on synchronized MIMD or even SIMD machines. This has been observed in [Casti, Richardson & Larson 1973; Guibas, Kung & Thompson 1979] and will be illustrated on the knapsack problem in Example 14.

Branch and bound methods generate search trees in which each node has to deal with a subset of the solution set. Since the instructions performed at a node very much depend on the particular subset associated with that node, it is more appropriate to implement these methods in a distributed fashion on asynchronous MIMD machines. An initial analysis of distributed branch and bound, in which the processors communicate only to broadcast new solution values or to redistribute the remaining work load, is given in [El-Dessouki & Huen 1980]. In a sequential branch and bound algorithm, the subproblems to be examined are given a priority and from among the generated subproblems the one with the highest priority is selected next. In a parallel implementation, it depends on the number of processors which subproblems are available and thus how the tree is searched. One can construct examples in which p processors together are slower than a single processor, or more

than p times as fast. These anomalies are analyzed in [Burton, Huntbach, McKeown & Rayward-Smith 1983; Lai & Sahni 1984] and illustrated on the traveling salesman problem in Example 15.

Example 14. *Knapsack.* Given n items j, each with a profit c_j and a weight a_j ($j=1,\ldots,n$), and given a knapsack capacity b, one wishes to find a subset of the items of maximum total profit and of total weight at most b. The problem is $\mathcal{NP}$-hard [Garey & Johnson 1979].

It is convenient to introduce the notation

$$C(m, n, b) = \max_{S \subseteq \{m,\ldots,n\}} \left\{ \sum_{j \in S} c_j \mid \sum_{j \in S} a_j \leq b \right\}.$$

According to Bellman's principle of optimality, one attains the maximum profit $C(1, n, b)$ by excluding item n and taking the profit $C(1, n-1, b)$ or by including item n and adding c_n to the profit $C(1, n-1, b-a_n)$. A recursive application of this idea gives the following dynamic programming algorithm [Bellman 1957]:

for $z \leftarrow 0$ **to** b **do** $C(1, 0, z) \leftarrow 0$;
for $j \leftarrow 1$ **to** n **do**
  for $z \leftarrow 0$ **to** $a_j - 1$ **do** $C(1, j, z) \leftarrow C(1, j-1, z)$,
  for $z \leftarrow a_j$ **to** b **do** $C(1, j, z) \leftarrow \max\{C(1, j-1, z), C(1, j-1, z-a_j)+c_j\}$.

The algorithm runs in $O(nb)$ time. (Note that this is exponential in the problem size. Since it is polynomial in the problem data, it is called 'pseudopolynomial'.) The obvious parallelization is to handle the stages j ($0 \leq j \leq n$) sequentially and, at stage j, to handle the states $(1, j, z)$ ($0 \leq z \leq b$) in parallel [Casti, Richardson & Larson 1973]:

Algorithm KS1
par $[0 \leq z \leq b]\, C(1, 0, z) \leftarrow 0$;
for $j \leftarrow 1$ **to** n **do**
  par $[0 \leq z < a_j]\, C(1, j, z) \leftarrow C(1, j-1, z)$,
    $[a_j \leq z \leq b]\, C(1, j, z) \leftarrow \max\{C(1, j-1, z), C(1, j-1, z-a_j)+c_j\}$.

This requires $O(n)$ time and $O(b)$ processors with a processor utilization of $O(1)$.

We can achieve a running time that is sublinear in n by observing that

$$C(1, n, b) = \max_{0 \leq y \leq b} \{C(1, m, b-y) + C(m+1, n, y)\}$$

for any $m \in \{1,\ldots,n-1\}$. It is of interest to note that this more general recursion was proposed in [Bellman & Dreyfus 1962] in the context of parallel computations. If we choose $m = n-1$, the previous recursion results as a special case. If we choose $m = n/2$, then we get another dynamic programming algorithm for the knapsack problem (where it is assumed that n is a power of 2):

Algorithm KS2

par $[1 \le j \le n]$ **par** $[0 \le z \le a_j]\, C(j, j, z) \leftarrow 0$,
$[a_j \le z \le b]\, C(j, j, z) \leftarrow c_j$;
for $l \leftarrow 1$ **to** $\log n$ **do**
$k \leftarrow 2^l$,
par $[0 \le j < n/k]$ **par** $[0 \le z \le b]\, C(jk+1, jk+k, z)$
$\leftarrow \max_{0 \le y \le z} \{C(jk+1, jk+\frac{1}{2}k, z-y) + C(jk+\frac{1}{2}k+1, jk+k, y)\}$.

The algorithm requires $O(nb^2)$ time on a single processor and $O(\log n \log b)$ time on $O(nb^2/\log b)$ processors. While the parallel running time is probably the best one can hope for (it might be called 'pseudopolylogarithmic'), the number of processors is huge. This number can be reduced by a factor of $\log n \log b$ by application of the first algorithm to produce starting solutions for the second algorithm. The modified algorithm has three phases:

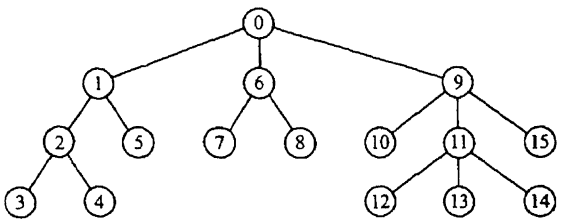
(i) Separate the n items into g groups of n/g items each.

(ii) Apply Algorithm KS1 to each group, in parallel: $O(n/g)$ time, $O(gb)$ processors.

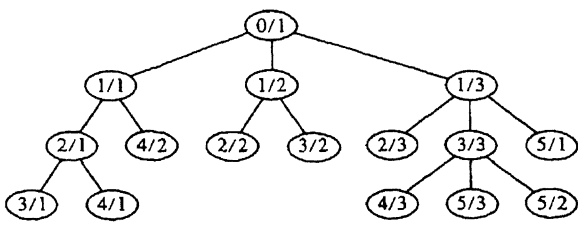
(iii) Apply Algorithm KS2, starting with g groups rather than with n items: $O(\log g \log b)$ time, $O(gb^2/\log b)$ processors.

We now set $g = \lceil n/(\log n \log b) \rceil$ to arrive at an algorithm that still requires $O(\log n \log b)$ time but using 'only' $O(nb^2/(\log n (\log b)^2))$ processors.

Example 15. *Traveling salesman* [Pruul 1975]. Given a complete graph with n vertices and a weight for each edge, one wishes to find a Hamiltonian cycle (i.e., a cycle passing through each vertex exactly once) of minimum total weight.



(a) Sequential search; node t is selected at time t.



(b) Parallel search by three processors;
node t/p is selected at time t by processor p.

Fig. 10. Depth first tree search.

A traditional branch and bound method for the solution of this $\mathcal{NP}$-hard problem uses a bounding mechanism based on the linear assignment relaxation, a branching rule based on subtour elimination, and a strategy for selecting new nodes for examination based on depth first tree search. The details are of no concern here and can be found in [Lawler, Lenstra, Rinnooy Kan & Shmoys 1985]. Fig. 10(a) shows a search tree in which the nodes have been labeled in order of examination.

Pruul designed a parallel version of this method for an asynchronous MIMD machine. Each processor performs its own depth first search; when it encounters a node that has already been selected by another processor, it selects in the subtree rooted by that node an unexamined node at the highest level. Fig. 10(b) illustrates the process.

The lack of parallel hardware forced Pruul to simulate the algorithm on a sequential computer. An empirical analysis for ten 25-vertex problems yielded average speedups that were greater than the number of processors. This may be confusing at first sight, but the explanation is simple and lies outside the area of parallel computing. The simulated parallel algorithm is nothing but a sequential algorithm that is based on a mixture of depth first and breadth first tree search. Such complex strategies have not yet been explored in any detail and might be quite powerful.

References

R.E. Bellman (1957), Dynamic Programming (Princeton University Press, Princeton, NJ).

R.E. Bellman and S.E. Dreyfus (1962), Applied Dynamic Programming (Princeton University Press, Princeton, NJ).

J.L. Bentley (1980), A parallel algorithm for constructing minimum spanning trees, J. Algorithms 1, 51–59.

J.L. Bentley and H.T. Kung (1979), A tree machine for searching problems, Proc. 1979 Internat. Conf. Parallel Processing, 257–266.

F.W. Burton, M.M. Huntbach, G.P. McKeown and V.J. Rayward-Smith (1983), Parallelism in branch-and-bound algorithms, Report CSA/3/1983, University of East Anglia, Norwich.

J. Casti, M. Richardson and R. Larson (1973), Dynamic programming and parallel computers, J. Optim. Theory Appl. 12, 423–438.

A.K. Chandra, D.C. Kozen and L.J. Stockmeyer (1981), Alternation, J. Assoc. Comput. Match. 28, 114–133.

S.A. Cook (1974), An observation on time-storage trade off, J. Comput. System Sci. 9, 308–316.

S.A. Cook (1981), Towards a complexity theory of synchronous parallel computation, Enseign. Math. (2) 27, 99–124.

E. Dekel, D. Nassimi and S. Sahni (1981), Parallel matrix and graph algorithms, SIAM J. Comput. 10, 657–675.

E. Dekel and S. Sahni (1983a), Binary trees and parallel scheduling algorithms, IEEE Trans. Comput. 32, 307–315.

E. Dekel and S. Sahni (1983b), Parallel scheduling algorithms, Oper. Res. 31, 24–49.

E.W. Dijkstra (1959), A note on two problems in connexion with graphs, Numer. Math. 1, 269–271.

D. Dobkin, R.J. Lipton and S. Reiss (1979), Linear programming is log-space hard for P, Inform. Process. Lett. 8, 96–97.

J. Edmonds and R.M. Karp (1972), Theoretical improvements in algorithmic efficiency for network flow problems, J. Assoc. Comput. Mach. 19, 248–264.

O.I. El-Dessouki and W.H. Huen (1980), Distributed enumeration on between computers, IEEE Trans. Comput. 29, 818–825. Note: in the title, read 'network' for 'between'.

M.J. Flynn (1966), Very high-speed computing systems, Proc. IEEE 54, 1901–1909.

M.R. Garey and D.S. Johnson (1979), Computers and Intractability: A Guide to the Theory of NP-Completeness (Freeman, San Francisco).

L.M. Goldschlager (1977), The monotone and planar circuit value problems are log space complete for P, SIGACT News 9.2, 25–29.

L.M. Goldschlager (1982), A universal connection pattern for parallel computers, J. Assoc. Comput. Mach. 29, 1073–1086.

L.M. Goldschlager, R.A. Shaw and J. Staples (1982), The maximum flow problem is log space complete for P, Theoret. Comput. Sci. 21, 105–111.

L.J. Guibas, H.T. Kung and C.D. Thompson (1979), Direct VLSI implementation of combinatorial algorithms, Caltech Conf. VLSI, 509–525.

U.I. Gupta, D.T. Lee and J.Y.-T. Leung (1979), An optimal solution for the channel-assignment problem, IEEE Trans. Comput. 28, 807–810.

D.B. Johnson and S.M. Venkatesan (1982), Parallel algorithms for minimum cuts and maximum flows in planar networks (preliminary version), Proc. 23rd Annual IEEE Symp. Foundations of Computer Science, 244–254.

D.S. Johnson (1983), The NP-completeness column: an ongoing guide; seventh edition, J. Algorithms 4, 189–203.

R.M. Karp, E. Upfal and A. Wigderson (1985), Constructing a perfect matching is in Random NC, Proc. 17th ACM Symp. Theory of Computing, 22–32.

L.G. Khachian (1979), A polynomial algorithm in linear programming, Soviet Math. Dokl. 20, 191–194.

G.A.P. Kindervater and J.K. Lenstra (1985), Parallel algorithms, in: M. O'hEigeartaigh, J.K. Lenstra and A.H.G. Rinnooy Kan, eds., Combinatorial Optimization: Annotated Bibliographies (Wiley, Chichester), Ch. 8.

R.E. Ladner (1975), The circuit value problem is log space complete for *P*, SIGACT News 7.1, 18–20.

T.-H. Lai and S. Sahni (1984), Anomalies in parallel branch-and-bound algorithms, Comm. ACM 27, 594–602.

E.L. Lawler (1976), Combinatorial Optimization: Networks and Matroids (Holt, Rinehart and Winston, New York).

E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (eds.) (1985), The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization (Wiley, Chichester).

R. McNaughton (1959), Scheduling with deadlines and loss functions, Management Sci. 6, 1–12.

N. Megiddo (1982), Poly-log parallel algorithms for LP with an application to exploding flying objects. Unpublished manuscript.

D.E. Muller and F.P. Preparata (1975), Bounds to complexities of networks for sorting and for switching, J. Assoc. Comput. Mach. 22, 195–201.

F.P. Preparata and J. Vuillemin (1981), The cube-connected cycles: a versatile network for parallel computation, Comm. ACM 24, 300–309.

R.C. Prim (1957), Shortest connection networks and some generalizations, Bell System Tech. J. 36 1389–1401.

E.A. Pruul (1975), Parallel processing and a branch-and-bound algorithm, M. Sc. thesis, Cornell University, Ithaca, NY.

J.T. Schwartz (1980), Ultracomputers, ACM Trans. Programming Languages and Systems 2, 484–521.

H.J. Siegel (1977), Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks, IEEE Trans. Comput. 26, 153–161.

H.J. Siegel (1979), A model of SIMD machines and a comparison of various interconnection networks, IEEE Trans. Comput. 28, 907–917.

J.S. Squire and S.M. Palais (1963), Programming and design considerations of a highly parallel computer, Proc. AFIPS Spring Joint Computer Conf. 23, 395–400.

H.S. Stone (1971), Parallel processing with the perfect shuffle, IEEE Trans. Comput. 20, 153–161.

S.H. Unger (1958), A computer oriented toward spatial problems, Proc. IRE 46, 1744–1750.
L.G. Valiant (1982), Reducibility by algebraic projections, Enseign. Math. (2) 28, 253–268.
P. van Emde Boas (1985), The second machine class: models of parallelism, in: J. van Leeuwen and J.K. Lenstra, eds., Parallel Computers and Computations, CWI Syllabus 9, Centre for Mathematics and Computer Science, Amsterdam. 133–161.