Annals of Operations Research 14(1988) 245-289

Parallel Computing in Combinatorial Optimization

G.A.P. Kindervater
Erasmus University
P.O. Box 1738, 3000 DR Rotterdam

J.K. Lenstra
Centre for Mathematics and Computer Science
P.O. Box 4079, Amsterdam
and
Erasmus University
P.O. Box 1738, 3000 DR Rotterdam

This is a review of the literature on parallel computers and algorithms that is relevant for combinatorial optimization. We start by describing theoretical as well as realistic machine models for parallel computations. Next, we deal with the complexity theory for parallel computations and illustrate the resulting concepts by presenting a number of polylog parallel algorithms and $\mathcal{P}$-completeness results. Finally, we discuss the use of parallelism in enumerative methods.

1980 Mathematics Subject Classification: 90C27, 68Q15, 68Q25, 68Rxx.
Key Words and Phrases: Parallel computer, computational complexity, polylog parallel algorithm, $\mathcal{P}$-completeness, sorting, shortest paths, minimum spanning tree, matching, maximum flow, linear programming, knapsack, scheduling, traveling salesman, dynamic programming, branch and bound.

1. Introduction

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.

We will attempt to review the literature on parallel computers and algorithms as far as it is relevant for the area of combinatorial optimization. In comparison with a previous survey [43], the present paper not only mentions theoretical results but also addresses practical aspects of parallel combinatorial computing. For a broader survey which is, however, up to date only until July 1983, we refer to our annotated bibliography [42].

The organization of the paper is as follows.

Section 2 is concerned with *machine models* designed for parallel computations. Theoretical as well as realistic 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

© J.C. Baltzer A.G., Scientific Publishing Company

a specific interconnection network. Such networks are illustrated on the problems of matrix multiplication, determining a transitive closure, and finding a minimum spanning tree. We also discuss the simulation of theoretical models by realistic ones. In Sections 3, 4 and 5, we will restrict ourselves to theoretical models; in Section 6, we consider existing parallel computers as well.

Section 3 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 *$\mathcal{P}$-complete* under log-space transformations.

Section 4 gives examples of *polylog parallel algorithms* for elementary problems like finding the maximum and sorting, for finding shortest paths, a minimum spanning tree and a traveling salesman tour by the double minimum spanning tree heuristic, and for three problems from scheduling theory. We also outline a *randomized* polylog parallel algorithm for the maximum cardinality matching problem.

Section 5 discusses the *$\mathcal{P}$-completeness* of a variety of problems: linear programming, finding a maximum flow in a network, list scheduling, and finding a traveling salesman tour by the nearest neighbor heuristic.

Section 6 reviews the use of parallelism in *enumerative methods* for $\mathcal{NP}$-hard problems. We will discuss results in three directions: practical experience with the implementation of dynamic programming and branch and bound on existing parallel computers; worst case examples exhibiting various forms of anomalous behavior; and some initial results on the design and analysis of a model for the distribution of a tree search procedure over several parallel processors.

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.

2. 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 [24]. He distinguishes four classes of machines (cf. Figure 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.

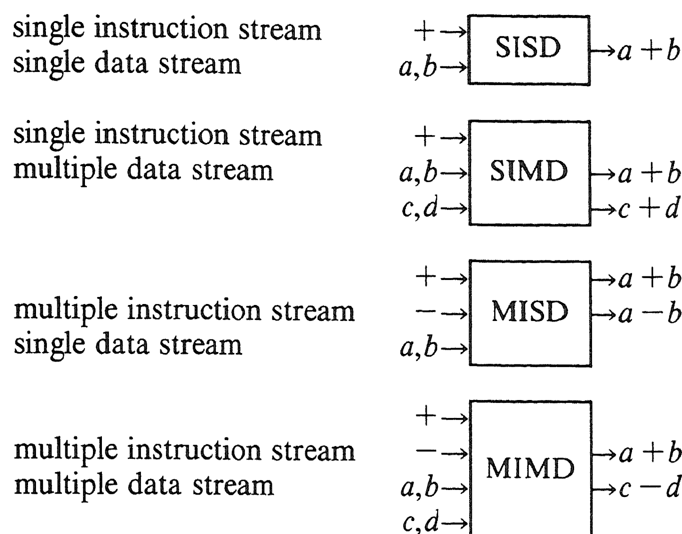


FIGURE 1. The classification of Flynn

(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) and the Goodyear/MPP (Massively Parallel Processor) belong 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 Intel/iPSC (Intel's Personal SuperComputer) is an example of an asynchronous MIMD machine.

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 6. *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 6) 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 [64], 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 help us in investigating the intrinsic parallelism in problems and algorithms. They are therefore of great theoretical interest, but current technology prohibits their realization.

In an *ultracomputer*, each processor has its own memory and 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.

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

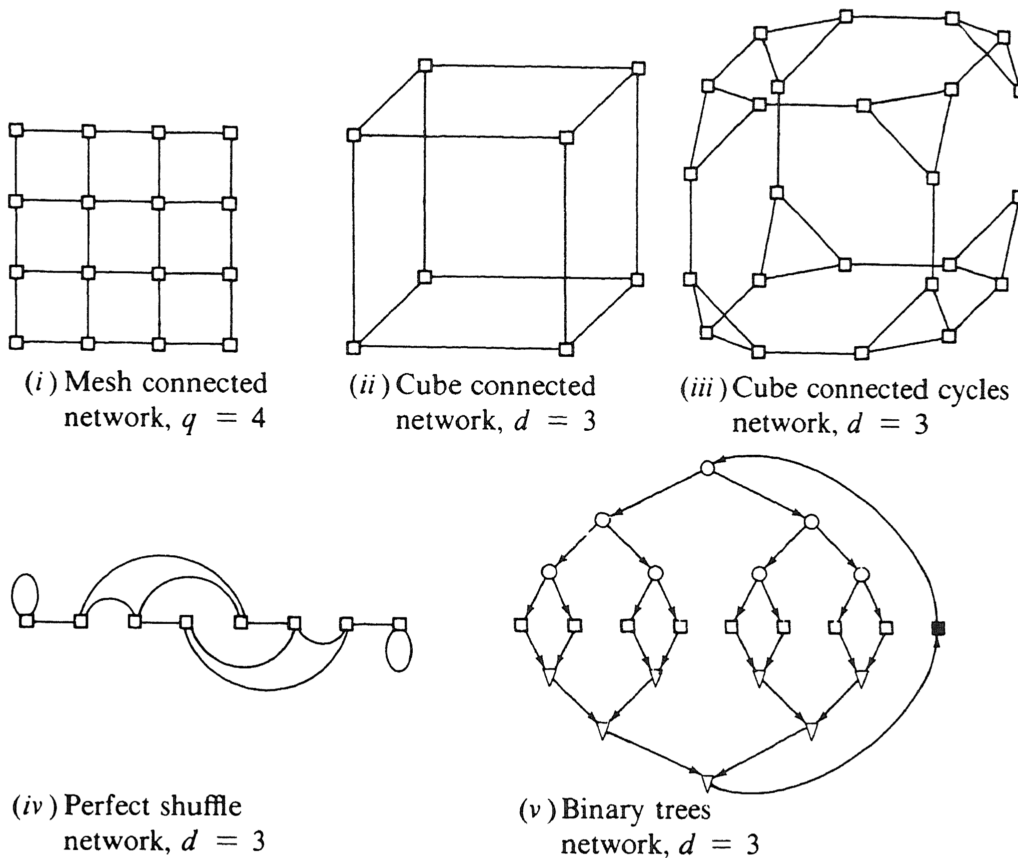
(*i*) *Two-dimensional mesh connected* network [70]. Each processor is identified with an ordered pair (i,j) $(i,j=1,...,q)$, and processor (i,j) is connected to processors $(i\pm1,j)$ and $(i,j\pm1)$, provided they exist. Note that $d_1=4$ and $d_2=2(q-1)=\Theta(\sqrt{p})$.

(*ii*) *Cube connected* network [67]. 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 [60]. 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 [68]. There are $p=2^d$ processors with interconnections $(i, 2i-1)$, $(i+p/2,2i)$, $(2i-1,2i)$ for $i=1,...,p/2$. 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 [7]. There are $p=3\cdot2^d-2$ 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 $d_2=\Theta(\log p)$.



(i) Mesh connected network, $q = 4$

(ii) Cube connected network, $d = 3$

(iii) Cube connected cycles network, $d = 3$

(iv) Perfect shuffle network, $d = 3$

(v) Binary trees network, $d = 3$

FIGURE 2. Five interconnection networks

All these networks can simulate each other quite efficiently; see SIEGEL [65,66] 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.

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 Figure 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 $\Sigma_k a_{ik} b_{kj}$ and that the procedure is best possible in terms of speedup and processor utilization. Furthermore, only one copy of each matrix element has to be kept in storage. This is a typical example of a systolic algorithm performed on an SIMD machine and suitable for VLSI implementation.

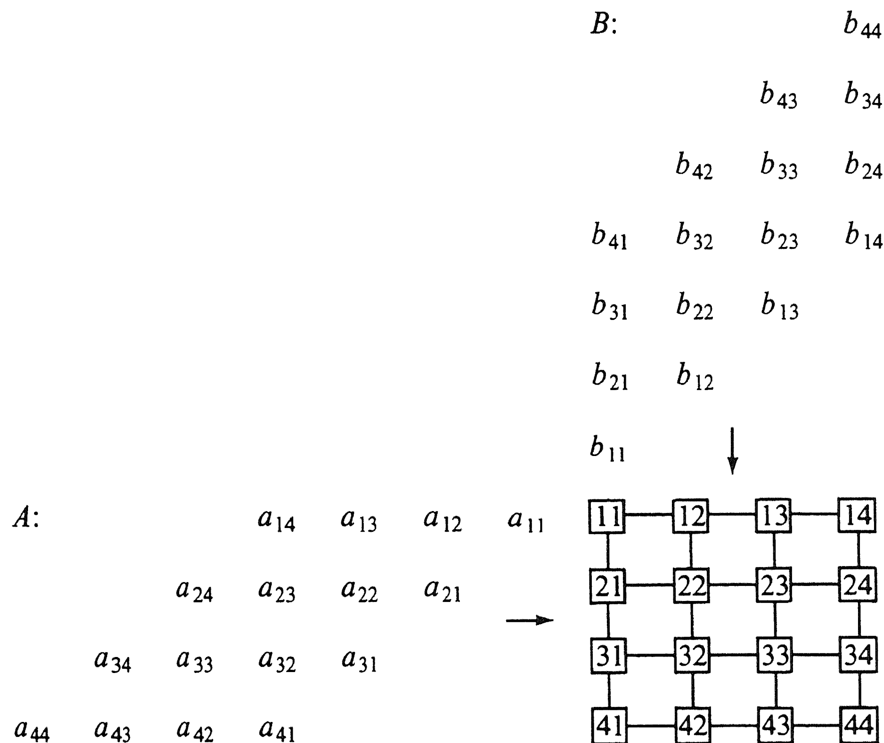


FIGURE 3. Matrix multiplication on a mesh connected network

EXAMPLE 2. *Transitive closure* [31]. 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 $(i=1,...,2^d)$ and suppose that P_i stores the ith element e_i of S $(i \leqslant n)$. 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 $e^{(1)},...,e^{(m)}$ 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* [6]. Given a complete undirected graph G with vertex set $\{1,...,n\}$ 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 [61] and DIJKSTRA [20]. 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 $c_{i\cdot}$ 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 $i = 2,...,n$, 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 $l_i \leqslant c_{ii^*}$ 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 sequen-

tially. This modified algorithm still runs in $O(n\log n)$ time, but now using $O(n/\log n)$ processors with a processor utilization of $O(1)$.

The most common paracomputer model is the PRAM (Parallel Random Access Machine). 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. Simulation of the theoretical PRAM model by ultracomputers with a bounded degree network that allows for fast communication is usually done in two phases.

First, the use of the shared memory is eliminated. We introduce an intermediate model, the MPC (Module Parallel Computer). In an MPC, each processor has its own memory and is connected to all other processors. By sending messages, a processor can access a variable stored in the memory of another processor. However, if several processors try to access a variable stored in the memory of the same processor simultaneously, only one will succeed and the others receive a message that the access failed. An n-processor MPC can simulate a computational step of an (n,m)-PRAM (a PRAM with n processors and a shared memory of size m) with high probability in time $O(\log n)$ [71] or in deterministic time $O(\log m)$ [2]. The proof of the probabilistic bound is constructive, but for the deterministic simulation only an existence proof is given. The problem of finding a constructive deterministic simulation of a PRAM step in logarithmic time is still open.

The second phase eliminates the use of the complete interconnection network. One step of an n-processor MPC can be simulated in $O(\log n)$ steps by a bounded degree network with n processors [2].

Combining the two phases, we conclude that a step of an (n,m)-PRAM requires probabilistic time $O(\log^2 n)$ or deterministic time $O(\log m \log n)$ on a bounded degree network.

KARLIN & UPFAL [38] describe a direct simulation of a PRAM. They show that T steps of an (n,m)-PRAM can be simulated in $O(T\log m)$ steps by a bounded degree network, with probability tending to 1 as n or T goes to infinity. Until today, no deterministic simulation with the same time characteristic is known.

In Sections 3, 4 and 5, we will restrict ourselves to the PRAM paracomputer model, which lends itself better to complexity considerations and to the explanation of parallel algorithms. In Section 6, we will encounter a variety of existing parallel architectures, some of which are quite different from the models described above.

3. 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 [15] for a more thorough exposition and to JOHNSON [37] 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* [12, 28]: *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 [73] for a survey. It holds, for example, in the case that the machine model is a PRAM and $T(n)=n^{O(1)}$ (i.e., a polynomial function of problem size).

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 tim* $(\log n)^{O(1)}$, i.e., in time that is polynomially bounded in the logarithm of th problem size n. Some examples are given in Section 4; other, more complicated, examples are finding a maximum flow in a planar graph [36] and linear programming with a fixed number of variables [57]. 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* $\mathcal{P}$ or, more precisely, $\mathcal{P}$*-complete* under log-space transformations: they belong to $\mathcal{P}$ and any other problem in $\mathcal{P}$ is reducible to them by a transformation using logarithmic work space. Examples will be discussed in Section 5; 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 therefore, 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 review. We refer to JOHNSON [37] for further details and more references.

4. POLYLOG PARALLEL ALGORITHMS

We will now describe polylog parallel algorithms for ten problems. Examples 5, 6 and 7 deal with basic operations on a set of numbers, Examples 8, 9 and 10 discuss the problems of finding shortest paths, a minimum spanning tree and a traveling salesman tour by the double minimum spanning tree heuristic, and Examples 11, 12 and 13 are concerned with the scheduling of a set of jobs on parallel machines. Example 14 outlines a randomized polylog parallel algorithm for the maximum cardinality matching problem. Other problems that are solvable in polylog parallel time have been mentioned in Section 3 and will return in Section 5.

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 (non-randomized) 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

$$\textbf{par}\ [a \leqslant i \leqslant z]\ s_i$$

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 $a_n, a_{n+1},...,a_{2n-1}$. Consider the following procedure:

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

The computation is illustrated by means of a *binary tree* in Figure 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.

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 $O(1)$.

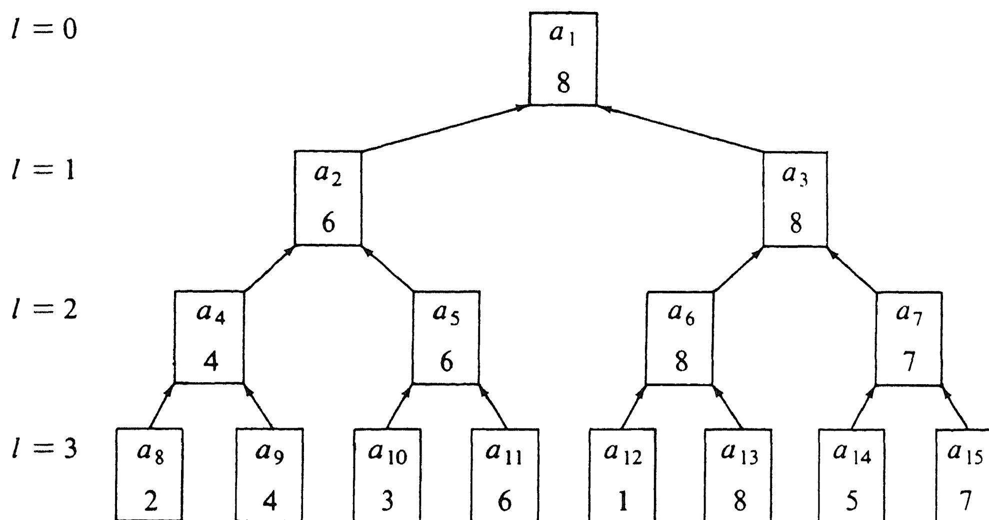


FIGURE 4. Maximum finding: an instance with $n=8$

EXAMPLE 6. *Partial sums* [17]. Given n numbers a_n, $a_{n+1},...,a_{2n-1}$ with $n=2^m$, one wishes to find the partial sums $a_n+...+a_{n+j}$ for $j=0,...,n-1$. Consider the following procedure:

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

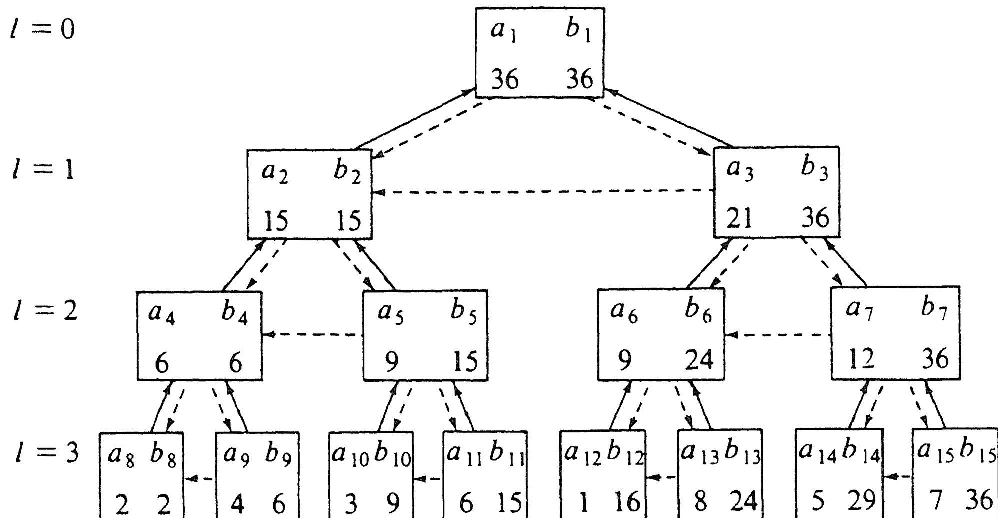


FIGURE 5. Partial sums: an instance with $n=8$

The computation is illustrated in Figure 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 $b_{n+j}=a_n+...+a_{n+j}$ for $j=0,...,n-1$.

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* [58]. Given n numbers $a_1,...,a_n$, one wishes to renumber them such that $a_1\leqslant...\leqslant a_n$. We assume, for simplicity, that $a_i\neq a_j$ if $i\neq j$. Consider the following procedure:

$$\textbf{par}\ [1\leqslant i,j\leqslant n]\ \rho_{ij}\leftarrow \textbf{if}\ a_i\leqslant a_j\ \textbf{then}\ 1\ \textbf{else}\ 0;$$

$$\textbf{par}\ [1\leqslant j\leqslant n]\ \pi_j\leftarrow \mathrm{sum}\{\rho_{ij}\,|\,1\leqslant i\leqslant n\};$$

$$\textbf{par}\ [1\leqslant j\leqslant n]\ a_{\pi_j}\leftarrow a_j.$$

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* [16]. Given a complete directed graph with vertex set $\{1,...,n\}$ and a length c_{ij} for each arc (i,j), one wishes to find the shortest path lengths for all pairs of vertices. LAWLER [51] gives an algorithm 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 $d_{ij}^{(2l)}=\min_{k\in\{1,...,n\}}\{d_{ik}^{(l)}+d_{kj}^{(l)}\}$. Taking into account that a shortest path, if it exists, contains at most $n-1$ arcs, we obtain the following algorithm:

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

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. *Minimum spanning tree* [63]. The Prim-Dijkstra algorithm for the minimum spanning tree problem was discussed in Example 4. A minimum spanning tree of a complete undirected graph G with vertex set $\{1,...,n\}$ and a length c_{ij} for each edge $\{i,j\}$ can also be found in $O(n^2)$ time by an algorithm due to SOLLIN [8]. We assume that the edge lengths are all distinct (if not, we number the edges in some arbitrary way and say that from two edges with the same length the one with the lowest number is smaller). The algorithm starts with n components, each consisting of a different vertex, and with an empty set of edges belonging to the tree. At each step of the algorithm, each component finds an edge of minimum length between any of its own vertices and a vertex of a different component. Since all edge lengths are different, the edges thus obtained do not form cycles between the components and are added to the minimum spanning tree. We now merge the components which are connected by the newly found edges into a new one, and perform a next step of the algorithm as long as there is more than one component left. Because the number of components is at least halved at each step, the algorithm terminates after at most $\lceil\log n\rceil$ steps.

In the algorithm below, for each component a representative is chosen. Two

vertices belong to the same component if they have the same representative. Let r_i $(i=1,...,n)$ denote the representative of the component to which vertex i belongs.

par $[1 \leq i \leq n]$ $r_i \leftarrow i$;

for $l \leftarrow 1$ **to** $\lceil \log n \rceil$ **do**

  par $[1 \leq i \leq n]$

    find k such that $r_k \neq r_i$ & $c_{ik} = \min\{c_{ij} | 1 \leq j \leq n, r_j \neq r_i\}$,

    if k does not exist **then** a minimum spanning tree has

      been found & the algorithm is stopped,

    $t_i \leftarrow k$;

  par $[1 \leq i \leq n]$

    find k such that $r_k = r_i$ & $c_{kt_k} = \min\{c_{jt_j} | 1 \leq j \leq n, r_j = r_i\}$,

    $s_i \leftarrow k$ & $t_i \leftarrow t_k$;

  par $[1 \leq i \leq n]$ $s_i \leftarrow$ **if** $t_{t_i} = s_i$ & $r_i < r_{t_i}$ **then** 0 **else** s_i;

  par $[1 \leq i \leq n]$ **if** $r_i = i$ & $s_i \neq 0$ **then** add edge $\{s_i, t_i\}$ to the tree;

  par $[1 \leq i \leq n]$ $r_i \leftarrow$ **if** $s_i = 0$ **then** r_i **else** r_{t_i};

  for $l^* \leftarrow 1$ **to** $\lceil \log n \rceil$ **do par** $[1 \leq i \leq n]$ $r_i \leftarrow r_{r_i}$.

Each step of the algorithm does the following. First, each component finds the edge of minimum length between any vertex of itself and one of a different component. Of the edges found twice at the same step, one copy is eliminated. The remaining edges are added to the tree. Finally, components are merged by finding a common representative, using a recursive doubling technique which will appear in Example 13. One step of the algorithm can be performed in $O(\log n)$ time on $O(n^2/\log n)$ processors by application of the procedure of Example 5 with maximization replaced by minimization. The complete algorithm requires $O(\log^2 n)$ time on $O(n^2/\log n)$ processors, with a processor utilization of $O(1/\log n)$.

EXAMPLE 10. *Double minimum spanning tree tour for the traveling salesman* [44]. In the traveling salesman problem, one is given a complete undirected graph G with vertex set $\{1,...,n\}$ and a length d_{ij} for each edge $\{i,j\}$ and one wishes to find a Hamiltonian cycle (i.e., a cycle passing through each vertex exactly once) of minimum total length. This is a well-known $\mathcal{NP}$-hard problem, and rather than trying to solve it to optimality one might decide to find an approximate solution in polynomial time. One such approximation algorithm is the double minimum spanning tree heuristic. It consists of three phases:

(*i*) Construct a minimum spanning tree. Using the routine of Example 9, we can perform this phase in $O(\log^2 n)$ time on $O(n^2/\log n)$ processors.

(*ii*) Double the edges of the minimum spanning tree and construct an Eulerian cycle. We do not go into the details here, but this phase can be done within the same time and processor bounds using the techniques from [3].

(*iii*) Start at a given vertex and traverse the edges, skipping vertices visited before. We first have to find the first occurrence of each vertex and then eliminate all duplications. Let $v_1,...,v_i,...,v_{2n-1}$ denote the Eulerian tour obtained in the previous phase, where v_i is the ith vertex of the tour. We proceed as follows.

$$\textbf{par } [1 \leqslant i,j \leqslant 2n-1]\ c_{ij} \leftarrow \textbf{if } v_i = v_j \textbf{ then } 1 \textbf{ else } 0;$$

$$\textbf{par } [1 \leqslant i \leqslant 2n-1]\ d_i \leftarrow \max\{0, 1-\text{sum}\{c_{ij} | 1 \leqslant j \leqslant i-1\}\};$$

$$\textbf{par } [1 \leqslant i \leqslant 2n-1]\ s_i \leftarrow \text{sum}\{d_j | 1 \leqslant j \leqslant i\}.$$

Note that $d_i = 1$ if v_i occurs for the first time in the tour, $d_i = 0$ otherwise, and that s_i denotes the number of different vertices in $v_1,...,v_i$. We obtain the tour $t_1 - t_2 - ... - t_n - t_1$ by:

$$\textbf{par } [1 \leqslant i \leqslant 2n-1]\ \textbf{if } d_i = 1 \textbf{ then } t_{s_i} \leftarrow v_i.$$

Using the partial sums algorithm from Example 6, we can implement phase (*iii*) within the same resource bounds as the previous phases. So, we end up with an algorithm that runs in $O(\log^2 n)$ time on $O(n^2/\log n)$ processors. Since the sequential algorithm takes $O(n^2)$ time, we have a processor utilization of $O(1/\log n)$.

EXAMPLE 11. *Preemptive scheduling of identical machines* [18]. Given m identical machines M_i $(i = 1,...,m)$ and n jobs J_j, each with a processing time p_j $(j = 1,...,n)$, 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 $1 \leqslant i \leqslant m$ and $0 \leqslant s \leqslant t$, 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.

An optimal schedule can be found in $O(n)$ time by the classical *wrap around rule* of MCNAUGHTON [56]. 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:

$t^* \leftarrow \max\{\max\{p_j \mid 1 \leqslant j \leqslant n\}, \text{sum}\{p_j \mid 1 \leqslant j \leqslant n\}/m\}$;

$s \leftarrow 0$; $i \leftarrow 1$;

for $j \leftarrow 1$ **to** n **do**

  if $s + p_j \leqslant t^*$

  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 Figure 6. There are two global parameters that are updated sequentially 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:

$t^* \leftarrow \max\{\max\{p_j \mid 1 \leqslant j \leqslant n\}, \text{sum}\{p_j \mid 1 \leqslant j \leqslant n\}/m\}$;

par $[1 \leqslant j \leqslant n]$ $q_j \leftarrow \text{sum}\{p_k \mid 1 \leqslant k \leqslant j-1\}$;

par $[1 \leqslant j \leqslant n]$

    $s_j \leftarrow q_j \bmod t^*$, $i_j \leftarrow \lfloor q_j / t^* \rfloor + 1$,

    if $s_j + p_j \leqslant t^*$

    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 $O(1)$.

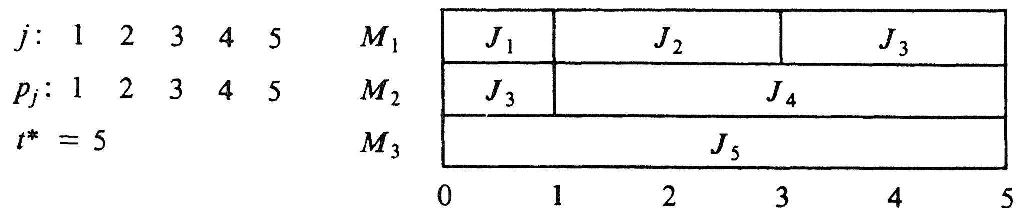


FIGURE 6. Preemptive scheduling: an instance with $m=3$ and $n=5$

EXAMPLE 12. *Preemptive scheduling of uniform machines* [55]. Given are m machines M_i, each with a speed s_i $(i=1,...,m)$, and n jobs J_j, each with a processing requirement p_j $(j=1,...,n)$. If J_j is completely processed on M_i, the processing time is p_j/s_i on machine M_i. One wishes to find a preemptive schedule of minimum length.

An optimal schedule can be found in $O(n + m\log m)$ time by an algorithm

due to GONZALEZ & SAHNI [30]. As in Example 11, the algorithm first finds an obvious lower bound t^* on the minimum schedule length and then constructs a schedule of length t^*. Assume that the machines are ordered according to nonincreasing speeds and that the $m-1$ largest jobs, ordered according to nonincreasing processing requirements, precede the $n-m+1$ remaining jobs. The Gonzalez-Sahni algorithm is as follows:

$t^* \leftarrow \max\{(p_1/s_1),(p_1+p_2)/(s_1+s_2),\ldots,$
$(p_1+\ldots+p_{m-1})/(s_1+\ldots+s_{m-1}),(p_1+\ldots+p_n)/(s_1+\ldots+s_m)\}$;
construct a composite machine with speed s_i in the interval $[(i-1)t^*,it^*)$ $(i=1,\ldots,m)$ and speed 0 in $[mt^*,\infty)$;
for $j \leftarrow 1$ **to** n **do**
find the latest possible interval $[s,s+t^*)$ such that the composite machine can process J_j, assign the interval $[s,s+t^*)$ to J_j, replace the speed of the composite machine at time $s+t$ by the original speed of the machine at time $s+t^*+t$, for all $t>0$.

After scheduling the $m-1$ largest jobs, the composite machine has in any interval of length t^* with positive speed a processing capacity that is greater than the processing requirement of any of the remaining jobs. The parallel algorithm first schedules the $m-1$ largest jobs; after that, the remaining jobs are scheduled in the same way as in Example 11. The first phase of Martel's algorithm is only sketched here; the full story can be found in his paper.

For each of the large jobs, we compute an interval to which we would like to assign that job. Martel observes that, if the intervals of two consecutive jobs overlap, we may combine them into one compound job with a processing requirement equal to the sum of the processing requirements of both jobs and find an interval of twice the original length on the composite machine. We group consecutively overlapping jobs together. If a group contains an odd number of jobs, we schedule the first job in its interval (and revise the composite machine as in the sequential algorithm) and combine the second with the third job, the fourth with the fifth job and so on, otherwise we combine the first with the second job, the third with fourth job and so on. We continue this process until there are at most two compound jobs left. These are scheduled sequentially. We now call the same procedure for each of the compound jobs, with the individual jobs of the compound job as job set and with the interval assigned to the compound job (extended to infinity with speed 0) as composite machine. Since at each recursive step the number of jobs in a new problem decreases by a constant factor, the algorithm terminates after a logarithmic number of such steps.

The entire algorithm can be implemented in $O(\log n+\log^3 m)$ time on $O(n)$ processors. It uses the sorting algorithm of AJTAI, KOMLÓS & SZEMERÉDI [1], which requires $O(\log n)$ time and $O(n)$ processors (and thereby provides a substantial improvement over the algorithm from Example 7).

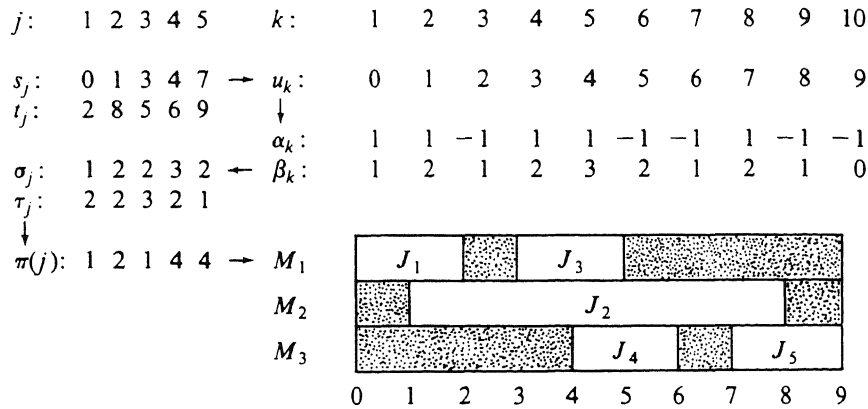


FIGURE 7. Scheduling fixed jobs: an instance with $n = 5$

EXAMPLE 13. *Scheduling fixed jobs* [18]. Given n jobs J_j, each with a starting time s_j and a completion time t_j ($j = 1,...,n$), 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 [32]. 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 $(s_1, t_1, ..., s_n, t_n)$ in nondecreasing order in $(u_1, ..., u_{2n})$ 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.

Figure 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 $j=1,...,n$:

sort $(s_1,t_1,...,s_n,t_n)$ in nondecreasing order in $(u_1,...,u_{2n})$ whereby,

if $t_j=s_k$ for some j & k, t_j precedes s_k;

par $[1 \leqslant k \leqslant 2n]$ $\alpha_k \leftarrow$ **if** $u_k \sim s_j$ **then** 1 **else** -1;

par $[1 \leqslant k \leqslant 2n]$ $\beta_k \leftarrow \text{sum}\{\alpha_l | 1 \leqslant l \leqslant k\}$;

par $[1 \leqslant k \leqslant 2n]$

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:

par $[1 \leqslant j \leqslant n]$

find k such that $\tau_k=\sigma_j$ & $t_k=\max\{t_l | t_l \leqslant s_j, \tau_l=\sigma_j\}$,

$\pi(j) \leftarrow$ **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 using recursive doubling. The chains formed by the arcs $(j,\pi(j))$ are collapsed simultaneously in a logarithmic number of steps (cf. Figure 8):

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

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

par $[1 \leqslant j \leqslant n]$ 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.

EXAMPLE 14. *Maximum cardinality matching* [39]. Given an undirected graph with vertex set V and edge set E, one wishes to find a matching of maximum cardinality. A matching is a set of vertex disjoint edges. It is perfect if each vertex is incident to an edge.

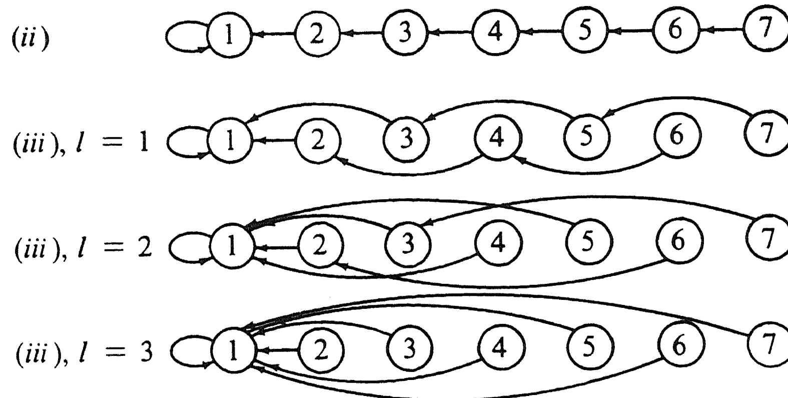


FIGURE 8. Scheduling fixed jobs:
finding the first preceding job on the same machine

LOVÁSZ [54] gave a randomized algorithm for deciding whether a graph has a perfect matching. It is based on the following theorem of Tutte: a graph on n vertices has a perfect matching if and only if the determinant of the $n \times n$ matrix $B=(b_{ij})$, with $b_{ij}=x_{ij}$ if $\{i,j\} \in E$ and $i<j$, $b_{ij}=-x_{ij}$ if $\{i,j\} \in E$ and $i>j$, and $b_{ij}=0$ otherwise, is not identically zero in the variables x_{ij}. Now, we choose a random number N, substitute for each variable x_{ij} a random number from $\{1,...,N\}$ and compute the determinant. If the determinant of B is identically zero, then we find the value zero. Otherwise, the probability that we get zero is very small. CSANKY [13] showed that computing a determinant belongs to $\mathfrak{NC}$. Therefore, the problem of deciding whether a graph has a perfect matching belongs to $\mathfrak{RNC}$, i.e., the class of problems solvable by a randomized algorithm in polylog time on a polynomial number of processors.

The randomized algorithm of KARP, UPFAL & WIGDERSON [39] which actually constructs a perfect matching in polylogarithmic time, if it exists, is also based on Tutte's theorem. It is quite complicated, and we refer to their paper. As a result, the problems of constructing a maximum cardinality matching and of constructing a matching of maximum weight in a graph whose edge weights are given in unary notation also belong to $\mathfrak{RNC}$. The complexity of the maximum cardinality matching problem with respect to deterministic parallel computations is an open question, even for bipartite graphs.

5. $\mathcal{P}$-COMPLETENESS

The first $\mathcal{P}$-complete problem was identified by COOK [14]. It involves the *solvability of a path system* and is proved $\mathcal{P}$-complete under log-space transformations 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.

EXAMPLE 15. *Circuit value* [46,27,29]. 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 or FALSE? Cf. Figure 9.

The circuit value problem is trivially in $\mathcal{P}$. LADNER [46] 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 $\mathcal{P}$-complete.

GOLDSCHLAGER [27] extended this result to the cases of *monotone* circuits, which have no NOT gates, and *planar* circuits, which have a cross free planar embedding, by giving log space transformations from the circuit value problem. Circuits which have in addition to input and output gates, only NAND gates (a NAND gate is an AND gate followed by a NOT gate) or NOR gates (a NOR gate is an OR gate followed by a NOT gate) are able to simulate arbitrary circuits; this not hard to see. Therefore, the circuit value problem is also $\mathcal{P}$-complete for circuits with only NAND gates or only NOR gates. GOLDSCHLAGER, SHAW & STAPLES [29] showed that all these results still hold if each input gate has fan-out one (it appears once as input to another gate) and each other gate has fan-out at most two.

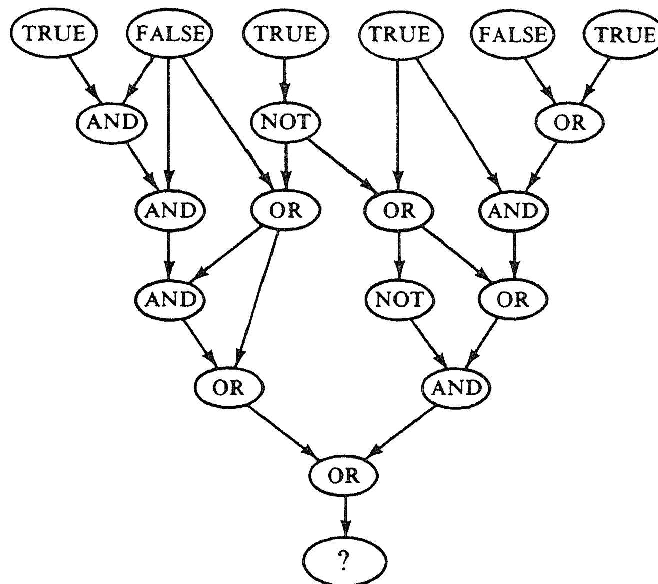


FIGURE 9. A logical circuit

EXAMPLE 16. *Linear programming* [21,72]. 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 $\mathcal{P}$ [40]. DOBKIN, LIPTON & REISS [21] established $\mathcal{P}$-completeness 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 $\mathcal{P}$-complete. VALIANT [72] 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 $\mathcal{P}$-complete 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 17. *Maximum flow* [29]. 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 $\mathcal{P}$ [22]. It was shown to be $\mathcal{P}$-complete 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 [36].

(*ii*) The problem is solvable in randomized 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, from the complexity status of the maximum cardinality matching problem as described in Example 14.

EXAMPLE 18. *List scheduling* [34]. In the multiprocessor scheduling problem, one is given m identical machines M_i ($i = 1,...,m$) and n jobs J_j, each with a processing time p_j ($j = 1,...,n$), and one wishes to find a nonpreemptive schedule of minimum length. A nonpreemptive schedule assigns to each J_j a pair (M_i,s), with $1 \leq i \leq m$ and $s \geq 0$, indicating that J_j is to be processed by M_i from time s to time $s + p_j$. A nonpreemptive schedule is feasible if the processing intervals on M_i are nonoverlapping for all i. It is optimal if the maximum job completion time is minimum.

This is an $\mathcal{NP}$-hard problem. A popular approximation algorithm is the list

scheduling heuristic, whereby a priority list of the jobs is given and at each step the earliest available machine is scheduled to process the first available job on the list. More formally:

> **for** $i \leftarrow 1$ **to** m **do** $s_i \leftarrow 0$;
>
> **for** $j \leftarrow 1$ **to** n **do**
>
> $\quad i^* \leftarrow \min\{i \mid s_i \leqslant s_k, k = 1,...,m\}$,
>
> $\quad$ assign (M_{i^*}, s_{i^*}) to J_j,
>
> $\quad s_{i^*} \leftarrow s_{i^*} + p_j$.

An example is given in Figure 10. The sequential algorithm requires $O(n \log m)$ time. We will show that the associated list scheduling problem of deciding about the resulting schedule length is $\mathcal{P}$-complete for $m \geqslant 2$.

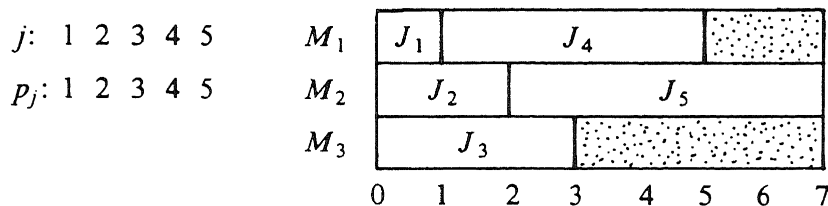


FIGURE 10. List scheduling: an instance with $m = 3$ and $n = 5$

Consider an instance of the circuit value problem with only input and NOR gates. First, we number the gates such that each NOR gate receives its inputs from higher numbered gates. We then give the incoming arcs to NOR gate i the weights 4^{2i} and 4^{2i+1}. The output arc gets weight 4. Cf. Figure 11. We construct the list of jobs as follows. The first has a processing time that equals the sum of the weights of all outgoing arcs of TRUE inputs. In decreasing order of i, we put seventeen jobs on the list for NOR gate i, one with length $2 \cdot 4^{2i+1}$, fourteen with length $4^{2i}/2$, and two with length $(4^{2i} + V_i)/2$, where V_i is the sum of the weights of the outgoing arcs of gate i. On two machines, the corresponding list schedule has the property that, after scheduling the first job or after scheduling all jobs associated with a gate, the difference in the completion times of both machines is equal to the sum of the weights of all arcs that have been computed to represent a TRUE value and have not yet been considered as input. In the end, the difference in the completion time is 4 if and only if the circuit computes the value TRUE. Checking these statements is left as an exercise to the reader. Since the transformation can be performed in logarithmic work space, the list scheduling problem is $\mathcal{P}$-complete for $m \geqslant 2$.

EXAMPLE 19. *Nearest neighbor tour for the traveling salesman* [44]. Given a complete undirected graph G with vertex set $\{1,...,n\}$, a length d_{ij} for each edge $\{i,j\}$ and two specified vertices v_1 and v_2, does the Hamiltonian cycle constructed by the nearest neighbor heuristic, when started at vertex v_1, visit

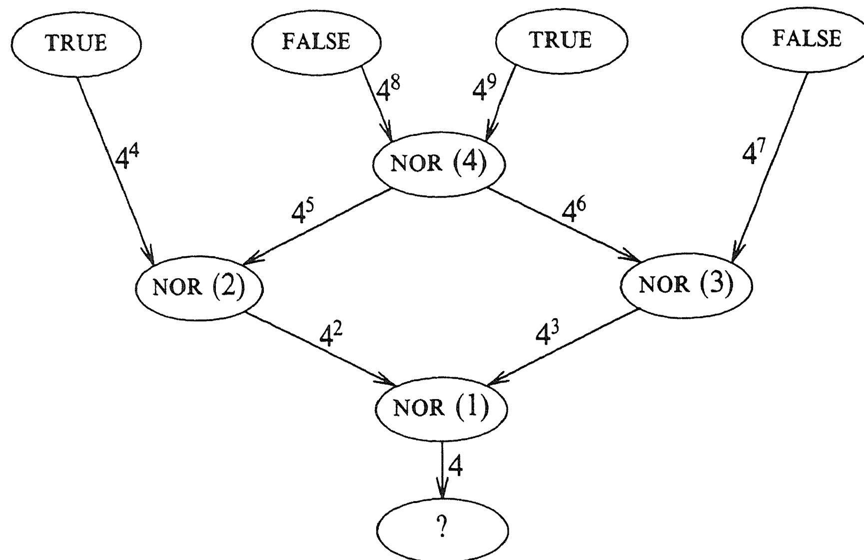


FIGURE 11. A circuit with numbered gates and weights assigned to the edges

vertex v_2 as the last one before returning to vertex v_1? The nearest neighbor heuristic is probably the simplest approximation algorithm for the traveling salesman problem. It proceeds as follows.

(*i*) Start at a given vertex.

(*ii*) Among all vertices not yet visited, choose as the next vertex the one that is closest to the current vertex. Repeat this step until all vertices have been visited.

(*iii*) Return to the starting vertex.

We will show that the nearest neighbor problem is $\mathcal{P}$-complete. For each instance of the circuit value problem with only input gates with fan-out one and NAND gates with fan-out at most two, we construct a graph in such a way that the circuit value of the considered instance is TRUE if and only if the nearest neighbor problem returns a 'yes' answer.

Let the circuit have m gates. We number them from 1 up to m such that they receive their inputs from gates with a lower number. Each gate in the circuit is represented by a subgraph. The nearest neighbor tour will visit the subgraphs in the order in which the corresponding gates are numbered in the circuit. This ensures that if the tour visits a subgraph corresponding to a non-input gate, it has passed the subgraphs corresponding to its input gates.

For NAND gate k ($k<m$) with fan-out two ($\alpha_k = \alpha_i$ NAND α_j), we construct the subgraph as shown in Figure 12. The vertex pairs ① - ② are used to connect the different subgraphs. If gate i is input to gate k, a ① - ② pair appears as output in the subgraph for gate i and also as input in the subgraph for gate k. The edge length zero assures that corresponding vertices 1 and 2 are always

neighbors in the obtained tour. If the fan-out is one (zero), we construct the same subgraph with one arbitrary ① - ② pair of output vertices (without output vertices). The subgraph is constructed in such a way that if the nearest neighbor tour enters the subgraph at vertex A from subgraph $k-1$, it leaves this subgraph through vertex B to subgraph $k+1$. We associate a TRUE (FALSE) value with this subgraph if the nearest neighbor tour on its way from A to B passes (does not pass) through the output vertices.

When the tour arrives at vertex A from subgraph $k-1$, there are three possibilities.

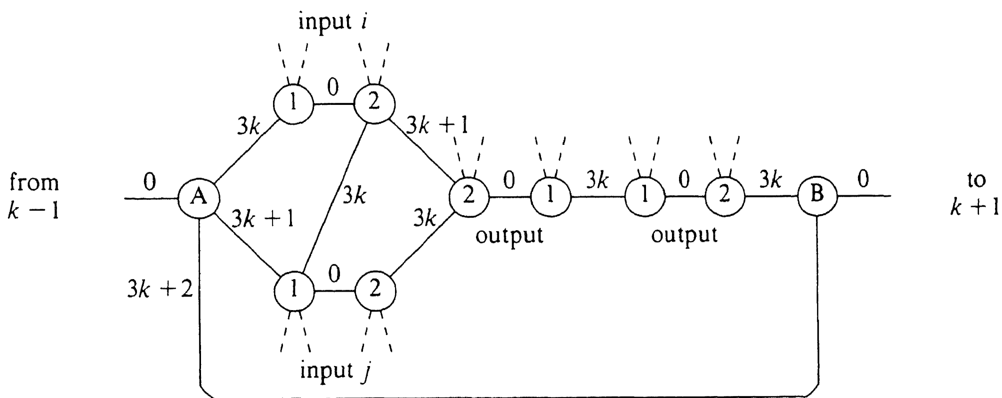


FIGURE 12. The representation of NAND gate k

(*i*) Inputs i and j have both been visited already. In this case the tour r go directly to vertex B and then it will choose the edge of length zer subgraph $k+1$. This will be the only case where the output vertices a not immediately visited. Note that as a result either output vertex 2 has it corresponding vertex 1 left as its only unvisited neighbor within the subgraph. See Figure 13.

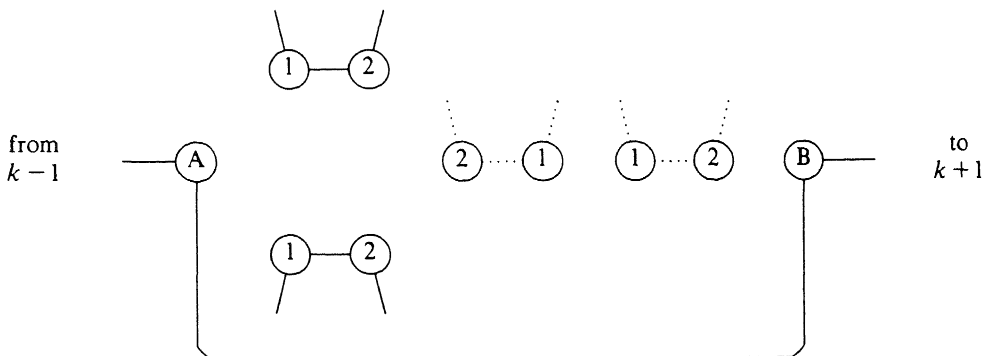


FIGURE 13. TRUE NAND TRUE → FALSE

(*ii*) Either input i or input j is still unvisited. The tour will choose vertex 1 of this unvisited input as next vertex, since the edge length is less than the distance to vertex B. From here it goes to the corresponding vertex 2 (edge length is zero). As noted under (*i*), this vertex 2 has no unvisited neighbors in the subgraph where it appears as output. Therefore, the next vertex must belong to subgraph k, i.e., the tour arrives at the outputs. Because edge lengths in a subgraph are proportional to the number of that subgraph and outputs belong to subgraphs with a higher number, the nearest neighbor algorithm will visit all output vertices and after that vertex B before leaving subgraph k to subgraph $k+1$. Cf. Figures 14 and 15.

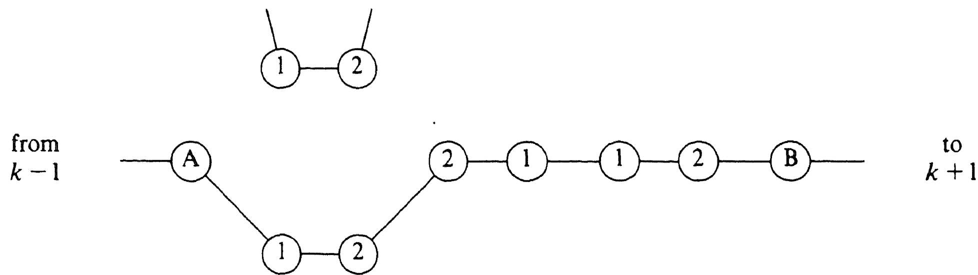


FIGURE 14. TRUE NAND FALSE → TRUE

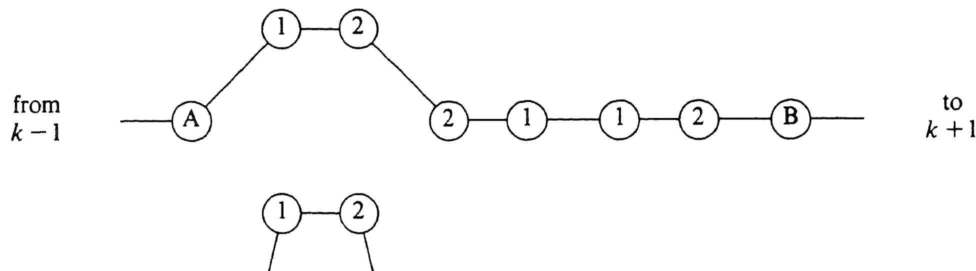


FIGURE 15. FALSE NAND TRUE → TRUE

(*iii*) Both inputs are unvisited. The tour will pass through all vertices of subgraph k before going to subgraph $k+1$ (Figure 16).

Note that in all cases all unvisited input vertices are included in the tour.

To summarize the results, the nearest neighbor tour from A to B passes through the output vertices if and only if at least one of the input vertices is not yet visited. In the circuit value problem, this corresponds to the fact that a NAND gate produces the value TRUE if and only if at least one of the inputs is FALSE.

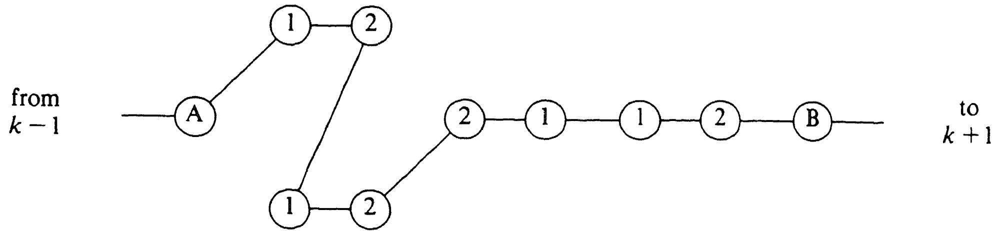
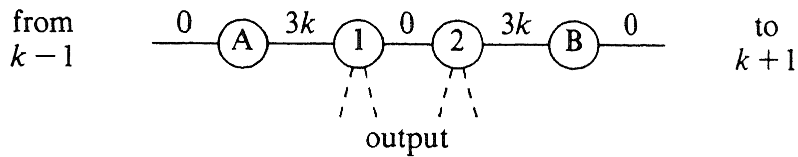


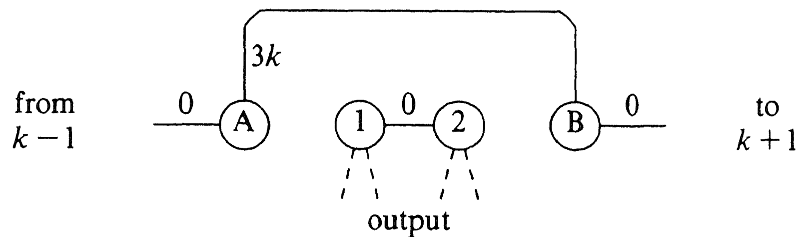
FIGURE 16. FALSE AND NAND FALSE → TRUE

For TRUE and FALSE inputs we construct the subgraphs as shown in Figure 17. The representation of NAND gate m (the last one) has a somewhat special structure. The output vertices are replaced by a vertex C. Both vertex B and C are connected to input 1 (see Figure 18). If the tour arrives at vertex A of this gate and we are in situation (i), the tour will go directly to vertex B and from there to vertex C before it leaves subgraph m. Otherwise vertex B will be the last vertex to be visited of this last subgraph.

It should now be clear that a nearest neighbor tour starting at the A-vertex of input 1 visits the B-vertex of the last gate as the last vertex if and only if the circuit computes the value TRUE. Since the transformation can be performed using work space which is logarithmic in the size of the circuit, the nearest neighbor problem is $\mathcal{P}$-complete. So, the construction of a nearest neighbor traveling salesman tour will probably require superpolylogarithmic work space or superpolylogarithmic parallel time.



(a) The representation of a TRUE input



(b) The representation of a FALSE input

FIGURE 17. The representation of input k

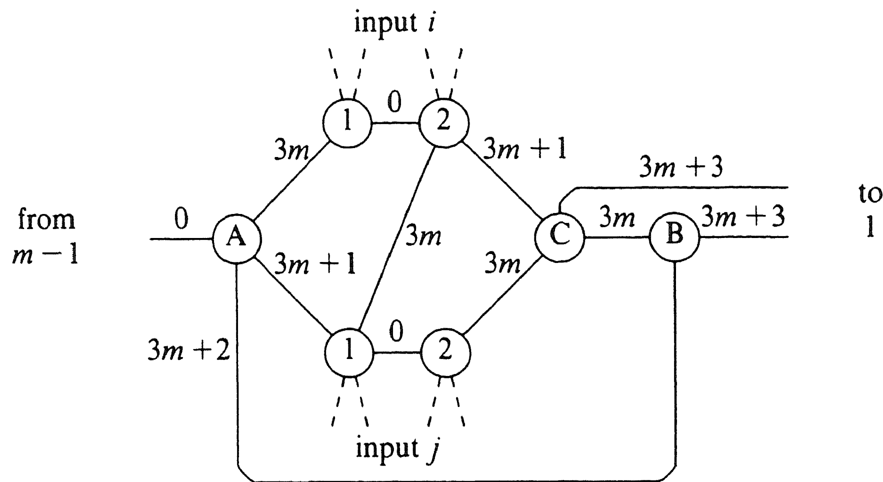


FIGURE 18. The representation of NAND gate m

6. 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.

6.1. Dynamic programming

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 by CASTI, RICHARDSON & LARSON [11] and GUIBAS, KUNG & THOMPSON [31], and will be illustrated on the knapsack problem in Example 20.

EXAMPLE 20. *Knapsack.* Given n items j, each with a profit c_j and a weight a_j ($j=1,...,n$), 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 [26].

It is convenient to introduce the notation

$$C(m,n,b) = \max_{S \subseteq \{m,...,n\}} \{\Sigma_{j \in S} c_j | \Sigma_{j \in S} a_j \leq b\}.$$

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 [4]:

> **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 $(0 \leqslant j \leqslant n)$ sequentially and, at stage j, to handle the states $(1,j,z)$ $(0 \leqslant z \leqslant b)$ in parallel [11]:

ALGORITHM KS1

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

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

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

for any $m \in \{1,\ldots,n-1\}$. It is of interest to note that this more general recursion was proposed by BELLMAN & DREYFUS [5] 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** $[1 \leqslant j \leqslant n]$ **par** $[0 \leqslant z < a_j]$ $C(j,j,z) \leftarrow 0$,
>
> $[a_j \leqslant z \leqslant b]$ $C(j,j,z) \leftarrow c_j$;
>
> **for** $l \leftarrow 1$ **to** $\log n$ **do**
>
> $k \leftarrow 2^l$,

$$\textbf{par } [0 \leqslant j < n/k] \textbf{ par } [0 \leqslant z \leqslant b] \; C(jk+1, jk+k, z)$$
$$\leftarrow \max_{0 \leqslant y \leqslant z} \{C(jk+1, jk+½k, z-y) + C(jk+½k+1, jk+k, y)\}.$$

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.

Algorithm KS1 has been implemented on two existing parallel computers. Before reporting on the results in Example 21, we describe the architectures in question.

The *ICL Distributed Array Processor* (DAP) [35] is a commercially available two-dimensional mesh connected SIMD computer with 64×64 processors. Each processor is connected to its four neighbors, with wraparound connections at the boundaries, and has its own local memory. System software makes it possible to look at the 4096 processing elements as if they were located in a one-dimensional array, each processor being connected to only two neighbors. The processors are capable of simultaneously performing the same instruction on local data, with the restriction that the data have to reside at exactly the same place of the respective local memories. Masking a processor has the effect that the result of the instruction executed is not stored; this makes conditional operations possible.

If for a particular problem the number of processors is not sufficient, the problem has to be decomposed into subproblems and the solutions to these subproblems have to be combined. This corresponds to simulating a DAP of size bigger than 64 by 64.

The performance of a program is measured by counting the number of instructions executed by the DAP. To estimate the CPU time, the number of instructions is multiplied by the average time needed for an instruction. However, differences between the frequencies of the various instructions in a particular program are neglected. There is no way to measure the CPU time of the DAP exactly.

The *Manchester dataflow machine* [33] is an experimental computer, based on the concept of dataflow. This is a technique for representing computations in terms of directed graphs. The nodes of the graph are instructions to be performed and the arcs are data routes. The data transmitted over the arcs are represented as *tokens*. A node accepts the tokens from its incoming arcs,

performs an operation on them, and sends the results away on its outgoing arcs. Whether or not two nodes can be executed concurrently depends on whether or not one of the two nodes needs the output of the other as input. Arcs not starting at a node receive the input data and arcs not ending at a node produce the output.

A node is *enabled* (can start its execution) as soon as the required tokens have arrived on the incoming arcs. The execution of a node may not be immediate, but will happen eventually. The time needed to execute instructions or to transport tokens from one node to another may vary. It is assumed, however, that all these times are finite. The computation is completely asynchronous. It can therefore happen that tokens have to wait for others on incident input arcs. A second consequence is that a dataflow graph in general allows for different execution sequences.

Figure 19 shows a possible execution sequence in a dataflow graph which calculates $x^2 - xy$ using primitive boxes DUP (which duplicates its input), $\uparrow 2$ (which produces the square of its input), $\times$ (which multiplies its inputs with each other) and $-$ (which subtracts the right input from the left input); stars ($\star$) represent the generated tokens moving through the graph.

Exploiting the parallelism contained in the dataflow model of computation requires an unconventional hardware organization. A general purpose dataflow machine needs a data structure of some sort to represent the dataflow graph of any particular problem. On the Manchester dataflow machine this data structure consists of labeled nodes containing the instruction to be performed and the destination of the results.

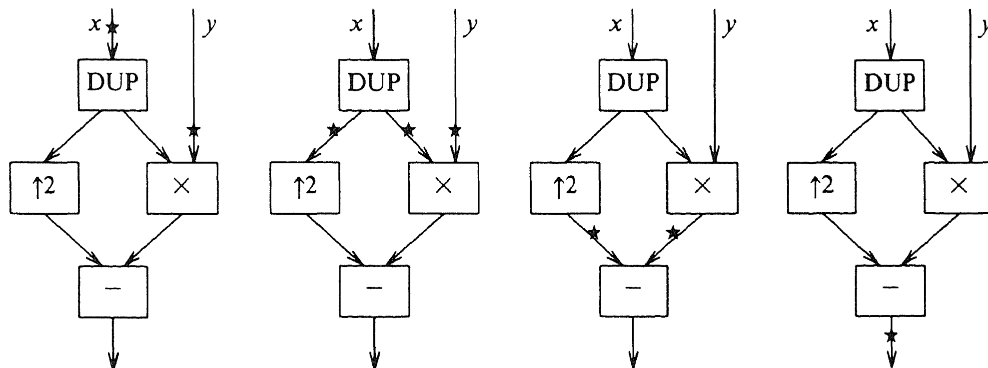


FIGURE 19. A dataflow graph with a possible execution sequence

The Manchester dataflow machine consists of a ring of elements each performing a special task (see Figure 20). A token consists of a value and a destination node. The *token queue* dispatches tokens, one at a time, to the *matching unit*. This is an associative memory, which groups tokens with the same destination node into packages and sends them to the node store. The matching

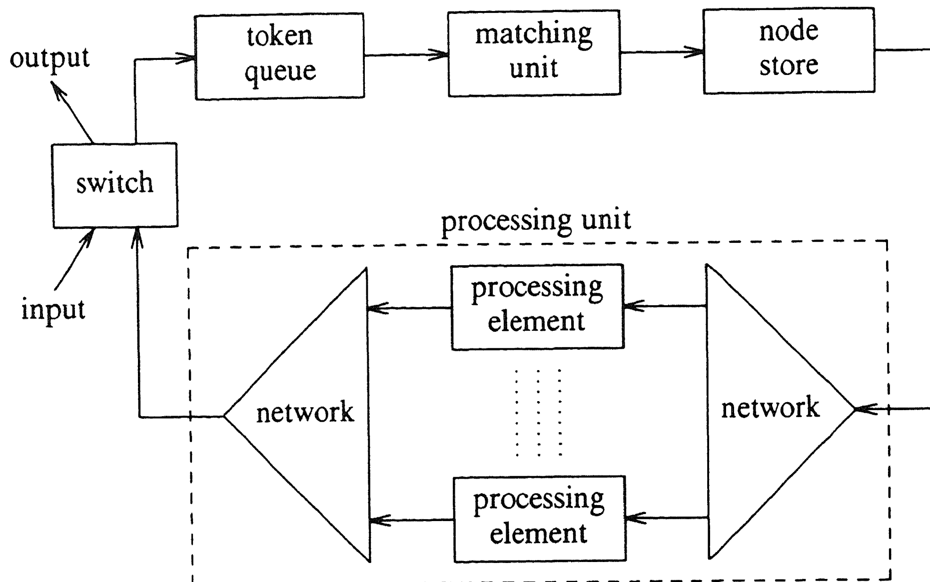


FIGURE 20. The Manchester dataflow machine

unit stores tokens until their partners have arrived. For efficiency reasons, only packages of one or two tokens are allowed. The *node store* contains the dataflow graph to be executed; each node of the graph consists of the instruction to be performed and the destination of the results. The node store adds this information to the package that arrives and sends the whole as an executable package to the processing unit. The *processing unit* sends the package via a distribution network to an idle processing element. After processing, the results arrive via an arbitration network at the switch. The *switch* inserts input tokens into the ring and removes output tokens; non-output tokens are sent along to the token queue.

The processing unit makes use of fine grained MIMD-type parallelism. The degree of parallelism depends on the number of processing elements. On a higher level, the units in the ring continuously perform operations on the flow of packages, which gives a parallelism as in an assembly line.

The critical part of the system is the matching unit. All units can be tailored to meet its maximum throughput capacity. For example, the speed of the processing unit can be adapted by adding or removing processing elements. One way to overcome this bottleneck is to construct several rings and connect them through the switch, which then becomes a full interconnection network. The Manchester dataflow machine presently consists of a single ring with twenty processing elements.

The performance of a program is measured by its CPU time. An emulator of the dataflow machine on a sequential computer can be used to obtain additional information. The emulator considers the dataflow machine as a synchronized MIMD machine with an unbounded number of processors, in which the output of a node is immediately available to successor nodes and enabled

nodes are executed without delay. The two fundamental time measurements are S_1, the number of time steps if only one processing element is available (i.e., the total number of instructions executed), and S_∞, the number of time steps with an unlimited number of processing elements (i.e., the critical path length of the underlying dataflow graph). The ratio $\pi = S_1/S_\infty$ measures the average parallelism in a program. A more detailed trace of the behavior of a program can be obtained if desired.

EXAMPLE 21. *Knapsack* [45]. For the implementation of Algorithm KS1 on the DAP, this machine is considered as a one-dimensional array of processors, numbered from 0 until 4095. As in Example 20, the values $C(1,j,z)$ are computed for $z=0,...,b$ in parallel and for $j=1,...,n$ in sequence, where processor z computes the values $C(1,1,z)$, $C(1,2,z),...,C(1,n,z)$. At stage j, a processor needs its own C-value, that of its a_j-th left-hand neighbor, and c_j. The computations and data movements can be accomplished for all processors in parallel, as long as b is no greater than 4095.

Three types of problem instances were generated. For type 1, the profits and weights were drawn uniformly from $\{1,..,64\}$. For types 2 and 3, 512 and 1024 were added to all profits and weights. For each type, three instances were created with 100, 200 and 300 items respectively; the capacity was set at 409 which is the largest problem size solvable on the DAP without decompositior

Results are more or less as expected. The estimated CPU time is linear i However, there is no distinction among the different types. Since the dista which data have to travel increases with the type number, one would expec increase in computing time as well. The only information which can retrieved from the DAP, however, is the number of instructions performed an that number appears to be the same for the three problem types. The running times are twenty times better than on the CDC/CYBER-170-750; cf. Figure 21.

n	type	DAP	CYBER-170-750
100	1	0.019	0.257
100	2	0.019	0.420
100	3	0.019	0.359
200	1	0.038	0.832
200	2	0.038	0.828
200	3	0.038	0.704
300	1	0.058	1.373
300	2	0.058	1.238
300	3	0.058	1.047

FIGURE 21. Knapsack Algorithm KS1 on the DAP and the CYBER-170-7 running times in seconds for instances with $b=4095$

On the Manchester dataflow machine, the computation is completely asynchronous. It may therefore happen that values of different stages are evaluated at the same time. However, the maximum speedup remains $O(b)$.

Since the dataflow computer is an experimental machine with limited hardware capacity, only very small problem instances could be run. The profits and weights are drawn from $\{1,...,100\}$. Instances with $n = 10, 20, 30, 40$ and $b = 100, 200, 300$ were generated.

n	$b = 100$	$b = 200$	$b = 300$
10	418	431	437
20	756	765	784
30	1091	1109	1122
40	1443	1466	1479

(a) Critical path length S_∞

n	$b = 100$	$b = 200$	$b = 300$
10	30	70	106
20	37	85	128
30	39	89	135
40	41	89	133

(b) Average parallelism π

FIGURE 22. Knapsack Algorithm KS1 on the emulator of the Manchester dataflow machine

Figure 22 shows the results of the emulator. The critical path length S_∞ is about linear in n, and the average parallelism π grows with b. With increasing b more elements fit into the knapsack, which explains the increase of S_∞ for constant n. For the problem instances considered, the hardware results are comparable: for less than ten processors, the speedup is almost linear; beyond that, hardly any gain is made (cf. Figure 23).

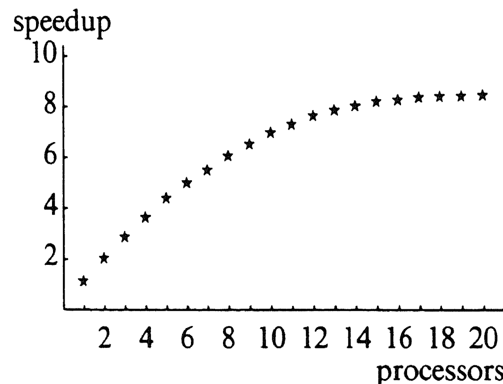


FIGURE 23. Knapsack Algorithm KSI on the Manchester dataflow machine: a typical speedup curve

6.2. Branch and bound

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 by EL-DESSOUKI & HUEN [23]. 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, several subproblems are examined at the same time. The point in time at which a subproblem becomes available depends on the number of processors, and this influences how the tree is searched. One can construct examples of anomalous behavior in which p processors together are slower than a single processor, or more than p times as fast.

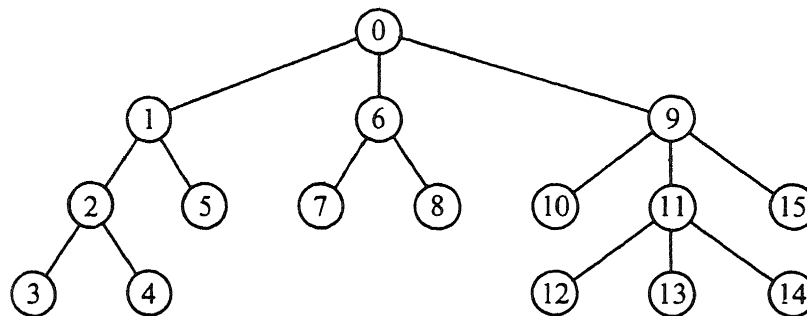
Examples 22, 23 and 24 discuss the implementation and anomalous behavior of branch and bound algorithms for the traveling salesman problem and the job shop scheduling problem. Example 25 deals with anomalies on a more theoretical basis. Example 26 reports on work in progress concerning the development of a theoretical model to analyze the distribution of work in a master-slave architecture.

EXAMPLE 22. *Traveling salesman* [62]. The traveling salesman problem was already described in Example 10. A traditional branch and bound method for its solution 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 the book by LAWLER ET AL. [52]. Figure 24(a) shows a search tree in which the nodes have been labeled in order of examination.

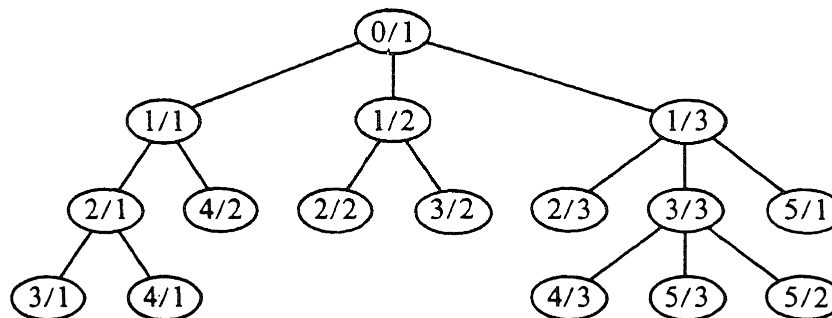
Pruul designed a parallel version of this method for an asynchronous MIMD machine. Each processor performs its own depth first search; when it encounters a node that has already been selected by another processor, it selects in the subtree rooted by that node an unexamined node at the highest level. Figure 24(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.

The *IBM Loosely Coupled Array of Processors* (LCAP) [19] consists of a master processor (IBM/4381-3) which is connected to ten slave processors (FPS/164); cf. Figure 25. On the master processor, at most ten processes run in parallel in a time sharing mode. To each of these, a slave processor can be assigned. A process can pass part of its work on to the slave processor, thereby creating true parallelism. As long as the slave is running, it cannot be influenced from outside and the invoking process on the master has to wait. Communicating with a slave processor is time consuming. Therefore, it does



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



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

FIGURE 24. Depth first tree search

not pay to send very small tasks.

For the communication between the processes on the master, one has basically to choose between two systems:

(*i*) The processors are considered as equivalent. They share part of the memory of the master processor.

(*ii*) The processes are considered as slave processes, and a master process is created. The master process is able to communicate with the slave processes; messages between slave processes have to be sent through the master process.

The limited control over the slave processors together with the restrictions on the interprocess communication makes the LCAP a rather rigid MIMD computer. In its present state, it is not well fit for algorithms in which the need for communication arises at run time.

EXAMPLE 23. *Job shop* [41]. Given are n jobs and m machines. A machine can handle at most one job at a time. A job consists of a chain of operations, each of which requires an uninterrupted given processing time on a given machine. The purpose is to find a schedule of minimum length. This $\mathcal{NP}$-hard problem [26] appears to be very difficult. Already small instances are hard to solve. The branch and bound algorithm from LAGEWEG, LENSTRA & RINNOOY KAN [47] computes lower bounds by relaxing the capacity constraints on all machines

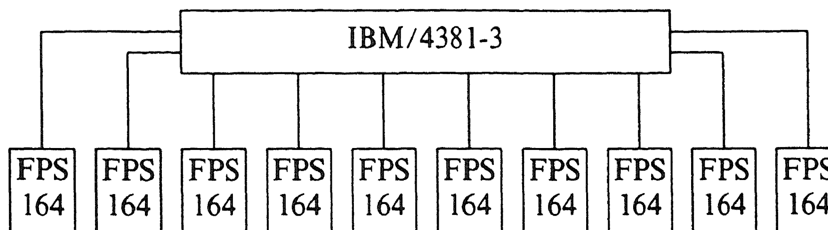


FIGURE 25. The IBM/LCAP

but one, creates subproblems by scheduling operations all of whose predecessors have been scheduled, uses depth first search, and obtains approximate solutions on a few equidistant levels of the search tree.

The implementation on the IBM/LCAP uses the second interprocess communication system. The master process generates the search tree up to a certain depth. Nodes neither branched from nor eliminated are ordered according to increasing lower bounds and put in a queue. The master process sends nodes from the front of this queue to idle slave processes. A slave performs a complete depth first search starting from the node it receives. If a better overall solution is found, it is sent to the master, which in turn informs the other slaves. If there are idle slaves and the queue of nodes of the master is empty, the master asks the busy slaves to pass on some of their work so as to refill its queue. The master process is run on the IBM machine and the slave processes pass the evaluation of the search tree on to the FPS systems. Since the software does not allow slaves to be interrupted by the master, it is necessary that they regularly report to the master. The report period has to be carefully chosen such that important news is quickly distributed and not too many unnecessary communications occur.

The algorithm shows a nondeterministic behavior. When the algorithm is run on the same instance several times, the distribution of the work over the processors varies, different search trees may be generated and different optimal solutions may be found.

The performance of the algorithm is illustrated on an instance with twenty jobs, each consisting of five operations, and five machines [59]. Reported are the maximum number of nodes branched by a slave, which indicates the parallel computing time, and the number of nodes branched by the master and slaves together, which represents the total amount of work. The master branches 65 nodes, resulting in an initial queue of 269 nodes. The slaves report to the master every 100 nodes. The results of a single run for each number of slaves are given in Figure 26. When the number of slaves increases from one to

number of slaves	maximum number of nodes branched by a slave	total number of nodes
1	11358	11423
2	2300	4609
3	1455	3320
4	900	2268
5	900	2667
6	900	3397
7	978	5143
8	700	3364
9	800	3457
10	800	3646

FIGURE 26. The job shop algorithm on the LCAP: an instance with twenty jobs and five machines

four, the maximum number of nodes branched by a slave decreases more than proportionally; this expresses a speedup anomaly. For higher numbers of processors, the maximum remains about the same. This is because the master gets into trouble. It is too slow for serving the communication requests of the slaves properly. A small number of slaves is served frequently, the others are waiting most of the time.

The *Boulder Distributed Processing Utilities Package* (DPUP) [25] has been developed to facilitate the use of a local area network of the University of Colorado at Boulder. The network consists of a small number of Pyramid and Sun work stations, which run the Berkeley Unix 4.2 operating system and are connected on an ethernet (see Figure 27). The ethernet makes it possible to send messages between processes on any two machines. The configuration can therefore be considered as an asynchronous MIMD computer.

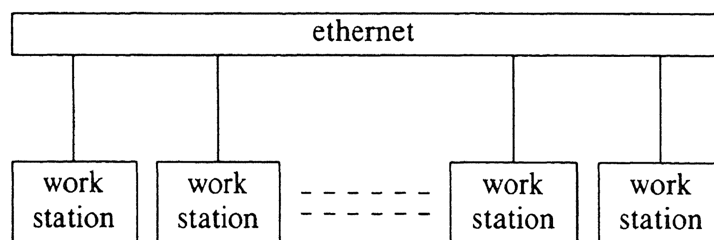


FIGURE 27. Work stations connected on an ethernet

DPUP enables a process to create remote processes on any desired machines and to establish communication links with them. In this way, a tree of processes can be created. In principle, it is possible to implement any

communication network. Communication between processes is completely asynchronous. The sending process stores the message in a buffer and may continue immediately after that. The receiving process empties the buffer as it is ready to do so. A process can be interrupted, for example to force important messages to be read at once. This software makes the system very flexible.

An ethernet allows for only one message to be sent at a time: communications are handled subsequently. In case of heavy traffic, the ethernet becomes the bottleneck of the system.

EXAMPLE 24. *Traveling salesman* [69]. The traveling salesman problem was described in Example 10 and a possible implementation of a parallel algorithm for its solution was discussed in Example 22. Trienekens considered a branch and bound algorithm with a lower bound based on 1-trees and a branching scheme of JONKER & VOLGENANT (cf. [52]).

The implementation using the Boulder DPUP is based on the master-slave principle. The master process keeps track of the nodes that are to be considered for branching. An idle slave process receives a node with the least lower bound from the master, branches this node, performs the lower bound computations, and sends the results back to the master. The advantage of this strategy over the one presented in Example 23 is that the master has full knowledge of the search tree generated so far. A disadvantage is the number of communications. Since a lot of work is involved in the lower bound computations, the time for node evaluation will dominate the time for interprocessor communication; in Example 23, the situation is the other way around.

The algorithm was run on a set of five Pyramid work stations, which have unequal processing power. Each work station executes a slave process; the most powerful work station also takes care of the master process.

The algorithm displays a nondeterministic behavior. The computational results are promising. Already for small search trees, with 30 to 60 nodes branched, a processor utilization (which is corrected for the different processor speeds) of more than 60 percent is achieved. The largest search tree, with 260 nodes branched for the solution of a Euclidean 75-city instance, gave a processor utilization of 93 percent.

EXAMPLE 25: *Anomalous behavior* [10, 48]. Assume that the evaluation of a node in a branch and bound tree takes constant time and that after the evaluation of the current set of nodes the processors collectively decide which set of nodes is to be evaluated next on the basis of a priority of each node. BURTON ET AL. [10] give examples in which two processors are more than twice as fast as a single processor, or slower than a single one. In Figures 28 and 29 both cases are illustrated. The numbers represent the priorities of the nodes; the node indicated by the box contains enough information to cause termination of the algorithm.

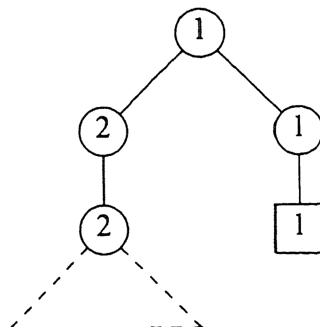


FIGURE 28. Anomalous behavior: best case for two processors

In the tree of Figure 28, a single processor first evaluates the root, creating two children. Since the right node has the lower priority of the two, the left node is evaluated first and the nodes of the large subtree follow. Only after the entire subtree is exhausted, the right node is evaluated, and one step later the optimal solution is found. A two-processor machine first evaluates the root. Then either processor takes a node, and the same happens at the next step. At that point the algorithm terminates. Hence, the two-processor system needs only three steps, while the number of nodes in the large subtree determines the running time for a single-processor computer.

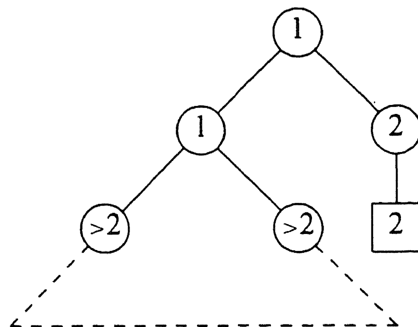


FIGURE 29. Anomolous behavior: worst case for two processors

In the tree of Figure 29, a single processor first evaluates the root, creating two children. Since the right node has the higher priority of the two, it is evaluated first. The box node is generated, and evaluated immediately, since it has a higher priority than the only other available node, the left son of the root. The algorithm terminates in three steps. A two-processor system evaluates the root at the first step, its two sons at the second step and after that the nodes of the subtree, since they have a higher priority than the box node. In this case, the algorithm runs longer with two processors than with only one.

LAI & SAHNI [48] also provide examples of anomalous behavior. This work has been extended by LAI & SPRAGUE [49, 50] and by LI & WAH [53], who

further investigate the conditions for the occurrence of anomalies in parallel branch and bound.

EXAMPLE 26. *Analysis of branch and bound algorithms on a master-slave architecture* [9]. The model for parallel branch and bound discussed in Example 24 is appealing. A master process keeps track of the set of nodes that have been generated but not yet evaluated, and a number of slave processes perform the evaluation and generation of nodes. The master orders the set of nodes according to a priority function. Each slave receives one node from the master and returns the results of its computations. If the search tree is big, the set of nodes the master has to handle will grow. At some point, the master becomes too slow to process the amount of incoming nodes. Assume that a slave receives a new node from the master as soon as it becomes idle, without waiting for the master to process its previous results. It is then possible to develop a queueing network model in which the trade-off between the speeds of master and slaves can be analyzed. It can be shown that for big search trees the number of nodes ordered by the master and awaiting release to the slaves will approach an asymptotic value, while the queue of nodes in front of the master will grow.

REFERENCES

[1] M. AJTAI, J. KOMLÓS and E. SZEMERÉDI, Sorting in $c \log n$ parallel steps, Combinatorica 3 (1983) 1-19.

[2] H. ALT, T. HAGERUP, K. MEHLHORN and F.P. PREPARATA, Deterministic simulation of idealized parallel computers on more realistic ones, eds. J. GRUSKA, B. ROVAN, J. WIEDERMANN, *Mathematical Foundations of Computer Science 1986,* Lecture Notes in Computer Science 233 (Springer, Berlin, 1986) 199-208.

[3] B. AWERBUCH, A. ISRAELI and Y. SHILOACH, Finding Euler circuits in logarithmic parallel time, Proc. 16th Annual ACM Symp. Theory of Computing (1984) 249-257.

[4] R.E. BELLMAN, *Dynamic Programming* (Princeton University Press, Princeton, NJ, 1957).

[5] R.E. BELLMAN and S.E. DREYFUS, *Applied Dynamic Programming* (Princeton University Press, Princeton, NJ, 1962).

[6] J.L. BENTLEY, A parallel algorithm for constructing minimum spanning trees, J. Algorithms 1 (1980) 51-59.

[7] J.L. BENTLEY and H.T. KUNG, A tree machine for searching problems, Proc. 1979 Internat. Conf. Parallel Processing (1979) 257-266.

[8] C. BERGE and A. GHOUILA-HOURI, *Programmes, Jeux et Réseaux de Transports* (Dunod, Paris, 1962).

[9] O.J. BOXMA and G.A.P. KINDERVATER, *A Queueing Network Model for Analyzing a Class of Branch and Bound Algorithms on a Master-Slave Architecture,* Report OS-R8717 (Centre for Mathematics and Computer Science, Amsterdam, 1987).

[10] F.W. BURTON, M.M. HUNTBACH, G.P. MCKEOWN and V.J. RAYWARD-SMITH, *Parallelism in Branch-and-Bound Algorithms,* Report CSA/3/1983

(University of East Anglia, Norwich, 1983).
[11] J. CASTI, M. RICHARDSON and R. LARSON, Dynamic programming and parallel computers, J. Optim. Theory Appl. 12 (1973) 423-438.
[12] A.K. CHANDRA, D.C. KOZEN and L.J. STOCKMEYER, Alternation, J. Assoc. Comput. Mach. 28 (1981) 114-133.
[13] L. CSANKY, Fast parallel matrix inversion algorithms, SIAM J. Comput. 5 (1976) 616-623.
[14] S.A. COOK, An observation on time-storage trade off, J. Comput. System Sci. 9 (1974) 308-316.
[15] S.A. COOK, Towards a complexity theory of synchronous parallel computation, Enseign. Math. (2) 27 (1981) 99-124.
[16] E. DEKEL, D. NASSIMI and S. SAHNI, Parallel matrix and graph algorithms, SIAM J. Comput. 10 (1981) 657-675.
[17] E. DEKEL and S. SAHNI, Binary trees and parallel scheduling algorithms, IEEE Trans. Comput. C-32 (1983) 307-315.
[18] E. DEKEL and S. SAHNI, Parallel scheduling algorithms, Oper. Res. 31 (1983) 24-49.
[19] P. DI CHIO and V. ZECCA, *IBM ECSEC Facilities: User's Guide,* Report G513-4080 (IBM European Center for Scientific and Engineering Computing, Rome, 1985).
[20] E.W. DIJKSTRA, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269-271.
[21] D. DOBKIN, R.J. LIPTON and S. REISS, Linear programming is log-space hard for P, Inform. Process. Lett. 8 (1979) 96-97.
[22] J. EDMONDS and R.M. KARP, Theoretical improvements in algorithmic efficiency for network flow problems, J. Assoc. Comput. Mach. 19 (1972) 248-264.
[23] O.I. EL-DESSOUKI and W.H. HUEN, Distributed enumeration on between computers, IEEE Trans. Comput. C-29 (1980) 818-825. Note: in the title, read 'network' for 'between'.
[24] M.J. FLYNN, Very high-speed computing systems, Proc. IEEE 54 (1966) 1901-1909.
[25] T.J. GARDNER, I.M. GERARD, C.R. MOWERS, E. NEMETH and R.B. SCHNABEL, *DPUP: a Distributed Processing Utilities Package,* Report CU-CS-337-86 (University of Colorado, Boulder, 1986).
[26] M.R. GAREY and D.S. JOHNSON, *Computers and Intractability: a Guide to the Theory of NP-Completeness* (Freeman, San Francisco, 1979).
[27] L.M. GOLDSCHLAGER, The monotone and planar circuit value problems are log space complete for P, SIGACT News 9.2 (1977) 25-29.
[28] L.M. GOLDSCHLAGER, A universal connection pattern for parallel computers, J. Assoc. Comput. Mach. 29 (1982) 1073-1086.
[29] L.M. GOLDSCHLAGER, R.A. SHAW and J. STAPLES, The maximum flow problem is log space complete for P, Theoret. Comput. Sci. 21 (1982) 105-111.
[30] T. GONZALEZ and S. SAHNI, Preemptive scheduling of uniform processor systems, J. Assoc. Comput. Mach. 25 (1978) 92-101.

[31] L.J. GUIBAS, H.T. KUNG and C.D. THOMPSON, Direct VLSI implementation of combinatorial algorithms, Caltech Conf. VLSI (1979) 509-525.

[32] U.I. GUPTA, D.T. LEE and J.Y.-T. LEUNG, An optimal solution for the channel-assignment problem, IEEE Trans. Comput. C-28 (1979) 807-810.

[33] J.R. GURD, C.C. KIRKHAM and I. WATSON, The Manchester prototype dataflow computer, Comm. ACM 28 (1985) 34-52.

[34] D. HELMBOLD and E. MAYR, *Fast Scheduling Algorithms on Parallel Computers,* Report CS-84-1025 (Stanford University, CA, 1984).

[35] R.W. HOCKNEY and C.R. JESSHOPE, *Parallel Computers: Architecture, Programming and Algorithms* (Hilger, Bristol 1981).

[36] D.B. JOHNSON, Parallel algorithms for minimum cuts and maximum flows in planar networks, J. Assoc. Comput. Mach. 34 (1987) 950-967.

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

[38] A.R. KARLIN and E. UPFAL, Parallel hashing - an efficient implementation of shared memory (preliminary version), Proc. 18th Annual ACM Symp. Theory of Computing (1986) 160-168.

[39] R.M. KARP, E. UPFAL, and A. WIGDERSON, Constructing a perfect matching is in Random NC, Combinatorica 6 (1986) 35-48.

[40] L.G. KHACHIAN, A polynomial algorithm in linear programming, Soviet Math. Dokl. 20 (1979) 191-194.

[41] G.A.P. KINDERVATER, A parallel branch and bound algorithm for the job shop problem, presentation, 8th European Conference on Operational Research, Lisbon, September 15-19, 1986.

[42] G.A.P. KINDERVATER and J.K. LENSTRA, Parallel algorithms, eds. M. O'HEIGEARTAIGH, J.K. LENSTRA and A.H.G. RINNOOY KAN, *Combinatorial Optimization: Annotated Bibliographies* (Wiley, Chichester, 1985) Ch. 8.

[43] G.A.P. KINDERVATER and J.K. LENSTRA, An introduction to parallelism in combinatorial optimization, Discrete Appl. Math. 14 (1986) 135-156.

[44] G.A.P. KINDERVATER and J.K. LENSTRA, *The Parallel Complexity of TSP Heuristics,* Report OS-R8609 (Centre for Mathematics and Computer Science, Amsterdam, 1986).

[45] G.A.P. KINDERVATER and H.W.J.M. TRIENEKENS, Experiments with parallel algorithms for combinatorial problems, European J. Oper. Res. 33 (1988) 65-81.

[46] R.E. LADNER, The circuit value problem is log space complete for *P*, SIGACT News 7.1 (1975) 18-20.

[47] B.J. LAGEWEG, J.K. LENSTRA and A.H.G. RINNOOY KAN, Job-shop scheduling by implicit enumeration, Management Sci. 24 (1977) 441-450.

[48] T.-H. LAI and S. SAHNI, Anomalies in parallel branch-and-bound algorithms, Comm. ACM 27 (1984) 594-602.

[49] T.-H. LAI and A. SPRAGUE, Performance of parallel branch-and-bound algorithms, IEEE Trans. Comput. C-34 (1985) 962-964.

[50] T.-H. LAI and A. SPRAGUE, A note on anomalies in parallel branch-and-bound algorithms with one-to-one bounding functions, Inform. Process.

Lett. 23 (1986) 119-122.
[51] E.L. LAWLER, *Combinatorial Optimization: Networks and Matroids* (Holt, Rinehart and Winston, New York, 1976).
[52] E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN and D.B. SHMOYS, eds., *The Traveling Salesman Problem: a Guided Tour of Combinatorial Optimization* (Wiley, Chichester, 1985).
[53] G.-J. LI and B.W. WAH, Coping with anomalies in parallel branch-and-bound algorithms, IEEE Trans. Comput. C-35 (1986) 568-573.
[54] L. LOVÁSZ, Determinants, matchings and random algorithms, ed. L. BUDACH, *Fundamentals of Computing Theory, FCT '79* (Akademie Verlag, Berlin, 1979) 565-574.
[55] C.U. MARTEL, *A Parallel Algorithm for Preemptive Scheduling of Uniform Machines,* Preprint (University of California, Davis, CA, 1986).
[56] R. MCNAUGHTON, Scheduling with deadlines and loss function, Management Sci. 6 (1959) 1-12.
[57] N. MEGIDDO, *Poly-log Parallel Algorithms for LP with an Application to Exploding Flying Objects* (1982) unpublished manuscript.
[58] D.E. MULLER and F.P. PREPARATA, Bounds to complexities of networks for sorting and for switching, J. Assoc. Comput. Mach. 22 (1975) 195-201.
[59] J.F. MUTH and G.L. THOMPSON, eds., *Industrial Scheduling* (Prentice Hall, Englewood Cliffs, NJ, 1963), 237.
[60] F.P. PREPARATA and J. VUILLEMIN, The cube-connected cycles: a versatile network for parallel computation, Comm. ACM 24 (1981) 300-309.
[61] R.C. PRIM, Shortest connection networks and some generalizations, Bell System Tech. J. 36 (1957) 1389-1401.
[62] E.A. PRUUL, *Parallel Processing and a Branch-and-Bound Algorithm,* M.Sc. thesis (Cornell University, Ithaca, NY, 1975).
[63] C. SAVAGE and J. JA'JA', Fast, efficient parallel algorithms for some graph problems, SIAM J. Comput. 10 (1981) 682-691.
[64] J.T. SCHWARTZ, Ultracomputers, ACM Trans. Programming Languages and Systems 2 (1980) 484-521.
[65] H.J. SIEGEL, Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks, IEEE Trans. Comput. C-26 (1977) 153-161.
[66] H.J. SIEGEL, A model of SIMD machines and a comparison of various interconnection networks, IEEE Trans. Comput. C-28 (1979) 907-917.
[67] J.S. SQUIRE and S.M. PALAIS, Programming and design considerations of a highly parallel computer, Proc. AFIPS Spring Joint Computer Conf. 23 (1963) 395-400.
[68] H.S. STONE, Parallel processing with the perfect shuffle, IEEE Trans. Comput. C-20 (1971) 153-161.
[69] H.W.J.M. TRIENEKENS, *Parallel Branch and Bound on an MIMD System,* Report 8640/A (Econometric Institute, Erasmus University, Rotterdam, 1986).
[70] S.H. UNGER, A computer oriented toward spatial problems, Proc. IRE 46 (1958) 1744-1750.

[71] E. UPFAL, A probabilistic relation between desirable and feasible models of parallel computation (preliminary version), Proc. 16th Annual ACM Symp. Theory of Computing (1984) 258-265.
[72] L.G. VALIANT, Reducibility by algebraic projections, Enseign. Math. (2) 28 (1982) 253-268.
[73] P. VAN EMDE BOAS, The second machine class: models of parallelism, eds. J. VAN LEEUWEN and J.K. LENSTRA, *Parallel Computers and Computations* (CWI Syllabus 9, Centre for Mathematics and Computer Science, Amsterdam, 1985), 133-161.