EXERCISES IN PARALLEL COMBINATORIAL COMPUTING

GERARD KINDERVATER

EXERCISES IN PARALLEL COMBINATORIAL COMPUTING

PROEFSCHRIFT
ter verkrijging van de graad van doctor
aan de Erasmus Universiteit Rotterdam
op gezag van de Rector Magnificus
Prof. dr. A. H. G. Rinnooy Kan
en volgens besluit van het College van Dekanen.
De openbare verdediging zal plaatsvinden op
donderdag 15 juni 1989 om 13.30 uur
door

GERARDUS ANTONIUS PETRUS KINDERVATER

geboren te Amsterdam.

1989
Centrum voor Wiskunde en Informatica, Amsterdam

Promotiecommissie

Promotor:	Prof. dr. J. K. Lenstra
Overige leden:	Dr. A. de Bruin
	Prof. dr. J. van Leeuwen
	Prof. dr. A. H. G. Rinnooy Kan

CONTENTS

0

Introduction

Over the last 40 years, computers have become faster by a steady series of improvements of their individual components, without fundamental changes in the concept as a whole. Operating speeds are now approaching their physical limits. In spite of all advances, however, there are still many problems which are unsolvable in reasonable time. Hence, more powerful architectures are required. A way to achieve further speedups is through the use of a collection of processors that cooperate in the solution process.

The first parallel computers were proposed in the late fifties. As in those days technological developments continued to improve the performance of traditional sequential machines enormously, the exploitation of parallelism in order to obtain faster computation times was generally regarded unnecessary. With the exception of the Illiac IV [Barnes, Brown, Kato, Kuck, Slotnick & Stokes 1968], no parallel computers were built. Around 1975, the situation changed: operating speeds of computers were so high that a much improved performance could only be expected through the introduction of parallelism. Since then, a diversity of parallel architectures has become available.

Operations research is one of the areas that are likely to benefit from advances in parallel computing. With respect to sequential computing, many operations research problems appear to be practically intractable, and for other problems a shorter solution time would be preferable. Today's parallel computers cannot solve all of these problems adequately either, but for future generations of parallel computers this may be different.

In this thesis, we discuss some aspects of the impact of parallel computing on combinatorial operations research. In the first place, it is necessary to investigate what one can and cannot expect from parallelism. The complexity theory for parallel computations provides the means to achieve this. On the other hand, there exists a formidable gap between theoretical models for

parallel computing and existing machines, and, in addition, available architectures differ very much from each other. It is therefore of interest to see what the capabilities of the current generation of parallel computers are. In particular, we would like to find out what kinds of architecture are most suitable for the field of combinatorial optimization, and what techniques can be used.

The organization of this thesis follows the structure of two of our survey papers [Kindervater & Lenstra 1986, 1988]:

Chapter 1 describes *machine models* for parallel computations. Machines can be classified according to *processor autonomy*, *interprocessor communication*, and *model of operation*. Examples of theoretical as well as realistic models are considered. The simulation of theoretical models by realistic ones is discussed.

Chapter 2 deals with the *complexity theory* for parallel computations. Given the basic distinction between *membership of* $\mathcal{P}$ and *completeness for* $\mathcal{NP}$ in sequential computing, 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. We will give examples of polylog parallel algorithms and discuss a number of $\mathcal{P}$-completeness results. In particular, we will concentrate on the construction of traveling salesman tours by some well-known heuristics [Kindervater, Lenstra & Shmoys 1989], and on the iterative improvement of such tours by local search methods [Kindervater, Lenstra & Savelsbergh 1989].

Chapter 3 discusses the *implementation* of standard algorithms for the change-making, shortest paths and knapsack problems on parallel computers that are suited for algorithms that make use of *fine-grained* parallelism; it is based on Kindervater & Trienekens [1988].

Chapter 4 analyzes the *coarse-grained* parallelization of *branch and bound* methods at the level of the parallel evaluation of nodes in the search tree. We describe experiments with branch and bound algorithms for the traveling salesman and the job shop scheduling problems. The *anomalous behavior* that these methods sometimes exhibit, is discussed in the last section of this chapter.

Chapter 5 gives a first attempt to the design and analysis of a model for the *distribution of a tree search procedure* over several parallel processors. A queueing network approach is taken to describe the various processes in a master-slave environment [Boxma & Kindervater 1987].

Chapter 6, finally, addresses issues that withstand a real breakthrough of parallel computing: the *diversity* among existing parallel architectures as well as the wide *gap* between theoretical models and available computers. It will also be necessary to develop formal techniques for the design and implementation of efficient parallel partitioning and tree search methods [Kindervater, Lenstra & Rinnooy Kan 1989].

The area of parallel computing is expanding very fast. It could have a beneficial influence on operations research. The current situation is chaotic, however, and it is not as yet clear where it will lead to. In this thesis, we discuss current developments and offer some suggestions of what would be desirable from an operations research point of view.

1

Computational Models

Many architectures for parallel computations have been proposed in the literature. Some of these machines actually exist or are being built. Unfortunately, parallel computers differ very much from each other, and the performance of algorithms is therefore highly architecture dependent. Accordingly, there exists no general theoretical model that effectively describes the broad spectrum of feasible parallel architectures. Some theoretical models are useful for the design and analysis of parallel algorithms, but their realization is usually not feasible due to physical limitations.

In the next sections, we will discuss three ways of classifying parallel architectures. The classifications are more or less orthogonal to each other and are based on *processor autonomy*, *interprocessor communication*, and *model of operation*. We end this chapter by describing a number of algorithms that illustrate the use of some specific architectures. Unless otherwise stated, a brief description of the parallel computers mentioned below can be found in Dongarra & Duff [1985].

1.1. Taxonomy of Flynn [1966]

The most widely used classification of parallel computers is due to Flynn. Flynn distinguishes four classes of machines (cf. Figure 1.1).

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

(2) SIMD (*single instruction stream, multiple data stream*). One type of instruction is performed at a time, possibly on different data. An enable/disable mask selects the processing elements that are allowed to perform the operation on their data. The ICL/DAP (Distributed Array Processor) (see Section 3.1.1), the Goodyear/MPP (Massively Parallel Processor) and the

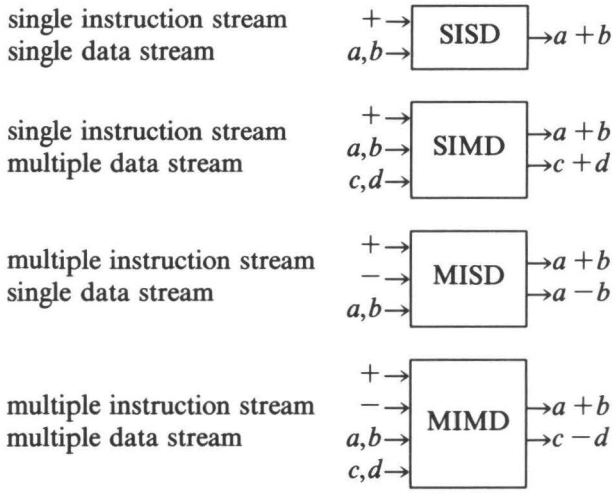


FIGURE 1.1. The classification of Flynn.

Connection Machine belong to this class. Also, *vector* computers such as the Cray-1 and the Cyber-205 (see Section 3.1.2) are often considered as SIMD machines.

In vector machines, an arithmetic operation is performed by a functional unit. The operation is split into a chain of small tasks. Each component of the functional unit performs a specific task and passes the result on to its neighbor. The computation is sped up by the *pipelining* of independent operations of the same type: as soon as a component has completed a task, it is ready to start the same task of the next operation. It turns out that developing algorithms for vector computers and SIMD machines can be done along the same lines.

(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 Alliant/FX8, the BBN/Butterfly, the IBM/LCAP (Loosely Coupled Array of Processors) (see Section 4.1.1) and the Intel/iPSC (Intel's Personal SuperComputer) are examples of MIMD machines.

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 algorithms to be presented in Section 1.4 and the dynamic programming recursions in Chapter 3. *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 Chapter 4) lends itself to this approach.

1.2. Taxonomy of Schwartz [1980]

Flynn's classification is not concerned with the way in which information is transmitted between the processors. This is dealt with by Schwartz, 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 many existing architectures, the processors have access to a common memory. As these machines only approximate a real shared memory by handling read and write instructions sequentially, they cannot be considered as paracomputers.

The most common paracomputer model is the PRAM (Parallel Random Access Machine). The PRAM is a synchronized MIMD 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.

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, seven are briefly described below. They are illustrated in Figure 1.2.

(*i*) *Complete* network. Each pair of processors is directly connected. In a p-processor system, $d_1 = p - 1$ and $d_2 = 1$. An example of this type of configuration is the MPC (Module Parallel Computer). The MPC is a theoretical model, in which each processor has its own memory and is connected to all other processors. By sending messages, a processor can access a variable stored

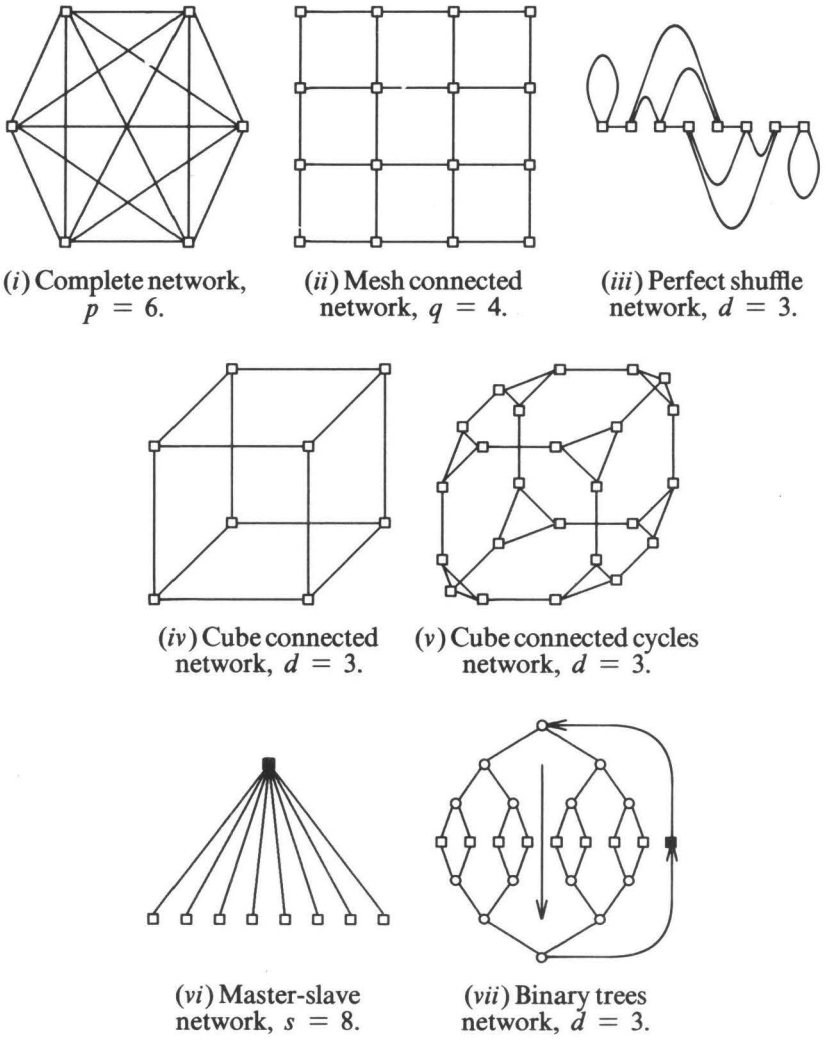


FIGURE 1.2. Seven interconnection networks.

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.

(*ii*) *Two-dimensional mesh connected* network [Unger 1958]. 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 $d_2 = 2(q-1) = \Theta(\sqrt{p})$. This interconnection network is used in the ICL/DAP and the Goodyear/MPP.

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

(*iv*) *Cube connected* network [Squire & Palais 1963]. This can be seen as a d-dimensional hypercube with 2^d processors at the vertices and interconnections along the edges. Note that $d_1 = d_2 = d = \log p$. The Intel/iPSC and the Connection Machine are organized this way.

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

(*vi*) *Master-slave* network. There are $s+1$ processors, organized as a one-level tree: one 'master' processor is connected to s 'slave' processors. Note that $d_1 = s$ and $d_2 = 2$. As an example we have the IBM/LCAP.

(*vii*) *Binary trees* network [Bentley & Kung 1979]. There are $p = 3 \cdot 2^d - 2$ processors, interconnected by two binary trees with common leaves. The 2^d processors corresponding to these leaves perform the actual computations. The other $2^d - 1$ processors in the first tree (an out-tree) send the data down to their descendants, and those in the second tree (an in-tree) combine the results from their ancestors. An additional master processor controls the network by providing the input for one root and receiving the output from the other. Note that $d_1 = 3$ and $d_2 = \Theta(\log p)$.

All these networks can simulate each other quite efficiently; see Siegel [1977, 1979] for details. Still, it appears that the cube connected cycles and perfect shuffle networks are reasonably flexible, while the mesh connected and binary trees networks have been designed for more restricted types of computations.

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. 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 $O(\log n)$ [Upfal 1984] or in deterministic time $O(\log m)$ [Alt, Hagerup, Mehlhorn & Preparata 1987]. 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 [Alt, Hagerup, Mehlhorn & Preparata 1987].

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 [1986] 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.

1.3. Control-driven, data-driven and demand-driven architectures

[Treleaven, Brownbridge & Hopkins 1982]

Parallel computers not only differ in the autonomy of the processing elements and the interprocessor communication, but also in the model of operation they use.

The main operational models are *control-driven*, *data-driven* and *demand-driven*. In control-driven architectures, the user specifies through his program the exact order in which the computations must be performed and also which operations are to be performed in parallel. In the data-driven model, an operation can be performed as soon as all its operands are available, and in the demand-driven model, an operation can be initiated as soon as its outcome is needed. In these last two models, the order in which operations are performed is completely determined by the program itself at run time. If we look at parallel processing as a multiple processor scheduling problem with precedence constraints where the statements of a program are the jobs and the dependencies of the statements are the precedence constraints, a data-driven computation corresponds in it most ideal form to the ordering of the statements according to the earliest-time scheduling algorithm, whereas a demand-driven architecture considers the statements according to the latest-time scheduling algorithm; cf., for example, Gondran & Minoux [1984].

All sequential computers use the control model of operation and, at present, most of the existing parallel computers - including the ones mentioned in the previous sections - also use this method. A number of data-driven computers, called dataflow machines, have been built, but these machines are still in their infancy; see, for example, Watson [1984]. We will discuss the Manchester dataflow machine in Section 3.1.3 in detail. Demand-driven architectures do not yet exist. Several proposals have been made, such as the ALICE machine at the Imperial College in London [Darlington & Reeve 1981].

1.4. Algorithms

As an illustration of the concepts defined above, we will end this chapter by giving some examples of parallel algorithms for elementary problems. They all use the control-driven model of operation and are developed for a specific type of interconnection network.

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.

1.4.1. 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 1.3. 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.

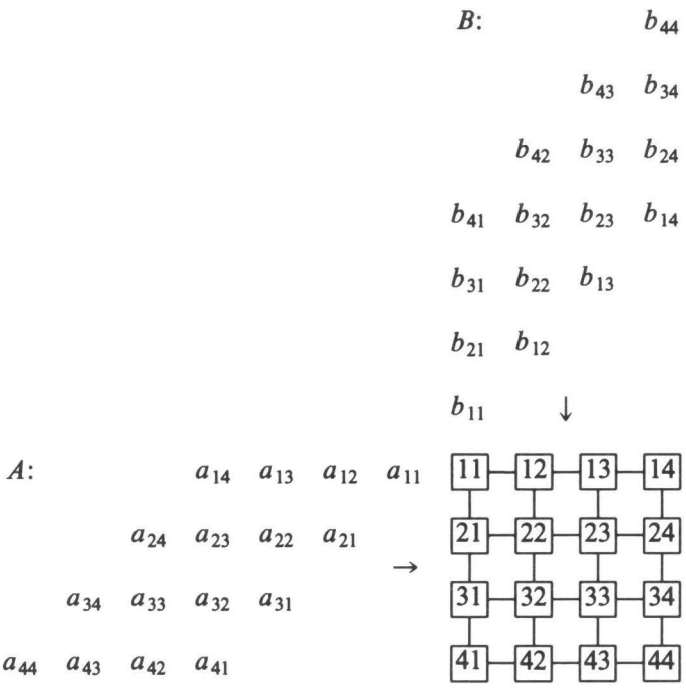


FIGURE 1.3. Matrix multiplication on a mesh connected network.

1.4.2. Transitive closure [Guibas, Kung & Thompson 1979]

The transitive closure of a directed graph G has an arc (i,j) if and only if G has a path from i to j. If G has n vertices, the algorithm from Section 1.4.1 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.

1.4.3. 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.

1.4.4. Minimum spanning tree [Bentley 1980]

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 [1957] and Dijkstra [1959]. The algorithm is based on the following principle. Let $T(V)$ be the collexion of edges in a minimum spanning tree of the subgraph of G induced by the subset V of vertices. If $i^* \notin V$ and $j^* \in V$ are such that $c_{i^*j^*} = \min_{i \notin V, j \in V}\{c_{ij}\}$, then $T(V \cup \{i^*\}) = T(V) \cup \{\{i^*,j^*\}\}$.

The algorithm starts with $T(\{1\}) = \varnothing$. 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 $c_{i.}$ 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 $T = \varnothing$ 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 sequentially. This modified algorithm still runs in $O(n \log n)$ time, but now using $O(n/\log n)$ processors with a processor utilization of $\Theta(1)$.

2

Parallel Complexity

The complexity theory for parallel computations explores the potential power and the inherent limitations of parallel computers. Section 2.1 presents an informal introduction to those concepts from the complexity theory for parallel computations that may have some impact on the field of combinatorial optimization. It turns out that parallelism introduces a distinction within the class $\mathcal{P}$: many problems in $\mathcal{P}$ are solvable in *polylog parallel time*, and others can be shown to be $\mathcal{P}$*-complete* under log space transformations. Examples of polylog parallel algorithms are given in Section 2.2, and a number of $\mathcal{P}$-completeness results are discussed in Section 2.3.

2.1. The parallel computation thesis

Complexity theory deals with the classification of problems based on the *running time* and the *work space* required by algorithms for their solution. When considering parallel algorithms, we can also take the *number of processors* into account. Although the complexity theory has been developed for *decision* problems (i.e., problems that produce a 'yes' or 'no' answer), this is not a severe restriction, since most other problems can be reformulated in terms of a limited series of decision problems. An optimization problem, for example, can be solved by posing questions on the existence of a feasible solution with at most or at least a specified value.

In this section, we discuss the complexity theory for parallel computations as far as it is of importance to the theory of combinatorial optimization. We do not intend to go into much detail, and refer to Cook [1981] for a more thorough exposition. First, we review the complexity theory with respect to sequential computations (cf. Garey & Johnson [1979]).

Sequential computers are reasonably represented by models of computation such as the Turing machine and the random access machine (RAM). Given

these models, we can define several complexity classes. The class $\mathcal{P}$ contains the problems that are solvable in *polynomial time*, i.e., the running time is bounded by a polynomial in the problem size. The problems in $\mathcal{P}$ are often called *well solved* or *easy*. $\mathcal{P}$SPACE contains the problems that are solvable using *polynomial space*, i.e., the work space is bounded by a polynomial in the problem size. A very well studied class included in $\mathcal{P}$SPACE is $\mathcal{NP}$, the class of problems for which a feasible solution can be recognized as such in polynomial time. It is obvious that $\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{P}$SPACE, and it is conjectured that both these inclusions are proper.

Another class contained in $\mathcal{P}$SPACE, which has not attracted much attention with respect to sequential computations, is POLYLOGSPACE. It consists of the problems that are solvable in *polylog space*, i.e., work space that is polynomially bounded in the logarithm of the problem size. Many problems in $\mathcal{P}$ belong to POLYLOGSPACE, but it is generally believed that $\mathcal{P} \not\subseteq$ POLYLOGSPACE. We do know, however, that POLYLOGSPACE $\neq \mathcal{P}$SPACE.

The classes $\mathcal{P}$SPACE and $\mathcal{NP}$ have their *complete* members. The $\mathcal{P}$SPACE-complete problems are generalizations of all other problems in $\mathcal{P}$SPACE in terms of transformations that require polynomial time. More precisely: a problem is $\mathcal{P}$SPACE-*complete under polynomial-time transformations* if it belongs to $\mathcal{P}$SPACE and if any other problem in $\mathcal{P}$SPACE is reducible to it by a transformation that requires polynomial time. It follows that, if any $\mathcal{P}$SPACE-complete can be shown to belong to $\mathcal{P}$, then $\mathcal{P}$SPACE = $\mathcal{P}$. Since this equality is not believed to be true, a polynomial-time algorithm for a $\mathcal{P}$SPACE-complete problem is very unlikely to exist. For the class $\mathcal{NP}$ and its complete members, the same properties hold.

$\mathcal{P}$ also has its complete problems. The $\mathcal{P}$-complete problems generalize all other problems in $\mathcal{P}$ in terms of transformations that require logarithmic work space. Formally: a problem is *log space complete for* $\mathcal{P}$ or, better, $\mathcal{P}$*-complete under log-space transformations*, if it belongs to $\mathcal{P}$ and if any other problem in $\mathcal{P}$ is reducible to it by a transformation using logarithmic work space. If any $\mathcal{P}$-complete problem would belong to POLYLOGSPACE, then $\mathcal{P} \subseteq$ POLYLOGSPACE. As this inclusion is believed to be false, an algorithm for a $\mathcal{P}$-complete problem that uses only polylogarithmic work space cannot be expected.

Sequential and parallel computations are related by a hypothesis known as the *parallel computation thesis* [Chandra, Kozen & Stockmeyer 1981; Goldschlager 1982]: *time bounded parallel machines are polynomially related to space bounded sequential machines*. That is, for any function T of the problem size n, the class of problems solvable by a machine with unbounded parallelism in *time* $T(n)^{O(1)}$ (i.e., polynomial in $T(n)$) is equal to the class of problems solvable by a sequential machine in *space* $T(n)^{O(1)}$. This thesis is a *theorem* for many 'reasonable' parallel machine models and 'well-behaved' time bounds; see Van Emde Boas [1985] for a survey. Fortune & Wyllie [1978], for example, showed that the class of problems solvable in $T(n)^{O(1)}$ time by a PRAM is equal to the class of problems solvable in $T(n)^{O(1)}$ work space by a Turing machine, if $T(n) \geqslant \log n$.

As a consequence, the class of problems solvable by a PRAM in polynomial

time is equal to $\mathcal{P}$SPACE. Since the PRAM is able to solve the apparently difficult problems in $\mathcal{P}$SPACE (such as the $\mathcal{P}$SPACE-complete and $\mathcal{NP}$-complete ones) in polynomial time, it is obviously an extremely powerful model. The theorem by Fortune & Wyllie also implies that the problems in POLYLOGSPACE are exactly the ones solvable by a PRAM in *polylog parallel time*, i.e., in time that is polynomially bounded in the logarithm of the problem size. This leads to a distinction within the class $\mathcal{P}$.

The problems in $\mathcal{P}$ belonging to POLYLOGSPACE are solvable in polylog parallel time. 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. (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, the $\mathcal{P}$-complete problems are unlikely to admit solution in polylog parallel time. If any such problem would be solvable in polylog parallel time, it would belong to POLYLOGSPACE, and it would follow that $\mathcal{P} \subseteq$ POLYLOGSPACE. Hence, their solution in polylog parallel time is not expected. 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. This does not imply, of course, that parallelism cannot yield substantial speedups.

We can, therefore, distinguish within $\mathcal{P}$ between the 'very easy' problems, which are solvable in polylog parallel time, and the 'not so easy' ones, for which such a 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, $\mathcal{NP}$-complete and $\mathcal{P}$SPACE-complete 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 [1983] for further details and more references.

2.2. Polylog parallel algorithms

The polylog parallel algorithms described below are designed to run on the PRAM model or on an SIMD machine with a shared memory. Simultaneous writes into the same memory location are prohibited. The simultaneous reads that occur are not essential and can be eliminated. We will use the notation

par $[B(i)]$ $S(i)$

to denote that the statement $S(i)$ is to be executed in parallel for all values of i satisfying the condition $B(i)$.

In some examples, we will encounter *randomized* algorithms, i.e., algorithms that produce the correct answer with probability greater than .5. From the complexity theory for randomized computations we only mention the class $\mathcal{RNC}$, i.e., the class of decision problems solvable by a randomized algorithm in polylog time on a polynomial number of processors.

We note that the (randomized) algorithms to be presented below require a polynomial number of processors, so that the related decision problems belong to $(\mathcal{R})\mathcal{NC}$.

2.2.1. *Maximum finding*

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

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

The computation is illustrated by means of a *binary tree* in Figure 2.1. 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.

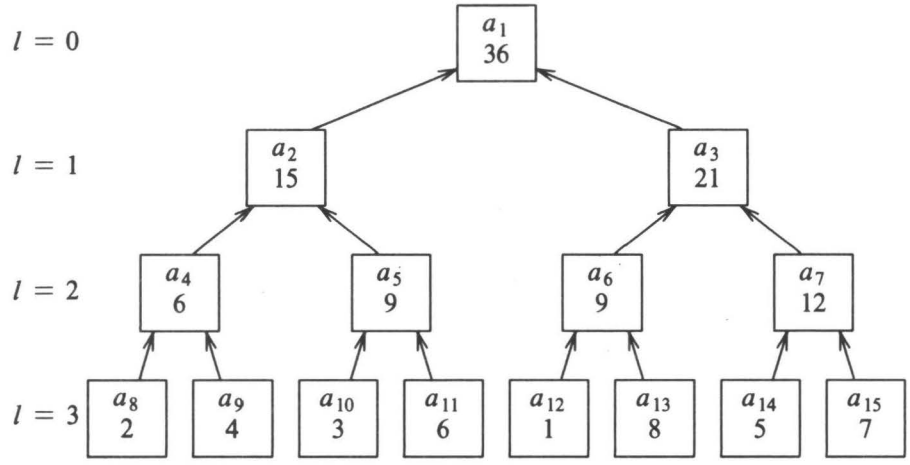


FIGURE 2.1. Maximum finding: an instance with $n = 8$.

The algorithm requires $O(\log n)$ time and $n/2$ processors. We can improve on this by applying a device similar to the one used in the last paragraph of Section 1.4.4. Suppose there are p ($p \leqslant n/2$) processors available, to which we assign n/p data. (For simplicity, we assume that p divides n.) Each processor first computes the maximum of the data assigned to it sequentially, before the above procedure is executed. The resulting algorithm has a running time of $O(n/p+\log p)$ with p processors. For $p = \lceil n/\log n \rceil$, this provides an algorithm that runs in $O(\log n)$ time, but now using only $\lceil n/\log n \rceil$ processors with a processor utilization of $\Theta(1)$.

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

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

The computation is illustrated in Figure 2.2. 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 Section 2.2.1. 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 the right child. 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 $b_{n+j} = a_n + \ldots + a_{n+j}$ for $j = 0, \ldots, n-1$.

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

Remark. In the form given above, the algorithm does not work for operations such as maximization. The partial sums algorithm uses subtraction, which has no equivalent in the case of maximization. We therefore present a version of the partial sums algorithm which is not quite so elegant as the original one, but which has the desired property since it makes use of addition only. It also runs in $O(\log n)$ time using $O(n/\log n)$ processors.

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

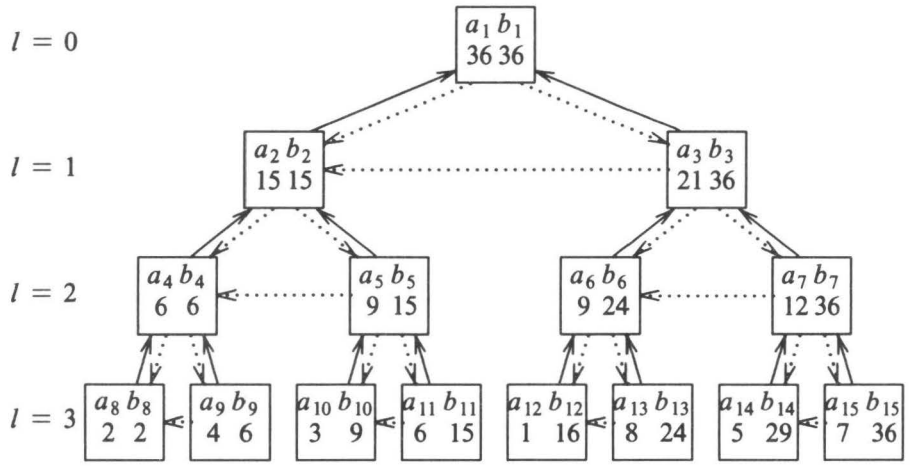


FIGURE 2.2. Partial sums: an instance with $n = 8$.

2.2.3. Sorting [Muller & Preparata 1975; Ajtai, Komlós & Szemerédi 1983] 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:

par $[1 \leqslant i,j \leqslant n]$ $\rho_{ij} \leftarrow$ **if** $a_i \leqslant a_j$ **then** 1 **else** 0;
par $[1 \leqslant j \leqslant n]$ $\pi_j \leftarrow \text{sum}\{\rho_{ij} \mid 1 \leqslant i \leqslant n\}$;
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 Section 2.2.2);

(*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 $\Theta(1)$.

A substantial improvement over the above algorithm was given by Ajtai, Komlós & Szemerédi. They developed a parallel sorting algorithm that had no sequential counterpart. It also runs in $O(\log n)$ time, but uses only $O(n)$ processors, which is best possible.

2.2.4. Shortest paths [Dekel, Nassimi & Sahni 1981]
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 [1976] 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 $[1 \leqslant i,j \leqslant n]$ $d_{ij}^{(1)} \leftarrow c_{ij}$;
for $m \leftarrow 1$ **to** $\lceil \log n \rceil$ **do**
  $l \leftarrow 2^m$,
  par $[1 \leqslant i,j \leqslant n]$ $d_{ij}^{(l)} \leftarrow \min\{d_{ik}^{(l/2)} + d_{kj}^{(l/2)} \mid 1 \leqslant k \leqslant n\}$.

Application of the routine of Section 2.2.1 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 $\Theta(1)$.

Greenberg, Ladner, Paterson and Galil [1982] showed that two $n \times n$ matrices can be multiplied on a PRAM in $O(\log n)$ time using $O(n^\alpha/\log n)$ processors, with α the exponent for matrix multiplication ($\alpha \leqslant 2.376$ [Coppersmith & Winograd 1987]). This improves the shortest paths algorithm to $O(\log^2 n)$ time and $O(n^\alpha/\log n)$ processors, with a processor utilization of $\Theta(1)$.

2.2.5. Minimum spanning tree [Savage & Ja'Ja' 1981; Chin, Lam & Chen 1982]
The Prim-Dijkstra algorithm for the minimum spanning tree problem was discussed in Section 1.4.4. 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 [Berge & Ghouila-Houri 1962]. 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 $[1 \leqslant i \leqslant n]$ $r_i \leftarrow i$;
for $l \leftarrow 1$ **to** $\lceil \log n \rceil$ **do**
  par $[1 \leqslant i \leqslant n]$
    find k such that $r_k \neq r_i$ & $c_{ik} = \min\{c_{ij} \mid r_j \neq r_i, 1 \leqslant j \leqslant n\}$,
    if k does not exist **then** a minimum spanning tree has been found & the algorithm is stopped,
    $t_i \leftarrow k$;
  par $[1 \leqslant i \leqslant n]$
    find k such that $r_k = r_i$ & $c_{kt_k} = \min\{c_{jt_j} \mid r_j = r_i, 1 \leqslant j \leqslant n\}$,
    $s_i \leftarrow k$ & $t_i \leftarrow t_k$;
  par $[1 \leqslant i \leqslant n]$ $s_i \leftarrow$ **if** $t_{t_i} = s_i$ & $r_i < r_{t_i}$ **then** 0 **else** s_i;
  par $[1 \leqslant i \leqslant n]$ **if** $r_i = i$ & $s_i \neq 0$ **then** add edge $\{s_i, t_i\}$ to the tree;
  par $[1 \leqslant i \leqslant n]$ $r_i \leftarrow$ **if** $s_i = 0$ **then** r_i **else** r_{t_i};
  for $l^* \leftarrow 1$ **to** $\lceil \log n \rceil$ **do par** $[1 \leqslant i \leqslant n]$ $r_i \leftarrow r_{r_i}$.

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 be explained later in an algorithm for scheduling fixed jobs on identical machines.

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 Section 2.2.1 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 $O(1/\log n)$.

By a careful analysis of the above algorithm, Chin, Lam & Chen proved that it can be implemented such that it runs in $O(\log^2 n)$ time on $O(n^2/\log^2 n)$ processors, with a processor utilization of $\Theta(1)$. Savage [1977] proved that the edges of a tree can be directed towards a given vertex within the same time and processor bounds.

2.2.6. Maximum cardinality matching [Karp, Upfal & Wigderson 1986; Mulmuley, Vazirani & Vazirani 1987]
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.

Lovász [1979] 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 [1976] showed that computing a determinant

belongs to $\mathcal{NC}$. Therefore, the problem of deciding whether a graph has a perfect matching belongs to $\mathcal{RNC}$.

The randomized algorithms of Karp, Upfal & Wigderson and Mulmuley, Vazirani & Vazirani which actually construct a perfect matching in polylogarithmic time, if it exists, are also based on Tutte's theorem. We refer to their papers for details. 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 $\mathcal{RNC}$; in particular, the last problem can be solved, with high probability, in $O(\log^2 n)$ time on $n^{3+\alpha} d_{\max}$ processors, where $d_{\max}$ is the maximum edge weight and α is the exponent for matrix multiplication (see also Section 2.2.4). The complexity of the maximum cardinality matching problem with respect to deterministic parallel computations is an open question, even for bipartite graphs.

2.2.7. Heuristics for the traveling salesman: double minimum spanning tree, Christofides, and nearest addition [Kindervater, Lenstra & Shmoys 1989]
In the traveling salesman problem (TSP), 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. We will consider three such algorithms in this section.

(1) *Double minimum spanning tree*

(*i*) Construct a minimum spanning tree and double its edges.

(*ii*) Construct an Eulerian cycle in the graph obtained in step (*i*) (i.e., a cycle passing through each of its edges exactly once).

(*iii*) Start at a given vertex and traverse the Eulerian cycle, skipping vertices visited before.

(2) *Christofides*

(*i*) Construct a minimum spanning tree and a minimum perfect matching on the vertices of odd degree in the tree.

(*ii*) Construct an Eulerian cycle in the graph obtained in step (*i*).

(*iii*) Start at a given vertex and traverse the Eulerian cycle, skipping vertices visited before.

(3) *Nearest addition*

(*i*) Start with a tour consisting of a given vertex and a self-loop.

(*ii*) Find vertices j and k with k belonging to the tour and j not for which d_{jk} is minimal, and insert j directly before k. Repeat this step until all vertices are inserted.

For each of these heuristics, we have a bound on the worst-case performance on TSP instances that satisfy the triangle inequality, i.e., $d_{ij} \leqslant d_{ik} + d_{kj}$ for all i,j,k. On these instances, the double minimum spanning tree and the nearest addition heuristics produce tours that are guaranteed to be less than twice as long as the optimum, and the Christofides heuristic always does better than one-and-a-half times the optimum. The crucial facts in proving these bounds

are that the minimum spanning tree is strictly shorter than the shortest tour, that the minimum perfect matching on any subset of vertices is no longer than half the shortest tour, and that no tour is longer than the Eulerian cycle from which it is obtained; see Lawler, Lenstra, Rinnooy Kan & Shmoys [1985] for details.

2.2.7.1. Double minimum spanning tree

Phase (*i*) of the double minimum spanning tree algorithm (constructing a minimum spanning tree and doubling its edges) can be performed in $O(\log^2 n)$ time with $O(n^2/\log^2 n)$ processors; see Section 2.2.5. Phase (*ii*) (finding an Eulerian cycle) can be done within the same time and processor bounds using the techniques from Awerbuch, Israeli & Shiloach [1984]. For phase (*iii*), 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.

par $[1 \leqslant i,j \leqslant 2n-1]$ $c_{ij} \leftarrow$ **if** $v_i = v_j$ **then** 1 **else** 0;
par $[1 \leqslant i \leqslant 2n-1]$ $d_i \leftarrow \max\{0, 1-\text{sum}\{c_{ij} \mid 1 \leqslant j \leqslant i-1\}\}$;
par $[1 \leqslant i \leqslant 2n-1]$ $s_i \leftarrow \text{sum}\{d_j \mid 1 \leqslant j \leqslant i\}$.

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:

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

Using the partial sums algorithm from Section 2.2.2, we can implement phase (*iii*) within the same resource bounds as the previous phases. Hence, we end up with an algorithm that runs in $O(\log^2 n)$ time on $O(n^2/\log^2 n)$ processors. Since the sequential algorithm takes $O(n^2)$ time, we have a processor utilization of $\Theta(1)$.

2.2.7.2. Christofides

The Christofides heuristic also consists of three phases. The second and third phases are identical to the corresponding phases of the double minimum spanning tree heuristic given above. Therefore, we need focus on implementing the first phase.

It is an open question if the minimum perfect matching problem belongs to $\mathcal{RC}$, but it can be solved in randomized polylog time (see Section 2.2.6). We will give a randomized *approximation scheme* for the Christofides heuristic, i.e., a family of algorithms that asymptotically approach its performance. More precisely, for each $\epsilon > 0$ we give an algorithm that runs in polylogarithmic time on a polynomial number of processors and, if the distances satisfy the triangle inequality, has probability of greater than .5 of delivering a tour of length less than $3/2+\epsilon$ times the shortest tour length; the running time is independent of ϵ and the number of processors is polynomial in $1/\epsilon$. The approach is based on the idea that an approximate minimum perfect matching will suffice to obtain an approximate Christofides tour and that an approximate minimum perfect

matching can be obtained by solving a matching problem with weights bounded by a polynomial in n. It will be useful to let $d(G) = \Sigma_{\{i,j\} \in E} d_{ij}$ for any graph $G = (V,E)$ and weight function d.

For the first phase of the heuristic we construct two Eulerian graphs and select the one of smallest total length. The first of these graphs is a double minimum spanning tree. For the second we proceed as follows.

(*i*) Find a minimum spanning tree T and identify the set V of vertices of odd degree in T.

(*ii*) Set $\mu = 2\epsilon d(T)/|V|$ and $E = \{\{i,j\} \subseteq V \mid d_{ij} \leqslant 2d(T)/3\}$. For all $\{i,j\} \in E$, set $\tilde{d}_{ij} = \lfloor d_{ij}/\mu \rfloor$.

(*iii*) Find a minimum perfect matching $\tilde{M}$ on $G = (V,E)$ with edge weights $\tilde{d}$ and add these edges to the minimum spanning tree.

We first show that this procedure has the claimed performance guarantee. To do this we show that one of the Eulerian graphs produced has total length less than $(3/2+\epsilon)d(C)$, where C is a shortest tour. Let M denote a minimum perfect matching on V with edge weights d_{ij}. If $d(T) \leqslant 3d(M)/2$, then the double minimum spanning tree has length at most $2d(T) \leqslant 3d(M) \leqslant (3/2)d(C)$.

Now assume that $d(T) > 3d(M)/2$. Note that for each $\{i,j\} \notin E$, $d_{ij} > 2d(T)/3 > d(M)$, so that $M \subseteq E$. Since $\mu\tilde{d}_{ij} \leqslant d_{ij} \leqslant \mu\tilde{d}_{ij} + \mu$ for $\{i,j\} \in E$, we have

$$\begin{aligned} d(\tilde{M}) &\leqslant \Sigma_{\{i,j\} \in \tilde{M}} (\mu\tilde{d}_{ij} + \mu) = \mu\tilde{d}(\tilde{M}) + \mu |V|/2 \\ &\leqslant \mu\tilde{d}(M) + \epsilon d(T) \leqslant d(M) + \epsilon d(T) \leqslant (1/2+\epsilon)d(C), \end{aligned}$$

and hence $d(T) + d(\tilde{M}) < (3/2+\epsilon)d(C)$.

As to the resource bounds, $O(\log^2 n)$ time and $(n^2 \log^2(1/\epsilon))/\log^2 n$ processors suffice for all of the computations except for finding the minimum perfect matching. This subroutine requires $O(\log^2 n)$ time and $n^{3+\alpha}\tilde{d}_{\max}$ processors, with $\tilde{d}_{\max}$ the maximum over all values $\tilde{d}_{ij}$ and α the exponent for matrix multiplication (cf. Section 2.2.6). By observing that

$$\max_{\{i,j\} \in V} \tilde{d}_{ij} \leqslant \left\lfloor \frac{2d(T)/3}{2\epsilon d(T)/|V|} \right\rfloor = \left\lfloor \frac{|V|}{3\epsilon} \right\rfloor = O(n/\epsilon),$$

we conclude that the number of processors required is $O(n^{4+\alpha}/\epsilon)$.

2.2.7.3. Nearest addition

Let v_1 be the given starting vertex. The order in which the nearest addition heuristic adds vertices to the tour corresponds to the way in which the algorithm from Prim and Dijkstra builds up a minimum spanning tree, starting from v_1; see Section 1.4.4. Therefore, we first construct a minimum spanning tree and direct its edges towards v_1. By means of this tree, we can determine for each two vertices i and j which one will be visited first. There are two possible situations (Figure 2.3). In the first situation, one vertex is a descendant of the other. Since each vertex is inserted immediately before its parent, the descendant will appear earlier in the tour than the ancestor. In the second situation, no vertex is a descendant of the other. Let k be the first common ancestor of i and j and let i' (j') be the last vertex on the path from i (j) to k; $i' = i$

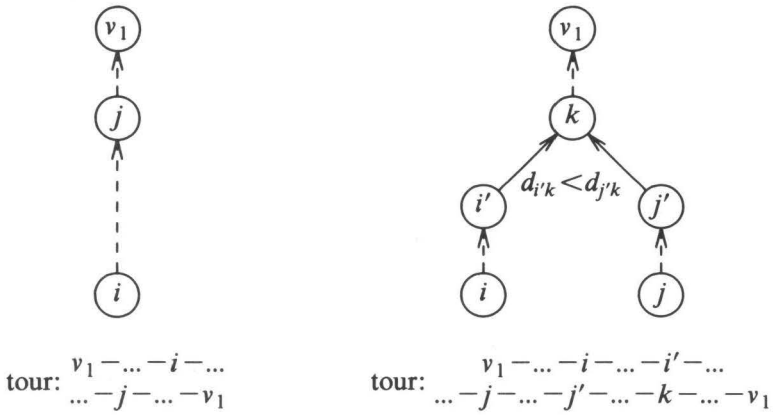


FIGURE 2.3. Nearest addition: the two possible situations for vertices i and j.

($j' = j$) if the path consists of only one arc. If $d_{i'k} < d_{j'k}$, then vertex i' will be inserted before vertex k, and after that vertex j' will be inserted in the tour immediately before vertex k and thus after vertex i'.

A detailed description of the algorithm is given below. It has a running time of $O(\log^2 n)$ on $O(n^2/\log^2 n)$ processors. Without loss of generality we assume that all distances are distinct.

(*i*) First, we construct a minimum spanning tree and direct it towards vertex v_1, generating arcs $(i,t(i))$ for $i \in \{1,\ldots,n\} \setminus v_1$. For convenience, we assume $t(v_1) = v_1$. This requires $O(\log^2 n)$ time and $O(n^2/\log^2 n)$ processors (cf. Section 2.2.5).

(*ii*) The next step is to construct an $n \times n$ 0-1 matrix (c_{ij}), representing the transitive closure of the tree ($c_{ij} = 1$ if there exist a path from vertex i to vertex j, $c_{ij} = 0$ otherwise). Let $u(i,l)$ denote the vertex at distance 2^l from vertex i, or v_1 if this vertex does not exist. The following statements do the job:

par $[1 \leqslant i \leqslant n]$ $u(i,0) \leftarrow t(i)$;
for $l \leftarrow 1$ **to** $\lceil \log n \rceil$ **do**
  par $[1 \leqslant i \leqslant n]$ $u(i,l) \leftarrow u(u(i,l-1),l-1)$;

par $[1 \leqslant i,j \leqslant n]$ **if** $i = j$ **then** $c_{ij} \leftarrow 1$ **else** $c_{ij} \leftarrow 0$
for $l \leftarrow \lceil \log n \rceil$ **downto** 0 **do**
  par $[1 \leqslant i,j \leqslant n]$ **if** $c_{ij} = 1$ **then if** $c_{iu(j,l)} = 0$ **then** $c_{iu(j,l)} \leftarrow 1$.

(The '**if** $c_{iu(j,l)} = 0$' condition is added to avoid simultaneous writes into c_{iv_1}.) These operations can be performed in logarithmic time with $O(n^2)$ processors. To reduce the number of processors, we have to observe that in each iteration of the last **for** l loop we only have to look at those pairs (i,j) for which $c_{ij} = 1$. The number of these pairs doubles in each iteration. Therefore, we perform the last iterations of the **for** loop in a different way. We replace the computation of the c-matrix by the following, where the parameter x will be chosen later:

for $l \leftarrow 1$ **to** x **do**
par $[1 \leq i \leq n,\ (l-1)\lceil n/x \rceil + 1 \leq j \leq \min\{l\lceil n/x \rceil, n\}]\ c_{ij} \leftarrow 0$;

par $[1 \leq i \leq n]\ c_{ii} \leftarrow 1$ & assign a processor to (i,i);

for $l \leftarrow \lceil \log n \rceil$ **downto** $\lfloor \log x \rfloor$ **do**
par [$1 \leq i,j \leq n$ & (i,j) has a processor assigned to it]
if $c_{iu(j,l)} = 0$ **then** $c_{iu(j,l)} \leftarrow 1$ & assign a free processor to $(i,u(j,l))$;

for $l \leftarrow x$ **downto** 0 **do**
par [$1 \leq i,j \leq n$ & (i,j) has a processor assigned to it]
if $c_{iu(j,l)} = 0$ **then** $c_{iu(j,0)} \leftarrow 1$ & assign the processor, assigned to (i,j), to $(i,u(j,0))$.

By choosing $x = \lceil \log^2 n \rceil$, we achieve a running time of $O(\log^2 n)$ with only $O(n^2/\log^2 n)$ processors.

(*iii*) Now, we compute the total number s_i of vertices in the subtree rooted by i:

par $[1 \leq i \leq n]\ s_i \leftarrow \text{sum}\{c_{ji} \mid 1 \leq j \leq n\}$.

Let r_i denote the number of descendants of the parent of vertex i which will be visited after vertex i in the tour:

par $[1 \leq i \leq n]\ r_i \leftarrow \text{sum}\{s_j \mid t(i) = t(j), d_{it(i)} < d_{jt(j)}, 1 \leq j \leq n\}$; $r_{v_1} \leftarrow 0$.

Finally, we compute for each vertex i the total number q_i of vertices visited after i:

par $[1 \leq i \leq n]\ q_i \leftarrow \text{sum}\{c_{ij}(1+r_j) \mid 1 \leq j \leq n\}$.

(if $c_{ij} = 1$, then 1 for j and r_j for the descendants of the parent of j), and a nearest addition traveling salesman tour has been determined. These last steps require the same time and processor bounds as the previous ones.

2.2.8. Local optimality of time-constrained traveling salesman tours [Kindervater, Lenstra & Savelsbergh 1989]
Given a traveling salesman tour (cf. Section 2.2.7), one wishes to decide whether it is k-optimal, i.e., whether it is impossible to obtain a shorter tour by the replacement of a set of k edges by another set of k edges. In the following, let the vertex set be {1,...,n}, let d_{ij} be the length of edge $\{i,j\}$ $(i,j = 1,...,n)$, and let $(v_1,...,v_n,v_{n+1})$ denote a TSP tour, where $v_{n+1} = v_1$. We consider the case $k = 2$ in detail; for $k > 2$, the same approach can be followed.

A 2-exchange replaces the edges $\{v_i,v_{i+1}\}$ and $\{v_j,v_{j+1}\}$ of the tour $(v_1,...,v_n,v_{n+1})$ by the edges $\{v_i,v_j\}$ and $\{v_{i+1},v_{j+1}\}$, thereby reversing the path from v_{i+1} to v_j; see Figure 2.4. Hence, a tour is 2-optimal if it can not be improved by 2-exchanges. It is an open problem whether there exists a polynomial-time algorithm that obtains 2-optimality by a sequence of 2-exchanges [Johnson, Papadimitriou & Yannakakis 1985]. We therefore restrict ourselves to deciding if a tour is 2-optimal.

Consider the following procedure. It verifies whether or not the given tour

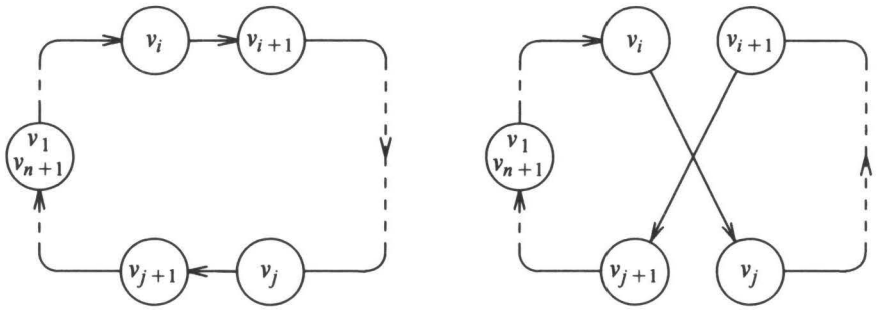


FIGURE 2.4. A 2-exchange.

$(v_1,...,v_n,v_{n+1})$ is 2-optimal. If not, a 2-exchange that produces a shorter tour is determined.

par $[1 \leqslant i < j \leqslant n]$ $\delta_{ij} \leftarrow d_{v_i v_j} + d_{v_{i+1} v_{j+1}} - d_{v_i v_{i+1}} - d_{v_j v_{j+1}}$;
$\delta_{\min} \leftarrow \min\{\delta_{ij} \mid 1 \leqslant i < j \leqslant n\}$;
if $\delta_{\min} \geqslant 0$
then the tour $(v_1,...,v_n,v_{n+1})$ is 2-optimal
else let i^* and j^* be such that $\delta_{i^*j^*} = \delta_{\min}$,
$(v_1,...,v_{i^*},v_{j^*},v_{j^*-1},...,v_{i^*+1},v_{j^*+1},...,v_{n+1})$ is a shorter tour.

By adapting the maximum finding algorithm from Section 2.2.1 such that it computes the minimum of a set of numbers and also delivers an index for which the minimum is attained, the above procedure can be implemented to require $O(\log n)$ time and $O(n^2/\log n)$ processors, which is optimal with respect to the $\Theta(n^2)$ possible 2-exchanges.

Let from now on the length of edge $\{i,j\}$ be the *travel time* between vertex i and vertex j. Assume that, as an extra condition, each vertex is given a *time window* in which it must be visited by the salesman on his tour. Arriving at a vertex before the opening of its time window introduces a waiting time at that vertex, but arriving at a vertex after the closing of its time window means infeasibility of the tour. A 2-exchange influences the arrival times at all vertices visited after the first change in the route. This may lead to infeasibility or a change in the waiting time.

The presence of time windows complexifies the problem. First, the problem of finding a feasible tour is $\mathcal{NP}$-complete. Secondly, processing a single 2-exchange requires $O(n)$ time, in contrast to $O(1)$ time in the unconstrained case. However, Savelsbergh [1988] showed that 2-optimality can still be verified in $O(n^2)$ time. We will give a parallel algorithm of verifying 2-optimality requiring $O(\log n)$ time and $O(n^2/\log n)$ processors, i.e., with the same resource requirements as in the unconstrained case.

Let $[s_i,t_i]$ denote the time window of vertex v_i $(i = 1,...,n)$, and let a feasible tour $(v_1,...,v_n,v_{n+1})$ be given. We start by considering paths along the given tour. With the use of these paths, we construct the tours that can be obtained

from the given tour by a 2-exchange. Our algorithm has five phases.

(*i*) Compute all partial sums of travel times along the tour:

par $[1 \leq i \leq j \leq n+1]$ $c_{ij} \leftarrow \text{sum}\{d_{v_k, v_{k+1}} \mid i \leq k < j\}$.

By application of the partial sums algorithm from Section 2.2.2, this phase requires $O(\log n)$ time and $O(n^2/\log n)$ processors.

(*ii*) We now investigate the effect of the time windows on the paths along the tour. For each pair of vertices v_i and v_j with v_i before v_j on the given tour, we define e_{ij} as the earliest possible departure time at v_j when traveling along the tour from v_i to v_j, and $\tilde{e}_{ij}$ as the earliest possible departure time at v_i when traveling from v_j to v_i in the reverse direction along the tour. Note that $e_{1,n+1}$ is the arrival time at v_1 of the given tour. Further, let l_{ij} denote the latest possible departure time at v_i such that the path from v_i to v_j remains feasible, and let $\tilde{l}_{ij}$ denote the latest possible departure time at v_j such that the path from v_j to v_i remains feasible.

par $[1 \leq i \leq j \leq n+1]$ $e_{ij} \leftarrow \max\{s_{v_k} + c_{kj} \mid i \leq k \leq j\}$;
par $[1 \leq i \leq j \leq n+1]$ $\tilde{e}_{ij} \leftarrow \max\{s_{v_k} + c_{ik} \mid i \leq k \leq j\}$;
par $[1 \leq i \leq j \leq n+1]$ $l_{ij} \leftarrow \min\{t_{v_k} - c_{ik} \mid i \leq k \leq j\}$;
par $[1 \leq i \leq j \leq n+1]$ $\tilde{l}_{ij} \leftarrow \min\{t_{v_k} - c_{kj} \mid i \leq k \leq j\}$.

Using the partial sums algorithm from Section 2.2.2 with addition replaced by maximization or minimization, we have the same time and processor requirements as in phase (*i*).

(*iii*) Given the earliest and latest possible departure times relative to paths along the given tour, we can compute the earliest departure time $dep_{ij}(k)$ at any vertex v_k and the earliest arrival time arr_{ij} at the origin after the substitution of the edges $\{v_i, v_{i+1}\}$ and $\{v_j, v_{j+1}\}$ by the edges $\{v_i, v_j\}$ and $\{v_{i+1}, v_{j+1}\}$.

par $[1 \leq i < j \leq n]$ $dep_{ij}(j) \leftarrow \max\{e_{1i} + d_{v_i v_j}, s_{v_j}\}$;
par $[1 \leq i < j \leq n]$ $dep_{ij}(i+1) \leftarrow \max\{dep_{ij}(j) + c_{i+1,j}, \tilde{e}_{i+1,j}\}$;
par $[1 \leq i < j \leq n]$ $dep_{ij}(j+1) \leftarrow \max\{dep_{ij}(i+1) + d_{v_{i+1}, v_{j+1}}, s_{v_{j+1}}\}$;
par $[1 \leq i < j \leq n]$ $arr_{ij} \leftarrow \max\{dep_{ij}(j+1) + c_{j+1,n+1}, e_{j+1,n+1}\}$.

For this phase, we need $O(1)$ time using $O(n^2)$ processors, or $O(\log n)$ time using $O(n^2/\log n)$ processors.

(*iv*) Under the assumption that the given tour $(v_1, \ldots, v_n, v_{n+1})$ is feasible, we can check feasibility of the tours obtained by 2-exchanges by:

par $[1 \leq i < j \leq n]$ $feas_{ij} \leftarrow (dep_{ij}(j) \leq \tilde{l}_{i+1,j}) \,\&\, (dep_{ij}(j+1) \leq l_{i+1,n+1})$.

The first condition checks feasibility at the vertices $v_{i+1}, \ldots, v_j$ and the second one at the vertices $v_{j+1}, \ldots, v_{n+1}$. As in the previous phase, we need $O(1)$ time using $O(n^2)$ processors, or $O(\log n)$ time using $O(n^2/\log n)$ processors.

(*v*) Finally, we can determine 2-optimality of the given tour in the same way as in the case without time windows.

$arr_{\min} \leftarrow \min\{arr_{ij} \mid feas_{ij},\ 1 \leqslant i < j \leqslant n\}$;
if $e_{1,n+1} \leqslant arr_{\min}$
then the tour $(v_1,...,v_n,v_{n+1})$ is 2-optimal
else let i^* and j^* be such that $feas_{i^*j^*}$ & $arr_{i^*j^*} = arr_{\min}$,
$(v_1,...,v_{i^*},v_{j^*},v_{j^*-1},...,v_{i^*+1},v_{j^*+1},...,v_{n+1})$ is a better feasible tour.

For this last part of the algorithm, the same time and processor bounds as before suffice. So, we end up with an algorithm that runs in $O(\log n)$ time using $O(n^2/\log n)$ processors, which is the same as in the case without time windows.

For each fixed $k>2$, we can derive a logarithmic-time algorithm along the the same lines. One has to take into account that, given k edges, several k-exchanges are possible. Further, the influence of a k-exchange on a tour is more complex. However, it is not hard to see that the running time remains $O(\log n)$ using $O(n^k/\log n)$ processors, which is optimal with respect to the number $\Theta(n^k)$ of k-exchanges.

2.2.9. Preemptive scheduling of identical machines [Dekel & Sahni 1983b]
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 [1959]. The algorithm first computes a value t^* which is an obvious lower bound on the minimum schedule length. It then constructs a schedule of length t^* by considering the jobs in an arbitrary order and scheduling them in the m periods $(0,t^*)$, carrying over the part of a job that does not fit at the end of the period on M_i to the beginning of the period on M_{i+1}. More formally:

$t^* \leftarrow \max\{\max\{p_j \mid 1 \leqslant j \leqslant n\}, \text{sum}\{p_j \mid 1 \leqslant j \leqslant n\}/m\}$;
$s \leftarrow 0$; $i \leftarrow 1$;
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 2.5 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

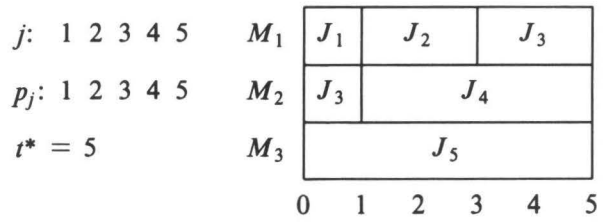


FIGURE 2.5. Preemptive scheduling: an instance with $m = 3$ and $n = 5$.

the maximum and the partial sums from Sections 2.2.1 and 2.2.2 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 $\Theta(1)$.

2.2.10. Preemptive scheduling of uniform machines [Martel 1988]
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 [1978]. As in the previous section, 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:

$t^* = \max\{(p_1/s_1), (p_1+p_2)/(s_1+s_2), ..., (p_1+...+p_{m-1})/(s_1+...+s_{m-1}),$
  $(p_1+...+p_n)/(s_1+...+s_m)\}$;
construct a composite machine with speed s_i in the interval $[(i-1)t^*, it^*)$ $(i = 1,...,m)$ 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 Section 2.2.9. 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 to run in $O(\log n + \log^3 m)$ time on $O(n)$ processors.

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

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

For a polylog parallel implementation, we need a more detailed sequential description of the algorithm [Gupta, Lee & Leung 1979]. We introduce an array u of length $2n$ containing all starting and completion times in nondecreasing order; the informal notation '$u_k \sim s_j$' ('$u_k \sim t_j$') will serve to indicate

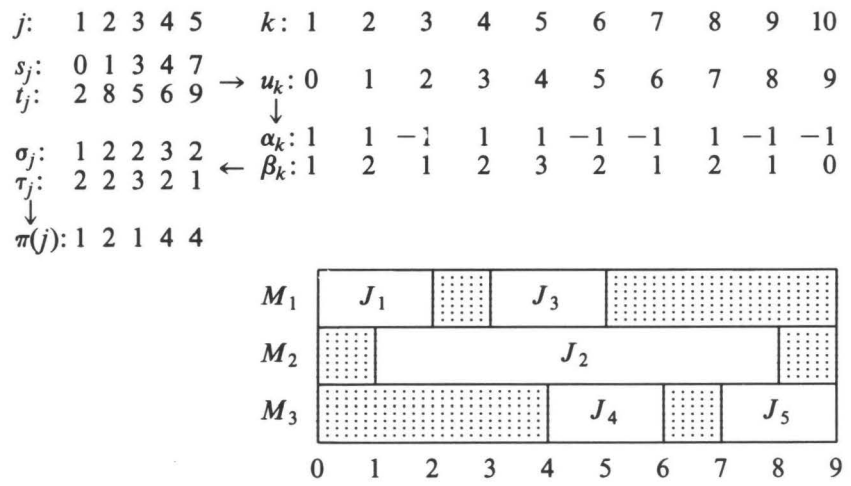


FIGURE 2.6 Scheduling fixed jobs: an instance with $n = 5$.

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, then 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 2.6 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, then 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 $[1 \leqslant k \leqslant 2n]$ $\beta_k \leftarrow \text{sum}\{\alpha_l \mid 1 \leqslant l \leqslant k\}$;
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:

sort $((\tau_1,t_1),...,(\tau_n,t_n))$ in $((V_1,v_1),...,(V_n,v_n))$ whereby
$(\tau_i,t_i)<(\tau_j,t_j))$ if and only if $\tau_i<\tau_j$ or $\tau_i=\tau_j$ & $t_i<t_j$;
par $[1\leqslant j\leqslant n]$
find k such that $\tau_k = \sigma_j$ & $t_k = \max\{t_l \mid t_l\leqslant s_j, \tau_l=\sigma_j\}$ by a
binary search of $(V_1,v_1),...,(V_n,v_n)$,
$\pi(j)\leftarrow$ **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 2.7):

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

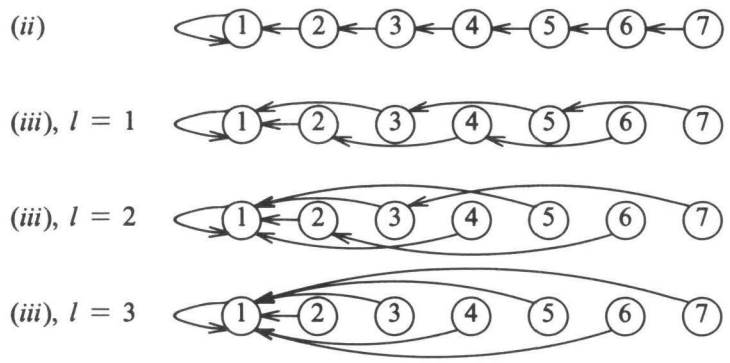


FIGURE 2.7. Scheduling fixed jobs: finding the first preceding job on the same machine.

(*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 partial sums and sorting routines from Sections 2.2.2 and 2.2.3, we can implement this algorithm to require $O(\log n)$ time and $O(n)$ processors, which improves the result of Dekel & Sahni because of a more economical sorting routine and an efficient implementation of step (*ii*).

2.3. $\mathcal{P}$-COMPLETENESS

The first $\mathcal{P}$-complete problem was identified by Cook [1974]. It involves the *solvability of a path system* and was 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.

After the identification of a first $\mathcal{P}$-complete problem P, one can prove that a problem Q in $\mathcal{P}$ is $\mathcal{P}$-complete by showing that every instance of P can be mapped to an instance of Q such that 'yes' instances of P are mapped to 'yes'

instances of Q and 'no' instances of P to 'no' instances of Q, where the transformation requires logarithmic work space [Garey & Johnson 1979]. It is said that P is *log-space transformable* to Q.

2.3.1 Circuit value [Ladner 1975; Goldschlager 1977; Goldschlager, Shaw & Staples 1982]
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 2.8.

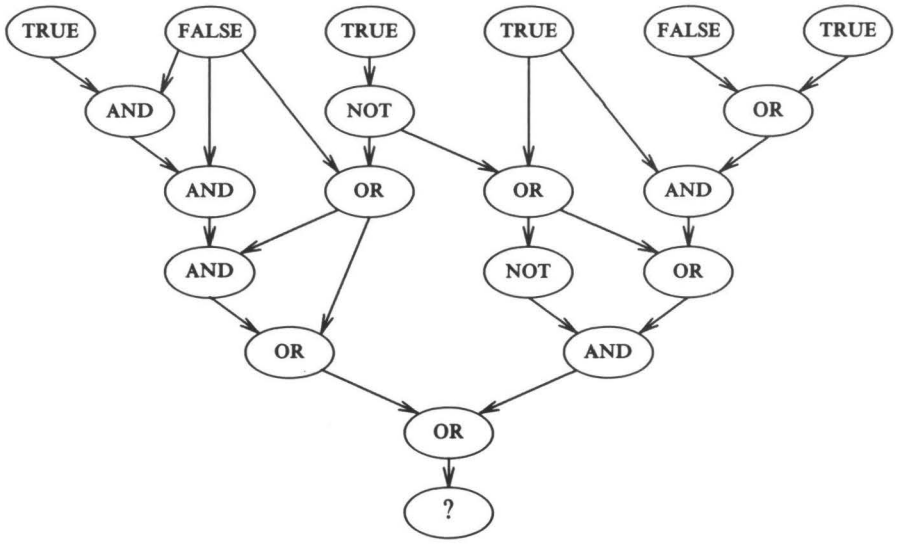


FIGURE 2.8. A logical circuit.

The circuit value problem is trivially in $\mathcal{P}$. Ladner indicated how to simulate any polynomial time deterministic Turing machine by a combinatorial circuit with only AND and NOT gates in logarithmic work space. It follows that the problem is $\mathcal{P}$-complete.

Goldschlager 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 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.

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

Linear programming is known to be in $\mathcal{P}$ [Khachian 1979]. Dobkin, Lipton & Reiss established $\mathcal{P}$-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 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,	· $x_j \leqslant x_h$, $x_j \leqslant x_i$, $x_j \geqslant 0$, $x_j \geqslant x_h + x_i - 1$,
· 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.

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

The maximum flow problem belongs to $\mathcal{P}$ [Edmonds & Karp 1972]. It was shown to be $\mathcal{P}$-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 end this section by mentioning two related results of a more positive nature.

(*i*) The maximum flow problem is solvable in polylog parallel time in the case of planar graphs, due to the relation of this case to the shortest path problem [Johnson 1987].

(*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 Section 2.2.6.

2.3.4. Heuristics for the traveling salesman: nearest neighbor, nearest merger, nearest insertion, cheapest insertion, and farthest insertion [Kindervater, Lenstra & Shmoys 1989]

In Section 2.2.7, we implemented a number of heuristics for the traveling salesman problem such that they run in polylog time. We will show in this section that for the nearest neighbor, nearest merger, nearest insertion, cheapest insertion and farthest insertion heuristics a polylog-time implementation is very unlikely to exist.

Each of the heuristics can be turned into a decision problem by posing a question about the result of the algorithm, such as 'does the tour obtained by starting the nearest neighbor heuristic in vertex v_1 visit vertex v_2 as the last one before returning to vertex v_1?' or 'does the tour obtained by the nearest merger algorithm contain edge $\{i,j\}$?'. We will prove that the nearest neighbor, nearest merger, nearest insertion, cheapest insertion and farthest insertion problems thus obtained are $\mathcal{P}$-complete by giving log-space transformations from the circuit value problem.

As in Section 2.2.7, let there be n vertices, numbered from 1 up to n, and let d_{ij} $(i,j = 1,...,n)$ denote the length of edge $\{i,j\}$ We will describe the heuristics in detail below.

(1) *Nearest neighbor*

(*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.

(2) *Nearest merger*

(*i*) Start with n partial tours, each consisting of a single city and a self-loop.

(*ii*) Merge the tours C_1 and C_2 for which $\min\{d_{ik} \mid i \in C_1, k \in C_2\}$ is as small as possible. Let $\{i,j\}$ be an edge of C_1 and $\{k,l\}$ an edge of C_2 for which $d_{ik} + d_{jl} - d_{ij} - d_{kl}$ is minimal. The merged tour is then constructed by replacing edges $\{i,j\}$ and $\{k,l\}$ by $\{i,k\}$ and $\{j,l\}$. Repeat this step until there is a complete tour.

(3) *Nearest insertion*

(*i*) Start with a tour consisting of a given vertex and a self-loop.

(*ii*) Find a vertex not on the tour which is closest to a vertex already contained in the tour.

(*iii*) Insert this vertex between two neighboring vertices on the tour in the cheapest possible way. If the tour is still incomplete go to step (*ii*).

(4) *Cheapest insertion*

(*i*) Start with a tour consisting of a given vertex and a self-loop.

(*ii*) Find a vertex not on the tour which can be inserted between two neighboring vertices on the tour in the cheapest possible way.

(*iii*) Insert this vertex between two neighboring vertices on the tour in the

cheapest possible way. If the tour is still incomplete go to step (*ii*).

(5) *Farthest insertion*

(*i*) Start with a tour consisting of a given vertex and a self-loop.

(*ii*) Find a vertex not on the tour for which the minimum distance to a vertex on the tour is maximal.

(*iii*) Insert this vertex between two neighboring vertices on the tour in the cheapest possible way. If the tour is still incomplete go to step (*ii*).

The nearest, cheapest and farthest insertion heuristics differ only in the second step from each other. They choose the next vertex to be inserted in the tour on different grounds but the actual insertion is done in the same way. With respect to the worst-case performance on TSP instances that satisfy the triangle inequality, we have that the nearest neighbor tour may be arbitrarily bad in comparison with the optimum, the nearest merger, nearest insertion and cheapest insertion heuristics produce tours that are no more than twice as long as the optimum, and the performance of the farthest insertion heuristic is unknown; cf. Lawler, Lenstra, Rinnooy Kan & Shmoys [1985].

The transformations that we present are partly defined by means of figures. Edges not shown in the figures are assumed to have a length ∞. To assure that the transformations require only logarithmic work space, we substitute $(-)cn$ for $(-)\infty$, where $c = 100$ can be seen to be sufficient.

The tours produced by the heuristics described above are the same when a constant is added to each edge length. If we add $10cn$ (with c the same as above) to each edge length in the TSP's constructed by the transformations, the resulting problems will satisfy the triangle inequality. So, if we show that the nearest neighbor, nearest merger, nearest insertion, cheapest insertion and farthest insertion problems are $\mathcal{P}$-complete, this is still true for the problems restricted to distance matrices that satisfy the triangle inequality.

2.3.4.1. Nearest neighbor

For the nearest neighbor heuristic we define the nearest neighbor problem in the following way: given a distance matrix and two vertices v_1 and v_2, does the nearest neighbor tour starting at vertex v_1 visit vertex v_2 as the last one before returning to vertex v_1? We will show that this decision problem is $\mathcal{P}$-complete. For each instance of the circuit value problem with only input and NAND gates, 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.

We number the gates such that each NAND gate receives its inputs from lower numbered gates. 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 traversed 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 2.9. 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

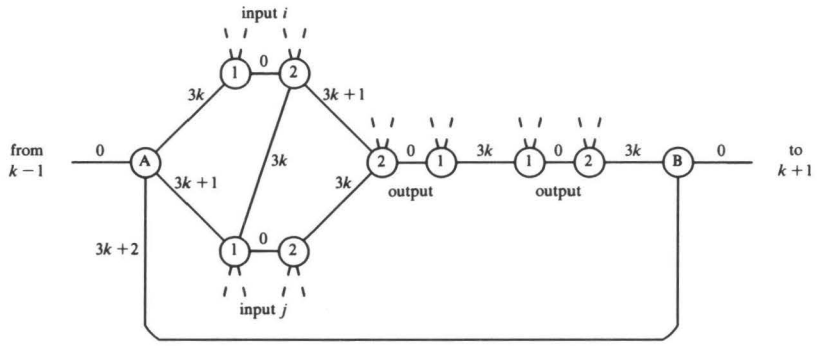


FIGURE 2.9. Nearest neighbor: the representation of NAND gate k.

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.

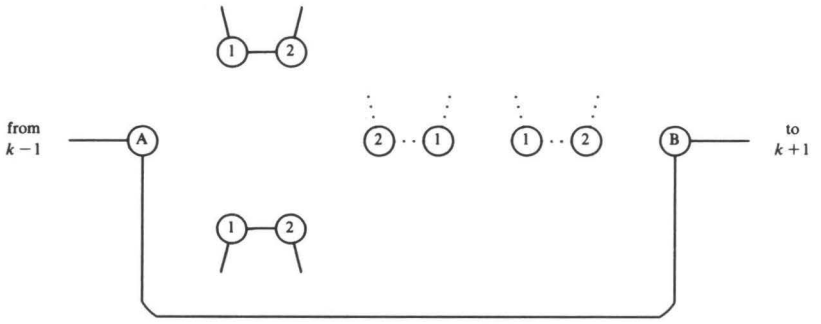


FIGURE 2.10. Nearest neighbor: TRUE NAND TRUE → FALSE

(*i*) Inputs i and j have both been visited already. In this case the tour must go directly to vertex B and then it will choose the edge of length zero to subgraph $k+1$. This will be the only case where the output vertices are not immediately visited. Note that as a result either output vertex 2 has its corresponding vertex 1 left as its only unvisited neighbor within the subgraph. See Figure 2.10.

(*ii*) Either input i or input j is still unvisited. The tour will choose vertex 1

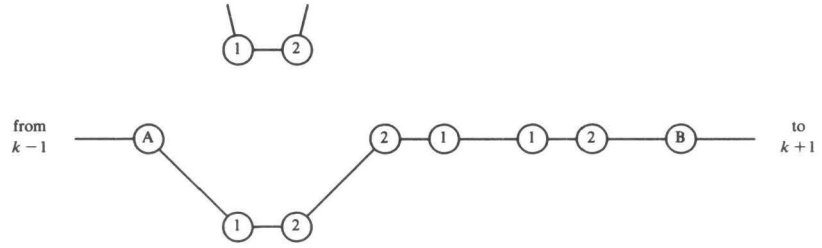


FIGURE 2.11. Nearest neighbor: TRUE NAND FALSE → TRUE

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 2.11 and 2.12.

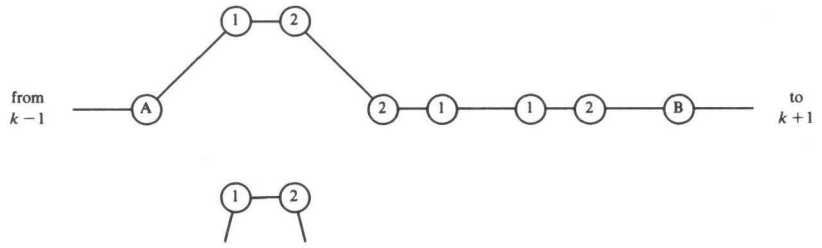


FIGURE 2.12. Nearest neighbor: FALSE NAND TRUE → TRUE

(*iii*) Both inputs are unvisited. The tour will pass through all vertices of subgraph k before going to subgraph $k+1$ (Figure 2.13).

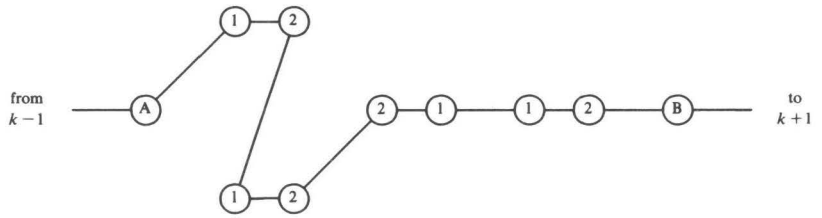
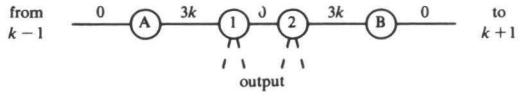


FIGURE 2.13. Nearest neighbor: FALSE NAND FALSE → TRUE

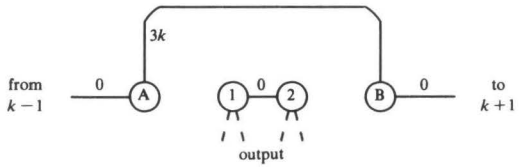
Note that in all cases all unvisited input vertices are included in the tour.

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.



(a) The representation of a TRUE input



(b) The representation of a FALSE input

FIGURE 2.14. Nearest neighbor: the representation of input k.

For TRUE and FALSE inputs we construct the subgraphs as shown in Figure 2.14. The subgraph corresponding to input 1 is a special case. Instead of the edge of length zero, it has two edges of length $3m+3$ which connect it to the subgraph corresponding to NAND gate m. The representation of this last gate 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 2.15). 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.

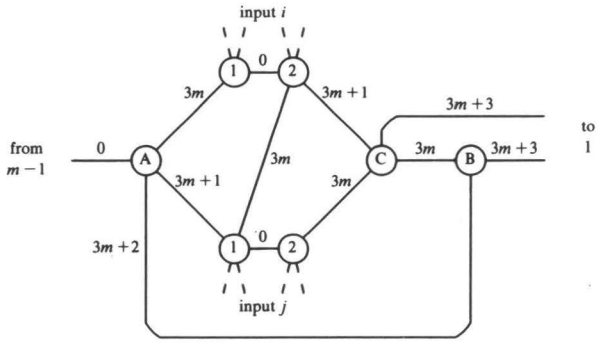


FIGURE 2.15. Nearest neighbor: the representation of NAND gate m.

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.

2.3.4.2. Nearest merger

Given a distance matrix and an edge $\{i,j\}$, the nearest merger problem is the problem of deciding whether the tour produced by the nearest merger heuristic contains $\{i,j\}$. We will show that this problem is $\mathcal{P}$-complete by giving a transformation from the circuit value problem.

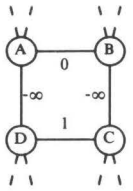


FIGURE 2.16. Nearest merger: the representation of an arc.

Consider an instance of the circuit value problem. For each arc, we construct a graph as shown in Figure 2.16. Gates with fan-out zero (for example, the last gate) are assumed to have an arc from itself to a dummy vertex. The dashed edges have a length greater than zero and will be described later. The nearest merger heuristic first builds the tours A-D-A and B-C-B, and then merges them to form the tour A-B-C-D-A.

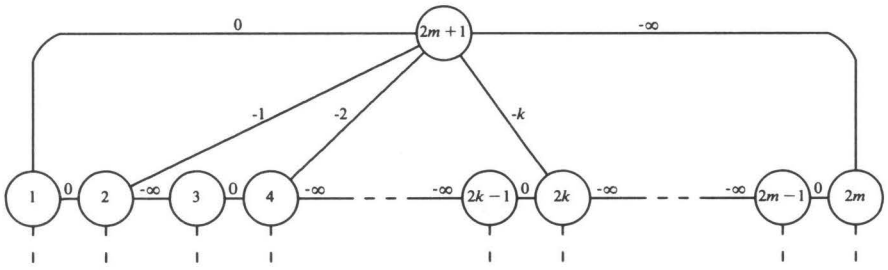


FIGURE 2.17. Nearest merger: the extra graph.

We also construct the graph of Figure 2.17, where m is the number of gates of the circuit. The algorithm starts by constructing partial tours of the form $(2k)-(2k+1)-(2k)$, for $k = 1,...,m$. The edge lengths $-(m-1),...,-1,0$ assure that the original tour $(2m)-(2m+1)-(2m)$ is merged with the other cycles of length two and finally with the self-loop $(1)-(1)$. The result is the tour $(2m+1)-(1)-(2)-...-(2k-1)-(2k)-...-(2m-1)-(2m)-(2m+1)$.

We will now describe how the graphs are connected. The edge lengths are

chosen such that the nearest merger heuristic will merge the cycles of Figure 2.16 with the (extended) tour of the extra graph of Figure 2.17 in the order of the numbers of the gates where the corresponding arcs begin. If a cycle of Figure 2.16 is added, an edge of length zero or one will remain in the tour. At this point, we associate a value TRUE (FALSE) with the arc if the edge of length one (zero) still belongs to the tour.

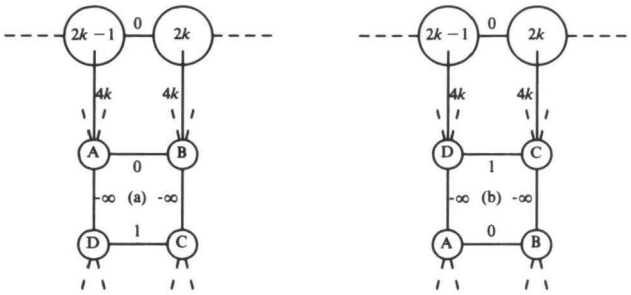


FIGURE 2.18. Nearest merger: the representation of the input gates; (a) input gate k is TRUE; (b) input gate k is FALSE.

Each cycle corresponding to an arc from an input gate is connected to the extra graph as shown in Figure 2.18. If the input is TRUE (FALSE), the edges $\{A,B\}$ and $\{2k-1,2k\}$ ($\{C,D\}$ and $\{2k-1,2k\}$) are replaced by $\{2k-1,A\}$ and $\{2k,B\}$ ($\{2k-1,D\}$ and $\{2k,C\}$).

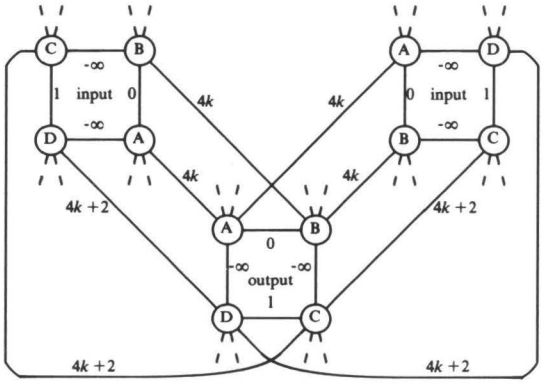
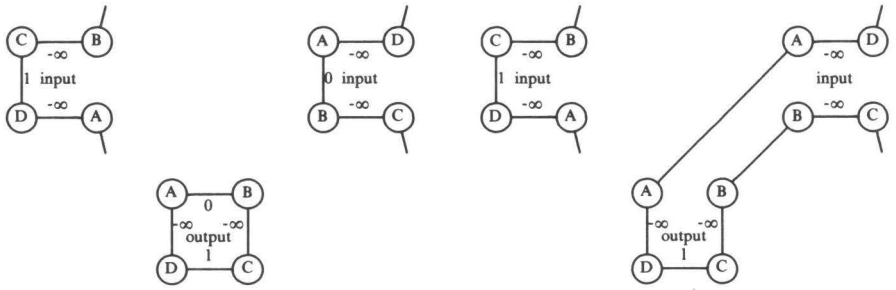


FIGURE 2.19. Nearest merger: the representation of NAND gate k (fan-out one).

For NAND gate k with fan-out one, the subgraphs are connected as shown in Figure 2.19. Let us assume that there are no edges in the tour connecting the two inputs. We consider the case where one input (left) has the associated value TRUE and the other one the value FALSE in detail (see Figure 2.20). There are two candidates for the merge operation: replace the edge $\{C,D\}$ of the left



(a) Situation before merging (b) Situation after merging

FIGURE 2.20. Nearest merger: TRUE NAND FALSE → TRUE.

input and the edge {C,D} of the output by the edges between the C and D vertices, or replace the edge {A,B} of the right input and the edge {A,B} of the output by the edges between the A and B vertices. The last replacement will be chosen, since it is cheaper.

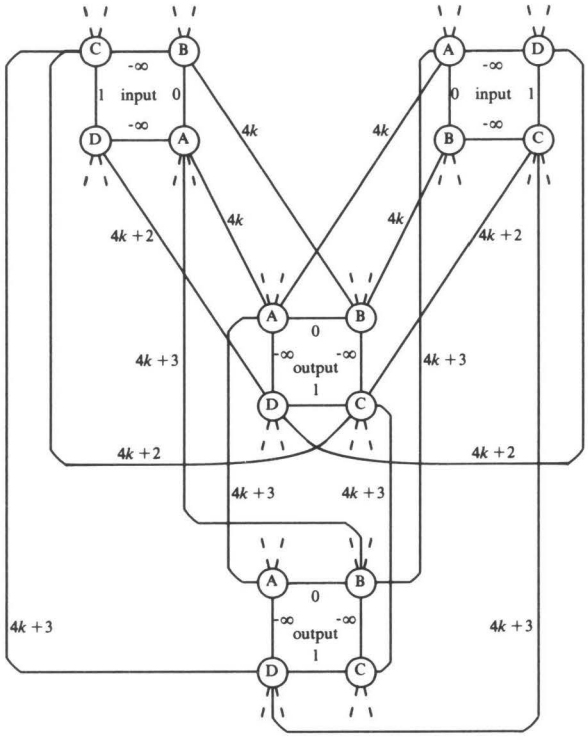
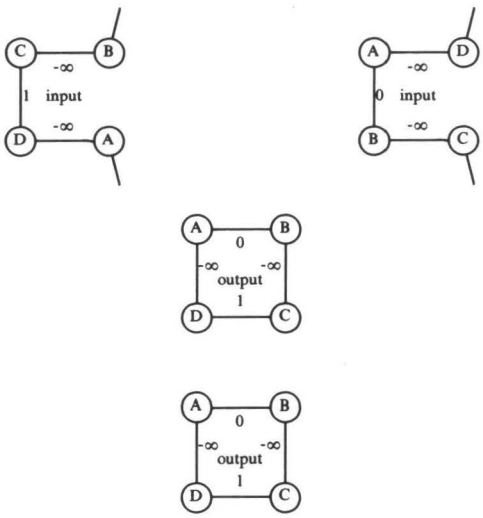
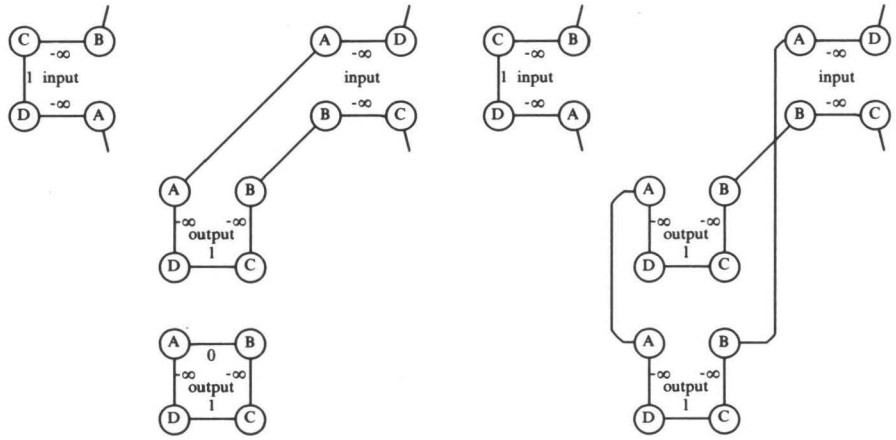


FIGURE 2.21. Nearest merger: the representation of NAND gate k (fan-out two).



(a) Situation before merging.



(b) The two merging steps.

FIGURE 2.22. Nearest merger: TRUE NAND FALSE → TRUE.

If NAND gate k has fan-out two, we connect the subgraphs as shown in Figure 2.21. The case TRUE NAND FALSE → TRUE is illustrated in Figure 2.22.

The other cases are left as exercises to the reader. Note that the output vertices of a subgraph are always inserted between two edges of length $-\infty$ of one of the inputs.

So far, we have assumed that there are no edges in the tour connecting both inputs of the same gate. Because of the way that output vertices are inserted in

the tour, connecting edges can only occur when the inputs are outputs from the same gate. These edges stretch between vertices with the same label (A or C). It is, however, impossible to remove them from the tour at low cost. Therefore, the same replacements will be made as in the case where there are no interconnecting edges between the inputs.

The above arguments imply that the circuit value of an instance of the circuit value problem is TRUE if and only if the nearest merger heuristic produces a tour which contains edge {C,D} of the subgraph corresponding to the arc starting from the last gate of the circuit. The transformation can be done in logarithmic work space. Hence, the nearest merger problem is $\mathcal{P}$-complete.

2.3.4.3. Nearest insertion, cheapest insertion, and farthest insertion

Given a distance matrix, a starting vertex and an edge $\{i,j\}$, the nearest insertion (cheapest insertion, farthest insertion) problem is the problem of deciding whether the nearest insertion (cheapest insertion, farthest insertion) heuristic produces a tour which contains edge $\{i,j\}$. The transformations from the circuit value problem showing that these problems are $\mathcal{P}$-complete are similar to the one for the nearest merger problem. We will give only the crucial part of the transformation, leaving the details and the verification to the reader.

Nearest insertion. For NAND gate k with fan-out zero or one to be simulated, we construct the graph of Figure 2.23. The output vertices are inserted in the tour in the order A, D, B and C between vertices A and B or B and C of one of the inputs. The representation if the fan-out is two is straightforward and not given here (edges between both outputs get a length -1). Representing the inputs to the circuits and starting up the algorithm is similar to the nearest merger case and also straightforward. The result of the nearest insertion algorithm is a tour which contains the edge of length one of the output arc of gate m if and only if the circuit produces the value true.

The transformation requires only logarithmic work space and hence the the nearest insertion problem is $\mathcal{P}$-complete.

Cheapest insertion. The same transformation as described for the nearest insertion problem works for the cheapest insertion problem, because in each step both algorithms will choose the same vertex to be inserted in the graphs simulating the logical circuit. So, the cheapest insertion problem is $\mathcal{P}$-complete as well.

Farthest insertion. We can use almost the same transformation as in the previous two cases. We construct the same graph as before. Let s_i be the number of the step in which vertex i would be included in the tour by the nearest insertion heuristic; this number is known before the heuristic is actually executed. We replace the starting vertex by starting vertices v_1, v_2, v_3, v_4 and v_5, such that the farthest insertion heuristic, when started in v_1, first builds the tour $v_1-v_2-v_3-v_4-v_5-v_1$. Edges originally incident to the starting vertex are made incident to v_3 and v_4. In this transformation we have to fill in a value

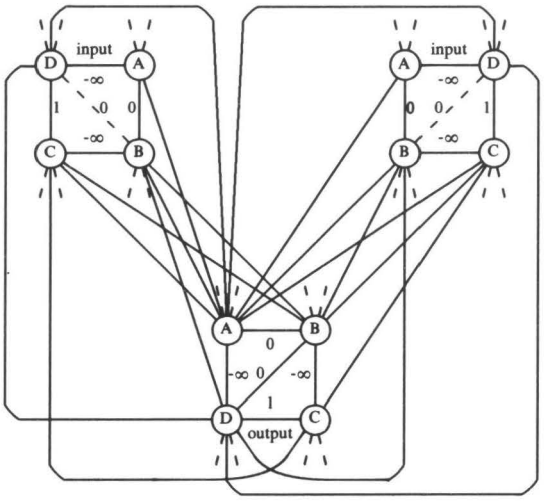


FIGURE 2.23. Nearest insertion: the representation of NAND gate k. The edges from output vertex A to the vertices A and B of both inputs have length $2k$; all other edges between input and output vertices have length $2k+1$.

for $(-)\infty$ explicitly; we take $(-)100n$. We add edges from all non-starting vertices i to v_1 of length $-100n-s_i$, and to v_2 and v_5 of length $200n$. The edges to v_1 have the smallest lengths and determine the order in which the vertices are added to the tour. The edges to v_2 and v_5 prohibit the exclusion of $\{v_1,v_2\}$ and $\{v_1,v_5\}$ from the tour. Now, the farthest insertion heuristic will add the vertices in the same order and in the same way to the tour as the nearest insertion heuristic. Herewith, the farthest insertion problem is $\mathcal{P}$-complete.

2.3.5. List scheduling [Helmbold & Mayr 1984]
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 $1 \leqslant i \leqslant m$ and $s \geqslant 0$, 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 selected to process the first available job on the list. More formally:

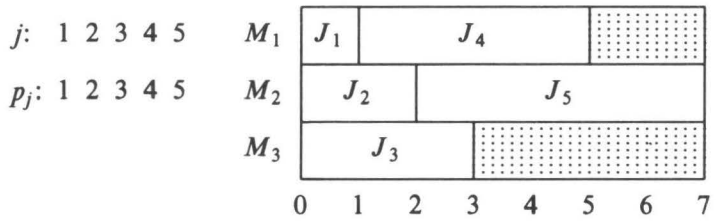


FIGURE 2.24. List scheduling: an instance with $m = 3$ and $n = 5$.

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

An example is given in Figure 2.24. 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 $m \geqslant 2$.

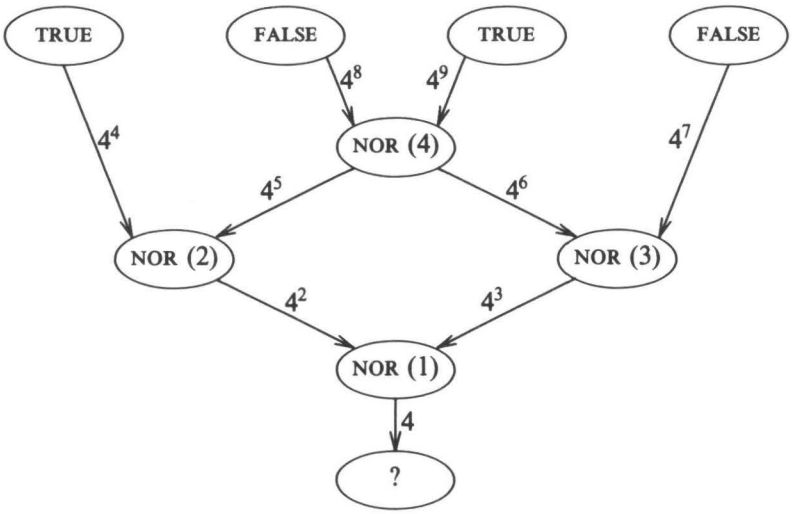


FIGURE 2.25. A circuit with numbered gates and weights assigned to the edges.

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 2.25. 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 $m \geqslant 2$.

3

Experiments with Fine-Grained Parallelism

The unbounded parallelism and the unit-time communication of the PRAM make it an attractive but unrealistic model. Neither of these properties will be encountered in practice. In existing parallel computers, the fixed number of processors bounds the parallelism and the interprocessor network sets a limit to the communication speed.

When programming parallel computers, one will notice the enormous diversity among them. The processor capabilities and the way data transfers are taken care of heavily influence the suitability of an architecture for certain types of algorithms. This situation is completely different from sequential computing, where computers can be considered equivalent and algorithms will show more or less the same performance when differences in processor speed are accounted for.

To execute an algorithm on a parallel computer, an adaptation with respect to the architecture at hand is necessary. This reformulation may increase the overall complexity of the algorithm or even obscure its essence. Sometimes, this effect is so severe that a particular parallel computer appears to be completely unsuited for executing specific types of algorithms.

Most parallel computers seem to be developed for solving problems from numerical analysis. In this area, remarkable speedups have been obtained. Algorithms for the solution of combinatorial problems are often of a different kind. It is, therefore, of interest to experiment with standard techniques, such as *dynamic programming*, *divide and conquer* and *branch and bound*.

In the dynamic programming approach, the solution to a problem is built stagewise. First, the problem is solved in a simple form, and then at each stage a new aspect of the problem is added until the solution to the original problem is found. The computations to be performed at each stage are usually elementary and always of the same type.

Divide and conquer solves a problem by splitting it into smaller ones, solving the smaller problems and combining their solutions into the solution to the original problem. The smaller problems are solved by recursively applying the same technique. In many applications, the generated tree is highly regular; it can most often be predicted which subproblems are to be solved. The work to be done for the solution of a subproblem can, however, very much depend on that subproblem.

Branch and bound methods generate search trees in which each node corresponds to a subset of the feasible solution set. A subproblem associated with a node is either solved directly, or its solution set is split and for each subset a new node is added to the tree. The process can be improved by computing a bound on the solution a node can produce. If this bound is worse than the best solution found so far, the node cannot produce a better solution and hence it can be excluded from further examination. The shape of the search tree to be generated by a branch and bound algorithm can, most of the time, not be described without actually performing the algorithm. Just as in the divide and conquer case, the computations within a node can be heavily subproblem dependent.

In this chapter and the next one, we will describe the implementation of a number of combinatorial algorithms on some of today's parallel computers. In Chapter 4, we will concentrate on architectures in which the processors are full-bodied sequential computers and the interprocessor communication is time consuming. For algorithms to be efficient, they need to have a high computation/communication ratio: we speak of *coarse-grained* parallelism. In the computers considered in the present chapter, the processors have limited capabilities and only small tasks can be assigned to them at a time. The interprocessor communication time is usually in the order of an arithmetic operation. In typical algorithms for these machines, the processors perform a few arithmetic operations and then communicate with each other. This type of parallelism is called *fine-grained.*

This chapter deals with three well-known combinatorial problems and three parallel computers. The problems are *change-making*, *shortest paths* and *knapsack*; the machines are the *ICL/DAP* (an SIMD processor array), the *CDC/CYBER-205* (a vector machine that might be classified as an SIMD machine) and the *Manchester dataflow machine* (an experimental MIMD dataflow computer). Details of the problems and the architectures are given in the next sections.

It will turn out that the SIMD machines are very efficient in executing synchronized algorithms that contain regular computations and regular data transfers. As soon as the computations or the data transfers become irregular or asynchronous, the SIMD machines become much less efficient. They are, therefore, very good for dynamic programming, but less suitable for divide and conquer and branch and bound.

The concept of dataflow appears to be very promising. The Manchester dataflow machine seems to capture all sorts of parallelism. Dynamic programming, divide and conquer and branch and bound algorithms give equally good

results. The performance of the Manchester dataflow machine is, however, limited by its experimental character. It has, amongst others, a small memory capacity and a small overall throughput. Only after an improvement of its performance, it will be clear how a dataflow computer will behave on more realistic problems and whether dataflow will fulfill its promise.

3.1. Architectures

In this section we will give a short description of the ICL/DAP, the CDC/CYBER-205 and the Manchester dataflow machine. We will emphasize those features that are relevant for the implementation of the combinatorial algorithms under consideration.

3.1.1. The ICL/DAP [Hockney & Jesshope 1981]

The ICL/DAP (Distributed Array Processor) is a commercially available two-dimensional mesh connected control-driven SIMD computer with 64×64 processors. Each processor is connected to its four neighbors, with wrap around connections at the boundaries (Figure 3.1), 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, where each processor is 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. It is effectuated by local data and makes the use of conditional operations possible.

Programs are executed on the DAP through a host computer. The host translates a program into DAP machine code and stores the machine code program and its input data in the DAP. After that, control is given to the DAP, which actually executes the program. When the DAP has finished, control is returned to the host and the host extracts the results from the DAP [ICL 1981].

The DAP can be programmed in the high-level language DAP-FORTRAN [ICL 1979]. This is an extension of standard FORTRAN with vector and matrix instructions, which can be used to process the elements of vectors and matrices in parallel. The DAP and the FORTRAN compiler do not detect any parallelism in a program on their own accord: the programmer has to detect the parallelism himself. By invoking the vector and matrix instructions of DAP-FORTRAN, the user can state explicitly which operations are to be performed in parallel. Although the DAP is capable of executing programs written in standard FORTRAN, no instructions of these programs are executed in parallel.

The vector and matrix instructions perform their parallel operations on vectors of dimension 64 or 4096 and on matrices of dimension 64 by 64 respectively. In performing operations on vectors of dimension 64, 64 processing elements cooperate in handling one vector element. If a particular problem instance is too big to fit in such a vector or matrix, the programmer has to

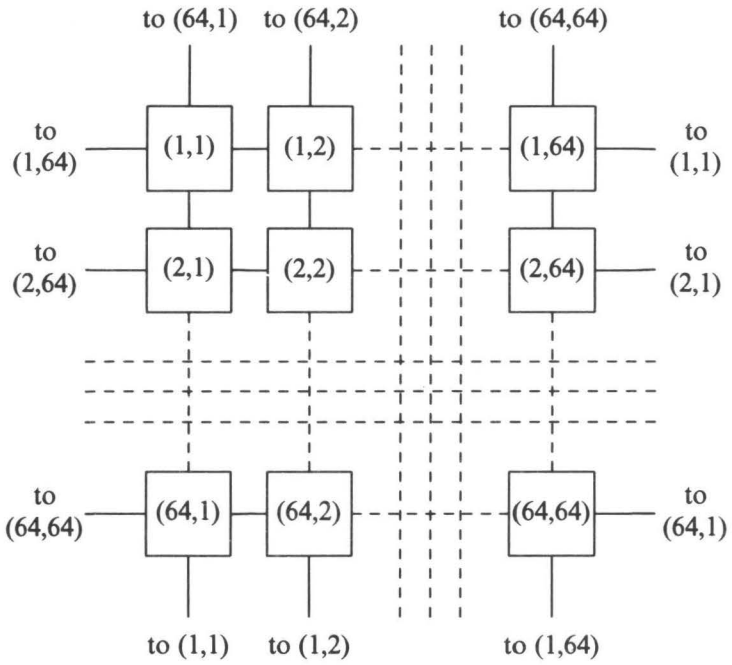


FIGURE 3.1. The DAP.

simulate a DAP of bigger dimensions on a DAP of dimension 64 by 64. He has to organize this himself, unlike, for example, in the Connection Machine, where the user only has to specify the dimension of the virtual machine to be simulated [Thinking Machines Corporation 1986].

The performance of a program is measured by counting the number of instructions executed by the DAP. To get an estimation of the CPU time, the number of instructions is multiplied by the average time needed for an instruction. This way of timing neglects the differences between execution times of the various instructions. There is no way to measure the CPU time used by the DAP exactly (since the DAP can be used shared with another user, the elapsed time between the start and the termination of a program does not give a reliable indication).

3.1.2. The CDC/CYBER-205 [Hockney & Jesshope 1981]

The CDC/CYBER-205 is a commercially available computer able to perform the same operation on all elements of vectors of variable length in a pipelined way. In order to do this, the functional units are segmented. Each segment does a small part of the operation to be performed and sends the results to its neighboring segment. In this way a pipeline is created; cf. Figure 3.2. The segmentation makes it possible to deliver, after a certain start-up time which is independent of the size of the vector, a result of such a vector operation at each clock cycle. When executing vector instructions, it is possible to specify

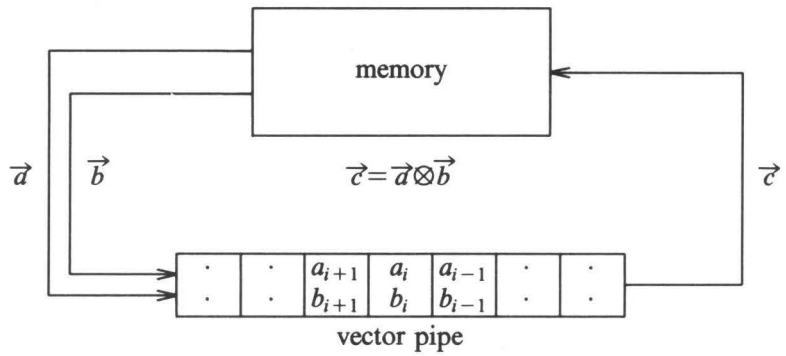


FIGURE 3.2. The CYBER-205.

whether or not a generated result must be stored. This enables the use of conditional operations.

Due to its capability of performing vector operations the CYBER-205 is very similar to an SIMD computer, although strictly spoken the results are generated in a sequential way.

The CYBER-205 can be programmed in the high-level language FORTRAN-200 [CDC 1983]. This extended standard FORTRAN contains vector instructions, which process vector elements in a pipelined manner. The FORTRAN-200 compiler is able to detect some parallelism in the program by trying to vectorize DO-loops, but far from every DO-loop can be vectorized in this way. By using the vector instructions, the programmer can specify which operations must be pipelined. However, this means that he has to analyze his algorithm and detect the parallelism himself. The CYBER-205 is capable of executing a program written in standard FORTRAN, but unless the compiler is told to try to vectorize this program and manages to vectorize at least part of it, no part of the program is executed in a pipelined manner.

The performance of a program on the CYBER-205 is measured by the CPU time needed to execute the program.

3.1.3. The Manchester dataflow machine [Gurd, Kirkham & Watson 1985]
Dataflow 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 data routes 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. Therefore, it can 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.

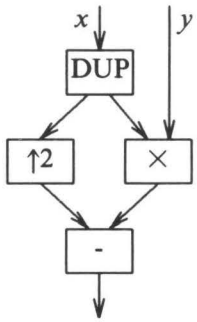


FIGURE 3.3. A dataflow graph.

Figure 3.3 shows a dataflow graph calculating $x^2 - xy$ using primitive boxes DUP (which duplicates its input), ↑2 (which produces the square of its input), × (which multiplies its inputs with each other) and − (which subtracts the right input from the left input). A possible execution sequence is shown in Figure 3.4; stars (⋆) represent the generated tokens moving through the graph.

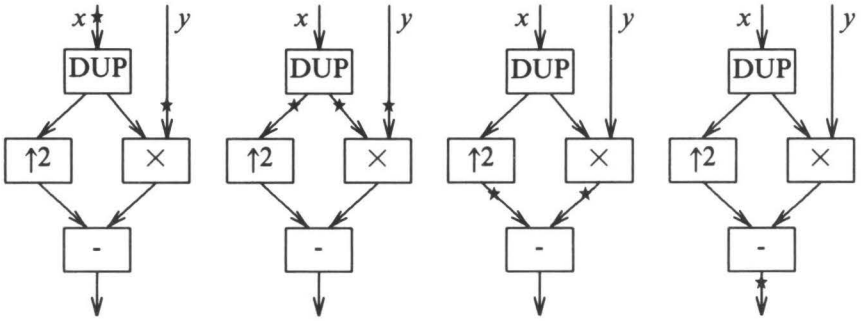


FIGURE 3.4. An execution sequence.

Exploiting the parallelism contained in the dataflow model requires an unconventional hardware organization. A general purpose dataflow machine needs a data structure of some sort to represent the dataflow graph of a 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.

The Manchester dataflow machine is an experimental computer, which consists of a ring of elements each performing a special task (see Figure 3.5). A

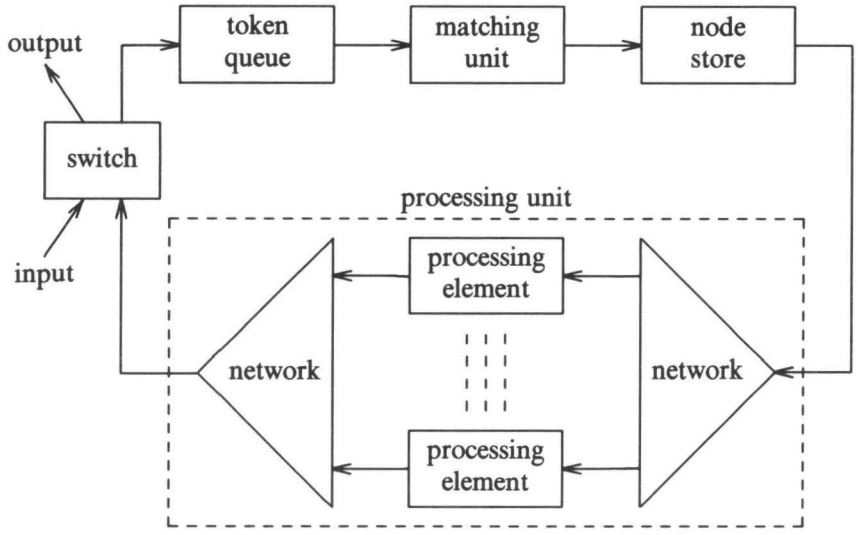


FIGURE 3.5. The Manchester dataflow machine.

token consists of a value and a destination node. The *token queue* buffers the incoming tokens and sends them, one at a time, to the *matching unit*. This is an associative memory, which groups tokens with the same destination node into packages and presents them to the node store. The matching 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 in the graph consists of the instruction to be performed, where an instruction is an elementary one such as in Figure 3.3, 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 a 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 (each processing element is able to take care of an executable package independently). 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. A 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 consists of a single ring with twenty processing elements.

The Manchester dataflow machine can be programmed in the high-level language SISAL (Streams and Iteration in a Single Assignment Language) [McGraw, Skedzielewski, Allan, Grit, Oldehoeft, Glauert, Kirkham & Noyce 1984]. SISAL has no concept of sequential execution and no direct control statements such as GOTO. To avoid the ambiguities that might arise from reassigning values to variables, the language allows each variable to be assigned only once in a program; in loops, a construct is provided to overcome the single assignment restriction. Further, SISAL has strict type and scope rules and prohibits all forms of side effects. More about single assignment languages can be found in Ackerman [1982]. The nature of a single assignment language makes it, in comparison with FORTRAN or PASCAL, easy to compile a program into a dataflow graph.

Due to the model of operation used, the parallelism in a program is detected by the dataflow machine itself. The only thing a programmer can do is trying to specify his program in such a way that the dataflow graph constructed is as broad as possible.

The Manchester dataflow machine is operated in the same way as the ICL/DAP. Program development and compilation is done on a host computer. The host first stores the generated dataflow graph in the node store and then inserts the input data via the switch into the ring. The data activate the dataflow machine. Output tokens leave the ring via the switch and are collected on the host.

To measure the performance of a program, the only information the dataflow machine provides is the execution time until the arrival of the first output token at the host. Therefore, a program has to be reorganized in such a way that it produces a single output token at the end of its execution, if a correct timing is needed.

In order to gain a better insight into the performance of programs, the Manchester Dataflow Group developed an emulator, which runs on a sequential computer. To keep this emulator manageable, some simplifying assumptions about the system architecture had to be made. The principal assumptions are the following:

(*i*) An unlimited number of processing elements is available and the throughput capacity of the ring is infinite.

(*ii*) The time needed to execute an instruction is equal for all instructions.

(*iii*) Output from an instruction can be transmitted to a successor instruction within the execution time period.

(*iv*) Enabled nodes are executed without delay.

As a consequence of these assumptions, the emulator considers the dataflow machine as a synchronized MIMD computer with an unbounded number of processors. Although the model created in this way is unrealistic, it is helpful in analyzing a program with respect to the parallelism detected by the dataflow concept and the expected running time on more mature dataflow computers.

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 $\pi = S_1/S_\infty$ gives a measure of the average parallelism in a program. A more detailed trace of the behavior of a program can be obtained if desired.

3.2. CHANGE MAKING [Kindervater & Trienekens 1988]

Given a coinage system with n types of coins, where coins of type i have value v_i $(i = 1,...,n)$, one wishes to determine the number of different combinations of coins with which amount Z can be paid without change.

Let $P(z,i)$ $(z \geqslant 0, i = 1,...,n)$ denote the number of different combinations amount z can be paid when coins of type 1 up to i may be used. In the change making problem, one wants to compute $P(Z,n)$. The following recursive equation holds:

$$P(z,i) = \sum_{k=0}^{\lfloor z/v_i \rfloor} P(z - kv_i, i-1) \quad (z \geqslant 0, i = 2,...,n),$$

with initial condition

$$P(z,1) = \begin{cases} 1 & \text{if } z = 0 \bmod v_1, \\ 0 & \text{otherwise.} \end{cases}$$

The change-making problem can also be seen as a network problem. Let $G = (V,A)$ be a directed graph. The set of vertices V consists of pairs (z,i) $(z = 0,...,Z, i = 0,...,n)$. There is an arc from vertex (z_1,i) to vertex $(z_2,i+1)$ if and only if $z_2 = z_1 + kv_{i+1}$ for some nonnegative integer k $(i = 0,...,n-1)$. The change-making problem is equivalent to the problem of determining the number of different paths in G from (0,0) to (Z,n). (A path α is different from a path β if α contains an arc not in β or β contains an arc not in α.)

The change-making problem can be solved by divide and conquer and by dynamic programming. Both techniques use the above recursion, but they use them in reverse directions. If the change-making problem is viewed as a graph problem, divide and conquer reduces to explicit enumeration of all paths in the graph whereas dynamic programming is a form of implicit enumeration of all paths.

3.2.1. Implementing divide and conquer

Divide and conquer boils down to a direct evaluation of $P(Z,n)$ through the recursion given above. In an SIMD machine, all processors must perform the same instructions. The number of subproblems in which a particular (sub)problem is split depends, however, entirely on the data of that instance. Fortunately, there exists an upper bound on this number and by adding dummy subproblems one can arrange that each subproblem is split in the same number as the others, i.e., a number equal to this upper bound. In this way, the processors can always execute the same instructions at a time.

On an SIMD machine, the obvious implementation is that each processor takes care of one subproblem. The number of subproblems created is exponential. Therefore, only very small-size problems can be solved using this strategy.

It is also possible to use different processors for solving the change-making problem for different amounts - where the coinage system remains the same - at the same time. Each processor then solves a problem sequentially and the time needed to do this equals the time needed to solve the largest problem. In our investigations, we only wanted to solve one instance of the change-making problem at a time. For this case, the SIMD machines are not well suited and we therefore implemented the divide and conquer approach only on the Manchester dataflow machine.

Recursion is a natural technique for programming divide and conquer. This technique results in a straightforward and elegant implementation on the Manchester dataflow machine. Due to the fine-grained parallelism of this machine, computations are performed asynchronously and in parallel wherever possible. The computations have to be synchronized for combining the solutions of the subproblems. The exact order in which the computations are performed is nondeterministic. The synchronization before combining the solutions of the subproblems ensures that this order is a feasible one.

3.2.2. Implementing dynamic programming

The dynamic programming algorithm can be stated as follows:

for $z \leftarrow 0$ **to** Z **do** $P(z,1) \leftarrow$ **if** $z = 0 \bmod v_1$ **then** 1 **else** 0;
for $i \leftarrow 2$ **to** n **do**
    for $z \leftarrow 0$ **to** Z **do**
        $P(z,i) \leftarrow 0$,
        for $k \leftarrow 0$ **to** $\lfloor z/v_i \rfloor$ **do** $P(z,i) \leftarrow P(z,i) + P(z - kv_i, i-1)$.

Note that if the change-making problem is solved for amount Z, it is solved for all smaller amounts as well.

The above algorithm can be implemented in a direct way on the Manchester dataflow machine. The computations are performed in some asynchronous feasible order. The parallelism is bounded by the synchronizations that occur because of the dependencies of consecutive iterations.

To implement the dynamic programming algorithm on the DAP and the CYBER-205, one has to analyze the parallelism in the program and state the detected parallelism explicitly using the tools the respective languages provide. The parallelism easiest to exploit in the dynamic programming algorithm resides in the **for** z loops. But to make this parallelism explicit, we must rewrite part of the algorithm.

We interchange the last **for** z loop with the **for** k loop. In the program thus obtained, it is immediately clear that the operations in the **for** z loops are independent and can be performed in parallel. In this way, we obtain the modified program:

for $z \leftarrow 0$ **to** Z **do** $P(z,1) \leftarrow$ **if** $z = 0 \bmod v_1$ **then** 1 **else** 0;
for $i \leftarrow 2$ **to** n **do**
    for $z \leftarrow 0$ **to** Z **do** $P(z,i) \leftarrow 0$,
    for $k \leftarrow 0$ **to** $\lfloor Z/v_i \rfloor$ **do**
        for $z \leftarrow kv_i$ **to** Z **do** $P(z,i) \leftarrow P(z,i) + P(z - kv_i, i-1)$.

The rewritten algorithm can be implemented straightforwardly on the DAP. In doing this, we have to view the DAP as a one-dimensional array of processors. Processor z computes the values $P(z,i)$ $(i = 1,...,n)$. The operations in a **for** z loop are executed in parallel. To compute the sum of the possible combinations, a processor needs the P-values of its kv_i-th neighbor of the previous iteration for $k = 1,...,\lfloor Z/v_i \rfloor$. This can be accomplished in parallel by using a DAP-FORTRAN shift routine. Such a routine has the nice property that is shifts in zeros for nonexisting values. Therefore, we can perform the last **for** z loop from z is zero up to Z, without using masks for the processors that compute the P-values for $z = 0,...,kv_i - 1$

Due to the fact that the DAP has only 4096 processors, the amount Z to be paid is limited to 4095.

For the CYBER-205 the same procedure can be applied, but instead of being processed in parallel, the operations in the **for** z loops are now processed in a pipelined (and strictly speaking sequential) manner.

3.2.3. Improving divide and conquer and dynamic programming

Divide and conquer as well as dynamic programming have their pros and cons for solving the change-making problem. Divide and conquer is very easy to program, but the subproblems generated are not mutually exclusive. So, it may happen that solutions to certain subproblems are recomputed. The computation could be sped up if these recomputations could be prevented. Dynamic programming solves all problems $P(z,i)$ $(z = 0,...,Z, i = 1,...,n)$ regardless whether or not the solution of a particular problem is needed to construct the solution of problem $P(Z,n)$. The computation could be sped up if there is a way to eliminate subproblems not needed in constructing the solution to $P(Z,n)$.

It is possible to combine the good sides of both methods. The idea is to use divide and conquer to construct the set of subproblems needed and thereafter dynamic programming to solve the problem using only this set of subproblems [Polya, Tarjan & Woods 1983]. This can be realized by adding a mechanism to the divide and conquer approach, which upon request for the solution of a particular subproblem takes the following steps:

- If the subproblem has already been solved, it returns the solution of this subproblem.
- If the subproblem is being solved at the moment, it queues the request for the solution of the subproblem and returns the solution as soon as it is available.
- If the subproblem has not been considered before, it solves this subproblem and stores the solution.

As in the case of the original divide and conquer algorithm, we implemented the improved algorithm only on the Manchester dataflow machine. The mechanism could not be written in SISAL due to the fact that its behavior is nonfunctional: given a certain input, the outcome is not completely determined by this input, but also by certain 'environmental' factors. The mechanism was written in TASS, an assembler language, and linked to the SISAL program.

3.2.4. Computational results

The ordering of the coins has consequences for the number of operations to be performed by the divide and conquer algorithm. Since the divide and conquer approach solves a subproblem by decomposition regardless whether or not this subproblem has been solved before, the approach is in essence a tree traversal, in which each leaf of the tree (a subproblem which can be solved without decomposition) must be visited exactly once. So, the work to be done is proportional to the number of edges in the tree. An optimal tree has the least number of edges. For such a tree, no node has more children than each of its children has. This is realized by splitting each subproblem using the remaining coin with the highest value. The coins should, therefore, be ordered by decreasing value.

The ordering of the coins has no consequences for the dynamic programming algorithm as long as the coin with the smallest value is used for the initializations. The first coin can be dealt with in $O(1)$ time, whereas the others need $O(Z/v_i)$ iterations for the combination of previous results ($i = 2,...,n$).

In all computational results shown, the ordering of the coins is optimal with respect to the method of solution used. The coinage system used is part of the Dutch system, made up of coins and bank notes of 1, 5, 10, 25, 100, 250, 500 and 1000 cents.

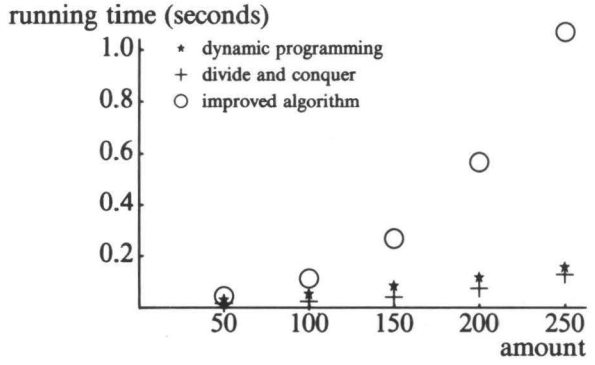


FIGURE 3.6. Change making: execution times on the Manchester dataflow machine with 20 processors.

Figure 3.6 shows some results of the dynamic programming, divide and conquer, and improved algorithms on the Manchester dataflow machine. Due to a limited memory capacity of the hardware, only small size problems could be

solved. The behavior of the programs on the Manchester dataflow machine can be explained from simulations on a sequential computer. These results are shown in Figures 3.7, 3.8 and 3.9. Due to memory restrictions, it was impossible to simulate bigger problems.

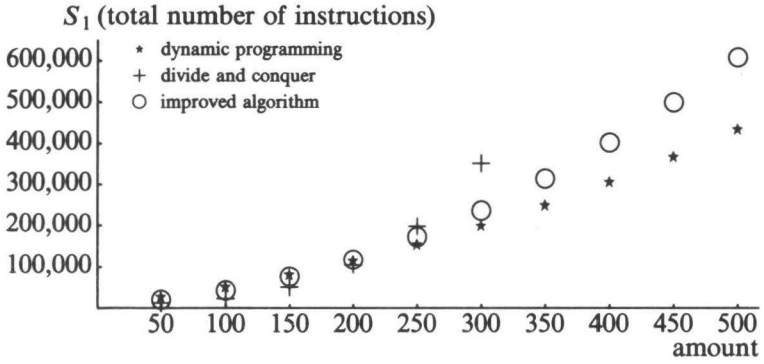


FIGURE 3.7. Change making: total number of instructions on the dataflow machine.

Figure 3.7 shows S_1 (the total number of instructions executed) versus the amount to be paid for the various programs. As expected, for small problems the divide and conquer program executes less instructions than the other two programs. But this reverses when the problem size increases. With increasing problem size, the improved algorithm executes less steps than divide and conquer but more steps than dynamic programming. The first is easily explained by the elimination of duplications. The second can only be explained if determining the state of a subproblem is more expensive than computing everything, needed or not.

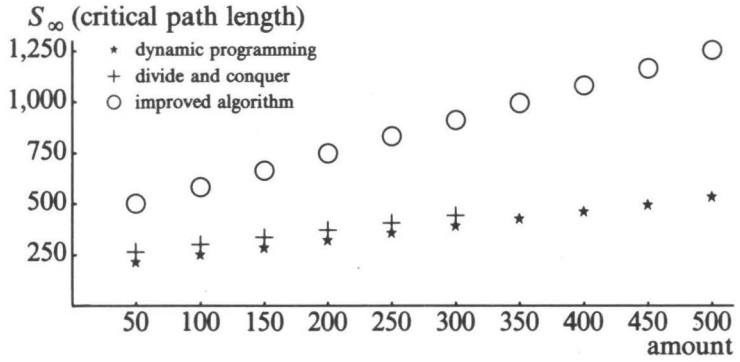


FIGURE 3.8. Change making: critical path length on the dataflow machine.

Figure 3.8 shows S_∞ (the total number of time steps needed if there was an unlimited number of processing elements) versus the amount to be paid. The S_∞ of divide and conquer and of dynamic programming behave in the same

way and differ by a constant. Both programs compute the solution by combining the solutions of subproblems. Since both use the same recursive formula, their S_∞'s have the same behavior. The difference is due to the work involved in the recursion. Since the recursion has always the same depth, the difference is a constant. The S_∞ of the improved program is larger. Determining the state of a subproblem appears to be a time consuming affair. Besides that, requests for the same subproblem have to be handled sequentially.

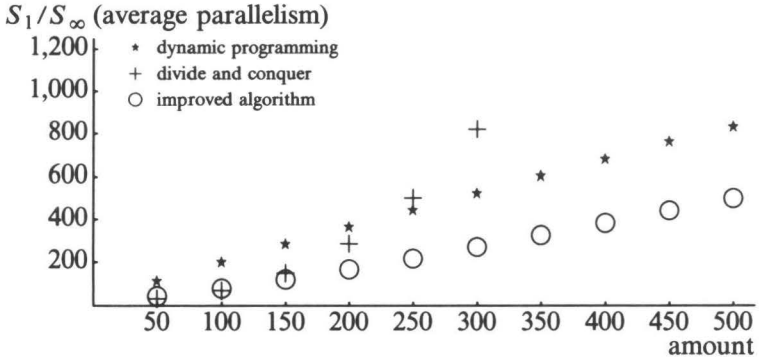


FIGURE 3.9. Change making: average parallelism on the dataflow machine.

Figure 3.9 shows the average parallelism π versus the amount to be paid. Divide and conquer shows an explosion in the parallelism with increasing problem size. This is because the subproblems generated are not mutually exclusive. If problem size increases, computing power is lost in solving an ever-increasing number of copies of the same subproblems in parallel. As expected, the average parallelism of the improved program is less than the average parallelism of dynamic programming. This is due to the sequential part of the mechanism which determines the state of a subproblem and to the fact that the solution of a problem must temporarily halt if one of its subproblems is being solved at the moment.

We conclude that, in the test environment under consideration, it is not worthwhile to be clever. It is much cheaper to compute everything.

Figure 3.10 shows our results on the execution of dynamic programming on the DAP and CYBER-205. For the problem sizes considered, the execution time on the DAP is linear. This execution time depends only on the number of subproblems to be combined. Taking the combinations can be performed in parallel and thus in constant time. The execution time on the DAP behaves in the same way as the critical path length of dynamic programming on the dataflow machine (Figure 3.8), because in the dataflow simulator we assume an unlimited number of processing elements for taking the combinations. As can be seen, the execution time on the CYBER-205 increases more than linear. This curve corresponds to the total number of instructions performed by the dataflow implementation (Figure 3.7).

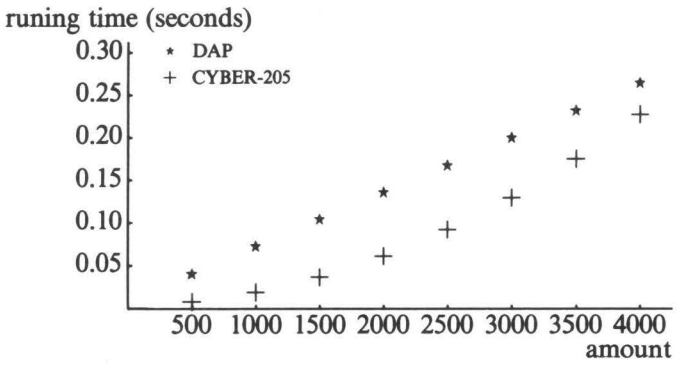


FIGURE 3.10. Change making: dynamic programming on the DAP and CYBER-205.

3.3. SHORTEST PATHS [Kindervater & Trienekens 1988]
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. We already came across the shortest paths problem in Section 2.2.4. There, we gave a polylog implementation of an algorithm from Lawler [1976]. In this section, we describe the implementation of algorithms due to Dijkstra [1959] and Floyd [1962] & Warshall [1962].

3.3.1. Dijkstra
Dijkstra's algorithm solves the one-to-all shortest paths problem in the case of nonnegative arc lengths. The nonnegativity of the lengths ensures that the total length of a path, when it is extended, cannot decrease. Therefore, it is possible to determine in the lth iteration of the algorithm the vertex lth closest to the origin. Denoting the origin by i^*, we have the algorithm:

$N \leftarrow \{1,...,n\} \setminus \{i^*\}$;
for all $j \in N$ **do** $d_j \leftarrow c_{i^*j}$; $d_{i^*} \leftarrow 0$;
for $l \leftarrow 2$ **to** n **do**
  $j^* \leftarrow \min\{j \mid d_j = \min\{d_k \mid k \in N\}, j \in N\}$,
  $N \leftarrow N \setminus \{j^*\}$,
  for all $j \in N$ **do** $d_j \leftarrow \min\{d_j, d_{j^*} + c_{j^*j}\}$.

In order to find all shortest paths, all vertices have to be considered as origin in turn. This can obviously be done in parallel.

On the DAP, this algorithm is implemented using vector instructions, where 64 processors take care of one vertex. Without considering the possibility of assigning more than one vertex to a set of 64 processors, this implementation restricts the problem size to 64. If a single processor would do the computations for one vertex, it would be possible to solve problems of size up to 4096. The memory capacity of a processor is, however, limited; only relatively small size problems fit into the DAP. Hence, to prevent processors from being idle,

they have to cooperate in the computations for a single vertex. Vector instructions then give about the best performance. The stages of the **for** l iteration are treated sequentially and the steps within a stage are performed in parallel. DAP-FORTRAN provides an (assembler) function which can compute the minimum of a vector using parallelism. The processors have to communicate with each other for finding the vertex with the next shortest distance from the origin. The number of this vertex and its corresponding distance have to be broadcast to all other processors. The '**for all** $j \in N$' instructions are executed for all $j \in \{1,...,n\}$ in parallel; with the use of a mask, which keeps track of the set N, only the relevant updates are performed. Since the computations are done in parallel and idle processors cannot do any useful work meanwhile, this is not a waste of computing power.

The CYBER-205 implementation is straightforward. The initialization and the instructions within the iterative loop can be pipelined. The language provides an (assembler) routine able to compute the minimum of a vector using the pipeline, and the conditional instructions are performed using masks.

We implemented Dijkstra's algorithm in two different ways on the Manchester dataflow machine. The first implementation closely resembles the SIMD implementation: a mask indicates the vertices of the set N to which the shortest distance still has to be computed; the operations are performed on all vertices and the obtained results are stored depending on the value of the mask. In the second implementation, a list of vertices belonging to the set N is maintained. The operations are performed on elements of this list; only values that are needed are computed. The first way is very easy to implement but has the disadvantage that computing power is wasted on vertices to which the shortest path is already known; the second way is harder to implement but does not waste computing power. Since the Manchester dataflow machine is an MIMD computer, processors that do unnecessary work could perform other available tasks, thus achieving a better overall performance. This is in contrast to an SIMD machine, where the overall performance is not influenced if some of the processors are silenced by a mask. Computations are performed asynchronously and in parallel wherever possible. In each iteration, however, the computations are synchronized on the point where the minimum value has to be computed. Updating the distances cannot be started unless the next shortest distance is known.

Both DAP and CYBER-205 FORTRAN provide an instruction for finding the index of an array element with minimum value. The SISAL language has a serious drawback in this respect: first the minimum value must be obtained and only then the corresponding index can be found.

3.3.2. Floyd-Warshall

The algorithm due to Floyd and Warshall computes the shortest path lengths for all pairs of vertices simultaneously. The arc lengths do not have to be non-negative and the occurrence of negative length cycles is detected. At the lth iteration, the shortest paths for all pairs of vertices are computed with intermediate vertices from the set $\{1,...,l\}$. The algorithm is as follows:

for $j \leftarrow 1$ **to** n **do for** $i \leftarrow 1$ **to** n **do** $d_{ij} \leftarrow c_{ij}$;
for $l \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do for** $i \leftarrow 1$ **to** n **do** $d_{ij} \leftarrow \min\{d_{ij}, d_{il} + d_{lj}\}$.

On the DAP processor (i,j) computes the length of a shortest path from vertex i to vertex j. At the lth iteration, processor (i,j) needs the current shortest distances computed by processor (i,l) and processor (l,j). This is achieved by broadcasting the lth column of the distance matrix rowwise and the lth row columnwise. This implementation restricts the problem size to 64. Bigger problems can be solved in this way by assigning more pairs of vertices to one processor.

On the CYBER-205, the initializing loops and the last **for** i loop of the algorithm are pipelined.

The Manchester dataflow machine will perform the algorithm in some arbitrary feasible order. Therefore, it might happen that values of different iterations are computed at the same time.

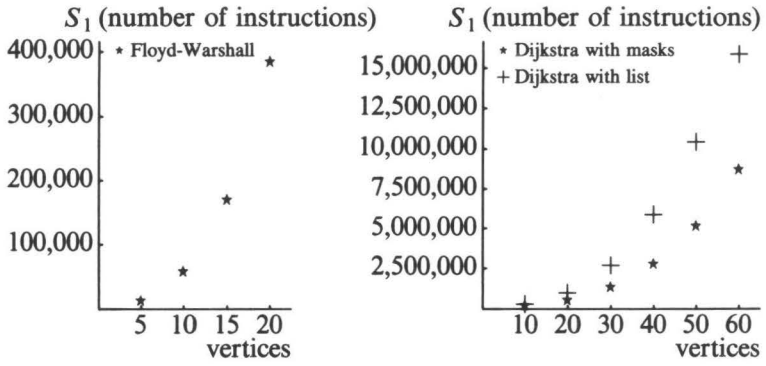
3.3.3. Computational results

On all machines we solved problems of size n up to 60, with distances drawn uniformly from [1,1000]. For each size, we generated three instances. The entries in the figures represent mean values. Dijkstra's algorithm is applied with all vertices as origin to make the results comparable to those of the Floyd-Warshall algorithm. On the DAP and CYBER-205 this has to be done sequentially, but on the Manchester dataflow machine simultaneous computation is possible.

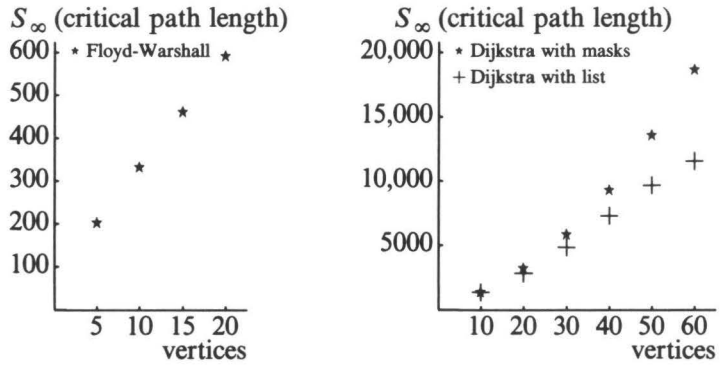
number vertices	Floyd-Warshall			Dijkstra		
	DAP	CYBER 205	CYBER 170-175	DAP	CYBER 205	CYBER 170-750
10	0.0025	0.001	0.002	0.021	0.001	0.002
20	0.0049	0.003	0.018	0.059	0.004	0.019
30	0.0073	0.007	0.057	0.124	0.010	0.058
40	0.0097	0.013	0.114	0.201	0.020	0.147
50	0.0121	0.022	0.215	0.311	0.034	0.271
60	0.0145	0.035	0.363	0.444	0.052	0.478

FIGURE 3.11. Shortest paths: running times (in seconds) on the DAP, CYBER-205 and CYBER-170-750.

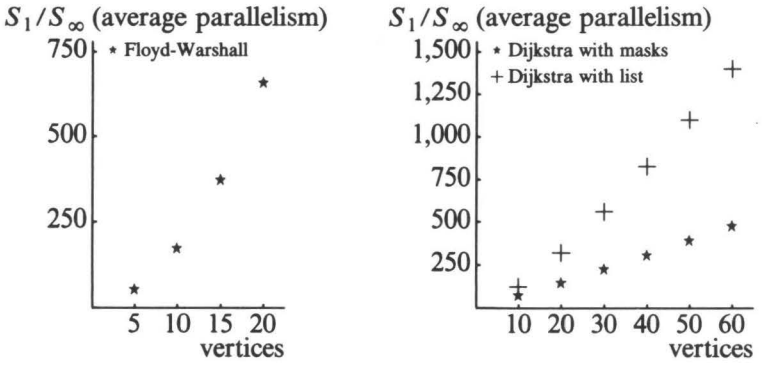
On the DAP, Floyd-Warshall shows a linear behavior and Dijkstra a quadratic one due to the fact that the basic routine has to be applied n times in sequence. At problem sizes which are a multiple of 64, a jump in the computing times will occur, after which the linear and quadratic behavior will continue. At those discontinuities, vectors and matrices outgrow their maximum size 64 and have to be split at the expense of longer computing times. On the CYBER-205, both algorithms have a cubic behavior, as on any sequential



(a) Total number of instructions.



(b) Critical path length.



(c) Average parallelism.

FIGURE 3.12. Shortest paths: performance on the dataflow machine.

computer, but the solution times for these small problems are about 10 times shorter than on a CYBER-170-750. Dijkstra's algorithm has a worse performance than Floyd-Warshall's. See Figure 3.11.

The simulator of the Manchester dataflow machine gives a linear behavior of S_∞ for the Floyd-Warshall algorithm. Due to the limited capacity of the matching store, the biggest problem we could handle with this algorithm was of size 20. Both versions of Dijkstra's algorithm have a nonlinear critical path length. This is because at each iteration a minimum has to be computed which takes $O(\log n)$ time in parallel. S_∞ is larger for the version using masks than for the one doing no useless work. For the total number of instructions performed and the overall parallelism it is the other way around. On a machine with a limited number of processors the former version will perform better, and on a powerful machine (or the simulator) the latter is to be preferred. Cf. Figure 3.12.

3.4. KNAPSACK [Kindervater & Trienekens 1988]

Given n items j, each with a profit c_j and a nonnegative weight a_j ($j = 1,...,n$), and given a knapsack with capacity b, one wishes to find a subset of the items of maximum total profit and of total weight no more than b. This can be formulated as an integer linear programming model of the following form:

$$\text{maximize } \sum_{j=1}^{n} c_j x_j$$

subject to

$$\sum_{j=1}^{n} a_j x_j \leqslant b,$$

$$x_j \in \{0,1\} \ (j = 1,...,n).$$

The problem is $\mathcal{NP}$-hard [Garey & Johnson 1979]. We consider two types of implicit enumeration: dynamic programming and branch and bound.

3.4.1. Dynamic programming

We introduce the notation $C(j,z) = \max_{S \subseteq \{1,...,j\}} \{\Sigma_{k \in S} c_k \mid \Sigma_{k \in S} a_k \leqslant z\}$. Using the optimality principle of dynamic programming, one attains the maximum profit $C(j,z)$ either by excluding item j and taking the profit $C(j-1,z)$ or by including item j and adding c_j to the profit $C(j-1,z-a_j)$. By recursively applying this idea, we get the following algorithm [Bellman 1957]:

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

On the DAP, the obvious implementation is to compute the values $C(j,z)$ for $z = 0,...,b$ in parallel and for $j = 1,...,n$ in sequence, where processor z computes the values $C(1,z)$, $C(2,z)$,...,$C(n,z)$. Here, the DAP is considered as an one-dimensional array of processors. In iteration j, a processor needs its own

C-value, that of its a_jth neighbor, and c_j. Using a DAP-FORTRAN shift routine, the data transfer of the C-values is accomplished for all processors in parallel. Because the shift routines fill in zeros for non-existing values, all states z can be dealt with in the same way. In this way, we get an $O(n)$ algorithm and a speedup of $O(b)$, provided b is no greater than 4095.

For the CYBER-205 basically the same procedure can be applied, although the parallel instructions are performed sequentially and a data shift is unnecessary. In the jth iteration, not all values $C(j,z)$ have to be evaluated explicitly. For all z with $\Sigma_{k\in\{1,\ldots,j\}} a_k \leqslant z \leqslant b$, all considered items fit together in the knapsack and hence $C(j,z) = \Sigma_{k\in\{1,\ldots,j\}} c_k$. In terms of the algorithm: in each iteration it is sufficient to compute the C-values up to the sum of the weights of the items considered. On a truly parallel computer (with enough processors), this observation would make no difference, but depending on the problem at hand it can lead to substantial savings on the sequential CYBER-205.

A SISAL version of Bellman's algorithm has been run on the Manchester dataflow machine. Since the computation is completely asynchronous, it might be possible that values of different iterations are evaluated at the same time, but a speedup of $O(b)$ remains best achievable.

3.4.2. Branch and bound

In the description of the branch and bound algorithm, we assume that the items have been ordered according to nonincreasing c_j/a_j.

An upper bound for the knapsack problem can be obtained by relaxing the integrality constraints $x_j \in \{0,1\}$ to $0 \leqslant x_j \leqslant 1$ $(j=1,\ldots,n)$. This linear-programming relaxation can be solved efficiently in $O(n)$ time, and in the solution at most one variable will be fractional. Setting the fractional variable to zero provides a feasible $\{0,1\}$-solution, which can be used for bounding the search tree. A node will be split by fixing variables to 0 or 1. Suppose a node has the first k variables fixed (denoted by $\tilde{x}_1,\ldots,\tilde{x}_k$), then we generate the subproblems $\{\tilde{x}_1,\ldots,\tilde{x}_k,1,\mathit{free},\ldots,\mathit{free}\}$, $\{\tilde{x}_1,\ldots,\tilde{x}_k,0,1,\mathit{free},\ldots,\mathit{free}\}$, $\{\tilde{x}_1,\ldots,\tilde{x}_k,0,0,1,\mathit{free},\ldots,\mathit{free}\}$, ..., $\{\tilde{x}_1,\ldots,\tilde{x}_k,0,0,0,0,\ldots,0\}$.

Since the evaluation of a node can hardly be parallelized efficiently on an SIMD-type computer, the parallelism has to be exploited on the DAP at the level of parallel evaluation of various nodes. By assigning each node to a different processor, at most 4096 nodes can be handled at the same time. In cases of branch and bound where the work to be done within a node very much depends on that node, the SIMD-restriction becomes a severe problem. Since the LP-relaxation of the knapsack problem can be solved in a regular way in linear time, all nodes can be dealt with concurrently. However, all processors of the DAP have to perform the same operation on data residing in the same place of their local memories. Therefore, specific information on a particular node cannot be taken into account satisfactorily. For example, fixed variables at one node may be free variables at another and the only way an SIMD machine can take care of this is by letting all processors look at all variables. Each time the nodes are split, the work has to be redistributed over the processors. If at any time more than 4096 nodes exist, only the 'best' 4096 nodes can

be evaluated concurrently. In our situation, we chose for a lexicographical enumeration scheme, i.e., a parallel depth first process. To achieve this, a priority queue is needed. This priority queue is maintained by all processors concurrently, but involves a lot of work.

On the CYBER-205, the same implementation will work. Here, the newly generated nodes have to be composed to a vector in order to use the pipeline and a priority queue is necessary if the vector length exceeds the maximum vector length 65535.

On the Manchester dataflow machine, we would like to have a completely asynchronous implementation of the algorithm. The MIMD-type parallelism allows for efficient implementation of the computation of upper and lower bounds. To kill subproblems that cannot yield the optimal solution, however, at each time the best feasible solution found so far has to be known by all subproblems under consideration. Since in SISAL, because of the single assignment rule, no global updatable variables exist, the only way to accomplish this within the language is by synchronizing the subproblem examinations after the computation of the lower bounds. But, synchronization means waste of computing power as processes have to wait for each other. Therefore we used the same assembler routine as in the improved divide and conquer algorithm (cf. Section 3.2.3) for simulating a global memory that contains the best overall feasible solution.

3.4.3. Computational results

For the DAP and CYBER-205, we generated three types of problems. In type 1 the profits and weights are drawn uniformly from [1,64]. To get types 2 and 3, we added 512 and 1024 to both the profits and the weights. For all three types we considered an instance with $n = 100$, 200 and 300; for dynamic programming b equals 4095, which is the largest problem size we can solve on the DAP without partitioning, and for branch and bound b equals 4200. From type 1 to 3, the knapsack problems are harder to solve by means of branch and bound methods. This comes from the empirical fact that, in general, knapsack problems are more difficult if the number of items that fit into the knapsack is smaller and the profit/weight-values are varying less.

Dynamic programming gives more or less the expected results on the DAP. The estimated CPU time grows linear with n, but there is no distinction for the different types. Since the distance which data have to travel increases with increasing type numbers, one expects an increasing computing time. The only information which can be retrieved from the DAP, however, is the number of instructions performed and that number is the same for all types of problems. The CYBER-205 computing times display the sequential nature of this machine. The running times are 20 times better than on the CYBER-170-750. Cf. Figure 3.13.

Branch and bound turns out to be inefficient on both the DAP and CYBER-205 (see Figure 3.14). The search trees for the type 1 problems are narrow. This implies for the DAP that only a small part of the processors is doing useful work and for the CYBER-205 that the vector lengths are small.

n	type	DAP	CYBER-205	CYBER-170-750
100	1	0.019	0.011	0.257
100	2	0.019	0.022	0.420
100	3	0.019	0.019	0.359
200	1	0.038	0.036	0.832
200	2	0.038	0.045	0.828
200	3	0.038	0.039	0.704
300	1	0.058	0.062	1.373
300	2	0.058	0.067	1.238
300	3	0.058	0.059	1.047

FIGURE 3.13. Knapsack: running times (in seconds) of dynamic programming on the DAP, CYBER-205 and CYBER-170-750 ($b = 4095$).

For the type 2 and 3 problems, the search trees are very broad. This ensures an economic use of the DAP processors and the CYBER-205 pipeline. But here the amount of work to redistribute the subproblems over the processors on the DAP and to rearrange the subproblems into a vector on the CYBER-205 is enormous. This part of the program completely dominates the computation of lower and upper bounds. For these reasons the traditional CYBER-170-750 performs better than the DAP and CYBER-205.

n	type	DAP	CYBER-205	CYBER-170-750
100	1	0.2	0.01	0.01
100	2	3.0	0.07	0.01
100	3	5.0	1.78	0.25
200	1	5.0	0.12	0.03
200	2	18.0	3.51	0.10
200	3	38.0	35.54	2.16
300	1	11.0	0.36	0.06
300	2	-	-	-
300	3	-	-	-

FIGURE 3.14. Knapsack: running times (in seconds) of branch and bound on the DAP, CYBER-205 and CYBER-170-750 ($b = 4200$).

On the Manchester dataflow machine, we only could run some very small problem instances. The profits and weights are drawn from [1,100]. We generated problems with $n = 10$, 20, 30 and 40 and $b = 100$, 200 and 300.

Dynamic programming shows an S_∞ linear in the problem size n and a parallelism growing with b. With growing b more elements fit into the knapsack. This explains an increasing S_∞ for constant n. Cf. Figure 3.15.

For the problem instances considered, the hardware results are comparable: for less than 10 processors the speedup increases almost linear in the number

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.

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.

FIGURE 3.15. Knapsack: dynamic programming on the dataflow machine.

of processors, after that hardly any gain is made (Figure 3.16). Apparently, the average parallelism is not enough to keep the processor elements busy.

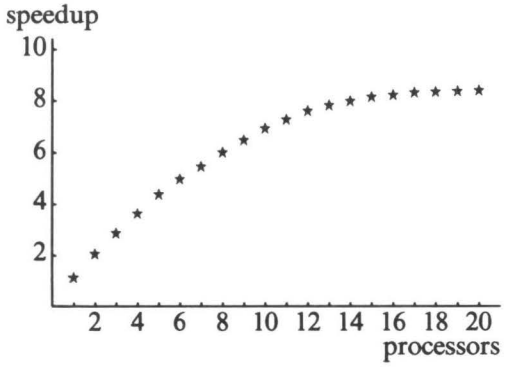


FIGURE 3.16. Knapsack: typical speedup curve for dynamic programming on the dataflow machine; $n = 40$ and $b = 300$.

Branch and bound results look promising. The S_∞ and π correspond to the depth and the width of the search tree; see Figure 3.17. Because communica-

n	$b = 100$	$b = 200$	$b = 300$
10	892	1226	750
20	1219	2300	1394
30	1287	2735	1767
40	4518	3407	5468

(a) Critical path length.

n	$b = 100$	$b = 200$	$b = 300$
10	9	8	9
20	14	21	12
30	19	21	15
40	48	24	77

(b) Average parallelism.

FIGURE 3.17. Knapsack: branch and bound on the dataflow machine.

tion is cheap and the parallelism is fine-grained, no time is lost in the assignment of tasks to processors. Therefore, it can be expected that problem instances for which broad search trees are needed can be solved efficiently on this sort of machines.

4

Experiments with Coarse-Grained Parallelism: Branch and Bound

In contrast to the computers considered in the previous chapter, many parallel architectures consist of a set of powerful processors interconnected by a network that is relatively slow compared to the individual processor speeds. To obtain a good performance of algorithms, the number of data transfers must be kept to a minimum; in many cases, it is even worthwhile to let several processors do the same work instead of assigning the job to one processor and then broadcast the results. In this chapter, we investigate this type of architectures with respect to their suitability for branch and bound algorithms.

The nodes in search trees generated by branch and bound methods each deal with a subset of the solution set (cf. Chapter 3). The observation that any two nodes, neither of which is an ancestor of the other, can be solved independently, provides a natural basis for the parallelization of branch and bound algorithms: an idle processor searches for an available but not yet expanded node, evaluates this node, thereby possibly creating new nodes, and informs the other processors on newly found better solutions. Except for the broadcasting of these feasible solutions, interprocessor communications may be necessary for finding an expandable node and for the storage of generated nodes. There are several possibilities for the storage of the nodes of the search tree.

(*i*) *One central queue.* The nodes are kept in a queue which is shared by all processors. Communication is always necessary for the selection of an expandable node and for the storage of generated nodes. This implementation has the advantage that all information of the generated tree is available at one place and that the processors can work on the most promising nodes.

(*ii*) *Local queues.* Each processor maintains a queue of nodes. When searching for a node to be evaluated, a processor first looks in its own queue and, if this queue is empty, in the queues of the other processors. New nodes are stored in the processor's own queue. In this situation, the number of

communications is limited, but the possibility exists that a processor's queue may contain only uninteresting nodes and that the processor is doing what turns out to be useless work.

For most multiprocessor systems, a hybrid variant of a central queue, queues shared by only part of the processors and local queues will give the best results. Which combination is to be preferred is highly architecture dependent.

The last issue of branch and bound to be cleared is the order in which nodes are considered for evaluation. In sequential computers, one usually makes a choice between two selection rules: *depth first search*, i.e., the nodes are considered in a lexicographical order (last in first out), and *best bound search*, i.e., the nodes are considered according to increasing lower bounds in the case of minimization (decreasing upper bounds in the case of maximization). In general, best bound search generates less nodes than depth first search (conditions can be found in [Fox, Lenstra, Rinnooy Kan & Schrage 1978]), but has higher changeover costs between the evaluation of two nodes. The selection rules used in parallel algorithms are mostly extensions of the two selection rules discussed here.

The behavior of branch and bound algorithms is sometimes unexpected: there exist examples of anomalous behavior in which $p+1$ processors are slower than p processors or more than proportionally faster. The reason for this lies in the fact that the point in time at which a node becomes available depends on the number of processors and that this influences how the tree is searched. Another way to express this is that, for each number of processors, we essentially have a different underlying sequential algorithm. We will return to this later on.

This chapter is organized as follows. Section 4.1 describes the *IBM/LCAP* and and a *local area network of workstations*, architectures in which the processors are powerful computers of their own accord. Section 4.2 considers the implementation of two branch and bound algorithms for the *traveling salesman problem* and Section 4.3 deals with the *job shop scheduling problem*. The anomalous behavior of branch and bound algorithms is discussed in Section 4.4.

4.1. Architectures

In this section we will briefly describe the IBM/LCAP and a multiprocessor consisting of workstations connected on an ethernet.

4.1.1. The IBM/LCAP [Di Chio & Zecca 1985]

The IBM/LCAP (Loosely Coupled Array of Processors) consists of a master processor (IBM/4381-3) which is connected to ten slave processors (FPS/164); cf. Figure 4.1. 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 not pay to send very small tasks.

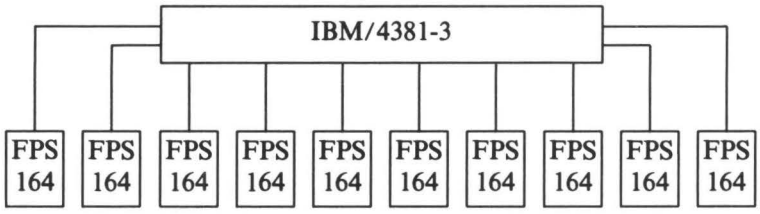


FIGURE 4.1. The IBM/LCAP.

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

(*i*) The processes 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 LCAP is programmable in FORTRAN. For the implementation of the shared memory model, an extension of the language is provided; the master-slave principle is effectuated by a set of communication routines that are accessible from standard FORTRAN programs.

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.

4.1.2. Local area network of workstations [Gardner, Gerard, Mowers, Nemeth & Schnabel 1986]

The Boulder Distributed Processing Utilities Package (DPUP) 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 4.2). 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.

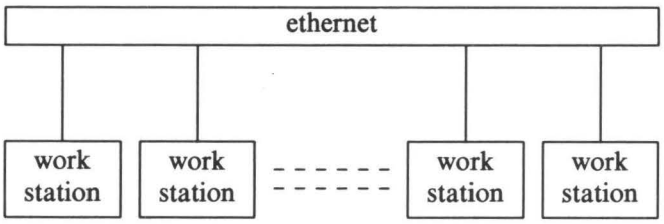


FIGURE 4.2. Work stations connected on an ethernet.

DPUP can be used from programs written in C and 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.

4.2. Traveling salesman

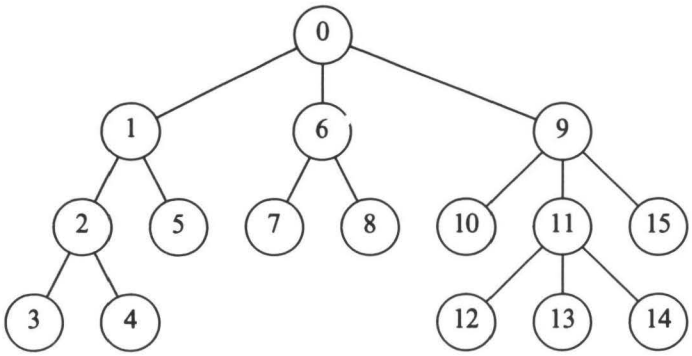
Recall that the traveling salesman problem (TSP) is the problem of finding a Hamiltonian cycle of minimum length in a complete undirected graph with given edge lengths (see Chapter 2). Since the TSP is a famous hard problem, it is an appealing object for parallel branch and bound. We describe two approaches in detail. Others can be found in, for example, Finkel & Manber [1987] and Pekny & Miller [1988]. (The last reference contains an algorithm for the asymmetric TSP.)

4.2.1. An assignment based algorithm on a shared memory computer [Pruul 1975; Pruul, Nemhauser & Rushmeier 1988]

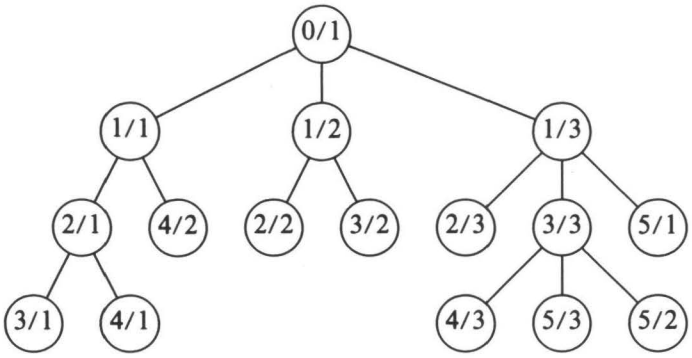
A traditional branch and bound method for the TSP uses a bounding mechanism based on the linear assignment relaxation, a branching rule based on subtour elimination, and a strategy for selecting new nodes for examination based on depth first tree search. The details are of no concern here and can be found in the book edited by Lawler, Lenstra, Rinnooy Kan & Shmoys [1985]. Figure 4.3(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 4.3(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.



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



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

FIGURE 4.3. Depth first tree search.

4.2.2. A 1-tree based algorithm on a local area network of workstations [Trienekens 1989b]

The branch and bound algorithm considered by Trienekens combines a lower bound based on 1-trees and a branching scheme of Jonker & Volgenant (cf. Lawler, Lenstra, Rinnooy Kan & Shmoys [1985]).

The implementation using the Boulder DPUP software 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. In this strategy, the master has full knowledge of the search tree generated so far. The number of communications is, however, high. Since a lot of work is involved in the lower bound computations, the time for node evaluation will dominate the

time for interprocessor communication.

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.

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.

4.3. Job shop scheduling

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 [Garey & Johnson 1979] appears to be very difficult. Already small instances are hard to solve. The branch and bound algorithm from Lageweg, Lenstra & Rinnooy Kan [1977] computes lower bounds by relaxing the capacity constraints on all machines 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. A parallel version of the algorithm was implemented on the IBM/LCAP and the Boulder local area network of workstations.

4.3.1. Implementation on the IBM/LCAP

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 [Muth & Thompson 1963]. Reported are the maximum number of nodes branched by a slave,

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

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 4.4. When the number of slaves increases from one to 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.

4.3.2. Implementation on a local area network of workstations

On the Boulder network of workstations, the algorithm is implemented using the master-slave principle in the same way as in Section 4.2.2, i.e., a master process keeps track of the nodes to be evaluated and idle slave processes receive a node with the least lower bound from the master process, evaluate this node and send the results back to the master process. The most powerful workstation (SUN 3/180) executes the master process and the other workstations (SUN 3/75) each perform a slave process.

The performance of the algorithm is illustrated on an instance with six jobs, each consisting of six operations, and six machines [Muth & Thompson 1963]. For each number of slave processes, the algorithm was run four times. The computing times fluctuated up to 10%, but the generated search tree was the same (in all test runs the algorithm branched 466 nodes). The results presented in Figure 4.5 are mean values.

Since in all cases the same search tree is generated, the master process has to perform an amount of work which is more or less independent of the number of slave processes. The time needed for this is a lower bound on the

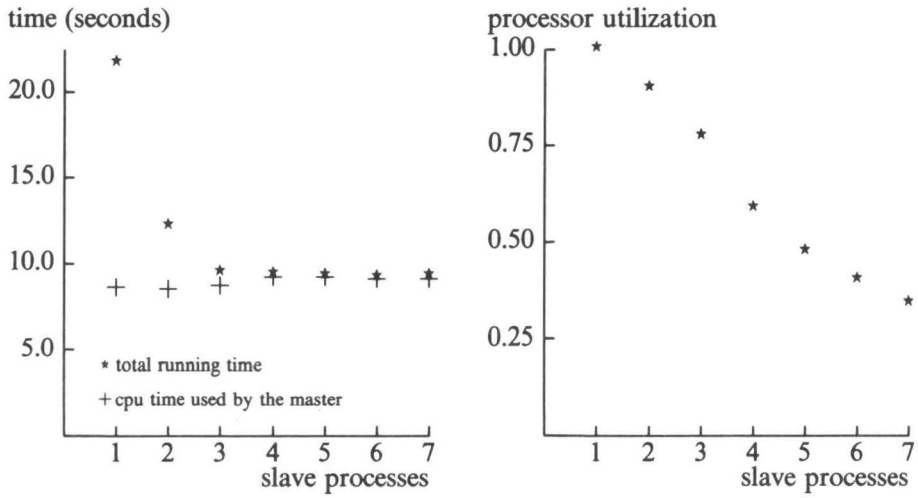


FIGURE 4.5. The job shop algorithm on a local area network of workstations.

computation time. As long as the number of slave processes is low, the master process succeeds in keeping the slaves busy. For a high number of processors, the work to be done by the master dominates and the slaves are waiting most of the time.

Although the instance considered is small, it clearly shows the weakness of the master-slave principle with only one central queue. The bottleneck of the master will show up for any instance as the number of processors increases. In that case, an approach in the same direction as described in the previous section has to be preferred.

4.4. ANOMALOUS BEHAVIOR [Burton, Huntbach, McKeown & Rayward-Smith 1983; Lai & Sahni 1984; Lai & Sprague 1985, 1986; Li & Wah 1986]

The branch and bound algorithms of the previous sections sometimes showed a pleasant behavior: adding a processor decreases the running time more than proportionally. Unfortunately, it is also possible that the addition of a processor slows down the computation. In this section, we discuss the conditions under which such anomalous behavior may occur. Branch and bound closely resembles the model described by Graham [1969], who considers a list scheduling algorithm for the multiprocessor scheduling problem (see Section 2.3.5) subject to precedence constraints between the jobs. It is not surprising that a number of Graham's observations apply here.

For simplicity, we consider a synchronized multiprocessor system in which the evaluation of a node in a branch and bound tree takes constant time and 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. The running time of the algorithm is measured by the number of iterations needed. These restrictions are not essential; Trienekens [1989a]

showed that the results can be extended to asynchronous models. We first analyze the running times for one and p (>1) processors and then consider the case where we have p_1 and p_2 processors with $1<p_1<p_2$.

Burton, Huntbach, McKeown & Rayward-Smith give examples in which two processors are more than twice as fast as a single processor, or slower than a single one. In Figures 4.6 and 4.7 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.

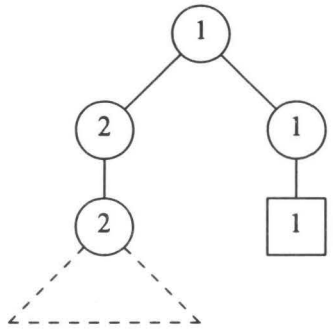


FIGURE 4.6. Anomalous behavior: best case for two processors.

In the tree of Figure 4.6, 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.

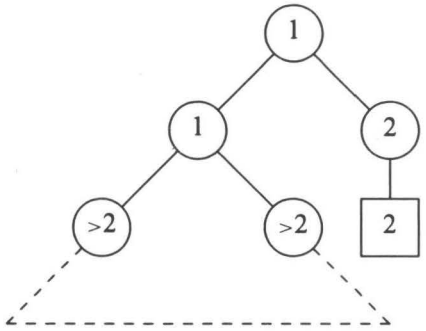
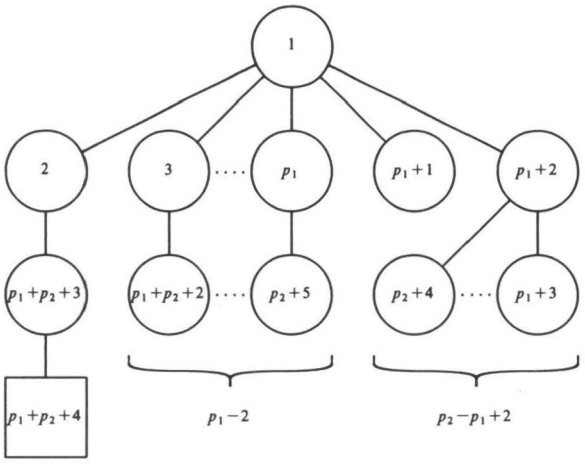
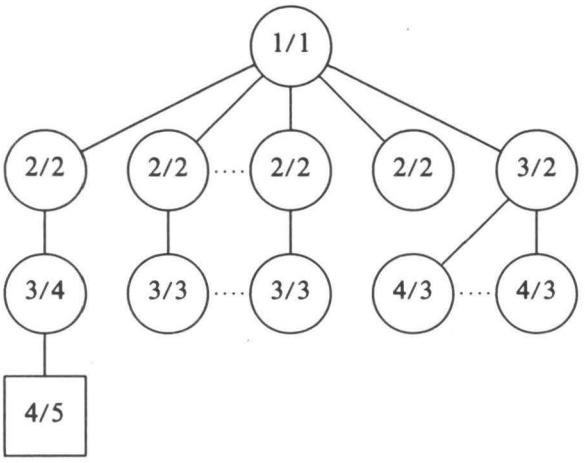


FIGURE 4.7. Anomalous behavior: worst case for two processors.

In the tree of Figure 4.7, 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 children 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.



(a) Lower bounds.



(b) Node t_1/t_2 is evaluated at time t_i with p_i processors ($i = 1,2$).

FIGURE 4.8. Best bound search: deceleration when increasing the number of processors from p_1 to p_2, with $p_1 < p_2 < 2(p_1 - 1)$.

The slow down anomaly of Figure 4.7 is easy to explain. The children of a node P_1 are more attractive than some other node P_2 in the tree, while father node P_1 is less attractive than node P_2. This situation can be prevented by requiring that the priority function is *monotone* in the sense that the priority of the children is always no greater than the priority of the father. However, Lai & Sahni show that this is not sufficient. It is also necessary that all priorities are *distinct*. Under these two conditions, it can be shown that a multiprocessor system at each iteration evaluates at least one node that must be expanded by the sequential algorithm. If there are no such nodes left, the multiprocessor system has all the information necessary to deliver the optimal solution of the problem. Hence, the multiprocessor system needs no more iterations than the single processor machine. These requirements for the priority function are not severe. For example, for depth first search the conditions are already fulfilled, and best bound search can be adapted by using the place of the node in the tree as an extra criterion.

In Figure 4.6, the enormous speedup is achieved because the two processors do not have to evaluate all the nodes expanded by the sequential algorithm. Indeed, it is easy to see that a speedup anomaly cannot occur unless a multiprocessor system does not consider the complete search tree which is treated by the sequential algorithm. This can only happen when the sequential algorithm evaluates nodes that have a lower bound which is no less than the optimal solution value. It is clear that this situation may appear in depth first search. For best bound search it is necessary that several nodes have a lower bound equal to the optimal solution value.

The case for p_1 and p_2 processors ($1<p_1<p_2$) is not so easy. As an example given by Lai & Sahni shows (see Figure 4.8), deceleration anomalies may occur in very natural situations. The conditions of the previous case are not sufficient to prevent a slowdown. To be sure that deceleration does not happen, additional information on the branch and bound tree to be generated is needed. Most of the time, this knowledge is not available before the branch and bound algorithm has finished. The good news is, however, that the (limited) experiments with branch and bound show that in practice one does not have to worry about this sort of anomalous behavior.

5

A Queueing Network Model for Distributed Enumeration

As we have seen in the previous chapter, the master-slave principle is very appealing for parallel branch and bound methods. In particular, the following implementation is attractive. The master process keeps track of the search tree generated so far, orders the nodes according to their priorities, and sends the node with the highest priority to a slave process as soon as one becomes idle. The slave processes evaluate the nodes they receive and send the results back to the master process. In this implementation, the master process has full knowledge of the search tree generated so far and can ensure that the most promising part of the search tree is examined by the slave processes. However, the processing speed of the master process must be high enough to handle the communication requests of the slave processes adequately, and to maintain the priority queue of available nodes. Otherwise, the benefits of this implementation are likely to disappear: valuable information may not reach the master process in time and the slave processes may be forced to do what turns out to be useless work, or the slave processes may become idle.

In Sections 4.2.2 and 4.3.2, we described two experiments with the master-slave principle with one central queue of nodes. One experiment was rather successful, the other was not. In this chapter, we want to obtain insight into the performance of this particular type of implementation via a queueing theoretic approach. We are interested in the effect of changing the speed of the master or the slaves and of changing the number of slave processes.

We consider two variants. In the next section, we will describe a queueing network which models the master-slave variant, where a slave process evaluates a node, puts the results in a queue at the master process, and immediately continues with a new node, already processed by the master process. The benefits of this variant are clear: the slave processes are only idle if there are no nodes available for evaluation. However, if the number of nodes available for

evaluation grows, the master process becomes slower and, as a result, a long queue of nodes waiting to be processed by the master process may form. This has the effect that valuable information may not reach the master process in time and that the slave processes may be forced to do what turns out to be useless work. In Section 5.2, the queueing model is analyzed by means of a fluid flow approximation, and the techniques developed are illustrated by some numerical examples in Section 5.3.

Section 5.4 studies the variant where a slave process receives a new node only after the master process has consumed the slave's latest results. This second variant avoids the possibility of a long queue in front of the master process. The disadvantage is that a slow master process causes idleness of the slave processes. Here, the appropriate queueing system turns out to be a so-called machine repair model.

Throughout this chapter, we assume that at any point in time there are enough nodes available for evaluation by the slaves. This is not a serious restriction since parallel computers are particularly useful for solving problem instances that require large search trees for finding the optimal solution.

5.1. Queueing model description

In the queueing network representation of the parallel processing of branch and bound nodes, the master process is represented by a single server M, and the P slave processes are represented by P parallel servers $S_1,...,S_P$, gathered in a service station S; cf. Figure 5.1. The nodes are represented by customers. The splitting of a node is performed by the birth and death process *B&D*. To further specify the queueing network, we have to describe the routing of customers and the service processes at M and S.

The routing of customers. When a customer arrives at M, he may have to wait in a queue until his service starts. After having obtained his service requirement, the customer leaves and immediately arrives at S, where he usually has to wait in a queue. In this queue, each customer has a priority which determines the order in which the customers are served by S. In the branch and bound algorithms under consideration, the priority queue is maintained by the master process. In the queueing network model, however, it is more natural to identify the priority queue with the queue at service center S. Now there are two possibilities:

(i) Before the customer is taken into service, the service center M receives information on which ground it decides to throw away a part of the queue at S, to which this customer belongs: the customer is instantaneously removed from the queueing network. This corresponds to the situation that the master obtains information from a node which makes the analysis of the nodes in a part of the priority queue obsolete. Customers who are thrown out of the queue at S are not replaced by other customers.

(ii) After a (possibly zero-length) waiting period, the customer is taken into service by one of the P servers; after having obtained his required service, he leaves S.

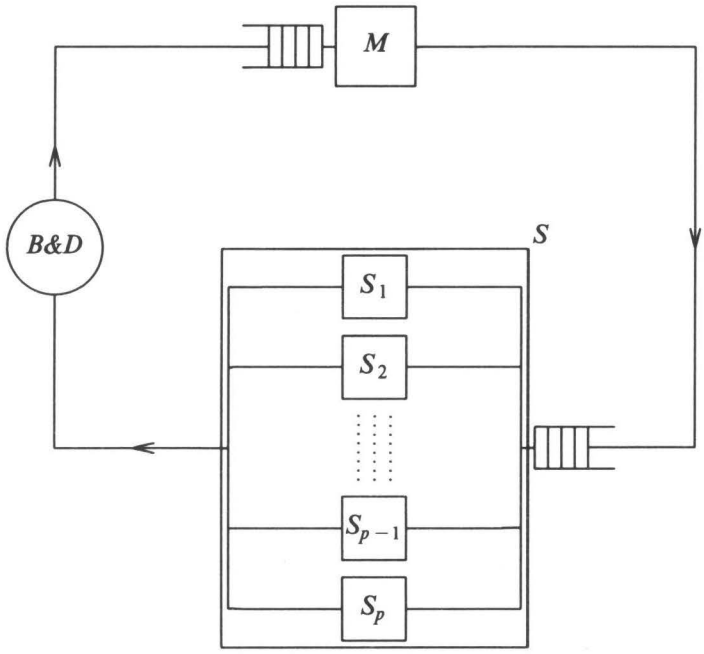


FIGURE 5.1. The queueing network model.

A customer who has successfully completed a service in S arrives at the birth and death process *B&D*. There, he leaves the queueing network, but he is immediately replaced by 0, 1 or 2 new customers, with probabilities p_0, p_1, p_2 respectively; $p_0+p_1+p_2 = 1$ (we assume that a branch and bound node has at most two descendants; the analysis to be presented in Section 5.2 remains valid when this assumption is relaxed). These new customers immediately arrive at M. The probabilities p_i may vary with time; we denote them by $p_i(t)$. The mean increase of the number of customers in the network after a service completion in S at time t will be denoted by

$$\phi(t) = p_1(t)+2p_2(t)-1. \tag{5.1.1}$$

In approximation, $\phi(t)$ will be a decreasing function of t, with $\phi(0)=1$ and $\phi(\infty)=-1$. In the branch and bound setting, this corresponds to the observation that the number of nodes generated by a node usually equals two in the beginning of a tree search, and that this number decreases to zero in the course of time. For most of the subsequent calculations, the exact form of ϕ is irrelevant.

The service process at M. The single server M serves customers in order of arrival ('first-come first-served'). M's service of a customer consists of two parts:

(i) a constant part of length a, which reflects the master's processing of the information contained in a node;

(ii) a part of length $b \ln(1+y)$, which reflects the master's putting a node in a priority queue of size y. Note that insertion in a priority queue requires $O(\ln y)$ time units when its size is y.

Hence the total service time of a customer in M, in the case that this customer has to be inserted in a priority queue of size y, equals

$$a + b \ln(1+y).$$

Instead of constants, a and b may also be stochastic variables; in the analysis, that will turn out to be of minor importance.

In the following, the queue length of waiting customers in M at time t will be denoted by $\mathbf{y}_M(t)$.

The service process at S. When a server in S becomes idle, the customer at the front of the queue (if any) is immediately taken into service. When a newly arriving customer finds several servers idle, he chooses an arbitrary idle server. We assume that the P slave processes - and hence the P servers - are identical.

The service times of customers at S are independent, identically distributed stochastic variables with mean $1/\alpha$. Generally it will not be necessary to specify the service time distribution at S further, but at a few places in the text we will consider the case of a negative exponential distribution.

Apparently the 'capacity' of S is $P\alpha$: S is able to handle $P\alpha$ customers per unit of time, on the average. We assume that $1/a \gg P\alpha$, i.e., M's maximum speed of handling customers is much higher than that of S. Of course a large queue at S will slow down M considerably.

The length of the queue at S at time t will be denoted by $\mathbf{y}_S(t)$.

Remark. In parallel computers, communication takes a certain amount of time. We assume that the time to send messages between the master and the slaves has been taken into account in the service times.

5.2. Mathematical analysis of the node processing mechanism

In the previous section, a queueing network model was introduced to describe the node processing mechanism in parallel branch and bound in a master-slave environment. In this section, we present a mathematical analysis of the queue length processes in that queueing network. This analysis is basically of a non-stochastic nature. Of course $\mathbf{y}_M(t)$ and $\mathbf{y}_S(t)$ are stochastic processes, which may exhibit considerable fluctuations. Information concerning the (random) behavior of $\mathbf{y}_S(t)$ and $\mathbf{y}_M(t)$ requires a detailed queueing analysis. The problem of analyzing the transient behavior of queues is notoriously difficult, even when arrival and service rates are constant. In our case, a detailed mathematical

analysis of the evolution of, say, $\mathbf{y}_M(t)$ requires analysis of the transient behavior of a single server queue with complex time dependent arrival and service rates. Hardly any results are available in the literature concerning such problems. Massey [1985] studies the asymptotic queue length behavior of an M/M/1 queue (i.e., a single server queue with Poisson arrival process and negative exponentially distributed service times) with time dependent arrival and service rates. Rider [1976] and Rothkopf & Oren [1979] derive approximations for the mean queue length at time t in this M/M/1 queue; their approximations are fairly complicated. These models are considerably less complex than the model under consideration, with its interaction between M and S. As there seems to be little hope of obtaining useful exact results, we have taken recourse to a standard type of approximation. The approximation, simple as it may be, will turn out to yield much insight into the behavior of both queue length processes. In queueing literature it is called a *fluid flow approximation* (cf. Newell [1971]).

Fluid flow approximations are based on the following observations: (i) In a system with a large queue, many customers must arrive and depart before the queue changes much (in a relative sense). (ii) In a period of time sufficiently long for many arrivals and departures to occur, the effect of random fluctuations - due to the stochastic nature of the arrival and service processes - will be relatively small. The latter observation can be theoretically supported by Laws of Large Numbers and Central Limit Theorems. As an example, consider the departure process from the saturated service station S. Assume that successive service times in S are independent, negative exponentially distributed stochastic variables with mean $1/\alpha$. Then successive departure intervals are independent, negative exponentially distributed stochastic variables with mean $1/P\alpha$. The number of departures, $D(t)$, in an interval of length t is Poisson distributed with mean $P\alpha t$ and variance $P\alpha t$. According to the Strong Law of Large Numbers,

$$\frac{D(t) - E[D(t)]}{E[D(t)]} = \frac{D(t) - P\alpha t}{P\alpha t} \to 0, \quad t \to \infty, \tag{5.2.1}$$

with probability one. Supplementary information is provided by the Central Limit Theorem, which shows that for large t,

$$Pr\{-y < \frac{D(t) - P\alpha t}{\sqrt{P\alpha t}} < y\} \approx \frac{1}{\sqrt{2\pi}} \int_{-y}^{y} \exp(-x^2/2)\, dx. \tag{5.2.2}$$

Based on the above observations, we can replace the discrete and random arrivals and departures at M and S by nonrandom continua (cf. Newell [1971]): we can view M and S as reservoirs, with fluids flowing in and out. In this setting, a reservoir can be considered to be empty for a lengthy period of time, without really being completely empty; it is empty only on a scale of measurement in which fluctuations in cumulative flows are negligible.

In our fluid flow analysis of the node processing mechanism, we distinguish two possible states in which the system can be, namely:

ME: M is empty;
MNE: M is not empty.

Once more, at a time t_0 at which the system is in state *ME*, $y_M(t_0)$ is not necessarily zero, but it is negligible. Note that y is not printed boldface, as the queue length process is no longer assumed to be stochastic.

Throughout the analysis, S will be considered to be nonempty, with all P servers being occupied. When $y_S(t)$ would become zero, M would serve so fast that S would very soon saturate again. This is no longer true when there are hardly any customers in the system, but that situation is of not much interest.

For arbitrary functions ϕ, the system state can switch back and forth between *ME* and *MNE* several times. In Subsection 5.2.1 we describe, in detail, the behavior of the queue length processes in each of these two states. In Subsection 5.2.2 we follow the evolution of $y_M(t)$ and $y_S(t)$ from beginning to end, in the case of a non-increasing function ϕ with $\phi(0)=1$ and $\phi(\infty)=-1$.

In Section 5.1 we have mentioned an important feature of branch and bound: the master obtains information from a node which makes the analysis of the nodes in a part of the priority queue obsolete. In the queueing network setting, this corresponds to the situation that, upon departure of a customer from M, the tail of the queue at S is removed from the network: $y_S(t)$ instantaneously is reduced by a certain number. In describing the queue length processes in states *ME* and *MNE*, we first ignore such *sudden reductions of the queue at S*. In Subsection 5.2.3 we point out which simple changes are required to take reductions of the queue at S into account.

5.2.1. Queue length behavior in the states ME and MNE

We shall mainly concentrate on the queue length process $y_S(t)$; $y_M(t)$ follows from the relation

$$y_S(t) + y_M(t) = P\alpha \int_0^t \phi(u)\,du, \qquad t \geqslant 0. \tag{5.2.3}$$

This relation holds for general ϕ, ignoring the possibility of a sudden reduction of the queue at S.

The state ME. In state *ME*, M is clearly nonsaturated: its input rate is lower than its maximum possible processing rate. The output rate of S is $P\alpha$ (all P servers are occupied); so the input rate to M, and accordingly the input rate to S, is $P\alpha(1+\phi(t))$. Hence, with t_0 the entrance time of the system in state *ME*,

$$y_S(t) = y_S(t_0) + P\alpha \int_{t_0}^t \phi(u)\,du = P\alpha \int_0^t \phi(u)\,du. \tag{5.2.4}$$

The last equality follows from (5.2.3) because, by definition,

$$y_M(t) = 0,$$

when the system is in state *ME*.

If the function ϕ is such that y_S grows, this may slow down M so much that M becomes saturated; the system will switch to state *MNE*. The epoch at which the system changes from state *ME* to state *MNE*, t_1, is determined by the condition

$$P\alpha(1+\phi(t_1)) = [a+b\ln(1+y_S(t_1))]^{-1} = [a+b\ln(1+P\alpha\int_0^{t_1}\phi(u)\,du)]^{-1}, \quad (5.2.5)$$

with t_1 the smallest solution larger than t_0.

The state MNE. Suppose that, at a time t_1, the system enters state *MNE*. The server in M is now continuously busy; the input rate at M still is $P\alpha(1+\phi(t))$, but its output rate - and the input rate to S - equals $[a+b\ln(1+y_S(t))]^{-1}$. The queue length process $y_S(t)$ (or rather its fluid flow approximation) evolves according to the following differential equation:

$$\frac{d}{dt}y_S(t) = -P\alpha + \frac{1}{a+b\ln(1+y_S(t))}, \quad t \geqslant t_1. \quad (5.2.6)$$

The initial condition is determined by (5.2.5):

$$y_S(t_1) = P\alpha\int_0^{t_1}\phi(u)\,du = \exp[\frac{1}{bP\alpha(1+\phi(t_1))} - \frac{a}{b}] - 1. \quad (5.2.7)$$

The differential equation (5.2.6) plays a central role in our analysis of the queueing effects of the parallel processing mechanism. Rewrite (5.2.6) into

$$\int dt = \int \frac{a+b\ln(1+y_S)}{1-P\alpha a - P\alpha b\ln(1+y_S)}\,dy_S,$$

or, with C_1 some yet unknown constant,

$$t + C_1 = -\frac{1}{P\alpha}y_S + \frac{1}{P\alpha}\int \frac{1}{1-P\alpha a - P\alpha b\ln(1+y_S)}\,dy_S. \quad (5.2.8)$$

Introduce

$$C := \frac{1}{b}[\frac{1}{P\alpha} - a], \quad (5.2.9)$$

and the exponential integral (cf. Abramowitz & Stegun [1965])

$$E_1(z) := \int_z^\infty \frac{\exp(-v)}{v}\,dv, \quad z>0. \quad (5.2.10)$$

Substitution of $v = C-\ln(1+y)$ in (5.2.10) shows that (5.2.8) can be rewritten into

$$t + C_1 = -\frac{1}{P\alpha}y_S(t) + \frac{1}{(P\alpha)^2 b}e^C E_1(C - \ln(1+y_S(t))). \quad (5.2.11)$$

The initial condition determines the constant C_1:

$$t_1 + C_1 = -\frac{1}{P\alpha}y_S(t_1) + \frac{1}{(P\alpha)^2 b}e^C E_1(C - \ln(1+y_S(t_1))). \quad (5.2.12)$$

Subtraction of the relations (5.2.11) and (5.2.12) finally gives us a relation between $y_S(t)$ and t:

$$t - t_1 = -\frac{1}{P\alpha}[y_S(t) - y_S(t_1)] + \tag{5.2.13}$$

$$\frac{1}{(P\alpha)^2 b} e^C [E_1(C - \ln(1+y_S(t))) - E_1(C - \ln(1+y_S(t_1)))].$$

It seems impossible to find an explicit expression for $y_S(t)$ as a function of t, $t \geqslant t_1$, but (5.2.13) is already very useful. Firstly, for each given value of $y_S(t)$ it is easy to explicitly calculate the corresponding t-value (the exponential integral E_1 is extensively tabulated [Abramowitz & Stegun 1965]). Secondly, standard knowledge about E_1 allows us to obtain useful insight into the behavior of $y_S(t)$.

It is clear from the differential equation (5.2.6) that, independently of the choice of ϕ, $y_S(t)$, $t \geqslant t_1$, increases as long as this differential equation holds, tending to the limit $\exp(C) - 1$. Let us now study the following question: at what time t_ϵ will $y_S(t) + 1$ reach the level $\exp(C(1-\epsilon))$? According to (5.2.13),

$$t_\epsilon - t_1 = -\frac{1}{P\alpha}[\exp(C(1-\epsilon)) - 1 - y_S(t_1)] + \tag{5.2.14}$$

$$\frac{1}{(P\alpha)^2 b} e^C [E_1(\epsilon C) - E_1(C - \ln(1+y_S(t_1)))].$$

Now we use the fact that (Abramowitz & Stegun [1965])

$$E_1(z) = -\gamma - \ln z - \sum_{n=1}^{\infty} \frac{(-1)^n z^n}{n n!}, \quad z > 0, \tag{5.2.15}$$

with $\gamma = 0.57721...$ denoting Euler's constant. Hence

$$E_1(\epsilon C) = -\gamma + \ln\frac{1}{\epsilon C} + O(\epsilon), \quad \epsilon \to 0, \tag{5.2.16}$$

so

$$t_\epsilon \approx \frac{1}{(P\alpha)^2 b} e^C (\ln\frac{1}{\epsilon} + O(1)), \quad \epsilon \to 0. \tag{5.2.17}$$

These calculations enable us to estimate the behavior of $y_S(t)$ close to its limiting value. In particular one can show that an $O(\epsilon)$ increase of $y_S(t)$ in this time region requires $O(1)$ time (one can, in fact, also derive this directly from the differential equation (5.2.6)). If $\phi(t) = 1$ in close approximation in a large time span in state *MNE*, $y_S(t) + y_M(t)$ grows linearly with $P\alpha$ customers per unit of time. Therefore, when $y_S(t)$ is close to its limiting value, the queue at M grows linearly with time in the time region under consideration.

The queue length process $y_M(t)$ follows from (5.2.3) once $y_S(t)$ has been determined. It depends on the choice of ϕ and of the various parameters whether a situation as sketched above (with the bulk of the growth of the customer population contributing to $y_M(t)$) actually occurs. See also the numerical

examples in Section 5.3.

For the system to switch back to state *ME*, it is required that *M*'s input rate $P\alpha(1+\phi(t))$ is less than its output rate $[a+b\ln(1+y_S(t))]^{-1}$ for some period of time. Let us suppose that ϕ and the various parameters are such that the system switches back to state *ME*. The epoch at which the system switches from state *MNE* to state *ME*, t_2, is determined by the condition $y_M(t_2)=0$, or equivalently:

$$y_S(t_2) = P\alpha \int_0^{t_2} \phi(u)\,du.$$

Substitution in (5.2.13) yields:

$$t_2 - t_1 = -\int_{t_1}^{t_2} \phi(u)\,du + \tag{5.2.18}$$

$$\frac{1}{(P\alpha)^2 b} e^C [E_1(C - \ln(1+P\alpha \int_0^{t_2} \phi(u)\,du)) - E_1(C - \ln(1 + P\alpha \int_0^{t_1} \phi(u)\,du))],$$

with t_2 the smallest solution, larger than t_1, of this equation. It has to be determined numerically.

5.2.2. Evolution of the queue length processes

We now restrict ourselves to the case of a non-increasing function ϕ with $\phi(0)=1$ and $\phi(\infty)=-1$. We follow the evolution of $y_M(t)$ and $y_S(t)$ from beginning to end.

Initially there is only one customer in the system (the root of the search tree). This customer is served in *M*, and subsequently in *S*; it is replaced by 2 new customers, who arrive at *M*; shortly thereafter there are 3 customers, etc. Very soon all processes of *S* are continuously busy. If, e.g., all service times at *S* are negative exponentially distributed with mean $1/\alpha$ and *M* is much faster than the *P* processes, the length of the initial period is approximately

$$\frac{1}{\alpha} + \frac{1}{2\alpha} + \ldots + \frac{1}{(P-1)\alpha}$$

(indeed, when j servers are active in *S*, the time until the first departure from *S* is negative exponentially distributed with mean $1/j\alpha$; the departing customer is almost certainly replaced by two other customers, who - after a very short visit to *M* - increase the number of active servers in *S* to $j+1$). After the initial period, $P\alpha$ customers leave *S* per unit of time (on the average), and $P\alpha(1+\phi(t))$ customers arrive at *M* per unit of time. *M* is extremely fast as long as the queue length at *S*, $y_S(t)$, is not too large: *M* has at first no difficulty handling its input stream, so its output stream also has intensity $P\alpha(1+\phi(t))$. In the fluid flow approach, *M* is still considered to be empty: the system is in state *ME*. $y_S(t)$ grows at a rate $P\alpha\phi(t)$, cf. (5.2.4). There are now two possibilities:

(i) *M* slows down so much that its maximal output rate equals its input rate: *M* starts to saturate, and the system enters state *MNE*;

(ii) M's speed is not reduced enough to reach the saturation point, and all customers are being processed without the system ever entering state *MNE*.

Case (i) obviously is the more interesting one. The system enters state *MNE*. The queue length process $y_S(t)$ now evolves according to the differential equation (5.2.6). M's queue length initially grows but, as a counteracting force, $\phi(t)$ decreases; finally, the input rate $P\alpha(1+\phi(t))$ in M becomes lower than the output rate and M's queue length starts to decrease. This process continues until M becomes empty again: the system switches back to state *ME*.

At this epoch, the input rate at S switches to $P\alpha(1+\phi(t))$. If ϕ has already become negative, the queue length at S immediately starts to decrease, and continues to do so (ϕ being a non-increasing function). Consequently M speeds up, and the system stays in state *ME* until there are no customers left. However, if ϕ still is positive, then in principle both possibilities (i) and (ii) discussed above again exist, and the system may switch back to *MNE*, etc. Such an alternating series of states *ME* and *MNE* may, for example, occur if shortly after entering state *MNE* the function ϕ drops from almost one to a small positive value and keeps this value for a substantial period. The system will react by a change from state *MNE* to state *ME*, and since the number of customers is still growing M will get saturated once more.

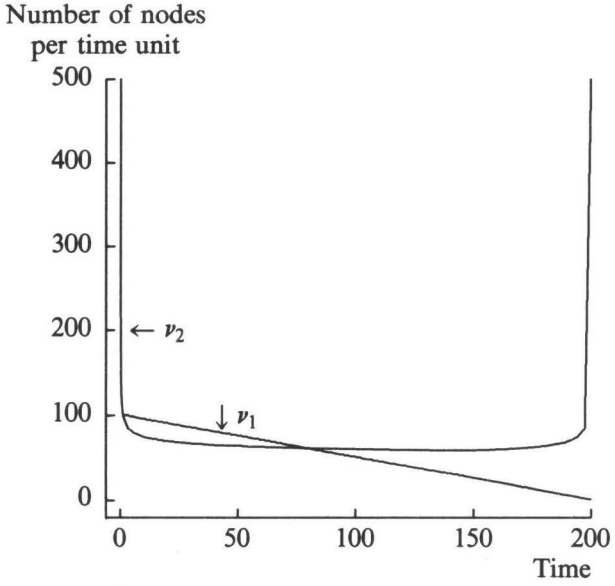


FIGURE 5.2. M's input rate ν_1 and service speed ν_2 ($\nu_1 = P\alpha(1+\phi(t))$ and $\nu_2 = [a+b\ln(1+y_S(t))]^{-1}$); $P\alpha = 50$, $a = b = 0.0020$, and ϕ is linearly decreasing.

Figure 5.2 depicts the typical behavior of M's input rate $P\alpha(1+\phi(t))$ and its service speed $[a+b\ln(1+y_S(t))]^{-1}$.

5.2.3. Reductions of the queue at S

Neither Figure 5.2, nor the global description of the queue length processes, considers the phenomenon that instantaneously part of the queue at S is thrown out of the system. This phenomenon, which also implies a sudden increase of M's speed, can easily be captured in the mathematical analysis. Suppose that a reduction of the queue at S occurs at an epoch t_d, and that x customers are removed from the network. If this happens while the system is in state ME, the output rate, $P\alpha(1+\phi(t_d))$, of M is not affected. Much more interesting is the situation in which the sudden drop in the queue length of S occurs while the system is in state MNE. Instantaneously the output rate of M increases to

$$[a+b\ln(1+y_S(t_d+))]^{-1} = [a+b\ln(1+y_S(t_d-)-x)]^{-1}.$$

The queue length at S still behaves according to the differential equation (5.2.6), but with a new initial value $y_S(t_d+)$. The speedup of M may soon lead to an empty queue at M, so that the system enters state ME. Of course, it is possible that *several* considerable reductions of the queue at S occur. Not much is known about the frequency with which this phenomenon occurs, nor about the sizes (x) of the corresponding jumps. Therefore we do not discuss the issue in much detail here. It suffices to observe that our model is able to determine the influence of sudden reductions of the queue at S on the speed of the master, and on the subsequent behavior of the queue sizes.

In Section 5.3 we present some numerical examples which, for various choices of the function ϕ and the parameters P, α, a and b, exhibit the global behavior of $y_S(t)$ and $y_M(t)$. In one example, the phenomenon of a reduction of the queue at S is also taken into account.

Remark. The function ϕ has so far been considered as a process independent function. In reality, ϕ may depend on the queue length process; it might in particular be realistic to decrease ϕ after the occurrence of a sudden reduction of the queue at S as described above (and this decrease should be related to the size of the reduction). Such process dependent behavior of ϕ can be incorporated in the model. The behavior of $y_S(t)$ would initially be still determined by the differential equation (5.2.6), but the input rate in M would suddenly decrease.

5.3. Numerical examples

To give a global idea of the behavior of $y_S(t)$ and $y_M(t)$, we will now present the results of some numerical computations. In all cases, we have considered a linearly decreasing function ϕ, with $\phi(0) = 1$. The process stops at a time T with $\phi(T) = -1$. The total number of customers served by the slaves at time T is $P\alpha T$. In all examples, we chose this number to be 10.000.

In Figure 5.3, the case $P\alpha = 50$ and $a = b = 0.0020$ is shown (see also Figure 5.2, and note that P and α are occurring as a product in all formulas). In

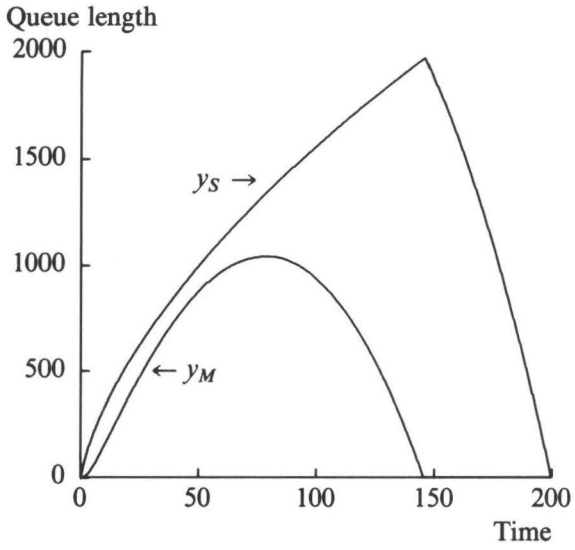


FIGURE 5.3. y_S and y_M for $P\alpha = 50$ and $a = b = 0.0020$.

the beginning, y_S is increasing very fast and M is getting saturated almost immediately. At that moment, the queue length y_M starts to grow. Since ϕ is a decreasing function, the number of customers arriving at M is decreasing. Therefore, M will eventually become empty and the system changes from state MNE to state ME. At this point in time, y_S starts to decrease since ϕ is already negative.

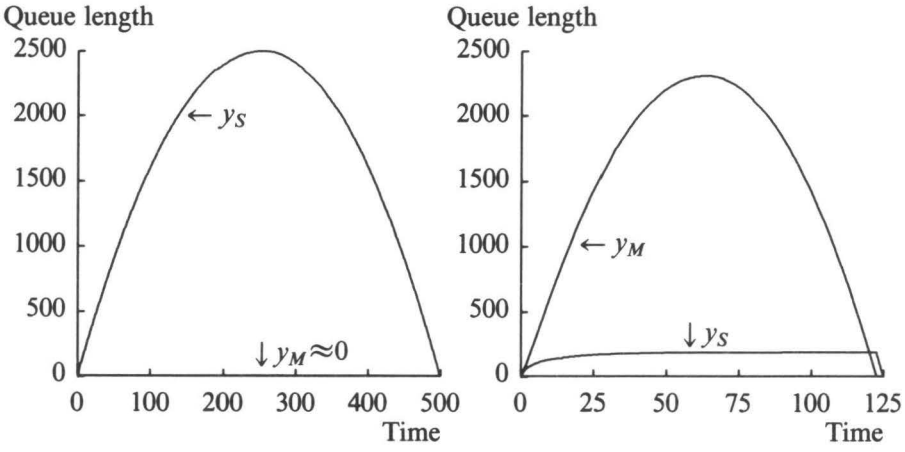


FIGURE 5.4. The effect of changing $P\alpha$; $a = b = 0.0020$, $P\alpha = 20$ (left), $P\alpha = 80$ (right).

Figure 5.4 shows the effect of changing $P\alpha$, which corresponds to altering the number of slaves or the processing speed of the slaves. For $P\alpha = 20$, the master is fast enough to serve the incoming customers and $y_M \approx 0$. If $P\alpha = 80$, the master gets into serious trouble. The speed of the master is much too slow compared with the number of incoming customers. Here, we can observe the fact that y_S is approaching an asymptotic value if the system is in state *MNE* for a long enough period.

There appears to be a delicate interaction between the processing capacities of the master and the slaves. Increasing the processing capacity of the slaves may change an almost continuously idle master into a saturated master with a very long queue. The beneficial effect of increasing the processing capacity of the slaves may now be reduced; for example, a node with information that would make a large part of the priority queue obsolete (i.e., a part of the queue at S would be thrown away), is delayed for a long time, thus possibly causing a deterioration of the running time of the algorithm.

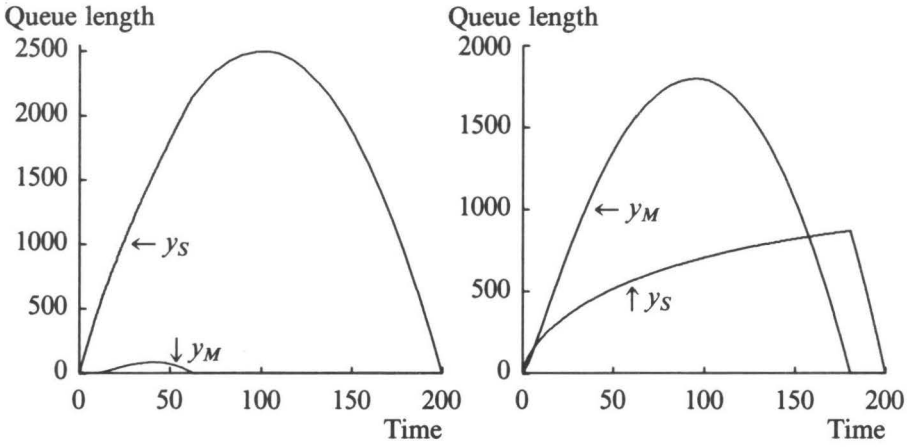


FIGURE 5.5. The effect of changing a and b; $P\alpha = 50$, $a = b = 0.0015$ (left), and $a = b = 0.0025$ (right).

In Figure 5.5, we consider different speeds of the master. The effects are about the same as when changing $P\alpha$.

Sudden reductions of the queue at S may cause an alternating sequence of the states *ME* and *MNE*. An example is given in Figure 5.6. In state *MNE*, a part of the queue at S is thrown away. As a consequence, M's speed increases so much that y_M becomes zero. Since the total number of customers in the system is still increasing rapidly, M gets saturated again, and the system enters state *MNE* again.

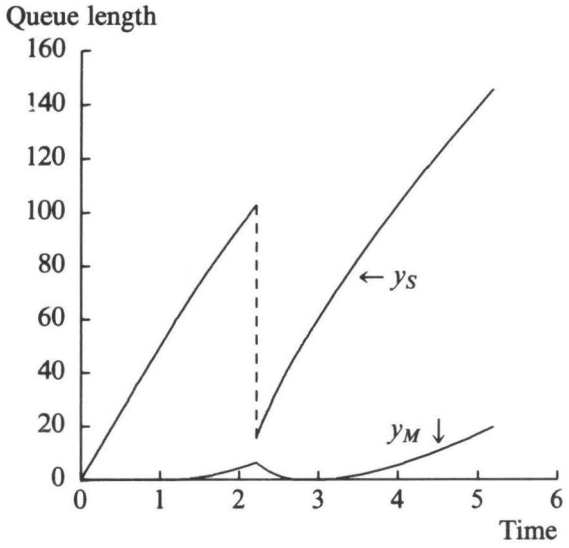


FIGURE 5.6. An example with a reduction of the queue at S; $P\alpha = 50$ and $a = b = 0.0020$.

5.4. THE MACHINE REPAIR MODEL

For the class of branch and bound algorithms considered in this chapter, it can be advantageous that the master has full knowledge of the search tree developed so far. An enormous queue length at the master can cause a slowdown of the computation. Therefore, we consider in this section branch and bound algorithms where a slave does not start with the evaluation of a new node until the master has processed the latest information the slave has sent.

This gives rise to the queueing model of Figure 5.7, with exactly P customers, each customer corresponding to one particular slave. This is a well known queueing model, often referred to as the machine repair model (the P customers being P machines which after breakdown have to be repaired in repair facility M). In a computer context, the model also represents a multiaccess system [Kobayashi 1978]. In such a case, the P slaves correspond to P terminal users. Each of these terminal users alternates between an active (think) phase and a passive phase; after a think phase, a job is sent to the central process M.

The machine repair model has been extensively studied in the queueing literature (see, for example, Gross & Harris [1985], Kobayashi [1978] and Tijms [1986]). Hardly any time-dependent results are known; however, under some distributional assumptions, quite simple explicit formulas for the *steady-state* queue length distribution at the repair facility, mean number of busy machines, etc., have been derived.

As in the previous model, it is assumed that the service times at the P servers $S_1,...,S_P$ of service station S are independent, identically distributed

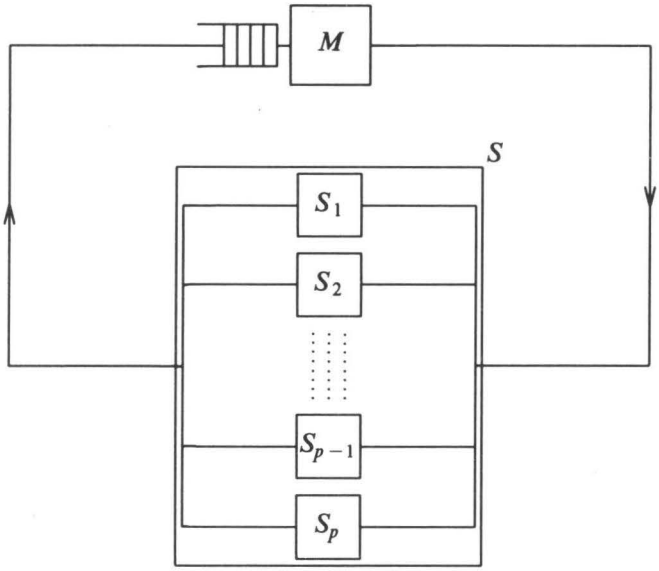


FIGURE 5.7. The machine repair model.

with mean $1/\alpha$. The assumptions concerning the service process in M differ from those in the previous model. For reasons of mathematical tractability, it is assumed that the service times at M are independent, negative exponentially distributed stochastic variables, with mean $1/\beta$. Note that the fluid flow approximation of Section 5.2 allowed us to leave the service time distribution at M unspecified. We return to this issue in the remark at the end of this section. The rate of the service times at the master is further assumed to be constant in time. However, the steady-state analysis for the case of constant master speed that we are about to present will already yield insight into the effect that a change in speed of the master has (see Figure 5.8 below).

It is easily seen that, under the assumption of negative exponentially distributed service times at M, S is equivalent - with respect to the number of busy servers - to the so-called M/G/P loss model. This is an open queueing model with a Poisson arrival process, P servers with generally distributed service times, and no waiting room; an arriving customer who finds all servers occupied is lost. Indeed, as long as the loss model and S contain less than P customers, their numbers of customers evolve in exactly the same way. When the loss model contains P customers, no more arrival is accepted until a customer has left; after an exponential period of time, a new arrival takes place. But exactly the same situation occurs in S, when it contains all P customers.

We restrict ourselves to the consideration of the limiting probability distribution of the number of busy servers, **B**, at S (which number equals P minus the number of customers in M). This amounts to studying the limiting distribution of the number of busy servers in the M/G/P loss model. This limiting

distribution, and hence the distribution of **B**, is given by (see, for example, Kelly [1979] or Tijms [1986]):

$$p_n := Pr\{\mathbf{B}=n\} = \frac{r^n/n!}{\sum_{j=0}^{P} r^j/j!}, \quad n=0,1,...,P, \tag{5.4.1}$$

with

$$r := \beta/\alpha.$$

The probability that an arriving customer in the M/G/P loss model is lost, $E_P(r)$, is given by Erlang's loss formula:

$$E_P(r) = p_P = \frac{r^P/P!}{\sum_{j=0}^{P} r^j/j!}. \tag{5.4.2}$$

The mean number of busy servers at S, N, follows from (5.4.1):

$$N := E[\mathbf{B}] = r[1 - E_P(r)]. \tag{5.4.3}$$

The relation between N and $E_P(r)$ is easily interpreted. Indeed, with r the amount of traffic offered to the M/G/P loss system per unit of time, N equals the mean amount of traffic handled per unit of time - and this should equal the mean number of busy servers. In this connection, note that αN represents the throughput of S, and hence also of M; so the mean cycle time of a job in the closed system is given by $P/\alpha N$.

In principle, (5.4.3) can be numerically evaluated. However, this evaluation may be cumbersome when P or r is large while, moreover, (5.4.2) does not yield much insight. Therefore, the behavior of N and $E_P(r)$ for large values of P and r has been extensively investigated. See Whitt [1984] for an interesting exposition and several early references, and see Newell [1984] for various asymptotic expansions. In particular, Newell presents a simple first-order approximation for $E_P(r)$ for $r\to\infty$, leading to

$$\begin{aligned} N &\approx r, \quad r \leqslant P, \\ N &\approx P, \quad r > P. \end{aligned} \tag{5.4.4}$$

Newell's second-order approximation (see also Whitt [1984]) leads to the following approximation for N. Introduce

$$\kappa := \frac{r}{\sqrt{P}}\left(\frac{P}{r} - 1\right),$$

and the standard normal distribution function

$$\Phi(x) := \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} \exp(-z^2/2)\, dz, \quad -\infty < x < \infty.$$

The mean number of busy servers in S is for large values of r approximated by:

$$N \approx r[1 - \frac{(r/P)^{P-1}e^{P-r}}{\sqrt{2\pi P}\,\Phi(\kappa P/r)}], \quad r \leqslant P,$$
$$N \approx r[1 - (\frac{P}{2\pi})^{1/2}\frac{\exp(-\kappa^2/2)}{r\Phi(\kappa)}], \quad r > P. \tag{5.4.5}$$

This approximation is based on Stirling's approximation for factorials, and the normal approximation to the Poisson distribution.

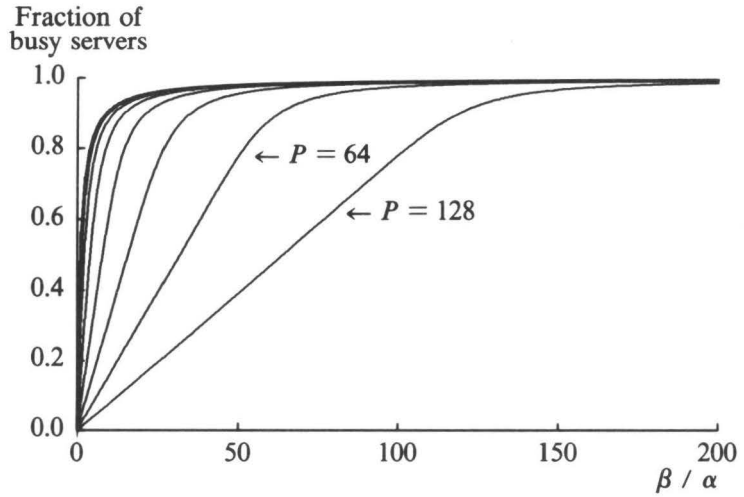


FIGURE 5.8. Fraction of busy servers as function of β/α for $P = 1, 2, 4, 8, 16, 32, 64, 128$.

Figure 5.8 displays the exact fraction of busy servers in S, N/P, as a function of $r = \beta/\alpha$ for $P = 1, 2, 4, 8, 16, 32, 64$, and 128. The figure clearly shows the usefulness of the simple first-order approximation (5.4.4). N grows linearly with β/α until the speed of the master M, β, almost equals $P\alpha$, the maximal speed of S; further increasing β has hardly any effect. The speed of the master varies with the number of generated but not yet examined nodes. The effect of such fluctuations in the speed of the master process on the fraction of busy servers can also be derived from Figure 5.8.

Figure 5.9 displays the fraction N/P as a function of $r/P = \beta/P\alpha$, for the same parameter choices as in Figure 5.8. The figure shows that, for $P > r$ $(\beta/P\alpha < 1)$, the fraction of busy servers decreases rapidly, when $\beta/P\alpha$ decreases. For fixed speeds of the master and the slaves, it is, therefore, only worthwhile to add slave processes as long as $P < r$ $(\beta/P\alpha > 1)$.

So far, we have been mainly concerned with the *mean* of the number of busy servers in S. Newell [1984] also presents approximations for the *distribution* of the number of busy servers in S. He states that, for P large and fixed, and

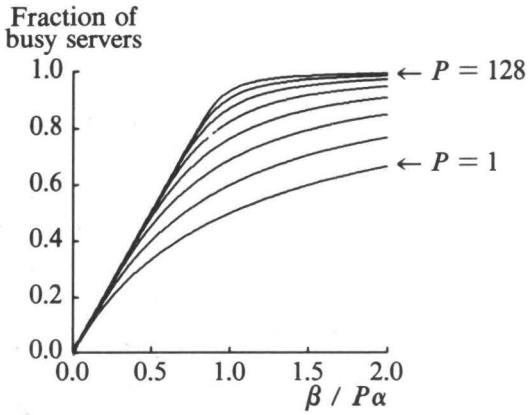


FIGURE 5.9. Fraction of busy servers as function of $\beta/P\alpha$ for $P = 1, 2, 4, 8, 16, 32, 64, 128$.

$r > P$ and in particular $1 - P/r >> r^{-1/2}$, the distribution of idle servers in S is approximately geometric:

$$Pr\{n \text{ idle servers}\} = p_{P-n} = (1 - P/r)(P/r)^n, \quad n = 0,1,...,P, \qquad (5.4.6)$$

with the mean number of idle servers in S approximately equal to $P/(r-P)$.

Remarks.

In this section the service times at S are generally distributed, whereas the service times at M are exponentially distributed. It is an interesting, and well-known, fact for the machine repair model (and the M/G/P loss model) that Formula (5.4.1) for the number of customers at S holds regardless of the form of the service time distribution at S. When M uses a processor-sharing discipline, (5.4.1) even holds when also the service times at M have a general distribution with mean $1/\beta$ (cf. Tijms [1986]). For the first-come-first-serve discipline at M under consideration, this *insensitivity* for the service time distribution at M is not true.

Formula (5.4.1) can be easily generalized to the case that the mean service time in M depends on the number of customers waiting in M, or equivalently, that the arrival rate at the M/G/P loss system depends on the number of busy servers. Let β_n denote the service speed in M when n customers are present in M. Then (5.4.1) should be replaced by

$$p_n := Pr\{\mathbf{B}=n\} = \frac{\prod_{k=1}^{n}(\beta_{P-k+1}/k\alpha)}{\sum_{j=0}^{P}\prod_{k=1}^{j}(\beta_{P-k+1}/k\alpha)}, \quad n=0,1,...,P. \qquad (5.4.7)$$

6

Perspectives

In the previous chapters, we have discussed the influence of parallel computing on combinatorial operations research. Parallel computers enable us to solve problem instances much faster than before, and make it possible to find the solution to instances which are beyond the capabilities of traditional sequential machines. The diversity in available architectures, however, forms a barrier for a broad use of the potential of parallelism.

From a theoretical point of view, the existing models for parallel computation lead to the identification of new complexity classes, but they do not provide a practical understanding of what can and what cannot be achieved, since the gap between theoretical models and available architectures is considerable.

In the following, we try to sketch the conditions that have to be met before parallelism can fulfill its promise and substantially expand the range of effectiveness of operations research methods.

6.1. Computational models

Traditional sequential computers are reasonably represented by models of computation such as the Turing machine. They are exchangeable to the extent that the relative efficiency of algorithms is largely machine independent. These observations form the basis of a meaningful complexity theory and a prospering computational practice.

In parallel computing, realistic models are lacking and existing machines are by no means equivalent. We have seen that there exist many different parallel architectures, which are suitable for very different types of algorithms. Not surprisingly, then, there is no single model of parallel computation that serves to represent reality. Indeed, a model that adequately reflects the actual burden of parallel computation and communication has to incorporate physical features of the computational environment that can be ignored in the

sequential case. Although a model like the PRAM, with its ability to create unbounded parallelism and to communicate in unit time, is not very realistic, it tells us a lot about the intrinsic parallelism in problems and algorithms.

One may try to cope with the idealistic nature of theoretical models of parallel computation by designing transformations to models that are closer to what we can expect in practice. Examples are the simulations of the PRAM by a network in which each processor has some constant number of connections [Alt, Hagerup, Mehlhorn & Preparata 1987; Karlin & Upfal 1986] (see also Section 1.2), and of big networks by smaller ones [Bodlaender 1987].

What is actually needed, however, is the investigation of severe restrictions on parallelism and communication. As a notable example, a robust theory for models with at most a linear number of processors that communicate over a bounded degree network would serve a very practical purpose.

6.2. Architectures

The main obstacle for the breakthrough of parallel computing is not the lack of reasonable models but the chaos in the real world of architectures. There is a broad variety of vastly different machines, each having its individual strengths and weaknesses. This has two important consequences. For a given algorithm, its implementation in parallel is highly dependent on the architecture in question. And for a given problem, the suitability of an algorithm is a function of the machine to be used.

As to the first point, a consensus will hopefully emerge on a single concept of a flexible MIMD computer. Given such a machine, the user should be able to define the type of parallelism he desires, by specifying the hierarchy and communication among his computational processes. The best way to develop this machine is by first building it in software and analyzing its performance (see, for example, De Bruin, Rinnooy, Kan & Trienekens [1988]); its hardware realization should be considered at a later stage. Such a unified architecture requires a flexible set of tools and, in particular, a versatile programming language which (unlike present practice in parallel programming) does not bother the user with the internal structure of the machine. While vector and SIMD computers will probably lose ground as independent machines, they will always be useful as processors that speed up the individual processes in an MIMD configuration.

As long as this ideal has not yet been reached, the second point deserves further investigation. Instead of considering the existing types of algorithms as given and designing a machine that is suitable for all of them, one might accept an existing machine as given and try to design algorithms for which it is particularly efficient. This may lead to new types of algorithms, but also to valuable insights into desirable properties of parallel architectures.

6.3. Computations

Parallel computers have provided a new playground for computational operations research. All kinds of algorithms have been implemented and tested on the parallel devices that happened to be available. The potential power of parallel computers is huge, but the effort required to harness this power should not be underestimated either. One will have to master new programming techniques, and one will have to get used to new algorithmic concepts that may fundamentally change the way problems are solved. For a simple problem like sorting, for example, there exist parallel algorithms that are drastically different from the traditional sequential sorting routines [Ajtai, Komlós & Szemerédi 1983] (see also Section 2.2.3).

A striking phenomenon in this respect is the use of randomization in combination with parallelism. Parallel algorithms that are able to toss a coin in order to decide how to proceed appear to be successful on an ever-increasing list of problems. This list includes a standard problem like matching [Karp, Upfal & Wigderson 1986; Mulmuley, Vazirani & Vazirani 1987] (see also Section 2.2.6), questions concerning parallel architectures themselves such as the communication between processing elements [Valiant 1982b; Valiant & Brebner 1981], but also problems in operations research such as the design of search strategies in branch and bound [Karp & Zhang 1988]. It seems safe to predict that this line of investigation will have a great impact on computational operations research.

Parallelism will enable us to solve problems faster and to solve larger problems than before (Beasley [1987], Mangasarian & Meyer [1988], Meyer & Zenios [1988] and Zenios [1989] mention some of the pioneering contributions in this field). This is especially promising for the development of real-time systems, that occur, for example, in the context of flexible manufacturing and interactive planning.

However, a healthy computational practice needs a sound theoretical basis. There is, in particular, a need for a theoretical approach towards the design and analysis of parallel algorithms for the broad class of hard combinatorial and nonlinear optimization problems. Among the few things that have been done in this respect are the investigation of anomalies in parallel branch and bound (cf. Chapter 4) and the analysis of parallel tree search in a master-slave environment (cf. Chapter 5).

For all the various kinds of search methods, the fundamental question is how the computational effort has to be distributed over the processors and how the communication has to be arranged so as to obtain a maximum speedup. Operations researchers are well positioned to model and solve this complicated design problem. Initial steps in this direction are being made; we mention the investigation of randomized search strategies as an example. In the words of Richard M. Karp [Frenkel 1986]: 'Even though you may never be able to go from exponential to polynomial, it's also clear that there is a tremendous scope for parallelism on those hard problems, and parallelism may really help us curb combinatorial explosions.'

References

M. ABRAMOWITZ, I.A. STEGUN (1965). *Handbook of Mathematical Functions*, Dover, New York.

W.B. ACKERMAN (1982). Data flow languages. *IEEE Computer 15(2)*, 15-25.

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

H. ALT, T. HAGERUP, K. MEHLHORN, F.P. PREPARATA (1987). Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM J. Comput. 16*, 808-835.

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

G.H. BARNES, R.M. BROWN, M. KATO, D.J. KUCK, D.L. SLOTNICK, R.A. STOKES (1968). The Illiac IV computer. *IEEE Trans. Comput. C-17*, 746-757.

J.E. BEASLEY (1987). Supercomputers and OR. *J. Oper. Res. Soc. 38*, 1085-1089.

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

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

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

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

H.A. BODLAENDER (1987). *Distributed Computing: Structure and Complexity*, CWI Tract 43, Centre for Mathematics and Computer Science, Amsterdam.

O.J. BOXMA, G.A.P. KINDERVATER (1987). *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.

F.W. BURTON, M.M. HUNTBACH, G.P. MCKEOWN, V.J. RAYWARD-SMITH (1983). *Parallelism in Branch-and-Bound Algorithms*, Report CSA/3/1983, University of East Anglia, Norwich.

CDC (1983). *FORTRAN 200 Version 1*, Reference Manual 60480200, CDC Sunnyvale, CA.

A.K. CHANDRA, D.C. KOZEN, L.J. STOCKMEYER (1981). Alternation. *J. Assoc. Comput. Mach. 28*, 114-133.

F.Y. CHIN, J. LAM, I-N. CHEN (1982). Efficient parallel algorithms for some graph problems. *Comm. ACM 25*, 659-665.

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

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

D. COPPERSMITH, S. WINOGRAD (1987). Matrix multiplication via arithmetic progressions. *Proc. 19th Annual ACM Symp. Theory of Computing*, 1-6.

L. CSANKY (1976). Fast parallel matrix inversion algorithms. *SIAM J. Comput. 5*, 616-623.

J. DARLINGTON, M. REEVE (1981). ALICE - a multi-processor reduction machine for the parallel evaluation of applicative languages. *ACM Proc. 1981 Conf. Funct. Progr. Lang. and Comput. Architecture*, 65-75.

A. DE BRUIN, A.H.G. RINNOOY KAN, H.W.J.M. TRIENEKENS (1988). A simulation tool for the performance evaluation of parallel branch and bound algorithms. *Math. Programming Ser. B 42*, 245-271.

E. DEKEL, D. NASSIMI, S. SAHNI (1981). Parallel matrix and graph algorithms. *SIAM J. Comput. 10*, 657-675.

E. DEKEL, S. SAHNI (1983a). Binary trees and parallel scheduling algorithms. *IEEE Trans. Comput. C-32*, 307-315.

E. DEKEL, S. SAHNI (1983b). Parallel scheduling algorithms. *Oper. Res. 31*, 24-49.

P. DI CHIO, V. ZECCA (1985). *IBM ECSEC Facilities: User's Guide*, Report G513-4080, IBM European Center for Scientific and Engineering Computing, Rome.

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

D. DOBKIN, R.J. LIPTON, S. REISS (1979). Linear programming is log-space hard for *P. Inform. Process. Lett. 8*, 96-97.

J.J. DONGARRA, I.S. DUFF (1985). *Advanced Architecture Computers*, Technical memorandum 57, Mathematics and Computer Science Division, Argonne National Laboratory.

J. EDMONDS, R.M. KARP (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *J. Assoc. Comput. Mach. 19*, 248-264.

R. FINKEL, U. MANBER (1987). DIB - a distributed implementation of backtracking. *ACM Trans. Programming Languages and Systems 9*, 235-256.

R.W. FLOYD (1962). Algorithm 97: shortest path. *Comm. ACM 5*, 345.

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

S. FORTUNE, J. WYLLIE (1978). Parallelism in random access machines. *Proc. 10th Annual ACM Symp. Theory of Computing*, 114-118.

B.L. FOX, J.K. LENSTRA, A.H.G. RINNOOY KAN, L.E. SCHRAGE (1978). Branching from the largest upper bound: folklore and facts. *European J. Oper. Res. 2*, 191-194.

K.A. FRENKEL (1986). Complexity and parallel processing: an interview with Richard Karp. *Comm. ACM 19*, 112-117.

T.J. GARDNER, I.M. GERARD, C.R. MOWERS, E. NEMETH, R.B. SCHNABEL (1986). *DPUP: a Distributed Processing Utilities Package*, Report CU-CS-337-86, University of Colorado, Boulder.

M.R. GAREY, D.S. JOHNSON (1979). *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, San Francisco.

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

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

L.M. GOLDSCHLAGER, R.A. SHAW, J. STAPLES (1982). The maximum flow problem is log space complete for P. *Theoret. Comput. Sci. 21*, 105-111.

M. GONDRAN, M. MINOUX (1984). *Graphs and Algorithms*, Wiley, Chichester.

T. GONZALEZ, S. SAHNI (1978). Preemptive scheduling of uniform processor systems. *J. Assoc. Comput. Mach. 25*, 92-101.

R.L. GRAHAM (1969). Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math. 17*, 418-429.

A.C. GREENBERG, R.E. LADNER, M.S. PATERSON, Z. GALIL (1982). Efficient parallel algorithms for linear recurrence computation. *Inform. Process. Lett. 15*, 31-35.

D. GROSS, C.M. HARRIS (1985). *Fundamentals of Queueing Theory*, Wiley, New York (2nd ed.).

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

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

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

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

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

ICL (1979). *DAP: FORTRAN Language*, Technical Publication 6918, ICL, London.

ICL (1981). *DAP: Developing DAP Programs*, Technical Publication 6920, ICL, London.

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

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

D.S. JOHNSON, C.H. PAPADIMITRIOU, M. YANNAKAKIS (1985). How easy is local search? (extended abstract). *Proc. 26th Annual IEEE Symp. Foundations of Computer Science*, 39-42.

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

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

R.M. KARP, Y. ZHANG (1988). A randomized parallel branch-and-bound procedure. *Proc. 20th Annual ACM Symp. Theory of Computing*, 290-301.

F.P. KELLY (1979). *Reversibility and Stochastic Networks*, Wiley, Chichester.

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

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

G.A.P. KINDERVATER, J.K. LENSTRA (1988). Parallel computing in combinatorial optimization. *Ann. Oper. Res. 14*, 245-289.

G.A.P. KINDERVATER, J.K. LENSTRA, A.H.G. RINNOOY KAN (1989). Perspectives on parallel computing. *Oper. Res.*, to appear.

G.A.P. KINDERVATER, J.K. LENSTRA, M.W.P. SAVELSBERGH (1989). *Complexity, Parallelism, and Interaction: the Traveling Salesman Revisited*, in preparation.

G.A.P. KINDERVATER, J.K. LENSTRA, D.B. SHMOYS (1989). The parallel complexity of TSP heuristics. *J. Algorithms*, to appear.

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

H. KOBAYASHI (1978). *Modeling and Analysis*, Addison-Wesley, Reading, MA.

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

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

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

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

T.-H. LAI, A. SPRAGUE (1986). A note on anomalies in parallel branch-and-bound algorithms with one-to-one bounding functions. *Inform. Process. Lett. 23*, 119-122.

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

E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN, D.B. SHMOYS (eds.) (1985). *The Traveling Salesman Problem: a Guided Tour of Combinatorial Optimization*, Wiley, Chichester.

G.-J. LI, B.W. WAH (1986). Coping with anomalies in parallel branch-and-

bound algorithms. *IEEE Trans. Comput. C-35*, 568-573.

L. LOVÁSZ (1979). Determinants, matchings and random algorithms. L. BUDACH (ed.). *Fundamentals of Computing theory, FCT '79*, Akademie Verlag, Berlin, 565-574.

O.L. MANGASARIAN, R.R. MEYER (eds.) (1988). *Parallel Methods in Mathematical Programming; Math. Programming Ser. B 42.*

C.U. MARTEL (1988). A parallel algorithm for preemptive scheduling of uniform machines. *J. Parallel Distributed Comput. 5*, 700-715.

W.A. MASSEY (1985). Asymptotic analysis of the time dependent M/M/1 queue. *Math. Oper. Res. 10*, 305-327.

J. MCGRAW, S. SKEDZIELEWSKI, S. ALLAN, D. GRIT, R. OLDEHOEFT, J. GLAUERT, C. KIRKHAM, B. NOYCE (1984). *SISAL: Streams and Iteration in a Single Assignment Language*, Language Reference Manual Version 1.2, Lawrence Livermore National Laboratory, Livermore, CA.

R. MCNAUGHTON (1959). Scheduling with deadlines and loss functions. *Management Sci. 6*, 1-12.

R.R. MEYER, S.A. ZENIOS (eds.) (1988). *Parallel Optimization on Novel Computer Architectures; Ann. Oper. Res. 14.*

D.E. MULLER, F.P. PREPARATA (1975). Bounds to complexities of networks for sorting and for switching. *J. Assoc. Comput. Mach. 22*, 195-201.

K. MULMULEY, U.V. VAZIRANI, V.V. VAZIRANI (1987). Matching is as easy as matrix inversion. *Combinatorica 7*, 105-113.

J.F. MUTH, G.L. THOMPSON (eds.) (1963). *Industrial Scheduling*, Prentice Hall, Englewood Cliffs, NJ, 237.

G.F. NEWELL (1971). *Applications of Queueing Theory*, Chapman and Hall, London.

G.F. NEWELL (1984). *The $M/M/\infty$ Service System with Ranked Servers in Heavy Traffic*, Springer, Berlin.

J.F. PEKNY, D.L. MILLER (1988). *A Parallel Branch and Bound Algorithm for Solving Large Asymmetric Traveling Salesman Problems*, Working paper, Carnegie Mellon University, Pittsburgh.

G. POLYA, R.E. TARJAN, D.R. WOODS (1983). *Notes on Introductory Combinatorics*, Birkhäuser, Boston.

F.P. PREPARATA, J. VUILLEMIN (1981). The cube-connected cycles: a versatile network for parallel computation. *Comm. ACM 24*, 300-309.

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

E.A. PRUUL (1975). *Parallel Processing and a Branch-and-Bound Algorithm*, M.Sc. thesis, Cornell University, Ithaca, NY.

E.A. PRUUL, G.L. NEMHAUSER, R.A. RUSHMEIER (1988). Branch-and-bound and parallel computation: a historical note. *Oper. Res. Lett. 7*, 65-69.

K.L. RIDER (1976). A simple approximation to the average queue size in the time-dependent M/M/1 queue. *J. Assoc. Comput. Mach. 23*, 361-367.

M.H. ROTHKOPF, S.S. OREN (1979). A closure approximation for the nonstationary M/M/s queue. *Management Sci. 25*, 522-534.

C. SAVAGE (1977). *Parallel Algorithms for Graph Theoretical Problems*, Ph.D.

Thesis, University of Illinois, Urbana-Champaign.

C. SAVAGE, J. JA'JA' (1981). Fast, efficient parallel algorithms for some graph problems. *SIAM J. Comput. 10*, 682-691.

M.W.P. SAVELSBERGH (1988). *Computer Aided Routing*, Ph.D. Thesis, Centre for Mathematics and Computer Science, Amsterdam.

J.T. SCHWARTZ (1980). Ultracomputers. *ACM Trans. Programming Languages and Systems 2*, 484-521.

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

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

J.S. SQUIRE, S.M. PALAIS (1963). Programming and design considerations of a highly parallel computer. *Proc. AFIPS Spring Joint Computer Conf. 23*, 395-400.

H.S. STONE (1971). Parallel processing with the perfect shuffle. *IEEE Trans. Comput. C-20*, 153-161.

THINKING MACHINES CORPORATION (1986). *Parallel Instruction Set (PARIS)*, Document Number 1-0002-2-7, Thinking Machines Corporation, Cambridge.

H.C. TIJMS (1986). *Stochastic Modeling and Analysis: a Computational Approach*, Wiley, Chichester.

P.C. TRELEAVEN, D.R. BROWNBRIDGE, R.P. HOPKINS (1982). Data-driven and demand-driven computer architecture. *Comput. Surveys 14*, 93-143.

H.W.J.M. TRIENEKENS (1989a). *Parallel Branch and Bound and Anomalies*, Report EUR-CS-89-1, Department of Computer Science, Erasmus University, Rotterdam.

H.W.J.M. TRIENEKENS (1989b). *Computational Experiments with an Asynchronous Parallel Branch and Bound Algorithm*, Report EUR-CS-89-2, Department of Computer Science, Erasmus University, Rotterdam.

S.H. UNGER (1958). A computer oriented toward spatial problems. *Proc. IRE 46*, 1744-1750.

E. UPFAL (1984). A probabilistic relation between desirable and feasible models of parallel computation (preliminary version). *Proc. 16th annual ACM Symp. Theory of Computing*, 258-265.

L.G. VALIANT (1982a). Reducibility by algebraic projections. *Enseign. Math. (2) 28*, 253-268.

L.G. VALIANT (1982b). A scheme for fast parallel communication. *SIAM J. Comput. 11*, 350-361.

L.G. VALIANT, G.J. BREBNER (1981). Universal schemes for parallel communication. *Proc. 13th Annual ACM Symp. Theory of Computing*, 263-277.

P. VAN EMDE BOAS (1985). The second machine class: models of parallelism. J. VAN LEEUWEN, J.K. LENSTRA (eds.). *Parallel Computers and Computations*, CWI Syllabus 9, Centre for Mathematics and Computer Science, Amsterdam, 133-161.

S. WARSHALL (1962). A theorem on boolean matrices. *J. Assoc. Comput. Mach. 9*, 11-12.

I. WATSON (1984). The dataflow approach - architecture and performance. F.B. CHAMBERS, D.A. DUCE, G.P. JONES (eds.). *Distributed Computing*, Academic Press, London, Ch. 2.

W. WHITT (1984). Heavy-traffic approximations for service systems with blocking. *AT&T Bell Labs. Techn. J. 63*, 689-708.

S.A. ZENIOS (1989). Parallel numerical optimization: current status and an annotated bibliography. *ORSA J. Comput. 1*, 20-43.

Samenvatting

Parallellisme en combinatoriek

In het overgrote deel van de huidige generatie computers is er één centrale processor die het rekenwerk stap voor stap uitvoert. Ondanks de soms enorme verwerkingssnelheid van deze sequentiële computers zijn er nog veel problemen die in de praktijk niet of niet snel genoeg kunnen worden opgelost. Voor deze problemen zijn vaak aanzienlijk snellere computers nodig. Aangezien de rekensnelheid van een processor niet onbeperkt te vergroten is, worden er computers ontwikkeld waarin verscheidene processoren tegelijkertijd aan de oplossing van een probleem werken. In theorie kunnen op deze manier willekeurig hoge verwerkingssnelheden worden gerealiseerd.

Parallelle computers zijn heden ten dage nog niet in staat veel grotere problemen op te lossen dan we met behulp van traditionele sequentiële machines al kunnen, maar verwacht mag worden dat daar langzaam verandering in komt. Dit proefschrift gaat in op een aantal aspecten van parallel rekenen die van invloed kunnen zijn op de theorie en de praktijk van de combinatorische optimalisering in de nabije toekomst.

Sequentiële computers zijn ruwweg equivalent als we hun rekensnelheden verdisconteren. Bij parallelle computers is dit helaas niet het geval. In hoofdstuk 1 worden drie veel gebruikte *classificatiemethoden voor parallelle computers* besproken. De classificaties zijn gebaseerd op de mate van zelfstandigheid van de processoren, de wijze waarop de processoren gegevens met elkaar uitwisselen, en de manier waarop een berekening uitgevoerd wordt. Tevens wordt een aantal theoretische modellen voor parallelle computers beschreven.

Wat we met parallellisme kunnen bereiken, hangt niet alleen van de hardware af. Het is ook van belang dat het op te lossen probleem in een voldoende aantal onafhankelijke stukken kan worden verdeeld. In hoofdstuk 2 komt daarom de *complexiteitstheorie* voor parallelle berekeningen aan de orde.

Centraal staat het *parallelle-berekeningstheorema*, dat zegt dat de benodigde werkruimte op een sequentiële computer ongeveer hetzelfde is als de benodigde tijd op een parallelle machine. Voor het theoretische 'Parallelle Random Access Machine' (PRAM) model houdt dit het volgende in. De problemen in $\mathcal{P}$-SPACE (de klasse van problemen die door een sequentiële machine in *polynomiale werkruimte* kunnen worden opgelost) zijn op een PRAM in *polynomiale tijd* oplosbaar. Binnen de klasse $\mathcal{P}$ van problemen die door een sequentiële computer in polynomiale tijd kunnen worden opgelost, ontstaat een tweedeling. Veel problemen blijken oplosbaar in *polylogaritmische werkruimte* door een sequentiële machine en daarmee in *polylogaritmische tijd* op een parallelle computer. Van andere problemen kunnen we aantonen dat ze *$\mathcal{P}$-volledig* zijn, en hun sequentiële oplosbaarheid in polylogaritmische ruimte (d.w.z. hun parallelle oplosbaarheid in polylogaritmische tijd) is daarmee zeer onwaarschijnlijk.

Het PRAM model is aantrekkelijk omdat het precies de grenzen aangeeft van wat met parallellisme mogelijk is. In de praktijk is het echter niet realiseerbaar. Algoritmen ontworpen voor de PRAM zullen voor ze op een parallelle architectuur uitgevoerd kunnen worden meestal behoorlijk aangepast moeten worden. De hoofstukken 3 en 4 gaan over de *implementatie* van een aantal standaardtechnieken uit de combinatorische optimalisering, zoals *dynamische programmering*, *'divide and conquer'* en *'branch and bound'*, op bestaande architecturen. In hoofdstuk 3 komen *fijnkorrelige* architecturen aan de orde waarin de processoren zeer begrensde rekenmogelijkheden hebben maar snel met elkaar kunnen communiceren. Hoofdstuk 4 behandelt de implementatie van branch and bound op *grofkorrelige* architecturen waarin de processoren volwaardige sequentiele computers zijn en communicatie relatief duur is. Bij branch and bound blijkt zich het verschijnsel voor te doen waarbij het toevoegen van een processor de berekening vertraagt of meer dan proportioneel versnelt. Hierop wordt nader ingegaan.

Aangezien branch and bound een veel gebruikte methode is, wordt in hoofdstuk 5 een eerste aanzet gegeven voor een theoretische beschrijving en analyse van zo'n algoritme in een *master-slave* omgeving. Het model is gebaseerd op een netwerk van wachtrijen.

Een echte doorbraak van parallel rekenen is er nog niet. Oorzaken hiervoor zijn ondermeer de grote verscheidenheid in beschikbare architecturen en de kloof tussen theoretische modellen en praktisch realiseerbare machines. Bovendien is er duidelijk behoefte aan een realistisch formeel model voor de ontwikkeling en de implementatie van parallelle algoritmen voor 'lastige' combinatorische problemen. Dit soort vragen komt aan de orde in hoofdstuk 6.

Parallel rekenen kan in de toekomst van grote betekenis worden. Op dit moment is de situatie echter nogal chaotisch. In dit proefschrift beschrijven we de huidige situatie en geven enige aanbevelingen voor wat vanuit een combinatorisch oogpunt wenselijk zou zijn.

STELLINGEN

behorende bij het proefschrift van

GERARDUS ANTONIUS PETRUS KINDERVATER

EXERCISES IN PARALLEL COMBINATORIAL COMPUTING

I

$10 \times 10 = 930$.

E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN, D.B. SHMOYS (1990). Sequencing and scheduling: algorithms and complexity. S.C. GRAVES, A.H.G. RINNOOY KAN, P. ZIPKIN (eds.). *Handbooks in Operations Research and Computer Science, Vol. 4: Logistics of Production and Inventory*, North-Holland, Amsterdam.

II

Polynomiale equivalentie van problemen betekent niet dat ze in de praktijk even snel oplosbaar zijn.

III

Gegeven zijn een instantie van het lineaire-toewijzingsprobleem en een optimale oplossing hiervoor. Dan zijn de problemen die ontstaan door als extra voorwaarde toe te voegen dat een bepaalde kant tot de oplossing moet behoren tegelijkertijd met een kortste-padalgoritme op te lossen.

G. KINDERVATER, A. VOLGENANT, G. DE LEVE, V. VAN GIJLSWIJK (1985). On dual solutions of the linear assignment problem. *European Journal of Operational Research 19*, 76-81.

IV

Handelsreizigersalgoritmen die ondergrenzen met behulp van polyhedrale methoden berekenen zijn superieur aan alle andere tot nu toe ontworpen optimaliseringsalgoritmen voor dit probleem.

E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN, D.B. SHMOYS (eds.) (1985). *The Traveling Salesman Problem: a Guided Tour of Combinatorial Optimization*, Wiley, Chichester.

V

'Simplexmethode' is een passende naam voor Karmarkars algoritme voor lineaire programmering.

N. KARMARKAR (1984). A new polynomial-time algorithm for linear programming. *Combinatorica 4*, 373-395.

VI

Voor De Bono's L-spel geldt dat de beginstand remise is en dat de langst mogelijke gedwongen winstvoering vanuit een willekeurige stand vijf zetten van de winnende partij vergt.

V.W. VAN GIJLSWIJK, G.A.P. KINDERVATER, G.J. VAN TUBERGEN, J.J.O.O. WIEGERINCK (1976). *Computer Analysis of E. de Bono's L-game*, Rapport 76-18, Mathematisch Instituut, Universiteit van Amsterdam.

VII

Het optreden van anomalieën bij parallelle implementaties van branch and bound methoden heeft implicaties voor het ontwerpen van zoekstrategieën bij sequentiële implementaties voor dergelijke methoden.

E.A. PRUUL, G.L. NEMHAUSER, R.A. RUSHMEIER (1988). Branch-and-bound and parallel computation: a historical note. *Oper. Res. Lett. 7*, 65-69.
G.A.P. KINDERVATER (1989). Dit proefschrift, Hoofdstuk 4.

VIII

De meeste parallelle algoritmen vertonen op een systeem met p processoren een versnelling kleiner dan p maar groter dan één. Dit is geen anomaal verschijnsel.

G.-J. LI, B.W. WAH (1986). Coping with anomalies in parallel branch-and-bound algorithms. *IEEE Trans. Comput. C-35*, 568-573.

IX

De baten van parallelle computers wegen op tegen de kosten.

X

Hardware-ontwikkelingen maken het zoeken naar betere algoritmen niet overbodig.