CWI Syllabi

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The CWI is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

CWI Syllabus 9

Parallel computers and computations

edited by
J. van Leeuwen & J.K. Lenstra

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

1980 Mathematics Subject Classification: 68A05, 68B20, 68Cxx, 68Exx, 65Fxx.
1983 CR Classification Scheme: C.1.2, D.1.3, D.4.7, F.1.0, F.2.1, F.2.2.
ISBN 90 6196 297 8

Printed in the Netherlands

Preface

In the realm of scientific computing there is an increasing demand for high-performance computers in applications that involve the solution of very large systems or the processing of complex and massive control information and data under strict timing constraints. New advances in computer technology and design have led to the availability of several very fast special purpose processors and 'supercomputers' that are now being installed at an increasing number of industrial facilities and computation centers. Several prototypes of even faster special purpose and general purpose computers are being designed and tested in a number of research centers over the world.

The advanced architectures of this new breed of computers are all centered around the concept of parallel processing. Vector computers, array processors and multiprocessors can all be viewed as parallel computers with some particular underlying architectural approach. The advent of parallel computers poses a large number of new problems for the scientific programmer in order that the extraordinary amount of (parallel) processing power can be fully utilized and exploited. Research efforts are under way to design and analyze new (parallel) algorithms for parallel computers and to develop libraries of software for the current applications in e.g. weather forecasting and aerodynamics simulations.

In the fall of 1983 a series of eight lectures was organized at the University of Utrecht to focus attention on the new developments in 'parallel computers and computations'. Eight experts of different backgrounds were invited to survey or describe an aspect of this field of research. The lectures covered concrete supercomputer architectures and their programming, the new challenges for systems programming, the design of numeric and non-numeric parallel algorithms, and the complexity of parallel computations.

This volume contains the full versions of the papers that were presented in the lecture series. We thank CDC/The Netherlands for its cooperation, the authors for their timely contributions to this book, and G.A.P. Kindervater for his editorial assistance.

J. van Leeuwen
J.K. Lenstra

Table of Contents

Parallel Computers and Algorithms

J. van Leeuwen
University of Utrecht

ABSTRACT

A variety of technological developments and algorithmic insights have led to the current designs of computing systems based on a small or large number of separate but cooperating processing units and data stores. Aim is to increase the overall processing speed and to allow that more and larger size scientific problems can be solved. We describe some of the algorithmic principles that underly many parallel algorithms.

1. INTRODUCTION

Ever since computers are being built, researchers and manufacturers have looked for ways of designing faster machines. Greater speed was obtained by improving the technology of individual components and applying techniques of instruction overlap and pipelining (see LORIN [37]) and by insisting on a sufficiently low level of programming to obtain efficient code. The insight developed that there are three essential ingredients to the overall speed of a computing system:

(i) the *speed* at which electronic circuits and wires can "switch" and transport information (signals),

(ii) the *organisation and interconnection* of the functional components in the hardware (architecture),

(iii) the *efficiency of data transfers* between processor(s) and memory and between memory and background stores (I/O).

The current, advanced computing systems have resulted from revolutionary developments in technology and algorithm design in each of these three directions. Hardware speed and compactness is enhanced by the advent of LSI- and of VLSI-technologies, which make it possible to have the power of a complete CPU in a few chips or on one board. It has stimulated the idea of having a

large supply of "processors" that cooperate in a computation. Secondly, already in the nineteen sixties it became apparent that the traditional von Neumann-type computer architecture would have to be changed to achieve substantial further speedups in execution. SCHWARTZ [50] wrote in 1965: "*The approach of present day computers to speeds at which the velocity of light becomes a significant design factor, and the continued fall in the price of computer components have directed attention to the use of parallelism as a device for increasing computational power.*" Presently a number of computers exist (see e.g. HOCKNEY & JESSHOPE [22]) that consist of a small or even a large number of "interconnected" processing units and memories. The ideas are also recognized in the approaches to large software systems viewed as systems of cooperating and communicating processes (see e.g. DIJKSTRA [12], HOARE [20]). Thirdly, the efficiency of instruction execution and data transfer is enhanced by letting processors act "in one sweep" on entire vectors of data that are available from special vector-registers (as in the CRAY-1 machines) or that are piped in from memory (as in the CYBER-205). I/O problems are (usually) solved by incorporating the "parallel" device in a host computer, or by providing a machine with suitable "front ends".

By now a number of different architectures of parallel computers have emerged, all based on some notion of how computations are to proceed and of how components in the architecture interact. A summary of the correspondences for present day architectures is given in figure 1 (from BÖHM [2]).

Model of Computation	Corresponding Computer Architecture
A. Sequential control on scalar data	A1. von Neumann-type computer
	A2. Multifunction CPU
	A3. Pipelined computer
B. Sequential control on vector data	B1. Vector computers
	B2. Array processors
C. Independent, communicating processes	C1. Shared memory multiprocessors
	C2. Ultra computers
	C3. Networks of small machines
D. Functional and data-driven computation	D1. Reduction machines
	D2. Dataflow machines

Figure 1. Computer architectures and their underlying computational model

The following five broad categories of parallel computers are often distinguished:

(i) *pipelined processors* (including e.g. the CRAY-1 and CYBER-205),

(ii) *SIMD machines* (including multiprocessor designs such as the ILLIAC IV and the Burroughs BSP),

(iii) *array processors* (a distinguished class of SIMD machines including e.g. the ICL-DAP and the AP-120B),

(iv) *MIMD machines* (distributed processor arrangements such as exemplified in the Denelcor HEP),

(v) *shared memory computers* (a class of MIMD machines including e.g. the CRAY-XMP).

The distinction between SIMD ("single instruction - multiple data") and MIMD ("multiple instruction - multiple data") machines is originally due to FLYNN [13] (see also STONE [56]), and refers to the distinction between all processors receiving the same stream of instructions (from a master processor) or possibly different ones. Many additional distinctions can be made (cf. HOCKNEY & JESSHOPE [22]), for example with respect to the amount of local memory available to each processor and/or the way global memory is shared (if there is a global memory at all) and the particular interconnection pattern used for the processor(s) and the memories.

All parallel computers fit the global form suggested in figure 2,

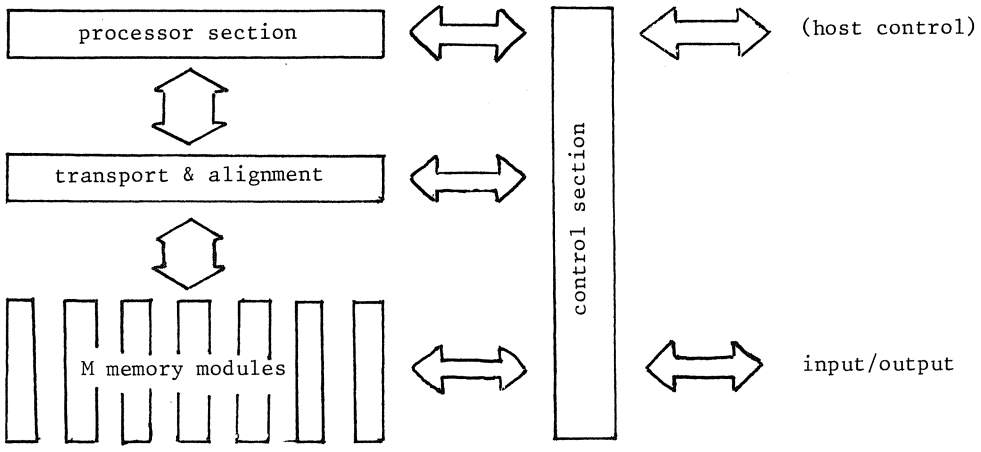


Figure 2. Global block diagram of a parallel computer

with many differences in the ways the various "sections" are realized. For example, in some machines the "processor section" will consist of one or two highly effective CPU's (as in vector/pipeline computers) and in other machines it will be an arrangement of 16 or more interconnected processors (as in array computers). The "transport section" is a highly pipelined data channel in some computers and a single stage or multi-stage processor/memory interconnection network (such as the shuffle-exchange network) in other designs. Memory is almost always partitioned into some M separate (but perhaps "interleaved") modules or "banks". In some machines M is a suitable power of two (M=8 or 16 for the CYBER 205, M=16 for the CRAY-1) whereas in other designs M was specifically chosen to be a prime number (M=17 in the Burroughs BSP). An excellent, brief survey of supercomputer organizations is given by HWANG, SU & NI [24].

The development of parallel computers and distributed systems has direct underpinnings in the theory of algorithms. A large number of studies (see e.g. KUCK [32] for an early example) have attempted to show the advantages and possible gains of a particular parallel architecture for scientific computation. Also, a sizeable literature developed on concrete parallel methods for use in e.g. numerical linear algebra (see e.g. the surveys by HELLER [19] and SAMEH [46]), sometimes under highly idealized assumptions about the capabilities of a parallel computer. In recent years the scope of this work has extended to all domains of discrete computing (see e.g. KINDERVATER & LENSTRA [26]). *Parallelism* has become a new dimension in algorithm design and analysis, of which the mathematical aspects are only beginning to be understood and for which the descriptional tools (viz. for programming) are still rather primitive. (Most parallel computers exploit a vector extension of FORTRAN, see PERROTT [42] for a possible alternative.) In this paper we shall present a brief impression of the new stimuli for algorithm research and the interaction with the ongoing development of parallel and distributed computing systems (see also VAN LEEUWEN [60]). In short 5 new classes of algorithms are arising because of this development:

(i) *vectorized algorithms* - the (re)formulation of (existing) algorithms in terms of uniform operations on vectors of data,

(ii) *systolic algorithms* - highly regular methods for dense processor arrays originally meant for implementation on a VLSI chip,

(iii) *parallel processing algorithms* - the formulation of algorithms as they are performed by a set of processors with a given interconnection pat-

tern or network,

(iv) *parallel algorithms* - methods for a set of processors that can communicate freely (and usually operate synchronously),

(v) *distributed algorithms* - methods for processors that communicate by exchanging messages (and usually operate asynchronously).

The distinction follows from the different domains of application of each of these classes and the different cost criteria used to evaluate algorithm performance. In the subsequent sections the distinctions between these types of algorithms will become clear.

2. INVITATION TO PARALLELISM

It is important to have a feeling for the ways parallelism can be discovered in a problem. Sometimes it is very hard or even impossible (cf. section 3). An example is the problem of computing the gcd of two n-bit numbers A and B by *Euclid's algorithm:*

{pre: $0 \leq A, B \leq 2^n$}
{post: a = gcd (A,B)}
a: = A;
b: = B;
while b ≠ 0 **do**

$$\begin{bmatrix} a \\ b \end{bmatrix} := \begin{bmatrix} b \\ a \bmod b \end{bmatrix} ;$$

It is well-known (Lamé's theorem, [27]) that Euclid's algorithm takes $O(n)$ steps, where each step involves a division of two $O(n)$-bit numbers. It is open whether a parallel algorithm can compute the gcd any faster, in a reasonable model of computation. At the bit-level one can do better than the $O(n^2)$ time-units of Euclid's algorithm. BRENT & KUNG [5] proposed the following method:

{pre: A odd, B ≠ 0, and $|A|, |B| \leq 2^n$}
{post: a = gcd (A,B)}
a := A;
b := B;
{use $\delta=\alpha-\beta$ with $|a| \leq 2^\alpha$, $|b| \leq 2^\beta$ and observe decrease of α,β}

```
δ := o;
repeat
    while b even do begin b := b div 2; δ := δ + 1 end;
    if δ ≧ o then begin swap (a,b); δ := - δ end;
    if (a + b) mod 4 = o then b := (a + b) div 2
                         else b := (a - b) div 2
until b = o;
```

The algorithm can be implemented by "streaming" A and B through the cells of a systolic array, low order bits first. The arithmetic on a and b is more or less done "in place", δ is represented by a separate sign bit and its absolute value in unary (a string of at most n ones). The algorithm terminates after at most 2n+1 iterations (because $\alpha + \beta$ strictly decreases during each round except possibly the first).

Theorem 2.1 *The gcd of two,* n*-bit numbers can be computed in linear time on a systolic array of* $O(n)$ *cells.*

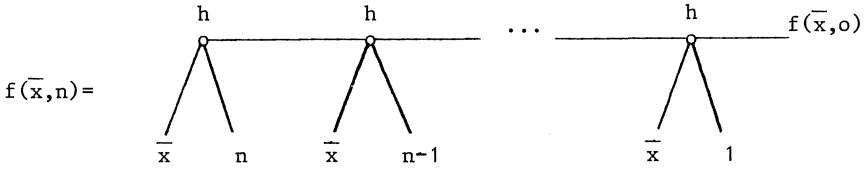
Fortunately it is not always this tricky to come up with a fast(er) parallel method. We shall discuss a number of important paradigms and the underlying techniques. We assume that processors are available in unlimited supply.

The best known examples of parallelism probably are the computations of x^n and of $a_o + \ldots + a_{n-1}$, both in $O(\log n)$ time. Both follow by the process of recursive doubling, which consists of the evaluation of subterms of size 2^i for i from o to logn. The true effect of parallelism is that in the same time-bound one can compute the values $\{x, x^2, x^3, \ldots\ldots, x^n\}$ and $\{a_o, a_o + a_1, a_o + a_1 + a_2, \ldots\ldots, \sum_{i=o}^{n-1} a_i\}$. There are several ways to see this. Consider a function $f(\bar{x},n)$ defined as follows:

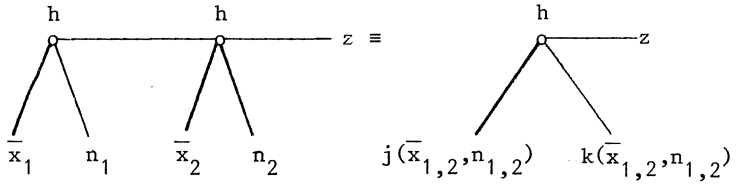
$f(\bar{x},o)=g(\bar{x})$

$f(\bar{x},n)=h(\bar{x},n,f(\bar{x},n-1))$ for $n > o$

with g and h "simple" functions. (One may recognize this as the defining scheme of primitive recursion.) It will be helpful to represent the evaluation of $f(\bar{x},n)$ by a graph:



Definition. A function $h(\bar{x},n,z)$ is calles *strongly reductive* if there are "simple" functions j and k such that for all $\bar{x}_1, \bar{x}_2, n_1, n_2$ and z we have



Functions like $\frac{x_1 z + x_2}{x_3 z + x_4}$ and $\sqrt{x + z^2}$ (provided they are non-degenerate) are strongly reductive.

Definition. A function $h(\bar{x},n,z)$ is called *reductive* if there are "simple" functions p and q and a strongly reductive function h' such that $h(\bar{x},n,z) = h'(p(\bar{x},n),q(\bar{x},n),z)$.

Theorem 2.2 *Let* f *be defined by primitive recursion using a reductive function* h. *Then the values* $\{f(\bar{x},o), f(\bar{x},1), \ldots, f(\bar{x},n)\}$ *can be computed by a parallel algorithm in* $O(\log n)$ *time.*

(The result is a slight extension of KOGGE & STONE [28].) In this way recursive doubling is applicable in a large number of instances. Figure 3 is taken from STONE [56]. In most cases $O(n)$ processors suffice.

Theorem 2.3 *The* LU*-decomposition of an* n x n *tridiagonal matrix (assuming it exists) can be computed in* $O(\log n)$ *time, using* n *processors.*

Function	Description
$x_i = x_{i-1} + a_i$	Sum the elements of a vector
$x_i = x_{i-1} \times a_i$	Multiply the elements of a vector
$x_i = \min(x_{i-1}, a_i)$	Find the minimum
$x_i = \max(x_{i-1}, a_i)$	Find the maximum
$x_i = a_i x_{i-1} + b_i$	First order linear recurrence, inhomogeneous
$x_i = a_i x_{i-1} + b_i x_{i-2}$	Second order linear recurrence
$x_i = a_i x_{i-1} + b_i x_{i-2} + \cdots$	Any order linear recurrence, homogeneous or inhomogeneous
$x_i = (a_i x_{i-1} + b_i)/(a_i x_{i-1} + d_i)$	First order rational fraction recurrence
$x_i = a_i + b_i / x_{i-1}$	Special case of first order rational fraction
$x_i = \sqrt{(x_{i-1})^2 + (a_i)^2}$	Vector norm

Figure 3. Functions suitable for recursive doubling

Proof.

The result is due to STONE [55]. Write A as

$$A = \begin{bmatrix} d_1 & f_1 & & & \\ e_2 & d_2 & f_2 & & \emptyset \\ & \ddots & \ddots & \ddots & \\ & & & & f_{n-1} \\ \emptyset & & & e_n & d_n \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ m_2 & \ddots & & \emptyset & \\ & \ddots & \ddots & & \\ & & & & \\ & & & m_n & 1 \end{bmatrix} \cdot \begin{bmatrix} u_1 & f_1 & & & \\ & u_2 & \ddots & & \emptyset \\ & & \ddots & & \\ & & & & f_{n-1} \\ \emptyset & & & & u_n \end{bmatrix} = \text{L.U.}$$

, then the following recursions are obtained:

$$m_i = e_i / u_{i-1} \quad (2 \leq i \leq n) \qquad \text{and} \qquad u_1 = d_1$$

$$u_i = d_i - e_i f_{i-1} / u_{i-1} \quad (2 \leq i \leq n)$$

The m_i's can be computed in one parallel time-step once the u_i's are available. Define $\{v_i\}_{0 \leq i \leq n}$ by $v_0 = 1$, $v_1 = d_1$ and for $i \geq 2$ $v_i = d_i v_{i-1} - e_i f_{i-1} v_{i-2}$. Then the v_i's are computable in $O(\log n)$ time and n processors using recursive doubling, and one easily verifies that $u_i = v_i / v_{i-1} (1 \leq i \leq n)$. □

The result can be extended to show that a tridiagonal linear system Ax=b can be solved in $O(\log n)$ time, using n processors. A rather more involved application of recursive doubling is used in the following result due to CHEN & KUCK [8] (also SAMEH & BRENT [47]) and, as for part (ii), to GREENBERG *et.al.* [17].

Theorem 2.4 *Let* L *be a non-singular triangular* n x n - *matrix with bandwidth* m + 1. *Then*

(i) *there is an algorithm for solving a system* Lx=b *in* $O(\log n . \log m)$ *time using* $O(n m^2)$ *processors,*

(ii) *there is an algorithm for solving a system* Lx=b *in* $O(\log n . \log m)$ *time using* $O(nm^{\alpha-1}/\log n . \log m)$ *processors where "α" is the exponent of an efficient, i.e.,* $O(n^{\alpha})$ *matrix multiplication algorithm.*

The theorem is important for its connection to the evaluation of m^{th} order linear recurrences. The best exponent α presently known is about 2.49.

A second technique to exploit parallelism is to decompose a problem into a number of independent sub-problems of which the solutions compose into the answer of the original problem, and to elaborate the sub-problems recursively in parallel by the same method. It is the well-known paradigm of *divide-and-conquer*, in a parallel setting. Using divide-and-conquer it is possible to understand the result expressed in theorem 2.4 for m=n-1.

Proposition 2.6 *A non-singular triangular linear system* Lx = b *can be solved in* $O(\log^2 n)$ *time, using* $O(n^3)$ *processors.*

Proof.

Compute L^{-1} as follows. Decompose (split) L into four equal size parts and observe that

$$\overbrace{\left\{\begin{bmatrix} A & 0 \\ B & C \end{bmatrix}^{-1}\right.}^{n/2} = \underbrace{\begin{bmatrix} A^{-1} & 0 \\ -C^{-1}BA^{-1} & C^{-1} \end{bmatrix}}_{n/2}$$

where A and C are again non-singular and triangular. Note that parallel matrix multiplication needs only $O(\log n)$ time on $O(n^3)$ processors, using recursive doubling to evaluate all component expressions. Hence, after computing A^{-1} and C^{-1} recursively in parallel only $O(\log n)$ further steps on $O(n^3)$ processors suffice to obtain L^{-1}. Altogether an algorithm of the desired complexity results. □

The implicit inversion method for triangular matrices can be viewed as a (very) special case of a much harder result due to CSANKY [11].

Theorem 2.7 *A non-singular* n x n *-matrix can be inverted in* $O(\log^2 n)$ *time, using* $O(n^4)$ *processors.*

It is open whether the $O(\log^2 n)$ bound can be improved. PREPARATA & SARWATE [43] have shown that Csanky's algorithm can be implemented using $O(n^{\alpha+\frac{1}{2}}/\log^2 n)$ processors, where α is the exponent of a matrix multiplication algorithm.

As another example of divide-and-conquer, consider the evaluation of an n^{th} degree polynomial $a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_o$ which, as is well-known, takes $O(n)$ steps using Horner's method.

Theorem 2.8 *A polynomial of degree* n *can be evaluated in* $O(\log n)$ *time using* n *processors.*

Proof.

Assume $n = 2^k-1$. Write p(x) of degree n as $q(x).\ x^{(n+1)/2} + r(x)$ for suitable polynomials q and r of degree $2^{k-1}-1$, and evaluate q and r by the same method recursively in parallel. In composing the answers from the "bottom" upwards, compute the necessary powers of x in $O(1)$ extra time per level. The entire computation takes about 2 log n "steps", using n processors. (This is *Estrin's algorithm*, see e.g. MUNRO & PATERSON [39] for more efficient splittings.) □

Divide-and-conquer algorithms suggest to organize a computation in a tree of processors, where we start with the undivided problem at the root (figure 4.a) and send off the "halved" instances of the problem to the son processors (figure 4.b) until sufficiently "simple" instances are obtained. The need to transfer data is solved by providing the "tree machine" with global memory, or sufficiently powerful "data paths". See HOROWITZ & ZORAT [23] for details.

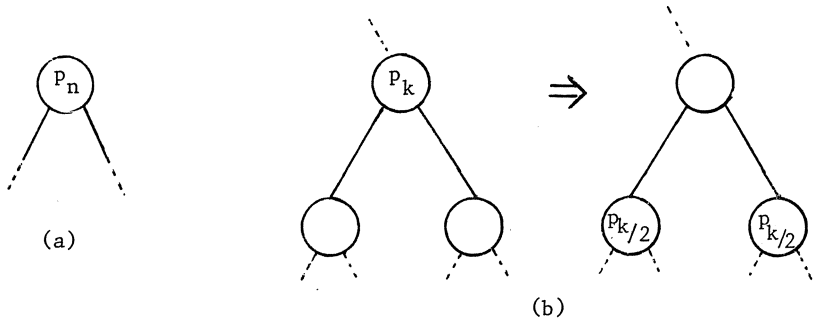


Figure 4

Observe that a computation on a problem P_n (n a power of two) requires a tree of 2n-1 processors, of which at most n will be active at the same time. BÖHM [2] had made the following observation:

Theorem 2.9 *A divide-and-conquer algorithm for a problem of "size"* n *can be implemented on a tree machine of* n *processors.*

A third technique of constructing parallel algoritms is the discovery of *independent subexpressions*. Given the fact that expressions are often given by parse-trees, one can try to extract sub-expressions that lead to a balanced decomposition for parallel evaluation. The following result is due to BRENT [4].

Theorem 2.10 *An arithmetic expression in* n *variables and constants using* +, * *and* / *and any depth of parenthesis nesting can be evaluated in* $O(\log n)$ *time using* $O(n/\log n)$ *processors.*

The technique has also been exploited for the evaluation of multivariate polynomials. Improving on a result of HYAFIL [25], SKYUM & VALIANT [52] proved the following remarkable fact.

Theorem 2.11 *A multi-variate polynomial of degree* d *that can be computed sequentially in* C *steps, can be computed in parallel in* $O(\log d.\log C + \log^2 d)$ *steps using a number of processors polynomial in* C.d.

It follows, for example, that the determinant of a n x n matrix can be evaluated in $O(\log^2 n)$ time using polynomially many processors. (This can also be derived from Csanky's results [11], see theorem 2.7.)

A fourth, and very common technique in parallel methods is the *change of the order of evaluation* (usually in complicated expressions). This is done very often in "vectorizing" existing software, but there are other applications too. An important example is the problem of computing the product C=A.B of two n x n matrices, which can be described by the n^2 expressions C_{ik} $(1 \leq i,k \leq n)$ with $C_{ik} = \sum_{j=1}^{n} A_{ij} B_{jk}$ or pictorially as

$$\begin{bmatrix} & \vdots & \\ \cdots & C_{ik} & \cdots \\ & \vdots & \end{bmatrix} = i \begin{bmatrix} \vdots & \vdots & & \vdots \\ A_{i1} & A_{i2} & \cdots & A_{in} \\ \vdots & \vdots & & \vdots \end{bmatrix} \cdot \begin{matrix} \\ k \\ \downarrow \end{matrix} \begin{bmatrix} & \cdots & \\ \cdots & B_{1k} & \cdots \\ \cdots & B_{2k} & \cdots \\ & \vdots & \\ & \vdots & \\ \cdots & B_{nk} & \cdots \end{bmatrix}$$

Direct evaluation would not take advantage of any vector-processing capability and also suggests that A and B are stored in different modes, row-wise and column-wise, which is not likely. There is a simple method, known as the "middle product" method (cf. HOCKNEY & JESSHOPE [22]), which computes C column-wise when A is stored column-wise and B is stored in any fashion:

$$\begin{bmatrix} C_{1k} \\ \vdots \\ C_{nk} \end{bmatrix} = \begin{bmatrix} A_{11} \\ \vdots \\ A_{n1} \end{bmatrix} \cdot B_{1k} + \begin{bmatrix} A_{12} \\ \vdots \\ A_{n2} \end{bmatrix} \cdot B_{2k} + \cdots + \begin{bmatrix} A_{1n} \\ \vdots \\ A_{nn} \end{bmatrix} \cdot B_{nk}$$

$(1 \leq k \leq n)$. The algorithm can be implemented as a scalar multiply of the n vectors of A followed by a vector add, and thus takes about n^2 multiplications and n.(n-1) vector additions. The algorithm is not very useful for e.g. banded matrices. MADSEN, RODRIGUE & KARUSH [38] have shown that in this case a reasonable vector algorithm can be designed based on the diagonals of A and B, requiring only about 2m+1-k vector multiplications and additions for accumulating all coefficients of a k^{th} column (m + 1 is the assumed band-

width). Storing matrices diagonal-wise has the added advantage that the transpose of a matrix is very easy to obtain. Finally it is possible to view C as the sum of n matrices of the form

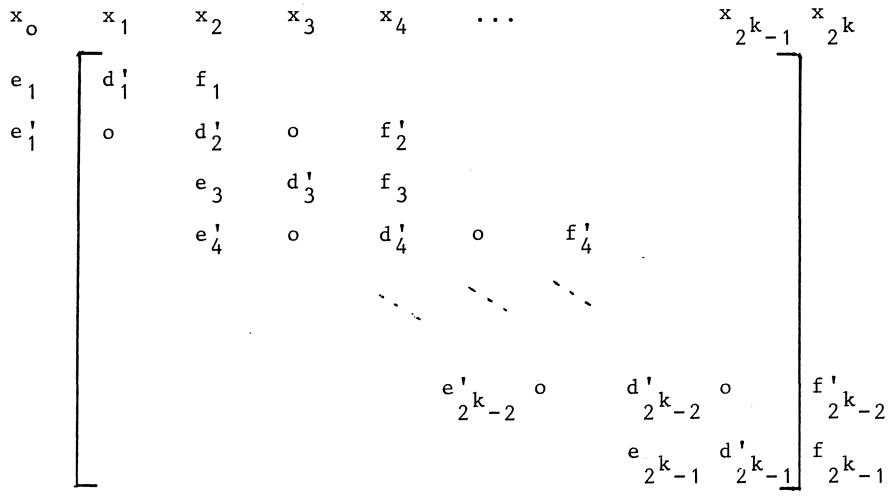
$$\begin{bmatrix} A_{1k} & A_{1k} & \cdots & A_{1k} \\ \vdots & \vdots & & \vdots \\ A_{nk} & A_{nk} & \cdots & A_{nk} \end{bmatrix} \times \begin{bmatrix} B_{k1} & B_{k2} & \cdots & B_{kn} \\ B_{k1} & B_{k2} & \cdots & B_{kn} \\ \vdots & \vdots & & \vdots \\ B_{k1} & B_{k2} & \cdots & B_{kn} \end{bmatrix}$$

, the multiplication taken component-wise, which can be advantageous for use on an array processor with rapid row- and column-transfer operations.

A fifth technique, specific to banded linear system solvers, is known as *cyclic reduction* or *odd-even reduction*. It is best explained using the example of a tridiagonal system Ax=b, where we assume A as in theorem 2.3 and of size 2^k-1. The method was apparently first used by HOCKNEY [21], and will be described without explicit mention of the necessary operations on b. Write A as follows

$$\begin{array}{c} \begin{matrix} x_o & x_1 & x_2 & x_3 & x_4 & \cdots & & x_{2^k-1} & x_{2^k} \end{matrix} \\ \begin{matrix} e_1 \end{matrix} \begin{bmatrix} d_1 & f_1 & & & & & & & \\ e_2 & d_2 & f_2 & & & & & & \\ & e_3 & d_3 & f_3 & & & & & \\ & & e_4 & d_4 & f_4 & & & & \\ & & & \ddots & \ddots & \ddots & & & \\ & & & & & e_{2^k-2} & d_{2^k-2} & f_{2^k-2} & \\ & & & & & & e_{2^k-1} & d_{2^k-1} & \end{bmatrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \\ f_{2^k-1} \end{matrix} \end{array}$$

where e_1 and f_{2^k-1} are added for consistency and use the convention that $x_o = x_{2^k} = o$. Bij a sweep using the odd-numbered equations we zero the e and f coefficients in the even-numbered equations, to obtain a aystem of the form



Now observe that of we have the values of x_o, x_2, x_4, ... then the values of x_1, x_3, ... follow in one further step from the odd-numbered equations. But the even-numbered equations form a tridiagonal system on the x_o, x_2, x_4 ... separately and we can continue recursively until a single equation in x_o, $x_{2^{k-1}}$ and x_{2^k} remains (assuming the algorithm nowhere degenerates). Since x_o and x_{2^k} were defined o we can solve for $x_{2^{k-1}}$, and "backsolve" at all levels of the recursion. Clearly, when it works, cyclic reduction solves a tridiagonal system in $O(\log n)$ time using n processors. The method has been extended to block-tridiagonal systems by SWEET [57] and to arbitrary banded linear systems by RODRIGUE, MADSEN & KARUSH [45] (who also gave conditions for the method to work).

A sixth method for constructing parallel algorithms is called *broadcasting*, although it is implicit already in some of the techniques we have seen. We rather use the term to denote the continued distribution of computed results throughout the stages of an algorithm to all processors. An example is the "*column sweep*" algorithm for solving a non-singular triangular linear system Lx=b (compare theorem 2.4).

Theorem 2.12 *A non-singular triangular* n x n *system* Lx=b *can be solved in* $O(n)$ *time using* n *processors.*

Proof.

(Observe that the time bound is worse than given in theorem 2.4 but the method will use fewer processors and is appreciably simpler.) Rewrite the

system into the form $x = Lx + b$ with L lower triangular. Clearly $x_1 = b_1$. Now use processors P_i $(2 \leq i \leq n)$ and assume that after eliminating x_{j-1} $(j \geq 2)$ the P_i with $i \geq j$ have the value $\ell_{i1} x_1 + \ldots + \ell_{ij-1} x_{j-1}$ in store. In the next cycle P_j can compute x_j. It subsequently broadcasts the value to all P_i with $i > j$, which compute $a_{ij} x_j$ and add it to the partial sum they accumulate. □

Broadcasting is often used in distributed algorithms.

A seventh technique to exploit parallelism is *pipelining*. It is encountered in all systolic algorithms (see e.g. KUNG [34] and KRAMER & VAN LEEUWEN [30]) and in several methods for parallel sorting. As an example we consider a sorting method due to TODD [58], based on the idea of merge sort.

Theorem 2.13 *A set of* n *elements can be sorted in* $O(n)$ *time using* $O(\log n)$ *processors.*

Proof.

Assume $n = 2^k$. Merge sort can be represented in a perfect binary tree, with the leaves holding the single elements to be sorted and the nodes at level i $(i \geq 1)$ having queues of size 2^i in which the (sorted) queues of the sons can be merged. Assign a processor P_i to every level of the tree. P_i merges "pairs" of consecutive queues into a block in P_{i+1}'s store. P_{i+1} starts as soon als P_i has produced one complete block (of length 2^{i+1}) and the first element of the next block, and continues at the same speed until all elements from level i+1 are merged upwards. One can verify that P_{i+1} never needs to wait for elements and (hence) that the P_i's form a perfect pipeline. It takes about 2.2^{i+1} time steps before a P_{i+1} can start, and it is guaranteed to finish in another n steps. The entire "pipeline" delivers the set as a single sorted queue in about 2n time. □

A similar method was recently used by CAREY & THOMPSON [7] to obtain a parallel dictionary algorithm that can "pipeline" searches, insertions and deletions using $O(\log n)$ processors (n is the number of elements in the set). They assign a processor to each level of 2-3-4 tree, for which a one-pass topdown update algorithm is known to exist. Faster parallel sorting methods exist but require more processors. The following classical result is due to BATCHER [1] (see also STONE [54]).

Theorem 2.14 *A set of* n *elements can be sorted in* $O(\log^2 n)$ *time using* $O(n)$ *processors.*

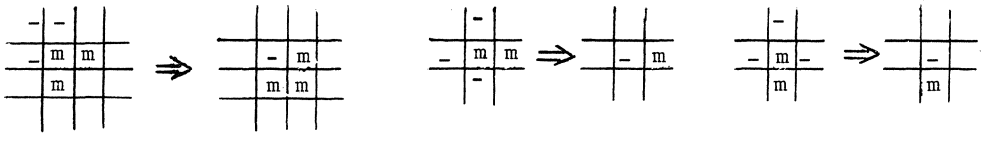
VALIANT [59] has shown that $O(\log n. \log\log n)$ parallel comparisons are sufficient to sort. An excellent survey of parallel sorting algorithms was given by FRIEDLAND [14].

An eighth technique for obtaining parallel methods is often found in graph algorithms and is known as *collapsing*. It normally consists of the processors cooperating in some way to accumulate information about larger and larger chunks of a graph, with processors effectively collapsing the information of a "neighborhood" of diameter 2^i for i from o on increasing into a single node. It explains (i.e., intuitively) why many graph algorithms have $O(\log^2 n)$ time bounds when many processors are used, because they involve log n phases of $O(\log n)$ parallel time each. See e.g. SAVAGE & JA'JA [48], or the survey by QUINN & DEO [44]. The algorithms are very sensitive to the way a graph is represented, in common memory (as is usually assumed) or by an adjacency map on a processor array (which leads to slower algorithms because of the communications over a grid, cf. KOSARAJU [29]). As an example of a collapsing (or "shrinking") algorithm we consider the following result of LEVIALDI [36].

Theorem 2.15 *Let some of the processors of an* n x n *processor array be marked. Connectivity of the marked processors can be recognized in* $O(n)$ *steps.*

Proof.

We only consider connectedness by shared "edges". Denote a marked processor by "m". Let the processors apply the following transformations in parallel:



The transformations preserve connectivity, and have the effect of shrinking a connected part towards the bottom right corner of the rectangle circumscribing it. In fact, the maximum rectilinear distance of this corner to a marked processor decreases by 1 at every iteration and a marked component will have shrunk to a single m-cell within 2n steps. Once an m-cell finds

itself without marked neighbors it must verify by broadcasting that it is the only marked processor left, which takes another $O(n)$ steps. □

It is open whether a linear time algorithm exists for testing the connectivity of a marked set in an n x n x n processor cube (cf.KOSARAJU [29]).

3. ISSUES TOWARDS REALIZATION

There are a number of reasons why parallel computers can be slower than anticipated in a theoretical analysis, slower perhaps than the fastest "sequential" computers. To obtain the optimum performance of a parallel computer one may have to decompose a problem and arrange a computation in a very machine dependent manner and "tune" an algorithm with due attention for processor structure, communication costs and data distribution. We will discuss the algorithmic aspects of some of the issues that arise.

The desired effect of having p processors available instead of just one is the *speed-up* of a computation by a factor of about p. The required distribution of work cannot always be achieved, and there even are problems for which no parallel algorithm can be substantially faster than the best sequential algoritm. (A simple example is the computation of x^n in $O(\log n)$ steps.) The following result is due to KUNG [33].

Definition. Let $f(x) = {}^{p(x)}/_{q(x)}$ be a rational function with p(x) and q(x) polynomials that are relatively prime. Then $\mathrm{Deg}(f) = \max\{\deg p, 1 + \deg q\}$.

Theorem 3.1 *The computation of a rational function* f *requires at least* log Deg (f) *time, regardless the number of processors used.*

An interesting application (also from KUNG [33]) can be obtained for the evaluation of first order recurrences of the form

$$x_o = y$$
$$x_{i+1} = \varphi(x_i)$$

with φ rational.

Definition. Let $\varphi(x) = {}^{p(x)}/_{q(x)}$ be a rational function with p(x) and q(x) polynomials that are relatively prime. Then $\deg \varphi = \max\{\deg p, \deg q\}$.

Theorem 3.2 *The computation of the* n^{th} *term of a first order recurrence with* φ *rational requires at least* $n.\log \deg \varphi$ *time, regardless the number of processors used.*

Proof.

We use the following fact: when φ and ψ are rational in x, then $\deg \varphi\circ\psi = \deg \varphi . \deg \psi$. Now observe that $x_n = \varphi^n(y) = \Phi(y)$ with $\deg \psi = (\deg \varphi)^n$, hence $\mathrm{Deg}(\Phi) \geq (\deg \varphi)^n$. By theorem 3.1 the computation of x_n must require at least $n . \log \deg \varphi$ time. □

As an example the computation of $\sqrt{a}$ using the recurrence $x_o = a$, $x_{i+1} = {}^1/_2(x_i + {}^a/x_i)$ cannot be sped up by more than a constant factor, no matter how many processors are supplied.

In section 2 we have always assumed that processors are available in unlimited supply ("*unbounded parallelism*"). This is clearly not the case in practice, but the assumption can be justified by a simple result known as "*Brent's Lemma*" (from [4]).

Theorem 3.3 *Assume a computation consisting of a total of* b *operations can be carried out in time* t *using unbounded parallelism. The the computation can be carried out with* p *processors in approximately* $t + {}^{(b-t)}/p$ *steps.*

Proof.

Suppose s_i operations are performed in parallel during step i $(1 \leq i \leq t)$, with $b = \sum_1^t s_i$. Using p processors we can simulate step i in $\lceil {}^{s_i}/_p \rceil$ time. The entire computation is thus rescheduled and takes a number of steps bounded by $\sum_1^t \lceil {}^{s_i}/_p \rceil \leq \sum_1^t (s_i + p-1)/p = (1-1/p)\ t + 1/p . \sum_1^t s_i = t + {}^{(b-t)}/_p$. □

In most parallel algorithms (viz. those based on the assumption of unbounded parallelism) the cost for communicating information is not taken into account. A better model is obtained if we view an algorithm as a directed acyclic graph in which the nodes represent operations, the edges are data paths and the levels represent stages of parallel activity. We assume that nodes have in-degree o (inputs) or, say, 2. An algorithm representation of this kind is called a *circuit*. If the algorithms can have a variable number of inputs n, then we normally want the corresponding circuits to be defined in a "uniform" manner for all admissible n. (For a more formal approach, see

COOK [9].) The important measures for circuits are the size s (the number of nodes), the depth d (the length of the longest path) and the number of levels t. BORODIN [3] has argued that circuitdepth is an adequate measure of "parallel time". We show this using a simple result due to GREENBERG *et.al.*[17].

Theorem 3.4 *A circuit of size* s *and depth* d *can be "evaluated" in time* $O(d)$ *using* $\lceil s/d \rceil$ *processors.*

Proof.

Consider the circuit and define S_i to be the set of nodes that are i edges away from the farthest input node ($o \leq i \leq d$). Clearly S_o consists of the input nodes, and the nodes of S_i can be evaluated once the nodes in $\bigcup_{o}^{i-1} S_j$ are. Thus the circuit can be "evaluated" in the order S_o, S_1, ... Evaluation of the nodes in S_i takes $\lceil |S_i|/p \rceil$ time using p processors. The entire computation takes $\sum_{o}^{d} \lceil |S_i|/p \rceil \leq \sum_{o}^{d} (|S_i|+p-1)/p = (1-1/p)\, d + 1/p \sum_{o}^{d} |S_i| = d + (s-d)/p$ steps. Choose p about s/d so the total amounts to $O(d)$. □

"Practical" parallel methods should use at most polynomially many processors and, in view of the results from section 2, $O(\log^k n)$ time for some k. This has led to the study of the class *NC* of problems that have polynomial size circuits of poly-log depth. See COOK [9] (or [10]) for an introduction.

The next step to understanding the complexity of parallelism requires a suitable model of a parallel computer. Closest to our present assumptions is the MIMD-model where processors communicate information through a global memory but can execute different programs. It immediately leads to the issue of conflicting read and/or write instructions. Usually simultaneous reads of a location are allowed, but simultaneous writes are not. See e.g. KUÇERA [31] and VISHKIN [62] for comments on this problem. Almost always it is assumed that the processors in the model are synchronized and even there is a global master - CPU. A very general and representative model of this kind was recently proposed by GOLDSCHLAGER [15] and is called the "*SIMDAG*" model, which combines the SIMD concept of a set of parallel processing units (PPU's) and global memory (see figure 5) and encompasses many earlier models. All processors have a full RAM-instruction set (no multiplication primitive), the CPU "contains" the program and occasionally broadcasts "parallel" instructions to all PPU's. Each PPU has its own index stored in a special signature

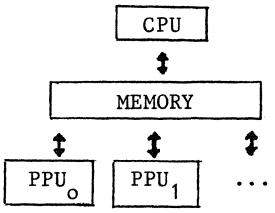


Figure 5

register (which thus provides a way to distinguish or mark processors). Simultaneous writes to a same location in global memory are resolved by giving priority to the lowest numbered PPU. Define SIMDAG-TIME (T(n)) as the class of problems solvable in (parallel) time T(n) on a SIMDAG, and define SPACE (S(n)) as the class of problems solvable in space S(n) on an ordinary random access machine. GOLDSCHLAGER [15] proves the following "*parallel computation thesis*" for the SIMDAG-model:

Theorem 3.5 *For every SIMDAG computable function* $T(n) \geq \log n$ *one has* $\bigcup_k$ SIMDAG-TIME $(T^k(n)) = \bigcup_k$ SPACE $(T^k(n))$.

The result supports the thesis that "parallel time" is equivalent (within a polynomial increase) to space on a Turing machine, which holds for other models of parallel computers that are sufficiently general too (see e.g. SAVITCH & STIMSON [49]).

Memory in a parallel computer is normally divided into a number of *banks* so complete "vectors" of data items from different banks can be fetched in one cycle. Assuming there are M banks, one can fetch vectors of up to M data items in every "cycle". Larger vectors must be broken up in chunks of size $\leq$ M and are retrieved by multiple parallel fetches. If the elements of an M-vector are not alle stored in different banks, then we say that a "conflict" occurs. KUCK [32] (see also BUDNIK & KUCK [6]) has shown already in the late nineteen sixties that the optimal benefit from "parallel memories" requires non-trivial distributions of the data and address-calculations, in order that vectors and blocks of data that are needed in the course of an algorithm are indeed available from distinct banks (and can be found!). For example, storing a N x N matrix ($N \leq M$) with one column

in every bank allows conflict-free access to every row and every diagonal in one cycle but forces sequential access for retrieving the elements of every column. On the other hand, a "skewed" organization as shown in figure 6 (with N = 4 and M = 5) alleviates these difficulties at least for rows, columns, and forward and backward diagonals. Any storage scheme s that maps

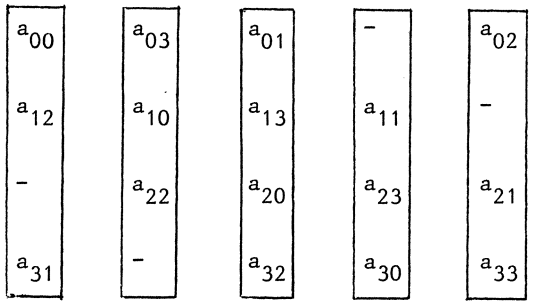


Figure 6. Storing a 4 x 4 matrix into 5 memory banks.

the elements of an N x N matrix into M memory banks ($M \geq N$) and provides for the conflict-free access to various vectors of interest is called a "*skewing scheme*". (We do not discuss the skewing of higher dimensional matrices.)

The simplest and most commonly used skewing schemes are the "*linear skewing schemes*" defined by formulae of the type

$$s(i,j)=ai + bj \pmod M$$

, for suitable integers a and b. We assume that s uses all memory banks and (hence) that (a,b,M)=1. Skewing schemes of this kind will be called "proper". (The convention is for theoretical purposes only, in practice one may want to store up to (a,b,M) distinct matrices in an interleaved manner using the same scheme with suitable shifts.) WIJSHOFF & VAN LEEUWEN [63] prove the following result

Theorem 3.6 *In order to have conflict-free access to rows, columns, and non-circulant forward and backward diagonals using a linear skewing scheme, the smallest number of memory banks required is*

$$M = \begin{cases} N & \text{if } 2 \nmid N \text{ and } 3 \nmid N \\ N+1 & \text{if } 2 \mid N \text{ and } N \equiv 0,1 \pmod 3 \\ N+2 & \text{if } 2 \nmid N \text{ and } 3 \mid N \\ N+3 & \text{if } 2 \mid N \text{ and } N \equiv 2 \pmod 3 \end{cases}$$

Moreover, it is possible to achieve this in all cases using the scheme $s(i,j) = i + 2j \pmod M$.

The result extends an observation of BUDNIK & KUCK [6] (see also LAWRIE [35]) that there is no linear skewing scheme to store an N x N matrix into N memory banks and have the desired types of conflict-free access when N is even.

Clearly different conditions arise when the set of vectors of interest is changed. Using linear skewing schemes most vectors will be stored with a fixed increment between the bank-numbers of consecutive elements.

Definition. A d-*ordered* k-*vector* is a vector of k elements whose i^{th} logical element ($o \leq i < k$) is stored in memory bank $c + di \pmod M$, for some constant c.

The following elementary result is essentially due to LAWRIE [35] (see also [63]).

Theorem 3.7 *A* d-*ordered* k-*vector can be accessed conflict-free if and only if* $M \geq k.\gcd(d,M)$.

Proof.

$\Rightarrow$. Consider a d-ordered k-vector and assume it can be accessed conflict-free. It means that for all $o \leq i_1, i_2 < k$, $i_1 \neq i_2$, we have $c + di_1 \not\equiv c + di_2 \pmod M$ hence $d.i \not\equiv o \pmod M$ for every $o < i < k$. This implies $\frac{M}{\gcd(d,M)} \geq k$, or $M \geq k.\gcd(d,M)$.

$\Leftarrow$. Observe that all steps in the given argument can essentially be reversed. □

WIJSHOFF & VAN LEEUWEN [63] prove a slightly more general result for the case of multiple parallel fetches:

Theorem 3.8 *A* d-*ordered* k-*vector can be accessed in precisely* $1 + \lfloor \frac{(k-1)\gcd(d,M)}{M} \rfloor$ *conflict-free fetches, and this is best possible.*

Given a linear skewing scheme s(i,j)= ai + bj (mod M) for storing an N x N matrix it is easily seen that (i)rows are b-ordered N-vectors, (ii) columns are a-ordered N-vectors, (iii) non-circulant diagonals of length k are (a+b)-ordered k-vectors ($1 \leq k \leq N$) and (iv) non-circulant anti-diagonals of length k are (a-b)-ordered k-vectors. Yet d-ordered vectors are only of limited scope. For example, the full circulant diagonals and anti-diagonals cannot be viewed as d-ordered N-vectors. Independently SHAPIRO [51] and HEDAYAT [18] proved the following result (compare theorem 3.6):

Theorem 3.9 *There exists a (proper) linear skewing scheme using* M=N *memory banks that provides conflict-free access to rows, columns, and all circulant diagonals and anti-diagonals if and only if* $2 \nmid N$ *and* $3 \nmid N$.

In general one may want to retrieve more general "*templates*" of matrix cells (e.g. blocks or L-shapes). For the practical case that M < N and vectors must be retrieved by multiple fetches WIJSHOFF & VAN LEEUWEN [63] prove the following result:

Theorem 3.10 *There exists a linear skewing scheme to store an* N x N *matrix in* M *memory banks such that every rookwise connected template of* t *cells can be retrieved by means of at most* $\lfloor \frac{t}{\sqrt{M}} \rfloor + 1$ *conflict-free fetches of vectors from the* M *memory banks.*

A survey of the general theory of skewing schemes was recently given by VAN LEEUWEN & VAN WIJSHOFF [61].

In SIMD-type architectures the processors (or perhaps even the processors and the memories) are connected in an *interconnection network* and yet another component is added to the problem of realizing a parallel computation, namely the problem of distributing a computation over the network and providing for the fast communication of intermediate results to the processors that need it (over the wires of the network). This leads to the problem of routing single data items from a source address to a destination address and to the (harder) problem of routing a number of source-destination pairs simultaneously, which is *the routing problem* for arbitrary permutations. A routing algorithm should route all messages in parallel with no queueing or conflicts, and essentially provide for the right switch settings at every stage to let the messages receive their destinations fast.

Processors could be interconnected by a simple crossbar switch, but in a number of designs more sophisticated networks have been used that use fewer than N^2 switches (N the number of processors).

Theorem 3.11 *Every network that realizes all connections between* N *processors must have* $\Omega(N \log N)$ *switches.*

Proof.

To route all permutations the network must admit at least N! different internal settings. If the network has s switches that can be in c states each (c some constant) then it can have at most c^s internal setting. Thus $c^s \geq N!$, and $s \geq {}^{\log N!}/_{\log c} = \Omega(N \log N)$ for any network. □

Let $N=2^n$, and assume that processors are indexed by n-bit binary numbers. Most networks are designed with some idea in mind of how to route information from address $a = a_{n-1}\, a_{n-2} \cdots a_o$ to address $b = b_{n-1}\, b_{n-2} \cdots b_o$ (with $o \leq a,b, < N$). The simplest idea is to put the processors at the vertices of a binary N-cube and to use the edges as wires. In $\log N = n$ iterations (at most) one can turn every bit of a into the corresponding bit of b and do the desired routing, but a disadvantage is that in every node n edges meet and thus n "switches" are put together. It is not possible to survey all networks here that have been proposed as alternatives with a bounded degree (e.g. 2) at every node, but many are essentially equivalent to the cube (see PARKER [40]). We shall only digress briefly to introduce *the omega network* that has received most attention.

Define the "*shuffle*" as the mapping σ defined by $\sigma(a_{n-1}\, a_{n-2} \cdots a_o) = a_{n-2} \cdots a_o\, a_{n-1}$ and define the (single stage) *shuffle-exchange network* as the "graph" of s with the edges leading pairwise into ${}^N/_2$ switches that can pass the data on or "exchange" it on the outgoing pair of lines. Effectively a switch either applies the identity of does an *exchange* e defined by $e(a_{n-1}\, a_{n-2} \cdots a_1\, a_o) = a_{n-1}\, a_{n-2} \cdots a_1\, \bar{a}_o$, i.e., it flips the last bit. Note that a message can be routed from a to b in (at most) $\log N = n$ shuffle-exchange steps, by simply rotating and flipping a's binary form into b's binary form. This suggests to either allow up to n iterations (or more) through the shuffle-exchange graph (as in STONE [54]) or to unfold it to an n-stage network of shuffle-exchange "steps" (as in LAWRIE [35].) The latter is known as the omega network, and shown in figure 7 for the case N=16. PARKER [40] has given the following characterization of the class Ω_N of permutations that can be routed in the omega network:

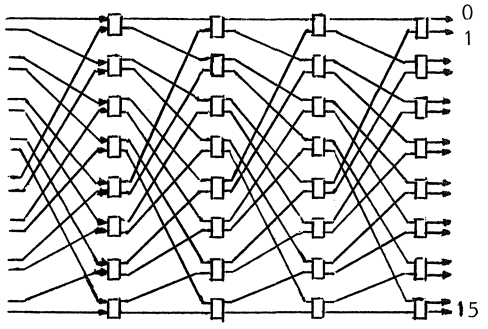


Figure 7.
The omega network
for N=16

Theorem 3.12 *Let* π *be a permutation mapping* a's *to* b's *(as above), then* $\pi \in \Omega_N$ *if and only if there are* n *boolean functions* $\{f_i\}_{o \leq i < n}$ *of* n-1 *variables such that for all* $o \leq i < n$ *bit* b_i *of* b *can be expressed as* $b_i = a_i \oplus f_i (b_{n-1}, \ldots, b_{i+1}, a_{i-1}, \ldots, a_o)$.

($\oplus$ is addition modulo 2.) The analysis of the class Ω_N is a tedious one. The following theorem combines deep results of PEASE [41], PARKER [40], and WU & FENG [64]. Let S_N be the permutation group on N elements and let ρ be the *bit-reversal permutation*, i.e., the permutation defined by $\rho (a_{n-1}\ a_{n-2} \cdots a_o) = a_o \cdots a_{n-2}\ a_{n-1}$.

Theorem 3.12

(i) $S_N \subseteq \Omega_N^{-1} o\ \Omega_N$

(ii) $S_N \subseteq \Omega_N \circ \rho \circ \Omega_N$

(iii) $S_N \subseteq \Omega_N^3$

(Recent results of STEINBERG [53] have simplified some of the proofs.) The theorem expresses the interesting result that every permutation can be routed in e.g. three forward passes through the omega network. WU & FENG [65], and also STEINBERG [53], have given an explicit algorithm to set the switches for a routing in $\leq 3 \log N$ steps. It is conjectured that $S_N \subseteq \Omega_N^2$, i.e., that at most two passes through the omega network suffice to route every permutation. (This is known as "*Parker's problem*".)

For computational purposes the shuffle-exchange network (or an iterated

form of it such as the omega network) appears to be very powerful as an interconnection network of "intelligent" processors that do a moderate amount of processing at every stage. To demonstrate this it is useful to consider an other network suggested by PEASE [41] first, the socalled *indirect binary n-cube*. Define the i^{th} order *"butterfly" permutation* ($1 \leq i \leq n$) as the permutation which interchanges the first and i^{th} bits of the address. The indirect binary n-cube consists of log N stages of N processors (in blocks of two) like the omega network, with the i^{th} order butterfly permutation connecting the $(i-1)^{st}$ and the i^{th} stage ($1 \leq i \leq n$) and an inverse shuffle at the end. The indirect binary n-cube for the case N=16 (n=4) is shown in figure 8. Let C_N denote the class of permutations that can be routed on the indirect binary n-cube. The following result is due to PEASE [41] and PARKER [40]:

Theorem 3.13 $C_N = \Omega_N^{-1} = \rho \circ \Omega_N \circ \rho$.

(ρ is the bit-reversal permutation.) It expresses the intriguing fact that the indirect binary n-cube is topologically equivalent to the "inverse" omega network, which itself is not much different from the omega network (with an added bit-reversal at the beginning and at the end). While the omega network is more regular in topology, the indirect binary n-cube may be handier for designing algorithms (in which the "boxes" do some processing of the data as well). This can be seen from the structure of the network (see figure 8), which suggests an intimate connection to the recursive doubling and divide-and-conquer paradigms. As an example we show that the N-*points FFT* can be evaluated in $O(\log N)$ time, in one pass, through the omega network. Recall that the FFT ("Fast Fourier Transform") can be viewed as a mapping: $(c_0, \ldots, c_{N-1}) \rightarrow (X_0, \ldots, X_{N-1})$ with $X_s = \sum_0^{N-1} c_k \omega^{sk}$ for $0 \leq s \leq N-1$, ω a primitive N^{th} root of unity.

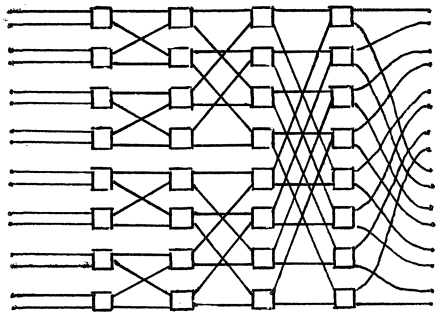
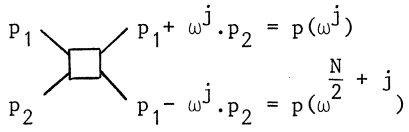


Figure 8.
The indirect binary n-cube for N=16

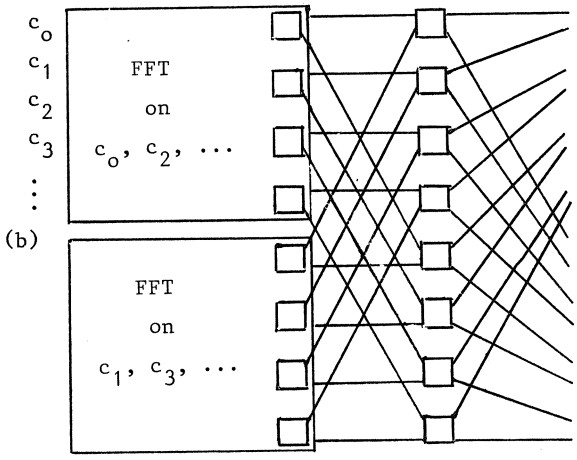
Theorem 3.14 *The* N*-points* FFT *can be evaluated in* $O(\log N)$ *time on* $\rho \circ C_N$ *or, equivalently, on* $\Omega_N \circ \rho$.

Proof.

View the N-points FFT as the problem of evaluating $p(x) = \sum_{0}^{N-1} c_k x^k$ on $\{1,\omega,\omega^2,\ldots,\omega^{N-1}\}$. Write $p(x) = \sum_{0}^{\frac{N}{2}-1} c_{2i} x^{2i} + x.\sum_{0}^{\frac{N}{2}-1} c_{2i+1}x^{2i} = p_1(x^2) + x.p_2(x^2)$, where $p_1(x)$ and $p_2(x)$ are the polynomials of degree $\frac{N}{2} - 1$ corresponding to the even and odd indexed coefficients respectively. It follows that the FFT on N points can be computed from two $\frac{N}{2}$ - points FFT's which produce the necessary values of $p_1(x^2)$ and $p_2(x^2)$. (Note that ω^2 is indeed a primitive $\frac{N}{2}$ th root of unity.) One pair of p_1, p_2 - values will be sufficient to compote both $p(\omega^j)$ and $p(\omega^{\frac{N}{2}+j}) = p(-\omega^j)$.



(a)



(b)

Figure 9. The FFT on N points

Using processing elements as shown in figure 9(a) the N-points FFT can be computed with a network of the recursive structure shown in figure 9(b), which is exactly the indirect binary n-cube. The coefficients must be put in reverse binary order before they can be input to the network. Thus the FFT can be evaluated in log N stages of computation on $\rho \circ C_N$. Using theorem 3.13 it follows that the computation can be scheduled also on $\rho \circ C_N = \rho \circ (\rho \circ \Omega_N \circ \rho) = \Omega_N \circ \rho$. □

A multi-stage network is ideally suited for pipelined computations, with $O(1)$ periods. STONE [54] has shown that Batcher's sorting algorithm on N data items can be implemented to run in $O(\log^2 N)$ time on a shuffle-exchange network, or in log N passes through the omega network. Many other examples of fast algorithms exist. The omega network has been proposed as the underlying interconnection network of the NYU ultracomputer (see e.g. GOTTLIEB *et.al* [16]).

The study of parallel processing algorithms leads to many intricate problems of algorithm design and forces to take all aspects into account that make an algorithm costly when run on a parallel computer.

4. REFERENCES

[1] BATCHER, K.E., *Sorting networks and their application*, in: Proc. AFIPS 1968 SJCC, vol 32, AFIPS Press, Montvale, NJ, pp 307-314.

[2] BÖHM, A.P.W., *Dataflow computation*, Ph.D.Thesis, Dept. of Computer Science, University of Utrecht, Utrecht, 1984.

[3] BORODIN, A., *On relating time and speace to size and depth*, SIAM J.Comput. 6 (1977) 733-744.

[4] BRENT, R.P., *The parallel evaluation of general arithmetic expressions*, J.ACM 21 (1974) 201-206.

[5] BRENT, R.P., and H.T. Kung, *Systolic VLSI arrays for linear time gcd computation*, in: F.Anceau and E.J. Aas (eds.), VLSI '83, Proc IFIP Int.Conf.VLSI, Trondheim 1983, North Holland Publ.Comp., Amsterdam, 1983, pp.145-154.

[6] BUDNIK, P., and D.J. KUCK, *The organisation and use of parallel memories*, IEEE Trans. Comput. C-20 (1971) 1566-1569.

[7] CAREY, M.J. and C.D. THOMPSON, *An efficient implementation of search trees on O(log N) processors*, Report UCB/CSD 82/101, Computer Science Div., University of California, Berkeley, 1982.

[8] CHEN, S., and D.J. KUCK, *Time and parallel processor bounds for linear recurrence systems*, IEEE Trans. Comput. C-24 (1975) 701-717.

[9] COOK, S.A., *Towards a complexity theory of synchronous parallel computation*, L'Enseignement Math. XXVII (1981) 99-124.

[10] COOK, S.A., *The classification of problems which have fast parallel algorithms*, in: M. Karpinski (ed.), Foundations of Computation Theory, Springer Lect.Notes in Computer Sci 158 (1983) 78-93.

[11] CSANKY, L., *Fast parallel matrix iversion algorithms*, SIAM J. Comput 5 (1976) 618-623.

[12] DIJKSTRA, E.W., *Cooperating sequential processes*, in: F.Genuys (ed.), Programming Languages, Acad.Press, New York, NY, 1968, pp.43-112.

[13] FLYNN, M.J., *Some computer organisations and their effectiveness*, IEEE Trans. Comput. C-21 (1972) 948-960.

[14] FRIEDLAND, D., *Taxonomy of parallel sorting*, Techn. Rep.CS 82-08, Dept. of Applied Math, the Weizmann Inst. of Science, Rehovot, Israel, 1982.

[15] GOLDSCHLAGER, L.M., *A universal interconnection pattern for parallel computers*, J.ACM 29 (1982) 1073-1086.

[16] GOTTLIEB, A., et.al., *The NYU ultracomputer - designing an MIMD shared memory parallel computer*, IEEE Trans. Comput. C-32 (1983) 175-189.

[17] GREENBERG, A.C., R.E. LADNER, M.S. PATERSON, and Z. GALIL, *Efficient parallel algorithms for linear recurrence relations*, Inf. Proc. Lett. 15 (1982) 31-35.

[18] HEDAYAT, A., *A complete solution to the existence and non-existence of Knut Vik designs and orthogonal Knut Vik designs*, J.Combin. Th., Ser. A., 22 (1977) 331-337.

[19] HELLER, D., *A survey of parallel algorithms in numerical linear algebra*, SIAM Rev. 20 (1978) 740-777.

[20] HOARE, C.A.R., *Communicating sequential processes*, C.ACM 21 (1978) 666-677.

[21] HOCKNEY, R.W., *A fast direct solution of Poisson's equation using Fourier analysis*, J.ACM 12 (1965) 95-113.

[22] HOCKNEY, R.W., and C.R. JESSHOPE, *Parallel computers*, A. Hilger Ltd, Bristol, 1981.

[23] HOROWITZ, E., and A. ZORAT, *Divide-and-conquer for parallel processing*, IEEE Trans. Comput. C-32 (1983) 582-585.

[24] HWANG, K., S.P. SU, and L.M. NI, *Vector computer architecture and processing techniques*, in: M.C.Yovits (ed.), Advances in Computers, vol. 20, Acad. Press., New York, NY, 1981, pp.115-197.

[25] HYAFIL, L., *On the parallel evaluation of multivariate polynomials*, SIAM J. Comput. 8 (1979) 120-123.

[26] KINDERVATER, G.A.P., and J.K. LENSTRA, *Parallel algorithms in combinatorial optimization: an annotated bibliograohy*, Techn. Rep. Mathem. Centre, Amsterdam, 1983.

[27] KNUTH, D.E., *The art of computer programming, vol 1: Fundamental algorithms*, Addison-Wesley Publ. Comp., Reading, Mass, 1968.

[28] KOGGE, P.M., and H.S. STONE, *A parallel algorithm for the efficient solution of a general class of recurrence equations*, IEEE Trans. Comput. C-22 (1973) 786-793.

[29] KOSARAJU, S.R., *Fast parallel processing array algorithms for some graph problems*, 1979, pp. 231-236.

[30] KRAMER, M.R., and J. VAN LEEUWEN, *Systolic computation and VLSI*, in: J.W. de Bakker and J. van Leeuwen (eds.), Foundations of Computer Science IV, part 1, Math. Centre Tracts 158, Mathem. Centre, Amsterdam, 1983, pp. 75-103.

[31] KUÇERA, L., *Parallel computation and conflicts in memory access*, Inf. Proc.Lett.14 (1982) 93-96.

[32] KUCK, D.J., *Illiac IV software and applications programming*, IEEE Trans. Comput. C-17 (1968) 758-770.

[33] KUNG, H.T., *New algorithms and lower bounds for the parallel evaluation of certain rational expressions*, Proc. 6th Annual ACM Symp. Theory of Computing, 1974, pp. 323-333.

[34] KUNG, H.T., *Let's design algorithms for VLSI systems*, Techn. Rep. CMU-CS-79-151, Dept. of Computer Science, CarnegieMellon University, Pittsburgh, Pa., 1979.

[35] LAWRIE, D.H., *Access and alignment of data in an array processor*, IEEE Trans. Comput. C-24 (1975) 1145-1155.

[36] LEVIALDI, S., *On shrinking binary picture patterns*, C.ACM 15 (1972) 7-10.

[37] LORIN, H., *Parallelism in hardware and software: real and apparent concurrency*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1972.

[38] MADSEN, N.K., G.H. RODRIGUE, and J.I. KARUSH, *Matrix multiplication by diagonals on a vector/parallel processor*, Inf. Proc. Lett. 5 (1976) 41-45.

[39] MUNRO, I., and M. PATERSON, *Optimal algorithms for parallel polynomial evaluation*, J.Comput. Syst.Sci 7 (1973) 189-198.

[40] PARKER, D.S., *Notes on shuffle/exchange-type switching networks*, IEEE Trans. Comput. C-29 (1980) 213-222.

[41] PEASE, M.C., *The indirect binary n-cube microprocessor array*, IEEE Trans. Comput. C-26 (1977) 458-473.

[42] PERROTT, R.H., *A language for array and vector processors*, ACM ToPlaS 1 (1979) 177-195.

[43] PREPARATA, F.P., and D.V. SARWATE, *An improved parallel processor bound in fast matrix inversion*, Inf. Proc. Lett. 7 (1978) 148-150.

[44] QUINN, M.J., and N. DEO, *Parallel algorithms and data structures in graph theory*, Techn. Rep. CS-82-098, Computer Science Dept., Washington State University, Pullman, Wash., 1982.

[45] RODRIGUE, G.H., N.K. MADSEN, and J.I. KARUSH, *Odd-even reduction for banded linear equations*, J.ACM 26 (1979) 72-81.

[46] SAMEH, A., *An overview of parallel algorithms in numerical linear algebra*, EDF Bull. Direct. Etud. Rech., Ser. C. No.1, 1983, pp. 129-134.

[47] SAMEH, A., and R.P. BRENT, *Solving triangular systems on a parallel computer*, SIAM J. Numer. Anal. 14 (1977) 1101-1113.

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

[49] SAVITCH, W.J. and M.J. STIMSON, *Time bounded random access machines with parallel processing*, J.ACM 26 (1979) 103-118.

[50] SCHWARTZ, J., *Large parallel computers*, J.ACM 13 (1966) 25-32.

[51] SHAPIRO, H.D., *Generalized latin squares on the torus*, Discr. Math. 24 (1978) 63-77.

[52] SKYUM, S., and L.G. VALIANT, *Fast parallel computation of polynomials using few processors*, in: J. Gruska and M. Chytil (eds.), Mathematical Foundations of Computer Science 1981, Springer Lect. Notes in Computer Sci 118 (1981) 132-139.

[53] STEINBERG, D., *Invariant properties of the shuffle-exchange and a simplified cost-effective version of the omega network*, IEEE Trans. Comput. C-32 (1983) 444-450.

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

[55] STONE, H.S., *An efficient parallel algorithm for the solution of a tri-diagonal linear system of quations*, J.ACM 20 (1973) 27-38.

[56] STONE, H.S., *Parallel computers*, in: H.S. Stone (ed), Introduction to computer architecture, SRA Inc., Chicago, III, 1980 (2^{nd} ed.), pp. 363-425.

[57] SWEET, R.A., *A cyclic reduction algorithm for solving blocktridiagonal*

systems of arbitrary dimension, SIAM J. Numer. Anal. 14 (1977) 706-719.

[58] TODD, S., *Algorithm and hardware for a merge sort using multiple processors*, IBM J.Res.Develop. 22 (1978) 509-517.

[59] VALIANT, L.G., *Parallelism in comparison problems*, SIAM J.Comput 3 (1975) 348-355.

[60] VAN LEEUWEN, J., *Distributed computing*, in J.W. de Bakker and J. van Leeuwen (eds.), Foundations of Computer Science IV, part 1, Math. Centre Tracts 158, Mathem. Centre, Amsterdam, 1983, pp. 1-34.

[61] VAN LEEUWEN, J., and H.A.G. WIJSHOFF, *Data mappings in large parallel computers*, Techn. Rep. RUU-CS-83-11, Dept. of Computer Science, University of Utrecht, Utrecht, 1983.

[62] VISHKIN, U., *Implementation of simultaneous memory address access in models that forbid it*, J. Algor. 4 (1983) 45-50.

[63] WIJSHOFF, H.A.G. and J. VAN LEEUWEN, *On linear skewing schemes and d-ordered vectors*, Techn. Rep. RUU-CS-83-7, Dept. of Computer Science, University of Utrecht, Utrecht, 1983.

[64] WU, C., and T. FENG, *The reverse-exchange interconnection network*, IEEE Trans. Comput. C-29 (1980) 801-811.

[65] WU, C., and T. FENG, *The universality of the shuffle-exchange network*, IEEE Trans. Comput. C-30 (1981) 324-332.

Comparative Performance Tests of Fortran Codes on the Cray-1 and Cyber 205

H.A. van der Vorst
University of Technology, Delft

ABSTRACT

The supercomputers CRAY-1 and CYBER 205 are among the most powerful numbercrunchers that are commercially available at the moment. Both types are so-called vector computers (pipelined processors) and therefore very well suited for many linear algebra computations. It has been recognized widely that most well-known algorithms have been designed for serial (or scalar) computers and that therefore one usually has to reformulate them, or to replace them by more suitable algorithms, in order to exploit the capabilities of the vector computers.

In spite of the evident similarities of both the CRAY-1 and CYBER 205, their mutual differences are so substantially that the selection of an algorithm for a given computational task and its Fortran implementation may be quite different for each of both supercomputers if one seeks for optimal performance (this may be even so for different configurations of the same computer type).

In this contribution we consider some relevant features of both supercomputers and we discuss their effects on the approach of a number of rather basic problems in numerical linear algebra in a Fortran programming environment.

1. INTRODUCTION

In the course of 1984 the Dutch scientific community will get the possibility of access to a CYBER 205 (SARA-Amsterdam) as well as to a CRAY-1 (SHELL-Rijswijk). This motivated us to collect some experience on both computers in order to be able to support future users of these systems.

Of course, much testing experience has been reported already (e.g., see [1,2,3,4]), mostly for more complex algorithms or even for complete Fortran codes. On relevant places we will refer to these results. In [5] it is shown in detail how two specific Fortran codes have been modified in order to achieve better execution rates for the CRAY-1 as well as for the CYBER 205.

In the coming sections we will consider the approach of a number of very elementary linear algebra algorithms rather than more complex ones and we will restrict ourselves to Fortran implementations of these algorithms. As we will demonstrate, the performance of the implementations of these algorithms may differ largely for each supercomputer, depending on their basic features. These features will be described globally in section 2. In section 3 we discuss some possible Fortran implementations of basic linear algebra algorithms and we analyse their performance, as it is observed in actual computation.

Finally we consider in section 4 how this works out for a more complex algorithm, namely the preconditioned conjugate gradient method. We will then also demonstrate that the differences between both computers, though classified in the same group, may lead to a different choice of algorithm for solving a given problem on each of them.

2. SOME FEATURES OF THE CRAY-1 AND THE CYBER 205

Before we consider a number of algorithms in more detail, some of the features found in both supercomputers will be described. We will restrict ourselves to those features that are relevant for the Fortran implementation of numerical algorithms.

Both the CRAY-1 and the CYBER 205 share a number of properties, by which they can be distinguished from other architectures:

- vector instructions, i.e., vectors can act as operands for some functional units.
- segmented functional units (pipelined processors), by which it is possible to deliver a result of certain vector operations each clock cycle (after a start-up time, which depends on the functional unit).
- overlap of vector instructions, e.g., Load V1 from memory and V0=V1+V2 can be executed almost simultaneously.
- 64 bits words (floating point 48 bits precision).
- 8 or 16 bank memory (important with respect to memory bank conflicts).

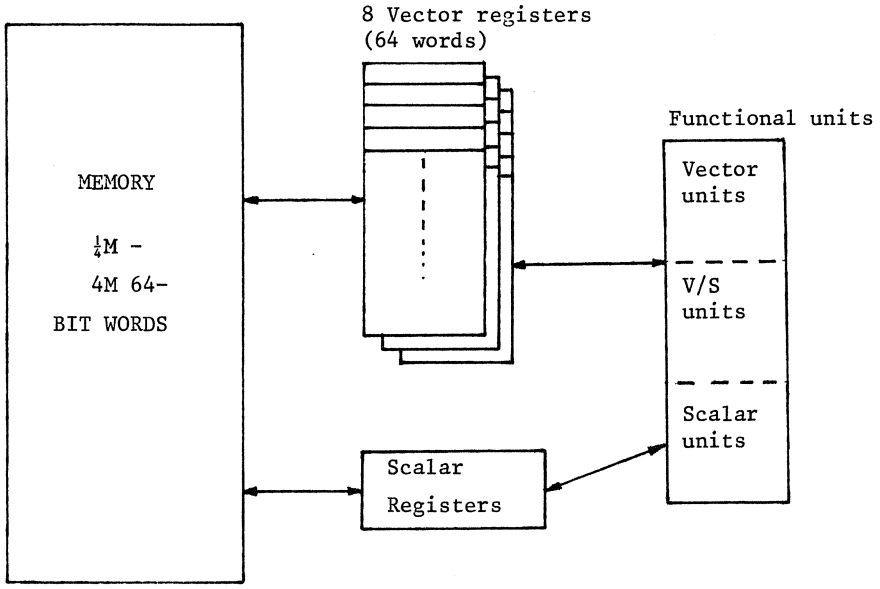


Figure 1. CRAY-1 Vector Register Concept

On the other hand they have a number of different features and some of these may require different implementations in Fortran in order to be exploited:

CRAY-1	CYBER 205
- clock cycle time 12.5 ns	- clock cycle time 20 ns
- bank cycle time 50 ns	- bank cycle time 80 ns
- 8 vector registers (64 words each, see figure 1)	- 1, 2 or 4 vector pipes (figure 2)
	- direct memory access (figure 2)
- only 1 path to memory (figure 3) (store cannot overlap any operation using the same register)	- 3 paths to memory: 2 loads and 1 store (figure 4)
- stride (constant)	- contiguous vectors (stride=1)
- chaining: e.g., Load V0, V2=V1+V0, V3=V2xS simultaneously (figure 5)	- linked triad capability
- vector code can be generated by special directives in comment lines	- vector code can be obtained using CDC Fortran vector syntax (extension to Fortran)

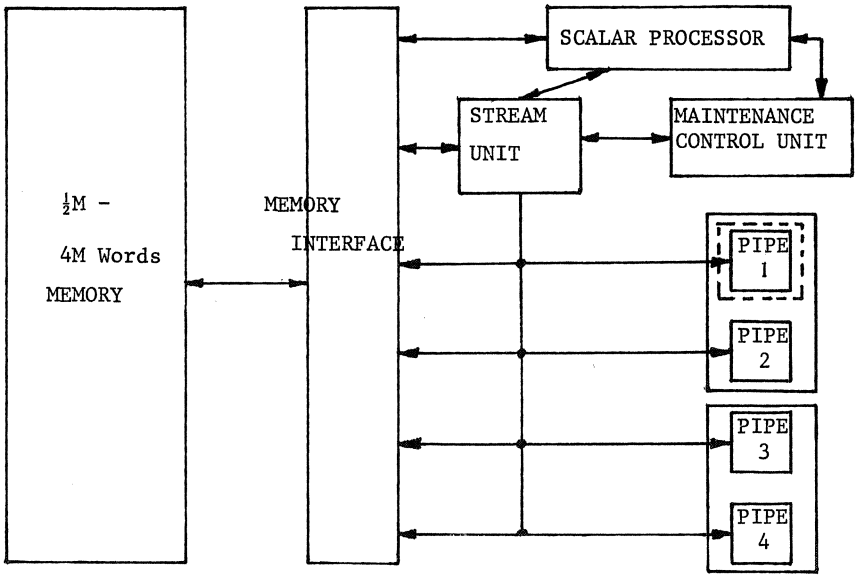


Figure 2. CYBER 205 Vector pipes and Direct Memory Access

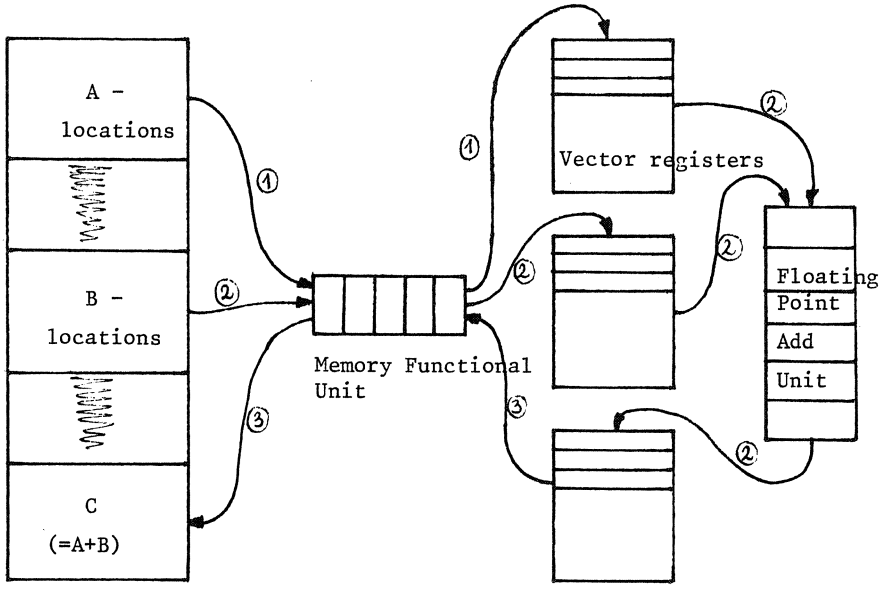


Figure 3. CRAY-1 Chaining: operations with the same number can overlap

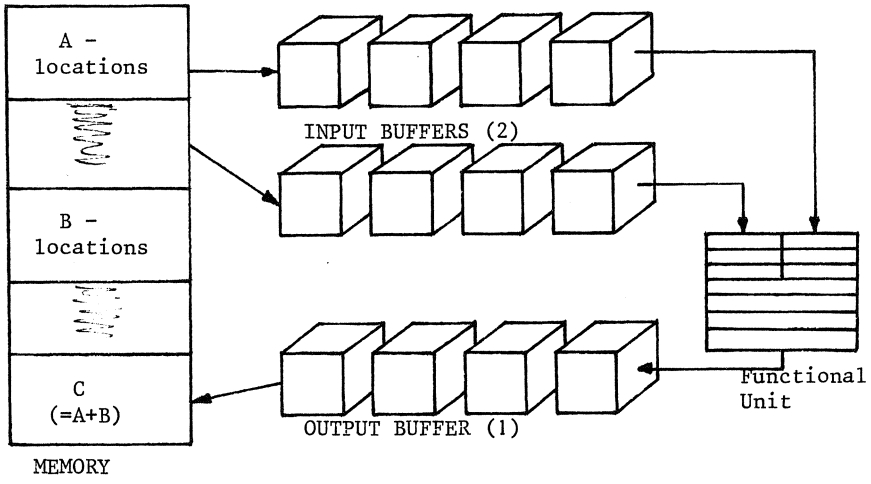


Figure 4. CYBER 205, 3 Paths to memory

A common measure for the speed of vector computers is the amount of MFLOPS that can be achieved (MFLOPS = 10^6 floating point operations per second; in our case the operations are add and multiply).

Since the add and multiply vector operations can almost completely overlap, the maximum execution rate for the CRAY-1 is 160 MFLOPS. For the CYBER 205 the maximum execution rate depends on the number of available vector pipes: 1-pipe 100 MFLOPS, 2-pipe 200 MFLOPS and 4-pipe 400 MFLOPS. Most experience reported in litterature is concerned with the 2-pipe version. It is also possible to execute floating point operations on the CYBER 205 in half precision (32-bit mode), this doubles the maximum possible execution rates.

As we will see it is not so easy to achieve the maximum execution rate for the CRAY-1 with Fortran code. The maximum rates for the CYBER 205 are only achieved for linked triads (see 3.1). In practice the terms scalar speed, vector speed and super vector speed are frequently used in order to classify the actual performance for (a part of) a code.

Super vector speed is reached when in average at least one functional floating point unit is constantly in use (e.g., for the CRAY-1: 50 - 150 MFLOPS). Vector speed is achieved when in average at most 1 floating point result is delivered each clockcycle, in situations where the vector instructions dominate (e.g., for the CRAY-1: 10 - 50 MFLOPS). Scalar speed results when almost no vector instructions are issued (e.g., for the CRAY-1: less

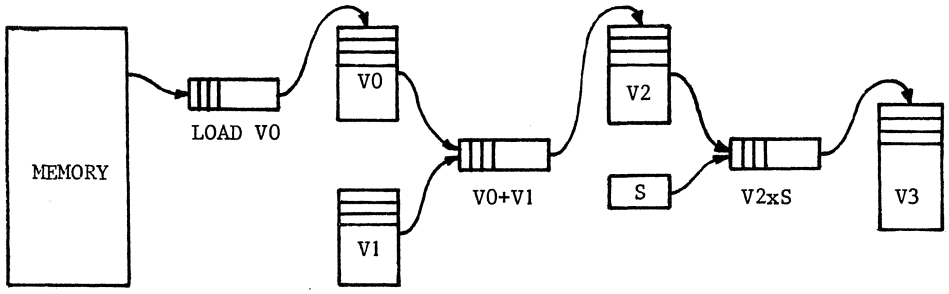


Figure 5. CRAY-1 Chaining Concept: Assuming that V1 and scalar S have been loaded previously, the 3 operations execute in parallel.

than 10 MFLOPS). Due to loop overhead and start-up times, the above given example rates are slightly less than their theoretical values. In order to get some feeling for the actual performance of a supercomputer we refer to DUFF and REID [6], who conclude from their investigations that 30 MFLOPS may be considered as a good rate for Fortran code and 100 MFLOPS is a good rate for assembler code (both rates are for the CRAY-1).

In the analysis of the performance of algorithms on vector computers, the term chime is often used. One chime denotes the number of clock cycles required to execute a given vector instruction, apart from start-up time. E.g., the vector instruction V0=V1+V2 takes 1 chime (assuming that V1 and V2 are available from vector registers on the CRAY-1).

Since many testing results do depend on variables such as compiler version, we give here the main characteristics of the testing circumstances.

CRAY-1A : ECMWF, Shinfield Park, Reading, U.K.
16-Bank Memory
Operating System COS 1.12
Fortran Compiler CFT 1.11, options ON=CELMPQRSTV

CYBER 205 : Control Data France, Paris, France (accessed through Control Data CYBERNET)
2-pipe, 8-Bank Memory
Operating System VSOS V20L575H
Fortran Compiler FORTRAN 2.1 Cycle OTS21N, options O=BOUV

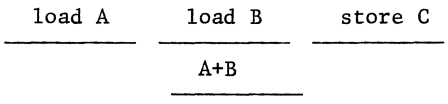
3. AN ANALYSIS OF SOME SIMPLE ALGORITHMS

3.1 Vector Summation

We consider the summation of two vectors as defined by

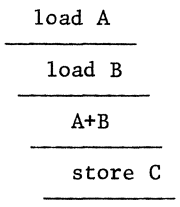
```
      DO 10 I=1,N
      C(I)=A(I)+B(I)
   10 CONTINUE
```

For the CRAY-1 the execution of the instructions generated by the Fortran compiler can be represented schematically as:



and we see that this operation takes 3 chimes, or more precisely, for $N \leq 64$, 29+3N clockcycles.

On the CYBER 205 these instructions can be executed in parallel:



which takes only 1 chime, or more precisely, $51+N/n_p$ clockcycles, where n_p denotes the number of vector pipes (1, 2 or 4).

It follows that, though the number of chimes for the CYBER 205 is 1/3 of that for the CRAY-1, the 1-pipe CYBER 205 only beats the CRAY-1 when N is larger than 37. For the 2-pipe CYBER 205 the turnoverpoint is N=23. We see for this rather extreme example that the CYBER 205 is slower than the CRAY-1 for short vectors, but (much) faster for (very) long vectors.

3.2 Linked Triads

A linked triad is an operation involving 2 vectors, 1 scalar, 1 add, 1 multiplication and with a vector result, like

```
      DO 10 I=1,N
      C(I)=A(I)+B(I)*D
   10 CONTINUE
```

This type of operation plays a role in, e.g., gaussian elimination, the updating of vectors in iterative processes, etcetera.

On the CRAY-1, with the present compiler, the execution of the above statements can be represented in chimes as follows

load B	load A	A+DxB()	store C
DxB()			

Though the add operation is a candidate for chaining with the load operation of A, it has to wait for the result of DxB(), which slightly overlaps the load of A. Since the possibility of chaining is only checked at the time that the first result of a vector operation comes available, and since the DxB() overlap is just beyond that point, the add instruction comes too late and has to waite until the load of A has been completed, therefore 4 chimes. Though in principal this situation could be detected during the compilation and then could be easily repaired, e.g., by issuing one superfluous instruction immediately before the load A instruction, this is not done by the present compiler. However, the instruction sequence can be changed by inserting parentheses:

```
      DO 10 I=1,N
      C(I)=(A(I))+B(I)*D
10    CONTINUE
```

which leads to

load A	load B	store C
	B()xD	
	A+B()xD	

: 3 chimes.

These 3 chimes involve 2 floating point vector operations and thus the maximum execution rate is (2/3)x80=53 MFLOPS. The memory references in this case (and similar situations) form the bottleneck and for this reason the CRAY-1 is often called memory limited.

The CYBER 205 offers the possibility of 3 simultaneous memory references and this leads to the execution sequence that is represented schematically as follows

load B
load A
*
+
store C

The result of the multiply is sent directly to the add unit without being stored in memory

: 1 chime (!)

Here there are 2 floating point vector operations in 1 chime and hence the maximum execution rate for the CYBER 205 is n_px100 MFLOPS (n_p=1,2 or 4).

3.3 The Innerproduct

We consider the following Fortran statements, representing the computation of an innerproduct:

```
      SUM=0.
      DO 10 I=1,N
      SUM=SUM+A(I)*B(I)
10    CONTINUE
```

The summation of vector elements is not a vector operation by itself, but if the vectors are long enough, then the summation can be arranged in parts and rather fast (vector) code can be generated for the CRAY-1, taking only 2 chimes, see JORDAN [1]. This has been realised for the CRAY-1 by the subroutine SDOT (see [12]).

For the CYBER 205 rather fast code could be obtained by a Folding Technique as described by SCHREIBER and TANG [7]: first compute $c_i=a_i x b_i$ (a vector operation), then add the second half part of c to the first half part and do this repeatedly until some minimum vector length is reached, the remaining part is summed in scalar mode. It can be shown that code based on this technique could compute the innerproduct in 2 chimes and therefore could have an execution rate of n_px50 MFLOPS (n_p = the number of vector pipes). Actually innerproduct computations on the CYBER 205 can be done with a special hardware instruction, which can be generated by a call to Q8SDOT (see [13]).

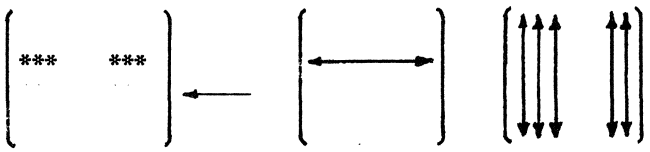
The Fortran compilers of both computers recognize the above Fortran statements and replace it by the appropriate in-line vector code. However, for N=5000 the CRAY-1 achieves an execution rate of 29 MFLOPS for the Fortran code, whereas SDOT does the computation with a rate of 72 MFLOPS. The difference is due to some rather strange overhead generated by the compiler in this situation.

For the CYBER 205 the actual execution rate for the above Fortran code, which is replaced by the compiler by an inline special hardware instruction, for N=5000, is 95 MFLOPS for the 2-pipe model. If one replaces the code by the corresponding call to Q8SDOT then the rate is precisely the same. We observe that this rate is quite close to what one might ultimately expect when using the Folding Technique.

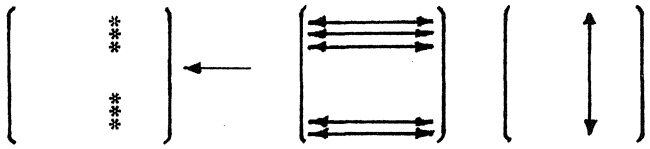
3.4 Matrix-Matrix Multiplication

There are 6 possibilities to carry out the full matrix multiplication A=BxC, where A, B and C are assumed to be full square matrices of order N. We will describe these possibilities graphically by diagrams (for details see DONGARRA [8]) and comment on them briefly. In the diagrams * denotes a scalar operand, ←⟶ denotes a vector operand.

a.



b.

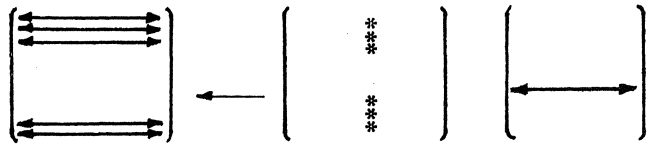


A ← B x C

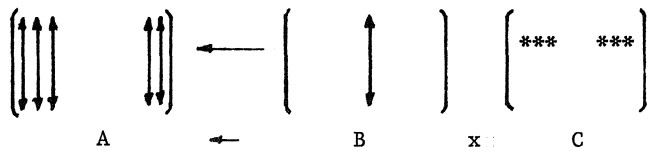
CRAY-1: The performance of a and b will be at vectorspeed at most, because of the innerproducts. Possibly there can be memory bank conflicts, depending on the rowdimension of B.

On the CYBER 205 the performance will be low (scalar speed), because of the stride unequal to 1 in the required elements of B.

c.



d.

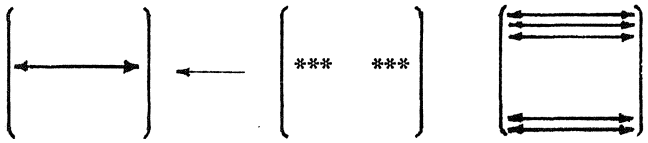


A ← B x C

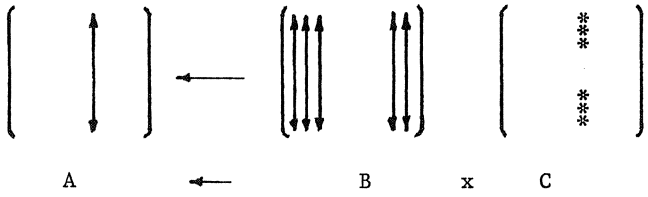
CRAY-1 : In both cases c and d the entire matrix A is loaded and stored in each outer loop and therefore vector speed at most.

CYBER 205: The rows in c degrade the performance (stride $\neq$ 1). Because of linked triads d is potentially fast.

e.



f.



CRAY-1 : Because of the column structure in f, this possibility should be preferred. The Fortran compiler does not recognize the possibility of accumulating the result vector in a vector register, before storing it. Therefore vector speed at most. Super vector speed can be reached either in CRAY Assembler Language or by a loop unrolling technique as proposed by DONGARRA and EISENSTAT [9], in that case performance rates close to 150 MFLOPS are feasible. The loop unrolling technique will be explained in 3.5.

CYBER 205 : For e we may again expect only scalar speed (stride $\neq$ 1), but f can be done at super vector speed (linked triads)

For both computers CRAY-1 and 2-pipe CYBER 205 we list the actually observed execution rates in MFLOPS in Table I.

N	matrix mult. version	CRAY-1	CYBER 205
100	a	38	5.7
	f	34	55
200	a	37	5.8
	a, with row dim. 201	49	
	f	37	86
300	a	57	5.8
	f	41	106

Table I. Execution rates in MFLOPS for the matrix multiplication

3.5 Matrix-Vector Product

This operation is in fact the kernel in the matrix multiplication as described in 3.4f. We consider the following Fortran statements, which do the matrix vector multiplication:

```
      DO 10 I=1,N
      Y(I)=0.
10    CONTINUE
      DO 30 J=1,N
      DO 20 I=1,N
20    Y(I)=(Y(I))+X(J)*A(I,J)
30    CONTINUE
```

Since X(J) is a constant in the innermost DO-loop (DO 20 loop), the expression there is a linked triad. As is shown in 3.2 the linked triad takes 3 chimes on a CRAY-1 and therefore the performance will be bounded by 53 MFLOPS, whereas the 2-pipe CYBER 205 has its execution rate in this case bounded by 200 MFLOPS. In actual computation the CRAY-1 achieves 41 MFLOPS and the CYBER 205 achieves 106 MFLOPS, for N=300.

As is mentioned in 3.4 the CRAY-1 Fortran compiler does not recognize the possible savings in loads and stores for the Y-vector. DONGARRA and EISENSTAT [9] propose to unroll the DO 30 - loop, e.g., for a depth of four:

```
      DO 30 J=4,N,4
      DO 20 I=1,N
      Y(I)=((((Y(I))+A(I,J-3)*X(J-3))+A(I,J-2)*X(J-2))+
     $     A(I,J-1)*X(J-1))+A(I,J)*X(J)
20    CONTINUE
30    CONTINUE
```

Since the X-elements act as scalars in the DO 20 - loop, there are now 6 memory references (for vectors) for each 8 floating point vector operations and therefore the maximum execution rate is 107 MFLOPS. In actual computation a rate of 96 MFLOPS has been observed, for N=300 and an unrolling depth of 16, see [9]. In CAL (Cray Assembler Language) the execution rate can be as high as 148 MFLOPS (for N=300, using the CAL-coded subroutine MXV, see [12]).

3.6 Band Matrix-Vector Products

As an example we consider the case that A is a pentadiagonal symmetric matrix of order 2000. The three relevant non-zero diagonals of the upper triangular part of A are denoted by a(i,1), a(i,2) and a(i,3), respectively,

where i is counted rowwise. The typical statement to be executed for the matrix-vector product b=Ax looks like

```
b(i)=a(i-2,3)*x(i-2)+a(i-1,2)*x(i-1)+a(i,1)*x(i)+
     a(i,2)*x(i+1)+a(i,3)*x(i+2) .
```

Because of the sparsity of A, an approach similar as in 3.5 is not very useful here and an approach that exploits the diagonalwise structure has to be preferred [15]. The CYBER 205 Fortran compiler apparantly recognizes that this vector statement can be done in 9 chimes on this machine, whereas the CRAY-1 Fortran compiler does not detect that it can be done in 11 chimes on the CRAY-1 and generates a code that requires 14 chimes. This explains that the CRAY-1 achieves 46 MFLOPS and the CYBER 205 (2-pipe) about $1.25 \times \frac{14}{9}$ times as much: 88 MFLOPS (for N=2000).

The CRAY-1 Fortran compiler can be forced to generate better code by inserting parentheses:

```
      N2=N-2
      DO 10 I=3,N2
      B(I)=((((A(I-2,3)*X(I-2))+A(I-1,2)*X(I-1))+
     $     A(I,1)*X(I))+A(I,2)*X(I+1))+A(I,3)*X(I+2)
   10 CONTINUE
```

The execution now takes 11 chimes and the actual observed execution rate is 58 MFLOPS (for N=2000).

A vector register serves as an operand to the vector functional units and we cannot access the individual elements in such a register. Therefore the same register cannot be used to deliver both X(I-2) and X(I-1) in the above statement (i.e., in Fortran this is not possible yet, in CAL one could use vector shift instructions in this case, see JORDAN [1]).

Note, however, that the register which contains the vector X(I-1) could act as an operand in the computation of B(I+1). This is also the case for the registers containing A(I,2), X(I), X(I+1) and X(I+2). For the CRAY-1 this can be used in order to reduce the number of vector loads, by unrolling the DO 10 - loop:

```
      DO 10 I=3,N2,2
      B(I)= . . .
      B(I+1)= . . .
   10 CONTINUE
```

Since there are only 8 vector registers available for storing the required vectors and the intermediate results, one has to rearrange the order of computation very carefully and then it appears to be possible to get the desired code (we have not considered the general m-diagonal case):

```
      DO 10 I=3,N2,2
      B(I)=((((A(I-2,3)*X(I-2))+A(I-1,2)*X(I-1))+A(I,1)*X(I))+
     $     A(I,2)*X(I+1))+A(I,3)*X(I+2)
      B(I+1)=((((A(I,2)*X(I))+A(I+1,2)*X(I+2))+A(I+1,1)*X(I+1))+
     $       A(I-1,3)*X(I-1))+A(I+1,3)*X(I+3)
   10 CONTINUE
```

Now a code is generated for the CRAY-1 that takes 17 chimes for each 18 vector floating point operations and therefore super vector speed may be expected. And indeed, actually we observe an execution rate of 68 MFLOPS.

In this case the CRAY-1 features can not optimally be used in Fortran, as they can be in Cray Assembler Language. JORDAN [1] describes a technique for similar sparse matrix vector products that could lead to MFLOPS rates of 100 and more.

3.7 The Vector Formula $b_i=x_i-a_i*(x_{i-1}-a_{i-1}*x_{i-2})$

On the CYBER 205 this vector statement requires 4 chimes for execution, as is schematically given by:

load x_{i-2}	load x_{i-1}	load a_i	load x_i
load a_{i-1}	load R_1	load R_1	load R_1
*	-	*	-
store R_1	store R_1	store R_1	store b_i

There are 4 vector floating point operations in these 4 chimes and hence the maximal execution rate for the 2-pipe CYBER 205 will be 100 MFLOPS. The actually observed rate for N=1000 is 83 MFLOPS.

The CRAY-1 Fortran compiler generates a code that takes 7 chimes:

load x_{i-2}	load a_{i-1}	load x_{i-1}	-	load x_i	-	store b_i
	*		load a_i			
			*			

Both add instructions can not be chained to the previous loads due to overlap problems (see also 3.2). In CAL this could be remedied, e.g., by inserting one instruction immediately before the load instruction, which reduces the sequence to 6 chimes.

There are 4 vector floating point operations for 7 chimes in Fortran and therefore the execution rate is bounded by $\frac{4}{7}$x80=46 MFLOPS. For N=1000 we actually observe 40 MFLOPS. By loop-unrolling, as in 3.6, the number of vector loads can be reduced on the CRAY-1:

```
      DO 10 I=3,N,2
      B(I)=(X(I))-A(I)*((X(I-1))-A(I-1)*X(I-2))
      B(I+1)=(X(I+1))-A(I+1)*(X(I)-A(I)*X(I-1))
10    CONTINUE
```

It is left to the reader to check that this can be carried out in 9 chimes (of vector length N/2). The actually observed execution rate in this case is 58 MFLOPS, for N=1000.

For most scalar computers relatively fast code is generated for:

```
      DO 10 I=3,N
      X1=X(I)
      AN=A(I)
      B(I)=X1-AN*(X2-AP*X3)
      X3=X2
      X2=X1
      AP=AN
10    CONTINUE
```

(with the appropriate initialisations, of course).

For the CYBER 175-100 this scalar code takes 1820 microsec., whereas the original "vector" statement requires 2120 microsec.,for N=1000. On the CRAY-1 the "scalar" loop costs 577 microsec., the CYBER 205 (2-pipe) takes 598 microsec. and we observe that in scalar mode both supercomputers are about equally fast, though the central processor of the CRAY-1 is 1.6 times faster than the CYBER 205 central processor.

4. VECTORIZATION OF A COMPLICATED ALGORITHM

In this section we consider the vectorization of the ICCG algorithm (vectorization of this algorithm has been considered many times, see, e.g., [1,16,17]). The Incomplete Choleski Conjugate Gradient algorithm for the iterative solution of the linear system Ax=b arises when the conjugate gradient method is applied to the preconditioned system $K^{-1}Ax=K^{-1}b$, where K is an incomplete Choleski decomposition of the symmetric positive real matrix A [10]. As a model problem we will consider the linear system Ax=b, where A has the structure as shown in figure 6.

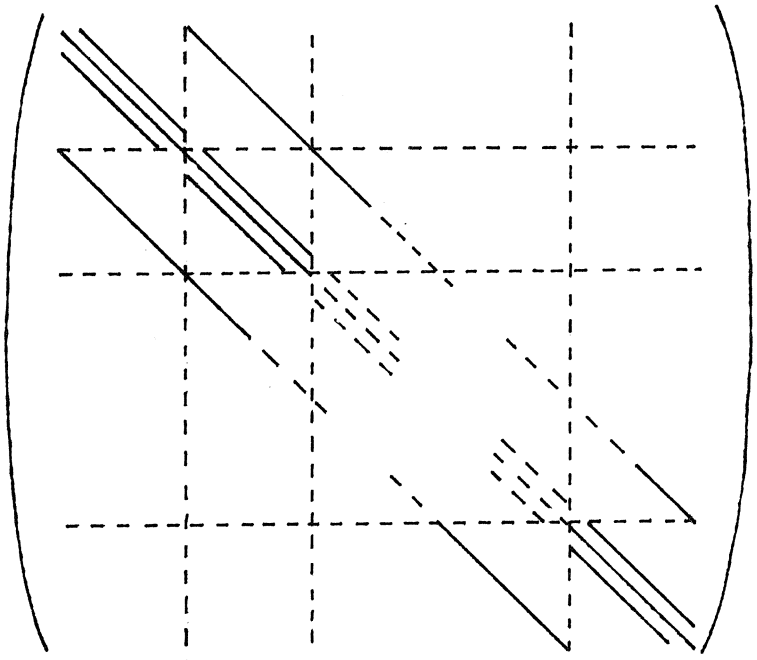


Figure 6. Structure of A

A very simple incomplete decomposition of A is defined by the splitting $A = L\,D^{-1}\,L^T - R$, where

(i) the strictly lower triangular part of L is equal to the corresponding part of A

(ii) $\mathrm{diag}(L\,D^{-1}\,L^T) = \mathrm{diag}(A)$ (4.1)

(iii) $\mathrm{diag}(L) = D$, D is a diagonal matrix.

Let the diagonal elements of A be denoted by $a_{i,1}$, the first codiagonal elements to the right by $a_{i,2}$ and the elements of the m-th codiagonal to the

right by $a_{i,3}$. We can partition the unknown vector x in parts, each part consisting of m successive unknowns, corresponding to the blockwise structure of the matrix; the blocks of the matrix A are also of dimension m.

From (4.1) the elements of D can be easily computed, for details see [10,11]. As a preconditioning matrix for A we choose K^{-1}, where $K = L\,D^{-1}\,L^T$. At this point we note that it is not necessary to compute K^{-1} explicitly, we only need an algorithm that generates the vector $K^{-1}z$, for any given z. In order to improve the efficiency of the ICCG algorithm we scale the matrix A in such a way that D = I. The ICCG algorithm is given by the following scheme.

$$
\begin{aligned}
&x_0 \text{ is an arbitrary initial approximation to } x\\
&r_0 = b - Ax_0\ ,\quad p_0 = K^{-1}r_0\\
&\text{for } i = 0,1,2,\ldots \quad \text{until } \|r_{i+1}\|_2 < \varepsilon\text{:}\\
&\alpha_i = \frac{(r_i, K^{-1}r_i)}{(p_i, Ap_i)}\\
&x_{i+1} = x_i + \alpha_i p_i\ ,\quad r_{i+1} = r_i - \alpha_i Ap_i\\
&\beta_i = \frac{(r_{i+1}, K^{-1}r_{i+1})}{(r_i, K^{-1}r_i)}\\
&p_{i+1} = K^{-1}r_{i+1} + \beta_i p_i
\end{aligned}
\qquad (4.2)
$$

The innerproducts (see 3.3), vector updates (see 3.2) and the computation of Ap_i (see 3.6) have been discussed already and they introduce no special vectorization problems. The bottleneck in this algorithm, with respect to vectorization, is the computation of $K^{-1}r_{i+1}$ in the i-th step. The computation of D in (4.1) is also not easily vectorizable (but it can be done, see e.g., JORDAN [1]). It has been neglected here because it has to be done once and it takes relatively only little computing time

We will now consider three different approaches to vectorizing the computation of $z=K^{-1}y'$ for a given input vector y. The output vector z is computed by solving Kz=y, which is done in two successive steps:

1. compute $\tilde{z}$ from $L\tilde{z}=y$, by forward elimination
2. compute z from $L^T z=\tilde{z}$, by backward elimination.

4.1 Straight-forward Computation of z

If we assume all vectors to be partioned in n successive parts of length m (corresponding to the block structure of A), then the components of $\tilde{z}$, in the k-th part, can be computed from

$$\tilde{z}_j = y_j - a_{j-1,2}\tilde{z}_{j-1} - a_{j-m,3}\tilde{z}_{j-m} \tag{4.3}$$

with $j = (k-1)*m+1, (k-1)*m+2, \ldots, k*m$.

Since the $\tilde{z}_{j-m}$ represent the already computed elements in the (k-1)-th part, the piece $\tilde{y}_j = y_j - a_{j-m,3}\tilde{z}_{j-m}$ vectorizes, so that we are left with the problem of computing $\tilde{z}_j$ from

$$\tilde{z}_j = \tilde{y}_j - a_{j-1,2}\tilde{z}_{j-1} \quad . \tag{4.4}$$

This is a forward recursion so that vectorization is inhibited, however optimized code for (4.4) is available for both the CRAY-1 (subroutine FOLR, see [12]) and the CYBER 205 (subroutine Q8SM011, see [13]). In this phase we have not considered the implementation of cyclic reduction or recursive doubling techniques, see, e.g., [16]. The computation of z from $L^T z=\tilde{z}$ can be done analogously.

4.2 Changing the Order of Computation of the Unknowns

If we still assume the vector elements to be partioned in successive parts of length m, then $\tilde{z}_j$ is computed by (4.3) from previous elements $\tilde{z}_{j-1}$ and $\tilde{z}_{j-m}$, as is shown schematically in figure 7.

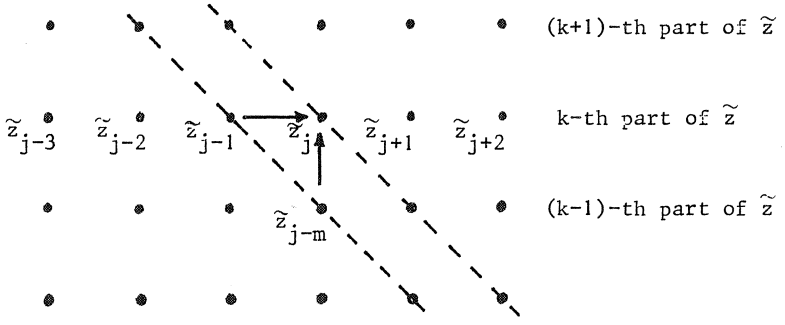


Figure 7. Dependency of unknowns

We see that in this case we can also compute the unknowns in a diagonal wise order, as is indicated by the dotted lines, and then each unknown is computed from already available elements on the previous diagonal. At the cost of twice as many DO - loops (with respect to the row-wise computation),

each loop now vectorizes on a CRAY-1 (constant stride), whereas on a CYBER 205 we still have problems (stride $\neq$ 1).

The stride problems can be solved by Gather-Scatter techniques, which can be applied either once (introducing problems in the computation of Ax), or for each DO - loop apart (introducing a considerable overhead). The stride problems can also be solved by renumbering the unknowns explicitly, which again introduces problems in the computation of Ax.

4.3 An Approximate Solution of $\tilde{K}\tilde{z}=y$

We now return to the row-wise computation of the vector $\tilde{z}$. By VAN DER VORST [14] it is proposed to compute the elements $\tilde{z}_j$ approximately from (4.4). Therefore we observe that (4.4) represents the solution of a bidiagonal system $(I+B)\tilde{z}^k = \tilde{y}^k$, where $\tilde{z}^k$ and $\tilde{y}^k$ are the k-th parts of the vectors $\tilde{z}$ and $\tilde{y}$, and B is the m-th order matrix that consists of only the first subcodiagonal in the k-th diagonal block of L. Formally the solution of (4.4) can be written as

$$\tilde{z}^k = (I - B + B^2 - B^3 + \ldots)\tilde{y}^k \ . \qquad (4.5)$$

It is shown in [14] that if we compute $\tilde{z}^k$ only approximately, by truncating the Neumann series in (4.5), then this results effectively in a perturbed preconditioning matrix, which differs only slightly from the original one, even if the truncation is carried out after a few terms, say 2 or 3.

The computation of $(I - B + B^2)\tilde{y}^k$ (truncation after 2 terms) or of $(I + B^2)(I - B)\tilde{y}^k$ (truncation after 3 terms) is completely vectorizable, avoiding the stride problems for the CYBER 205 (see 4.2). The truncation after 2 terms leads to the vector formula which has been analysed in 3.7.

4.4 Timing Results

For a system of order 3540 (59 rows with 60 unknowns, m=60) we have measured the computer CPU-times that are required to achieve a certain accuracy. In this case 99 ICCG iterations have been carried out.

For the computation of $A\tilde{x}$, $K^{-1}y$, the innerproducts and the updating of iterands, we have selected those Fortran implementations which led to the highest MFLOPS rates, as described in section 3. The CPU-times observed for the different approaches in section 4.1, 4.2 and 4.3 are listed below.

a. The standard ICCG algorithm, with the non-vectorized expression (4.3).
CRAY-1 : 0.455 seconds CYBER 205 : 0.631 seconds

b. The expression (4.3) replaced by the partly vectorizable expression (4.4), in combination with either FOLR (CRAY-1) or Q8SM011 (CYBER 205).
CRAY-1 : 0.392 seconds CYBER 205 : 0.512 seconds

c. The use of a 2-term truncated Neumann series, as is described in 4.3. The matrix B^2 was not computed explicitly, in fact the following expression has been used:

$$\tilde{z}_j = \tilde{y}_j - a_{j-1,2} * (\tilde{y}_{j-1} - a_{j-2,2} * \tilde{y}_{j-2})$$

Expressions of this type have been considered in 3.7. With this truncation 104 iterations were necessary in order to achieve a similar accuracy as achieved by the standard ICCG algorithm in 99 iterations.
CRAY-1 : 0.269 seconds CYBER 205 : 0.218 seconds

d. Computation of the unknowns in a diagonal wise order, as is outlined in 4.2. For the CRAY-1 one should be aware of possible memory bank conflicts (not in this case: stride = 59). Because of the stride $\neq 1$, we decided for the CYBER 205 to renumber the unknowns explicitly according to the diagonal wise ordering.
CRAY-1 : 0.247 seconds CYBER 205 : 0.442 seconds

From the above results we conclude that apparantly the CRAY-1 reaches the best performance for our problem when the unknowns are computed in the diagonal wise ordering (with standard ICCG), this in contrast to the CYBER 205 which is most efficient when we use the modified ICCG algorithm.

ACKNOWLEDGEMENTS

Computertime for experiments on the CRAY-1 of ECMWF at Reading, U.K., has been made available by the Dutch Meteorological Centre KNMI at De Bilt. The experiments on the CYBER 205 of Control Data at Paris, France, have been made possible by the Dutch Working Group on Supercomputers. These CYBER 205 experiments have been carried out by Jan van Kats (ACCU) with help of Gerrit van de Velde (Control Data B.V.).

REFERENCES

1. T.L. Jordan,"A Guide to Parallel Computation and Some CRAY-1 Experiences", LANL Report LA-UR-81-247, Los Alamos National Lab., Los Alamos, NM, 1981
2. I.J. Butcher and J.W. Moore, "Comparative Performance Evaluation of two Supercomputers: CDC CYBER 205 and CRI CRAY-1", LANL Report, Los Alamos National Lab., Los Alamos, NM, 1981
3. J.J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment", Argonne National Lab., Memorandum, Argonne, IL, 1983
4. J. Levesque et al., "Efficient Fortran Techniques for Vector Processors", Student Workbook, Pacific-Sierra Research Corp., Revision no. 4, Los Angeles, CA, 1982
5. H.A. van der Vorst and J.M. van Kats, "Comparative Performance Tests on the CRAY-1 and the Cyber 205", ACCU-Reeks No. 36, Academisch Computer Centrum, Utrecht, 1983
6. I.S. Duff and J.K. Reid, "Experience of Sparse Matrix Codes on the CRAY-1", Report CSS-116, C.S.S.D., AERE Harwell, 1981
7. R. Schreiber and W.P. Tang, "Vectorizing the Conjugate Gradient Method", Dept. of Computer Science, Stanford University, Stanford, CA, 1983
8. J.J. Dongarra, "Redesigning Linear Algebra Algorithms", in: ed. A. Bossavit, Calcul Vectoriel et Parallele", Bulletin de la Direction des Etudes et Recherche, Electricité de France, Serie C, no. 1, 1983
9. J.J. Dongarra and S.C. Eisenstat, "Squeezing the most out of an algorithm in CRAY Fortran", Argonne National Lab., Rep. ANL/MCS-TM-9, Argonne, IL, 1983
10. J.A. Meijerink and H.A. van der Vorst, "An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix", Math. of Comp., Vol. 31, pp.148-162, 1977
11. J.A. Meijerink and H.A. van der Vorst, "Guidelines for the Usage of Incomplete Decompositions in Solving Sets of Linear Equations as They Occur in Practical Problems", J. Comp. Phys., Vol. 44, pp.134-155, 1981
12. CRAY-1 Scientific Applications Subprograms, Pub. Number SR-0014
13. CDC CYBER 200 FORTRAN ref. Manual, Pub. Number 60457040

14. H.A. van der Vorst, "A Vectorizable Variant of Some ICCG Methods", SIAM J.Sci.Stat.Comput., Vol. 3, pp.350-356, 1982

15. N.K. Madsen, G.H. Rodrigue and J.I. Karush, "Matrix Multiplication by Diagonals on a Vector/Parallel Processor", Inf. Proc. Letters, Vol. 5, pp.41-45, 1976

16. A. Greenbaum and G. Rodrigue, "The Incomplete Choleski Conjugate Gradient Method for the STAR (5-point Operator)", Lawrence Livermore Lab. Report UCID-17574, Livermore, CA, 1977

17. A.Lichnewsky,"Some Vector and Parallel Implementations for Preconditioned Conjugate Gradient Algorithms", NATO Advanced Research Workshop on HIGH-SPEED COMPUTATIONS held at Julich, West Germany, 1983

Parallel Algorithms in Computational Linear Algebra

D.J. Evans
University of Technology, Loughborough

ABSTRACT

In this paper some techniques for exposing parallelism in a problem are surveyed and some new parallel algorithms for the direct and iterative solution of linear systems presented and compared with the existing sequential methods. Finally, a new explicit method for the finite difference solution of parabolic partial differential equations is derived. The new method uses stable asymmetric approximations to the partial differential equation which when coupled in groups of 2 adjacent points (4 points for 2 dimensions) on the grid result in implicit equations which can be easily converted to explicit form and offer many advantages especially for use on parallel computers. By judicious use of alternating this strategy on the grid points of the domain results in new explicit parallel algorithms which possess unconditional stability.

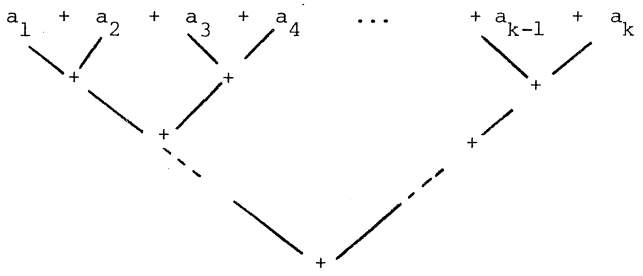
1. INTRODUCTION

Parallelism can arise at many different levels within a computational problem, which if exposed can be efficiently exploited by parallel

computers. Some well known techniques of doing this are:

1. *Vectorising existing software.* This is often achieved by changing the order in the evaluation of terms in a complicated expression so that a vector or matrix of components can be handled in one operation.

2. To decompose the problem into a number of independent sub-problems all of which can proceed independently. The solutions of these sub-problems are then combined in some way to yield the answer of the original problem. This technique is usually known as a *Divide and Conquer* strategy or partitioning, e.g. for the evaluation of $\sum_{i=1}^{k} a_i$ it is possible to decompose the problem in the following manner.



Thus, if t is the time unit for the addition operation, then the total times for the sequential and parallel computations are,

$$T_{sequential} = (k-1)t,\ T_k = (\log_2 k)t\ ,$$

while the speed-up of the computation due to parallel evaluation is determined as,

$$S_k = (k-1)/(\log_2 k) = O(k/\log_2 k)\ .$$

This result is true if we neglect:

i) the interconnection cost for S.I.M.D. computers,

ii) the synchronisation and shared memory conflicts for M.I.M.D. computers.

3. By the discovery of independent sub-expressions in the calculation which can proceed in parallel. Often this is termed *Implicit Parallelism* and examples such as *recursive decoupling and cyclic reduction* are such that the extraction of these sub-expressions can lead to a more balanced decomposition for parallel evaluation.
4. By developing new parallel methods such as the *Quadrant Interlocking Methods* for Computational Linear Algebra which will be described in Sections 2 and 3.
5. Another technique of achieving parallelism in a numerical algorithm is by the use of *Explicit methods*. Usually, such algorithms are the oldest methods for the solution of many problems. Unfortunately, they suffer from major defects such as poor stability and convergence characteristics and require unacceptable large solution times. Undoubtedly the more recent *Implicit Methods* are better but often we are not able to exploit to the full any Implicit Parallelism within the algorithm. Thus, the discovery of new *Explicit methods* of solution as given in Section 5 is important for the development of parallel algorithms.
6. Other techniques such as *pipelining*, *broadcasting* and *streaming* are more usually associated with hardware features of the computer and are not the subject of interest in this paper.

2. DIRECT METHODS FOR THE SOLUTION OF LINEAR SYSTEMS

The usual approach for solving linear systems is by Gaussian Elimination or triangular decomposition.

Given the matrix, A,

$$\text{i.e., } A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}, \text{ where } \det A^{-1} \neq 0 \text{ so that A is non-singular.} \tag{2.1}$$

We now attempt to find the matrix factors L and U of the form:

$$L = \begin{bmatrix} 1 & & & \\ \ell_{21} & 1 & & \\ \ell_{31} & \ell_{32} & 1 & \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ & u_{22} & u_{23} & u_{24} \\ & & u_{33} & u_{34} \\ & & & u_{44} \end{bmatrix}, \tag{2.2}$$

such that,

$$A \equiv LU \ . \tag{2.3}$$

By equating the coefficients in the matrix product (2.3), the following relations can be obtained to determine the coefficients of L and U. These are for rows 1,2 and 3, i.e.,

$$u_{11} = a_{11}, \quad u_{12} = a_{12}, \quad u_{13} = a_{13}, \quad u_{14} = a_{14}$$

$$\ell_{21}u_{11} = a_{21}, \ \ell_{21}u_{12}+u_{22} = a_{22}, \ \ell_{21}u_{13}+u_{23} = a_{23}, \ \ell_{21}u_{14}+u_{24} = a_{24} \tag{2.4}$$

$$\ell_{31}u_{11} = a_{31}, \ \ell_{31}u_{12}+\ell_{32}u_{22} = a_{32}, \ \ell_{31}u_{13}+\ell_{32}u_{23}+u_{33} = a_{33}, \text{ etc.}$$

with similar results for the last row.

These equations are essentially all *sequential* relations, since each of the unknowns $\ell_{i,j}$ and $u_{i,j}$ are brought into the above relations one at a time recursively and then determined in a similar manner.

The reason why such a factorisation is sought specifically in L.U. form is that the matrix factors L and U are known as easily inverted matrix forms and so the solution of the linear system,

$$\underline{Ax} = \underline{b} \ , \tag{2.5}$$

can be obtained by making use of the substitution A = LU to reduce the problem to the solution of the coupled systems,

$$\underline{Ly} = \underline{b} \tag{2.6}$$

and

$$\underline{Ux} = \underline{y} \ , \tag{2.7}$$

where $\underline{y}$ is an intermediate vector.

The linear systems (2.6) and (2.7) are easily solvable systems and can be solved by well known forward or backward substitution processes, i.e.,

$$\underline{Ly} = \underline{b} \ ,$$

$$\begin{bmatrix} 1 & & & \\ \ell_{21} & 1 & \bigcirc & \\ \ell_{31} & \ell_{32} & 1 & \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \ , \tag{2.8}$$

can be solved as follows:

$$\begin{aligned} y_1 &= b_1 && \rightarrow y_1 = b_1 \\ \ell_{21}y_1+y_2 &= b_2 && \rightarrow y_2 = b_2-\ell_{21}y_1 \\ \ell_{31}y_1+\ell_{32}y_2+y_3 &= b_3 && \rightarrow y_3 = b_3-\ell_{31}y_1-\ell_{32}y_2 \\ \ell_{41}y_1+\ell_{42}y_2+\ell_{43}y_3+y_4 &= b_4 && \rightarrow y_4 = b_4-\ell_{41}y_1-\ell_{42}y_2-\ell_{43}y_3 \end{aligned} \tag{2.9}$$

Similarly, for the system $\underline{Ux} = \underline{y}$.

These relations are again all *sequential processes*.

The question now is can we find a matrix factorisation that is more suitable for parallel computation?

Consider then a factorization of the matrix A of the form,

$$A = WZ \ , \tag{2.10}$$

where,

$$W = \begin{bmatrix} 1 & 0 & 0 \\ w_{21} & 1 & 0 & w_{24} \\ w_{31} & 0 & 1 & w_{34} \\ 0 & 0 & & 1 \end{bmatrix} \ , \text{ and } \ Z = \begin{bmatrix} z_{11} & z_{12} & z_{13} & z_{14} \\ 0 & z_{22} & z_{23} & 0 \\ & z_{32} & z_{33} & \\ z_{41} & z_{42} & z_{43} & z_{44} \end{bmatrix} . \tag{2.11}$$

In general, the matrices W and Z will have the forms,

$$W = \begin{bmatrix} & 0 & \\ & & \\ & 0 & \end{bmatrix} \qquad Z = \begin{bmatrix} & & \\ 0 & & 0 \\ & & \end{bmatrix}$$

and are termed the quadrant interlocking factors (Q.I.F.) of A. It can be noticed that they have a butterfly shape (Evans & Hatzopoulos, 1979).

To determine the coefficients of W and Z we equate the coefficients of A and WZ in (2.10). Thus, for rows I and IV we have,

$$\begin{aligned} &\text{I} \quad z_{11} = a_{11} \ , \ z_{12} = a_{12} \ , \ z_{13} = a_{13} \ , \ z_{14} = a_{14} \ , \\ &\text{IV} \quad z_{41} = a_{41} \ , \ z_{42} = a_{42} \ , \ z_{43} = a_{43} \ , \ z_{44} = a_{44} \ . \end{aligned} \tag{2.12}$$

Whilst for row 2, we have the equations,

$$\begin{aligned} &\text{II} \quad w_{21}z_{11}+w_{24}z_{41} = a_{21} \ , \ w_{21}z_{12}+z_{22}+w_{24}z_{42} = a_{22} \ , \\ &\qquad w_{21}z_{13}+z_{23}+w_{24}z_{43} = a_{23}; \ w_{21}z_{14}+w_{24}z_{44} = a_{24} \ . \end{aligned} \tag{2.13}$$

From the first and last equations we obtain w_{21} and w_{24} and by substitution in the 2nd and 3rd equations we obtain z_{22} and z_{23}.

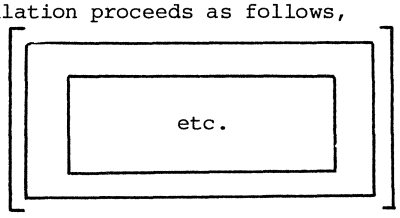
Similarly for row 3, we have the equations,

$$\begin{aligned} &\text{III} \quad w_{31}z_{11}+w_{34}z_{41} = a_{31} \ , \quad w_{31}z_{12}+z_{32}+w_{34}z_{42} = a_{32} \ , \\ &\qquad w_{31}z_{13}+w_{34}z_{43} = a_{33} \ , \quad w_{31}z_{14}+w_{34}z_{44} = a_{34} \ . \end{aligned} \tag{2.14}$$

As before we obtain from the first and last equations the values of w_{31} and w_{34} and by substituting in the 2nd and 3rd equations, we obtain z_{32} and z_{33}.

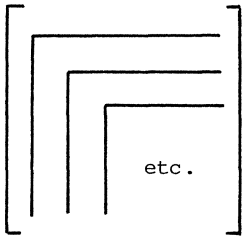
Thus, we can see that the first and last rows of Z are given immediately. Then, (2×2) sets of linear equations are solved to obtain $w_{i,1}$ and $w_{i,4}$ for i=2,3.

Thus, the calculation proceeds as follows,



where the outermost peripheral elements of the matrices W and Z are obtained. Then, the calculation proceeds to the innermost next layer of elements. Thus only $(\frac{n-1}{2})$ stages are required to compute all the elements of W and Z.

In comparison, the determination of the coefficients in the LU decomposition is given as,



Solution of the Linear Systems

Using the relationship $A = WZ$,

then the linear system $A\underline{x} = \underline{b}$,

can now be reformulated as the solution of 2 related linear systems,

$$\underline{Wy} = \underline{b} \ , \quad \text{and } \underline{Zx} = \underline{y} \ .$$

To solve $\underline{Wy} = \underline{b}$ we proceed as follows:

$$\begin{bmatrix} 1 & & 0 & \\ w_{21} & 1 & 0 & w_{24} \\ w_{31} & 0 & 1 & w_{34} \\ 0 & & & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

We see immediately, that,

$$y_1 = b_1 \quad \text{and} \quad y_4 = b_4 \ ,$$

$$w_{21}y_1+y_2+w_{24}y_4 = b_2 \quad \text{and} \quad w_{31}y_1+y_3+w_{34}y_4 = b_3 \ ,$$

or
$$y_2 = \tilde{b}_2 = (b_2-w_{21}y_1-w_{24}y_4) \ ,$$

and
$$y_3 = \tilde{b}_3 = (b_3-w_{31}y_1-w_{34}y_4) \ .$$

The solutions for $\underline{y}$ are obtained in pairs working from the top and bottom components of the vector.

Once the vector $\underline{y}$ has been determined then to solve the system $\underline{Zx}=\underline{y}$ we proceed as follows,

$$\begin{bmatrix} z_{11} & z_{12} & z_{13} & z_{14} \\ & z_{22} & z_{23} & \\ & z_{32} & z_{33} & \\ z_{41} & z_{42} & z_{43} & z_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \ .$$

Starting at the centre we solve the (2×2) linear system,

$$z_{22}x_2 + z_{23}x_3 = y_2 \ ,$$

$$z_{32}x_2 + z_{33}x_3 = y_3 \ ,$$

to evaluate
$$x_2 \text{ and } x_3 \ .$$

Then, we proceed outwards and solve the (2×2) linear system,

$$z_{11}x_1 + z_{14}x_4 = \tilde{y}_1 = (y_1 - z_{12}x_2 - z_{13}x_3)$$

$$z_{41}x_1 + z_{44}x_4 = \tilde{y}_4 = (y_4 - z_{42}x_2 - z_{43}x_3)$$

to evaluate x_1 and x_3 .

Thus, the solution $\underline{x}$ can be obtained in O(n) stages on a Parallel Computer with $O(n^2)$ processors.

Thus, for general (n×n) matrices, a factorisation of the form,

$$A = WZ \quad ,$$

is possible where W and Z have the matrix forms,

$$W = \begin{bmatrix} 1 & & & & & 0 \\ w_{2,1} & 1 & & & w_{2,n} & \\ & w_{3,2} & & & & \\ & & 1 & & & \\ & w_{n-2,2} & & & & \\ w_{n-1,1} & & & & w_{n-1,n} & \\ 0 & & & & 1 & \end{bmatrix} \quad , \quad Z = \begin{bmatrix} z_{1,1} & z_{1,2} & \cdots & & z_{1,n} \\ & z_{2,2} & & & z_{2,n-1} \\ & & \ddots & & \\ 0 & & z_{i,i} & & 0 \\ & & & \ddots & \\ z_{n,1} & & & & z_{n,n} \end{bmatrix}$$

where the elements of W and Z are given by,

$$W_{i,j} = \begin{cases} 1, & i=j, \\ 0, & i=1(1)\frac{n+1}{2},\ j=i+1(1)n-i+1, \\ 0, & i=\frac{n+2}{2}(1)n,\ j=n-i+1(1)i-1, \\ w_{i,j} & \text{otherwise.} \end{cases} \qquad Z_{i,j} = \begin{cases} z_{i,j}, & i=1(1)\frac{n+1}{2} \\ z_{i,j}, & i=\frac{n+2}{2}(1)n, \\ & j=n-i+1(1)i \\ 0, & \text{otherwise.} \end{cases}$$

Computation of the Matrices W and Z

By comparing terms of A and WZ we have:

1. the elements of the first and last row of Z are given immediately

$$z_{1,i} = a_{1,i} \quad \text{and} \quad z_{n,i} = a_{n,i} \quad \text{for all } i=1(1)n$$

2. then the sets of (2×2) linear systems given by,

$$z_{1,1}w_{i,1} + z_{n,1}w_{i,n} = a_{i,1} ,$$

$$z_{1,n}w_{i,1} + z_{n,n}w_{i,n} = a_{i,n} ,$$

are solved to obtain the values of $w_{i,1}$ and $w_{i,n}$ for i=2(1)n-1. This then completes the first stage and the calculation of the outermost elements of the matrices W and Z.

At least $(\frac{n-1}{2})$ such stages are required to compute all the elements of the matrices W and Z.

Solution of the Linear System

By using the relationship $A = WZ$,

the linear system $A\underline{x} = \underline{b}$,

can be reformulated as the solution of the 2 related linear systems

$$Z\underline{x} = \underline{y} \quad \text{and} \quad W\underline{y} = \underline{b} .$$

These are linear systems of the form,

$$\begin{bmatrix} 1 & & & & 0 \\ w_{2,1} & 1 & & w_{2,n} & \\ & w_{3,2} & & & \\ & & 1 & & \\ w_{n-1,1} & & & w_{n-1,n} & \\ & 0 & & & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \\ \\ y_{n-1} \\ y_n \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \\ \\ s_{n-1} \\ s_n \end{bmatrix}$$

We see that y_1 and y_n are calculated first then y_2, y_{n-1} and so on in pairs working from the top and rear of the vector $\underline{y}$ and $\underline{s}=\underline{b}$.

In general, at the i^{th} step, we have,

$$y_i = s_i, \quad y_{n-i+1} = s_{n-i+1} ,$$

and we reset the s_j in the following manner,

$$s_j = s_j - w_{j,i}y_i - w_{j,n-i+1}y_{n-i+1} , \quad j=i+1(1)n-i.$$

Similarly, the system, $Z\underline{x} = \underline{y}$,

can be treated in a similar manner.

For parallel computers with $O(n^2)$ processors - this is an $O(n)$ method.

Finally, it can be shown that by suitably chosen permutation matrices the method is identical to a (2×2) block Gaussian Elimination technique.

3. ITERATIVE METHODS FOR THE SOLUTION OF LINEAR SYSTEMS

Sequential iterative methods are derived using the principles of splitting the matrix into easily inverted forms. Thus, given a matrix A of the form,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

where $\det A^{-1} \neq 0$.

Then, the standard iterative approach is to assume the splitting,

$$A = D - L - U \ , \tag{3.1}$$

where,

$$D = \begin{bmatrix} a_{11} & & & \\ & a_{22} & & \\ & & a_{33} & \\ & & & a_{44} \end{bmatrix}$$

$$-L = \begin{bmatrix} 0 & & & \\ & \ddots & & \\ a_{21} & & \ddots & \\ a_{31} & a_{32} & & 0 \\ a_{41} & a_{42} & a_{43} & \end{bmatrix} \quad \text{and} \quad -U = \begin{bmatrix} 0 & a_{12} & a_{13} & a_{14} \\ & \ddots & a_{23} & a_{24} \\ & & \ddots & a_{34} \\ & & & 0 \end{bmatrix} .$$

Then, to solve the linear system $A\underline{x} = \underline{b}$, the Gauss or Jacobi method can be written as,

$$D\underline{x}^{(n+1)} = +L\underline{x}^{(n)} + U\underline{x}^{(n)} + \underline{b} , \tag{3.2}$$

and the Gauss-Seidel method as,

$$D\underline{x}^{(n+1)} = +L\underline{x}^{(n+1)} + U\underline{x}^{(n)} + \underline{b} , \tag{3.3}$$

or

$$(D+L)\underline{x}^{(n+1)} = +U\underline{x}^{(n)} + \underline{b} .$$

This is a sequential equation where the unknowns are brought in one at a time, i.e.,

$$\begin{aligned} a_{11}x_1^{(n+1)} &= -a_{12}x_2^{(n)} - a_{13}x_3^{(n)} - a_{14}x_4^{(n)} + b_1 , \\ a_{22}x_2^{(n+1)} &= -a_{21}x_1^{(n+1)} - a_{23}x_3^{(n)} - a_{24}x_4^{(n)} + b_2 , \\ a_{33}x_3^{(n+1)} &= -a_{31}x_1^{(n+1)} - a_{32}x_2^{(n+1)} - a_{34}x_4^{(n+1)} + b_3 , \\ a_{44}x_4^{(n+1)} &= -a_{41}x_1^{(n+1)} - a_{42}x_2^{(n+1)} - a_{43}x_3^{(n+1)} + b_4 . \end{aligned} \tag{3.4}$$

Can we apply the Quadrant Interlocking approach of the previous section to derive a class of parallel iterative methods?

Quadrant Interlocking Splitting (Q.I.S.) Methods

Suppose we write A in the form,

$$A = X - W - Z , \tag{3.5}$$

where X is defined as,

$$\begin{bmatrix} a_{11} & & & a_{14} \\ & a_{22} & a_{23} & \\ & a_{32} & a_{33} & \\ a_{41} & & & a_{44} \end{bmatrix} \tag{3.6a}$$

and,

$$-W = \begin{bmatrix} 0 & & 0 \\ a_{21} & & a_{24} \\ a_{31} & & a_{34} \\ 0 & & 0 \end{bmatrix} \quad \text{and} \quad -Z = \begin{bmatrix} 0 & a_{12} & a_{13} & 0 \\ & 0 & 0 & \\ 0 & a_{42} & a_{43} & 0 \end{bmatrix} . \tag{3.6b}$$

Then, a parallel iterative method can be written as,

$$X\underline{x}^{(n+1)} = +W\underline{x}^{(n)} + Z\underline{x}^{(n)} + \underline{b} , \tag{3.7}$$

similar to Jacobi form, and

$$X\underline{x}^{(n+1)} = +W\underline{x}^{(n+1)} + Z\underline{x}^{(n)} + \underline{b} , \tag{3.8}$$

or

$$(X-W)\underline{x}^{(n+1)} = +Z\underline{x}^{(n)} + \underline{b} ,$$

similar to Gauss-Seidel form.

Thus, the equations (3.7) in point form are given by,

$$\begin{aligned} a_{11}x_1^{(n+1)} + a_{14}x_4^{(n+1)} &= -a_{12}x_2^{(n)} - a_{13}x_3^{(n)} + b_1 , \\ a_{41}x_1^{(n+1)} + a_{44}x_4^{(n+1)} &= -a_{42}x_2^{(n)} - a_{43}x_3^{(n)} + b_4 , \end{aligned} \tag{3.9a}$$

followed by,

$$\begin{aligned} a_{22}x_2^{(n+1)} + a_{23}x_3^{(n+1)} &= -a_{21}x_1^{(n+1)} - a_{24}x_4^{(n+1)} + b_2 , \\ a_{32}x_3^{(n+1)} + a_{33}x_3^{(n+1)} &= -a_{31}x_1^{(n+1)} - a_{34}x_4^{(n+1)} + b_3 . \end{aligned} \tag{3.9b}$$

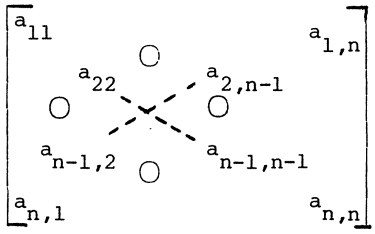
This parallel method again requires the solution of (2×2) linear systems for each pair of equations within each iteration.

Finally, for n=odd the centre element is treated by a separate equation.

In general, then, the matrix A can be written in the form,

$$A = X - W - Z \ , \tag{3.10}$$

where X is defined as,



with,

$$-W = \begin{bmatrix} 0 & & & 0 \\ a_{21} & & & a_{2,n} \\ & \ddots & \ddots & \\ a_{n-1,1} & & & a_{n-1,n} \\ 0 & & & 0 \end{bmatrix} \quad \text{and} \quad -Z = \begin{bmatrix} 0 & a_{12} \cdots & a_{1,n-1} & 0 \\ & \ddots & \ddots & \\ 0 & a_{n,2} \cdots & a_{n,n-1} & 0 \end{bmatrix}$$

Alternatively, the elements of X,W and Z can be given as,

$$X = \begin{cases} a_{i,i} \\ a_{i,n-i+1} \end{cases}, \ i=1,2,\ldots,n; \qquad -W = (a_{ij}) \begin{cases} 1 \leq j < \left\lfloor \frac{n-1}{2} \right\rfloor, \ j<i<n-j+1 \\ \left\lfloor \frac{n+2}{2} \right\rfloor < j < n, \ n-j+1<i<j \\ 0, \text{ elsewhere} \end{cases}$$

and

$$-Z = (a_{ij}) \begin{cases} 1<i<\left\lfloor \frac{n+1}{2} \right\rfloor, \ i<j<n-j+i \\ \left\lfloor \frac{n+2}{2} \right\rfloor < i < n, n-i+1<j<i \\ 0, \text{ elsewhere.} \end{cases} \tag{3.11}$$

Thus, we have split the coefficient matrix A into the sum of interlocking quadrant components A = X - W - Z and analogous to the familiar splitting A = D - L - U, (Varga, 1963) we can formulate the following parallel iterative methods:

Simultaneous QI method: $$X\underline{x}^{(k+1)} = (W+Z)\underline{x}^{(k)} + \underline{b} \ , \tag{3.12}$$

Successive QI method: $$(X-W)\underline{x}^{(k+1)} = Z\underline{x}^{(k)} + \underline{b}, \quad (3.13)$$

Simultaneous Overrelaxation QI method: $$X\underline{x}^{(k+1)} = (W+Z)\underline{x}^{(k)} + (1-\omega)X\underline{x}^{(k)} + \omega\underline{b}, \quad (3.14)$$

Successive Overrelaxation QI method: $$(X-\omega W)\underline{x}^{(k+1)} = (\omega Z + (1-\omega)X)\underline{x}^{(k)} + \omega\underline{b}, \quad (3.15)$$

where k is the iteration index and ω is the overrelaxation parameter chosen to maximise the convergence rate of the iterative method.

The following properties of these methods can be established (Evans and Haghighi, 1982).

1. The simultaneous QI method converges if A is diagonally dominant.
2. The QI matrix splitting is a regular splitting of A.
3. The successive QI method converges if A is irreducible and possesses weak diagonal dominance.
4. The successive QI method converges if A is real and positive definite, and
5. The successive overrelaxation QI method converges for $0<\omega<2$ and corresponds to a (2×2) block S.O.R. method.

Finally, again it can be shown that these are nothing more than (2×2) block iterative methods and one is naturally bound to enquire whether there is an optimum size block where the extra computational effort of inverting the block is more than compensated by the increase in convergence rate of the method.

4. EXPLICIT BLOCK ITERATIVE METHODS

We now consider a novel approach where the blocks are inverted

explicitly and new iterative methods developed for the simple (4×4) system of linear equations,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \tag{4.1}$$

in the following manner.

We sub-divide the system up into smaller (2×2) systems, i.e.,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 - a_{13}x_3 - a_{14}x_4 \ , \\ a_{21}x_1 + a_{22}x_2 &= b_2 - a_{23}x_3 - a_{24}x_4 \ , \text{ etc.} \end{aligned} \tag{4.2}$$

which can be written in iterative form as,

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^{(k+1)} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} - \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}^{(k)} , \tag{4.3}$$

and

$$\begin{bmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}^{(k+1)} = \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} - \begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^{(k)} , \tag{4.4}$$

or in *Explicit* form,

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^{(k+1)} = A_1^{-1} \left\{ \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} - \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}^{(k)} \right\} , \tag{4.5}$$

$$\begin{bmatrix} x_3 \\ x_4 \end{bmatrix}^{(k+1)} = A_2^{-1} \left\{ \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} - \begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^{(k)} \right\} . \tag{4.6}$$

Now since A_1 and A_2 are small (2×2) systems then they can be inverted explicitly, i.e.,

$$A_1^{-1} = \frac{1}{\Delta_1} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix} , \quad A_2^{-1} = \frac{1}{\Delta_2} \begin{bmatrix} a_{44} & -a_{34} \\ -a_{43} & a_{33} \end{bmatrix} , \tag{4.7}$$

where $\Delta_1 = (a_{11}a_{22}-a_{21}a_{12})$, $\Delta_2 = (a_{33}a_{44}-a_{34}a_{43})$. (4.8)

Thus, for the linear system,

$\underline{Ax} = \underline{b}$, where A is sparse, and of large order

and where the splitting

$A = D - L - U$, with D=block diagonal is assumed

then all previous implicit methods have used the assumption that $\underline{Dx} = \underline{b}$ can be determined by a simple efficient algorithm, i.e.,

$$\underline{Dx}^{(k+1)} = (L+U)\underline{x}^{(k)} + \underline{b}, \text{ the Block Jacobi method,} \quad (4.9)$$

and

$$\underline{Dx}^{(k+1)} = L\underline{x}^{(k+1)} + U\underline{x}^{(k)} + \underline{b}, \text{ the Block Gauss-Seidel method,} \quad (4.10)$$

with their overrelaxation counterparts. Such schemes are well known, i.e. 1-line, 2-line, k-line block methods (Varga, 1963).

However, if we assume that D^{-1},$D^{-1}L$ and $D^{-1}U$ are small block systems which can be explicitly determined, then new methods of Explicit Block form, e.g.

$$\underline{x}^{(k+1)} = (D^{-1}L + D^{-1}U)\underline{x}^{(k)} + \tilde{\underline{b}},$$

or

$$\underline{x}^{(k+1)} = (L^E + U^E)\underline{x}^{(k)} + \tilde{\underline{b}}, \text{ the Explicit Block Jacobi method,} \quad (4.11)$$

and

$$\underline{x}^{(k+1)} = L^E\underline{x}^{(k+1)} + U^E\underline{x}^{k} + \tilde{\underline{b}}, \text{ the Explicit Block Gauss-Seidel method,} \quad (4.12)$$

and their overrelaxation counterparts can be developed which will be appropriate for parallel implementation.

Now for what grouping of points can we determine $D^{-1}L$ and $D^{-1}U$ explicitly since when you invert the line block, the block fills up and sparsity disappears and the computational complexity of the method increases. However, for small (2×2) blocks this does not happen. So for the components x_1 and x_2 of the first block, we have,

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^{(k+1)} = \frac{1}{\Delta_1} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix} \left\{ \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} - \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}^{(k)} \right\} \tag{4.13}$$

and similarly for x_3 and x_4.

Thus, multiplying out we obtain the new *explicit* block equations,

$$\begin{aligned}
\Delta_1 x_1^{(k+1)} &= (a_{22}b_1 - a_{12}b_2) - [(a_{22}a_{13} - a_{12}a_{23})x_3^{(k)} + (a_{22}a_{14} - a_{12}a_{24})x_4^{(k)}] \\
\Delta_1 x_2^{(k+1)} &= (-a_{21}b_1 + a_{11}b_2) - [(-a_{21}a_{13} + a_{11}a_{23})x_3^{(k)} + (-a_{21}a_{14} + a_{11}a_{24})x_4^{(k)}] \\
\Delta_2 x_3^{(k+1)} &= (a_{44}b_3 - a_{34}b_4) - [(a_{44}a_{31} - a_{34}a_{41})x_1^{(k)} + (a_{44}a_{32} - a_{34}a_{42})x_2^{(k)}] \\
\Delta_2 x_4^{(k+1)} &= (-a_{43}b_3 + a_{33}b_4) - [(-a_{43}a_{31} + a_{33}a_{41})x_1^{(k)} + (-a_{43}a_{32} + a_{33}a_{42})x_2^{(k)}]
\end{aligned} \tag{4.14}$$

which can be considered as a viable computational approach.

The normal procedure is to precalculate these quantities and then solve the equations,

$$\begin{aligned}
\Delta_1 x_1^{(k+1)} &= b_1' - a_{13}' x_3^{(k)} - a_{14}' x_4^{(k)} , \\
\Delta_1 x_2^{(k+1)} &= b_2' - a_{23}' x_3^{(k)} - a_{24}' x_4^{(k)} , \\
\Delta_2 x_3^{(k+1)} &= b_3' - a_{31}' x_1^{(k)} - a_{32}' x_2^{(k)} , \\
\Delta_2 x_4^{(k+1)} &= b_4' - a_{41}' x_1^{(k)} - a_{42}' x_2^{(k)} ,
\end{aligned} \tag{4.15}$$

where $b_1' = (a_{22}b_1 - a_{12}b_2)$, $a_{13}' = (a_{22}a_{13} - a_{12}a_{23})$, $a_{14}' = (a_{22}a_{14} - a_{12}a_{24})$, etc., which leads to a new explicit Jacobi scheme which can then be developed into similar explicit block Gauss-Seidel and S.O.R. schemes.

The new explicit block iterative schemes in certain circumstances (i.e. when A is diagonal dominant, etc.) will have a greater convergence factor but also will involve more computational work per iteration. So the total amount of computational work to achieve the solution will have to

be considered. The interesting question is for what block size is the maximum efficiency obtained. Preliminary investigations have confirmed that the largest gains are achieved for a (3×3) block size.

5. CONVERSION OF IMPLICIT METHODS TO EXPLICIT FORM

Another technique of achieving parallelism in a numerical algorithm is by the use of explicit methods.

However these methods are the oldest methods and suffer from poor stability and convergence characteristics that require unacceptable computer solution times.

The newer implicit methods are better but often we are not able to exploit to the full the implicit parallelism in the solution algorithm.

Hence we must find new explicit methods with improved stability and convergence characteristics.

Consider the simple heat-conduction problem, (Fig.5.1),

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \quad , \quad 0 \leq x \leq 1, \; t > 0 \quad , \tag{5.1}$$

with initial conditions, $u(x,0) = f(x)$, $0 \leq x \leq 1$,

and boundary conditions, $u(0,t) = g_0(t)$, $0 < t \leq T$,

$u(1,t) = g_1(t)$, $0 < t \leq T$.

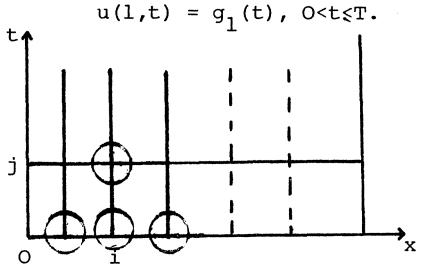


FIGURE 5.1

The simplest explicit method uses a forward difference operation approximation to $\frac{\partial u}{\partial t}$ and a central difference operator approximation to $\frac{\partial^2 u}{\partial x^2}$. The formula,

$$u_{i,j+1} = ru_{i-1,j} + (1-2r)u_{i,j} + ru_{i+1,j} + O(\Delta t + \Delta x^2) \tag{5.2}$$

is well known (Fig. 5.2) but is unstable for values of $r = \frac{\Delta t}{\Delta x^2} > \frac{1}{2}$. Hence, the algorithm is ideal for parallel application since every point on the grid can be evaluated at the same time. The method requires long solution times due to the small time step of integration.

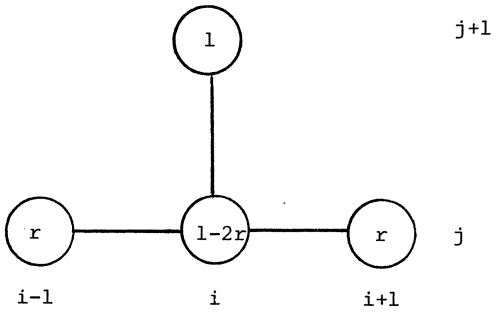


FIGURE 5.2

An implicit method uses a backward difference operator approximation to $\frac{\partial u}{\partial t}$ and a central difference operator approximation to $\frac{\partial^2 u}{\partial x^2}$. The equation,

$$-ru_{i-1,j+1} + (1+2r)u_{i,j+1} - ru_{i+1,j+1} \approx u_{i,j} , \tag{5.3}$$

is also well known and is stable for all values of r (Fig.5.3). However, the algorithm requires the solution of a system of 3 term finite difference equations at every time step in which we are not able to exploit the parallelism to the full.

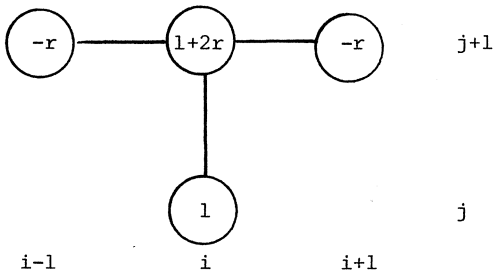


FIGURE 5.3

In order to facilitate the solution of these implicit equations, asymmetric techniques due to Saul'yev (1964) have been used, i.e. the computational molecule Fig. 5.4 representing the equation,

$$-ru_{i-1,j+1}+(1+r)u_{i,j+1} = (1-r)u_{i,j} + ru_{i+1,j} + O(\Delta t+\Delta x^2 + \frac{\Delta t}{\Delta x}) \qquad (5.4)$$

is explicit if solved from left → right and the computational molecule Fig. 5.5 representing the equation,

$$-ru_{i+1,j+1}+(1+r)u_{i,j+1} = (1-r)u_{i,j} + ru_{i-1,j} + O(\Delta t +\Delta x^2 - \frac{\Delta t}{\Delta x}) \qquad (5.5)$$

is explicit if solved from right → left.

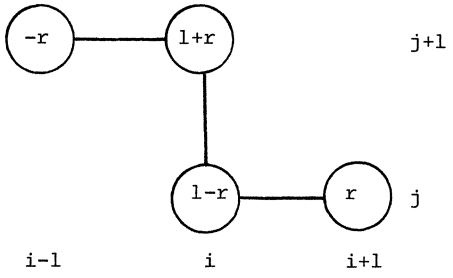


FIGURE 5.4

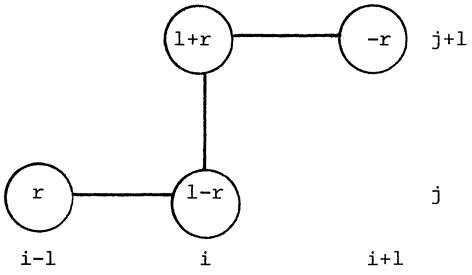


FIGURE 5.5

These two schemes are often referred to as semi-explicit formulae.

A New Group Explicit Method

If we now couple the use of the asymmetric equations (5.4) and (5.5) at 2 adjacent points, i.e.,

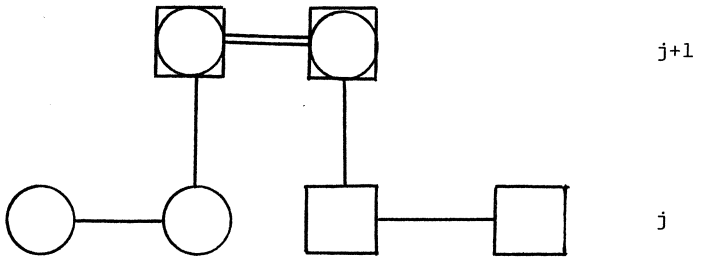


FIGURE 5.6

then they result in a (2×2) set of implicit difference equations.

For the group of two points, i.e. $\{i\Delta x,(j+\frac{1}{2})\Delta t\}$ and $\{(i+1)\Delta x,(j+\frac{1}{2})\Delta t\}$ in which equations (5.5) and (5.4) are used simultaneously to calculate the values of u at these points respectively. Therefore, at point $\{i\Delta x,(j+\frac{1}{2})\Delta t\}$ the solution is approximated by,

$$-ru_{i+1,j+1} + (1+r)u_{i,j+1} \approx ru_{i-1,j} + (1-r)u_{i,j} , \qquad (5.4a)$$

whilst at point $\{(i+1)\Delta x,(j+\frac{1}{2})\Delta t\}$, the solution is approximated by,

$$-ru_{i,j+1} + (1+r)u_{i+1,j+1} \approx (1-r)u_{i+1,j} + ru_{i+2,j} . \qquad (5.5a)$$

If we now rewrite equations (5.4) and (5.5) in matrix form,

$$\begin{bmatrix} 1+r & -r \\ -r & 1+r \end{bmatrix} \begin{bmatrix} u_{i,j+1} \\ u_{i+1,j+1} \end{bmatrix} = \begin{bmatrix} 1-r & 0 \\ 0 & 1-r \end{bmatrix} \begin{bmatrix} u_{i,j} \\ u_{i+1,j} \end{bmatrix} + \begin{bmatrix} ru_{i-1,j} \\ ru_{i+2,j} \end{bmatrix} \qquad (5.6)$$

in which the (2×2) matrix of coefficients can easily be inverted so that the equation can be written in explicit form as,

$$\begin{bmatrix} u_{i,j+1} \\ u_{i+1,j+1} \end{bmatrix} = \frac{1}{|A|} \begin{bmatrix} 1+r & r \\ r & 1+r \end{bmatrix} \left\{ \begin{bmatrix} 1-r & 0 \\ 0 & 1-r \end{bmatrix} \begin{bmatrix} u_{i,j} \\ u_{i+1,j} \end{bmatrix} + \begin{bmatrix} ru_{i-1,j} \\ ru_{i+2,j} \end{bmatrix} \right\} \tag{5.7}$$

where $A = 1+2r$. This simplifies to,

$$\begin{bmatrix} u_{i,j+1} \\ u_{i+1,j+1} \end{bmatrix} = \frac{1}{|A|} \begin{bmatrix} r(1+r)u_{i-1,j}+(1-r^2)u_{i,j}+r(1-r)u_{i+1,j}+r^2u_{i+2,j} \\ r^2u_{i-1,j}+r(1-r)u_{i,j}+(1-r^2)u_{i+1,j}+r(1+r)u_{i+2,j} \end{bmatrix} \tag{5.8}$$

For any ungrouped (single) points near the right and left boundaries equations (5.4) and (5.5) can be used respectively, i.e. for the right boundary,

$$u_{m-1,j+1} = \frac{1}{(1+r)} \left(ru_{m,j+1} + ru_{m-2,j} + (1-r)u_{m-1,j}\right) , \tag{5.9}$$

and for the left boundary,

$$u_{1,j+1} = \frac{1}{(1+r)} \left(ru_{0,j+1} + ru_{2,j} + (1-r)u_{i,j}\right) . \tag{5.10}$$

Finally, equation (5.6) can be easily converted to explicit form resulting in the computational molecule (Fig. 5.7).

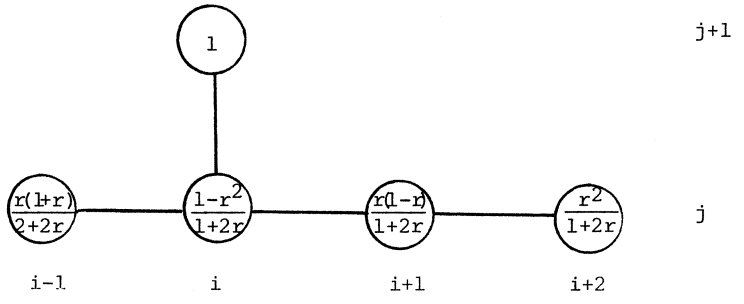


FIGURE 5.7

representing the equation,

$$u_{i,j+1} = \frac{1}{(1+2r)} \left[r(1+r)u_{i-1,j}+(1-r^2)u_{i,j}+r(1+r)u_{i+1,j}+r^2u_{i+2,j}\right] \tag{5.11}$$

and the molecule,

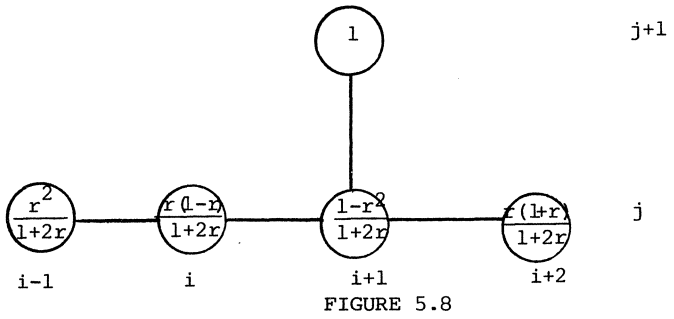


FIGURE 5.8

representing, $$u_{i+1,j+1} = \frac{1}{(1+2r)}[r^2u_{i-1,j}+r(1-r)u_{i,j}+(1-r^2)u_{i+1,j}+ r(1+r)u_{i+2,j}] \ , \qquad (5.12)$$

which when used in the alternating group explicit (AGE) method results in a stable explicit algorithm which is ideal for parallel application (Evans & Abduallh, 1983).

The given problem (5.1) was solved using the AGE algorithm on the NEPTUNE 4 processor parallel MIMD system at Loughborough University and the results obtained when compared with the standard explicit method confirm its suitability for parallel implementation.

TABLE 5.1

No. of points	No. of processors	The Explicit Method		The Group Explicit Method	
		Speed-up	Efficiency	Speed-up	Efficiency
1920	0,1	1.93	0.9650	1.98	0.9900
	0,1,2	2.85	0.9500	2.95	0.9833
	0,1,2,3	3.77	0.9425	3.91	0.9775
The relative speed up = $\frac{\text{explicit}}{\text{Group explicit}}$ =1.1619					

REFERENCES

[1] D.J. EVANS and M. HATZOPOULOS, *A Parallel Linear System Solver*, Int.J. Comp.Math. 7 (1979), 227-238.

[2] R.S. VARGA, *Matrix Iterative Analysis*, Prentice Hall, 1963.

[3] D.J. EVANS and R. SOJOODI-HAGHIGHI, *Parallel Iterative Methods for Solving Linear Equations*, Int.J.Comp.Math. 11 (1982), 247-284.

[4] V.K. SAUL'YEV, *Integration of Equations of Parabolic Type by the Method of Nets*, Macmillan, New York, 1964.

[5] D.J. EVANS and A.R.B. ABDULLAH, *A New Explicit Method for the Diffusion Equation*, pp.330-347, in 'Numerical Methods in Thermal Problems III, edit. R.W. Lewis et al. Pineridge Press, 1983.

An Internal View of the Cyber 205 Operating System

C.J. Purcell
Control Data Corporation, St. Paul, MN

1. THE COMPUTATIONAL REQUIREMENT

The Control Data CYBER 205 system is manufactured and built to serve as the major computational facility within a network of diverse computer systems (see Figure 1). All of the design parameters have been selected so as to minimize the size and the overhead of the operating system while maximizing system user facilities. Thus, this paper will focus on those aspects of the hardware design which have helped to minimize various aspects of the operating system. This minimization includes the time required to perform operating system functions, as well as the memory space required to support the operating system functions.

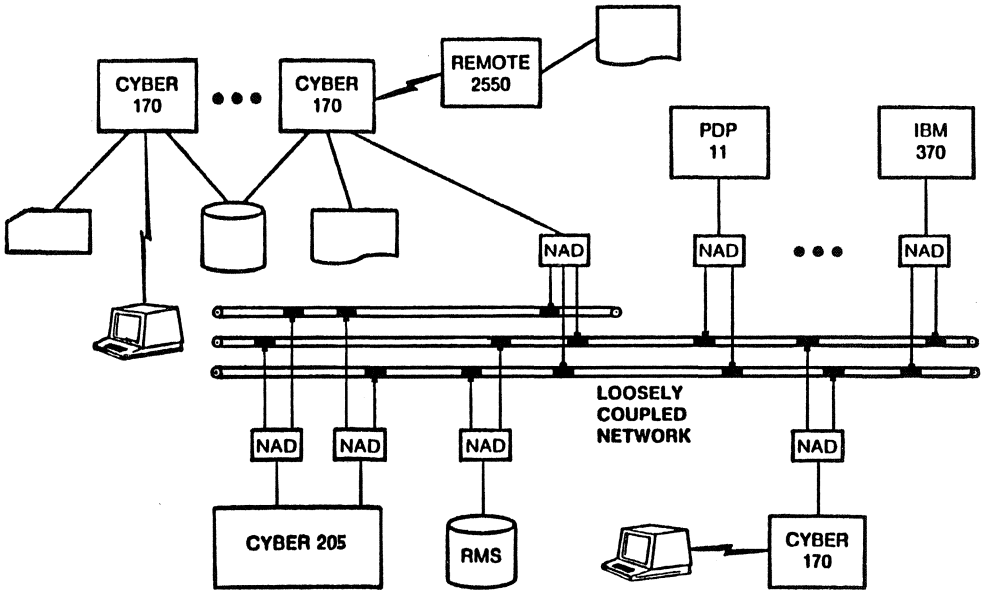


Figure 1. A typical network

ESTABLISHED SYSTEMS

MODEL	MEMORY	SINGLE PROCESSOR SUSTAINED PERFORMANCE	DATE
CDC 6600	131K	2 MEGAFLOPS	1965
CDC 7600	512K	8 MEGAFLOPS	1970
CDC STAR-100	1M	16 MEGAFLOPS	1975
CDC CYBER 205	4M	100 MEGAFLOPS	1981

CANDIDATES FOR FUTURE REQUIREMENTS

CYBER 2XX	32M	400 MEGAFLOPS	1987
CYBER 2XY	256M	1200 MEGAFLOPS	?

Figure 2. Some representative computer classes

The requirement to efficiently perform systematic operations on large quantities of floating point operations was quickly recognized in applied scientific computing. The architecture of the Control Data STAR-100 computer system was established in 1965 in order to provide a means of efficient large scale computation for exactly this reason (Purcell [1]). The computational capability provided by the STAR-100 can be compared to other classes of systems (see Figure 2). Various members of the CYBER 200 family are carrying the concepts of the STAR-100 further.

Engineers and scientists access this systematic capability within a FORTRAN language environment. FORTRAN is not an ideal language from the standpoint of computer science. The simplicity of FORTRAN, however, has attracted usage and utility throughout the world, especially as a method of information exchange. FORTRAN is also used in a large number of universal application packages. Use of these packages does not require any knowledge of FORTRAN at all. The only requirement is the need to enter data within specific format structures and the ability to understand the resultant data. This facility defines to large extent the requirement for the operating system, especially for the purposes of engineers and scientists. In addition, the owner of the computational resource requires that the many users are identified, counted, billed and protected, all at minimum disturbance of the overall system productivity.

Experience within Control Data Corporation led to an early implementation of the concept of distributed processing whereby each support process is to be performed at a level that is least expensive in terms of either main processor time or hardware expense. The main processor is designed to perform the computational function. Other functions are distributed to a variety of support units ("functional parallelism") as implemented in a number of auxiliary processors. In addition to a computational processor, functional parallelism requires facilities for queued access, batch preparation, telecommunication, output display, file support, and maintenance/ monitor functions.

The operating system is distributed in a manner which closely follows the distribution of the hardware. Thus, there are operating system functions in each support processor as well as in the central processor. The connecting links between the several processors are controlled by a diverse set of system messages, so that message handling becomes of great importance in the distributed system.

The development of the message philosophy over the last 20 years in Control Data Corporation has resulted in standard products of this type of facility. The product family is known as the "Loosely Coupled Network" with remote host facility. A remote host facility is any structure of hardware and software elements that supports Control Data local computer networks consisting of Control Data and non-Control Data hosts. The objectives of the remote host facility are:

- to support a local network of distributed hosts,
- to distribute network supervision tasks among host computer systems,
- to provide remote host access to local host system applications without concern for network topology,
- to provide efficient and effective use of network resources, and
- to ensure network integrity and maintainability.

All functions needed to implement the distributed functions required in the concept are readily available as three kinds of transfer mechanisms: queue files, permanent files and interactive files.

2. THE CYBER 205

The CYBER 205 with its immediate storage is simply another processor within the system, now identified as one of many "hosts" and in no way equipped with any extra authority. The only authority that it does have is

to process requests for service as directed by the message formats placed in the CPU memory. The support hardware provided within the CYBER 205 in order to facilitate the implementation of a minimum overhead operating system includes:

- monitor mode vs. job mode execution state,
- interrupt processing,
- virtual memory organization,
- variable size page structure,
- monitor mode instructions, and
- masked vector search instruction.

See the CYBER 205 Reference Manual ([2], Section 5) for a complete description of the various instructions and support features of the CYBER 205 hardware and software system.

The central operating system operates in a privileged state located in the beginning of memory. At the time of system initialization, the operator causes a master program to be written into that beginning of memory, then starts execution. Extensive hardware properties are provided in conjunction with an auxiliary processor (for maintenance purposes) so that the first instruction to be executed is found in a location specified by the contents of Register 6. The first eight registers of the 256 program registers have similar identified roles, as indicated in Figure 3.

```
REGISTER 0: TRACE REGISTER (OF LAST BRANCH)
REGISTER 1: DATA FLAG RETURN ADDRESS
REGISTER 2: DATA FLAG BRANCH ADDRESS
REGISTER 3: JOB MODE ILLEGAL.............TO MONITOR ADDRESS
REGISTER 4: MONITOR MODE ILLEGAL.........TO MONITOR ADDRESS
REGISTER 5: JOB EXIT FORCE...............TO MONITOR ADDRESS
REGISTER 6: EXTERNAL INTERRUPT...........TO MONITOR ADDRESS
REGISTER 7: JOB PAGE FAULT...............TO MONITOR ADDRESS
```

Figure 3. Monitor mode registers

Virtual memory is organized to provide each of 4096 potential users with a full address range of 48 bits. The use of a very large address space has precluded the need for segment tables for mapping virtual to real memory addresses. Thus, an associative search mechanism was developed in order to establish the location of "virtual" locations within "real"

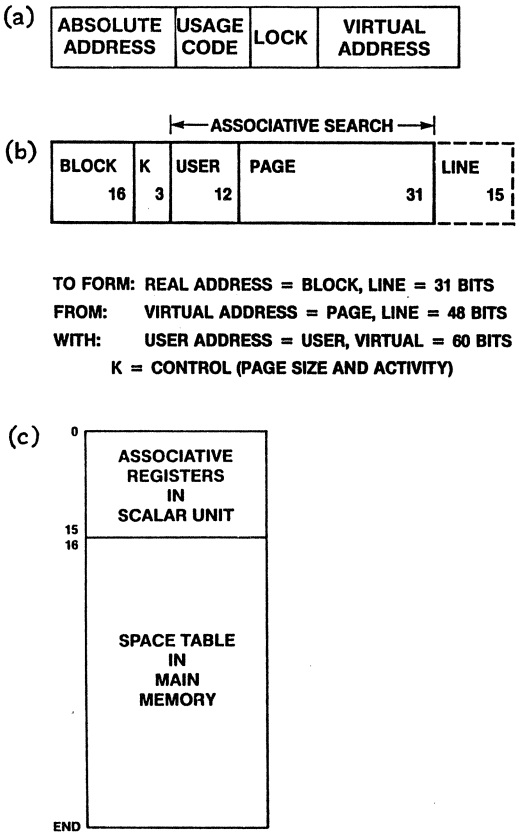


Figure 4. (a) Associative word (b) Page table entry format
(c) Page table format

memory. Figure 4 shows the format of the associative words and of the page table entries. The associative search is implemented so as to reorder the list in order to force the most recently used entries at the top of the list. The oldest entries are caused naturally to fall to the bottom of the list where they become prime candidates for replacement by more desirable page entries. A specific vector search instruction is provided for the operating systems to facilitate associative space table management (the "cc" instruction).

Figure 5 summarizes the organization of the CYBER 205 virtual memory system and Figure 6 lists some of the salient parameters. The maximum real memory size was established at 32 million words (of 256 million bytes) in 1965. This grand goal seemed to be sufficient at that time to carry us to the year 2000. We face implementation of this amount of memory by 1985, instead. I do not believe we will ever need the full 48 bits, however.

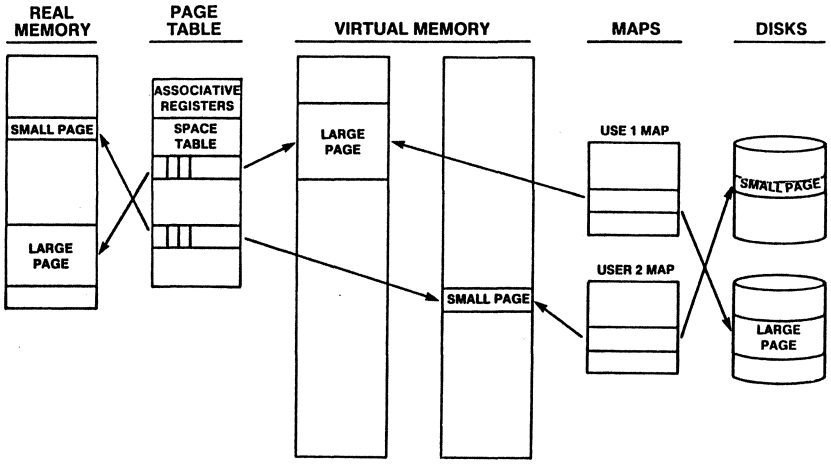


Figure 5. CYBER 205 virtual memory management (overview)

- **LOCK AND KEY PROTECTION**
- **16 VIRTUAL ADDRESS REGISTERS PLUS PAGE TABLE**
- **ADDRESS SPACE: 2×10^{12} WORDS**
- **SELECTABLE PAGE SIZES**
  - **SMALL PAGE SIZES: 512, 2048, AND 8192 WORDS**
  - **LARGE PAGE SIZE: 65,536 WORDS**
- **TRANSPARENT TO USER**

Figure 6. Some features of the CYBER 205 virtual memory system

Interrupt processing in the CYBER 205 is facilitated by categories established in conjunction with the information found in the previously noted registers for use in the monitor mode assignments. The interrupts transfer control as follows:

- page fault...to monitor,
- external interrupt...per channel,
- data fault...to own program,
- job time out...to monitor,
- monitor time out...to accounting,
- hardware fault...to maintenance by time out.

The system recovers from interrupts without loss of function or data. A partial interrupt (suspension) is required in order to allow for the asso-

ciative page table search within the span of execution of the memory-to-memory instructions of the CYBER 205. Full interrupt is supported by retention of all necessary restart "user program" parameters, at the expense of time and hardware but in exchange for substantial convenience.

3. OPERATING SYSTEM IMPLEMENTATION

There are four levels of program to be found within the CYBER 205. These levels include the central operating system operating in (absolute addressed) monitor mode, system tasks running in virtual mode, user services running in virtual mode and user programs in virtual mode. The time required to perform any identified function performed for or by a specific user, is charged to that user as much as possible.

The generalized configuration required to support a multi-programming environment with a large number of users is shown in the idealized networking scheme of Figure 7. The configuration of the CYBER 205 system is shown in more detail in Figure 8. Distributed processing is achieved by providing a computation facility with lower level processors. The CYBER 205 handles large scientific computation, with minimal support and I/O activity. Front-end processors handle unit record and tape I/O, remote access and data communication, and data management. The tape subsystem handles high performance tape I/O, and the disk subsystem handles high performance disk I/O.

Data flow for each user's program is managed by either the system or by the user, in conjunction with files primarily found in the disk system. This data is managed by the system when data and code spaces are referenced within

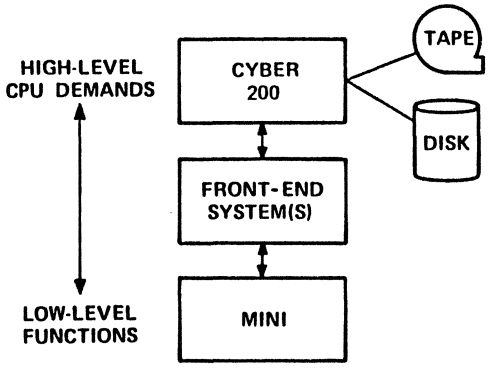


Figure 7. CYBER 205 distributed processing

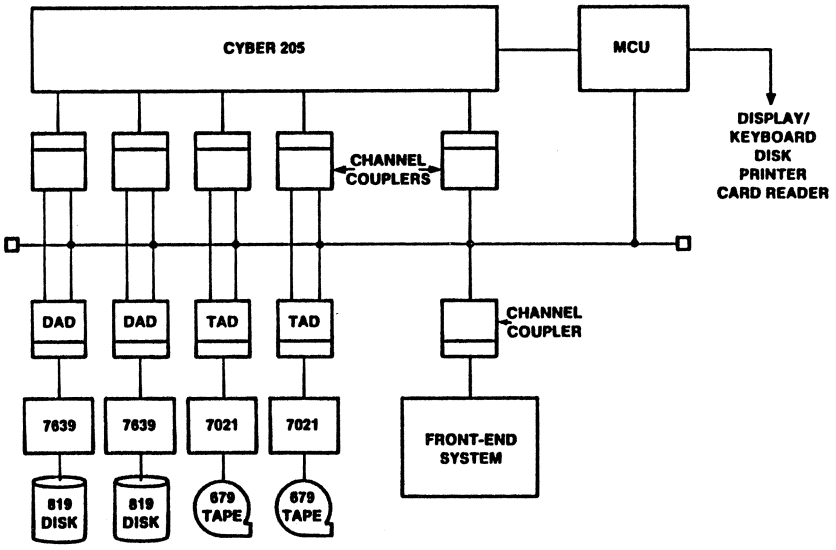


Figure 8. CYBER 205 general configuration

the rules of virtual memory. This mode, called demand paging, can work well for casual development of potential production codes.

Actual production code performance can be improved, as necessary, by manual operations such as "advise" or "buffer". These manual operations are invoked by specialists in performance programming whenever the cost or the time duration of a given production code becomes excessive. We have implemented the "advise" function so that the requested program or data can be retained or removed virtually with reasonable efficiency. Classical Control Data buffer operations (BUFFIN or BUFFOUT) require real memory operations in order to preserve compatibility with older practices, at some expense in efficiency.

Performance measurements on the CYBER 205 have been performed on a variety of user codes in both mono-programming and multi-programming environments. Successful programs have been reported by many scientists outside of Control Data Corporation. Information concerning operating system performance is usually difficult to establish outside or inside the company. An average efficiency of the current VSOS 2.1.5 has been measured by use of special instrumentation covering several multi-programming mixes at 75 percent CPU delivered to the user community. Figure 9 summarizes some of the results. The average time utilized by the system (not allocatable to specific users) is 10 percent. The average time spent in unoverlapped input/

	AVERAGE	RANGE
CPU DELIVERED TO USER	75%	55-95%
OPERATING SYSTEM OVERHEAD	10%	18-2%
NONPRODUCTIVE I/O WAIT	15%	27-3%
NEED FOR MORE REAL MEMORY	100%	
NEED FOR INCREASED DISK SUPPORT	100%	
VECTOR UNIT AVAILABLE FOR MORE WORK	..	

Figure 9. CYBER 205 performance measurements

output wait is 15 percent. These averages are established over a large number of cases and do not represent any specific case. The measurement technique includes analysis of system day files, CPU instrumentation, and the use of two pipe / four pipe performance comparisons.

The direction of the CYBER 200 Operating System Development is clear. The future requirements include the need for developing more support facilities in order to deliver more computations to the ever increasing simulation requirements of engineers and scientists (see Figure 10). We are told by the user community to develop faster processors, larger memories, bigger disk systems, more extensive terminal facilities and comprehensive graphics. All of this while reducing the cost of each computation and minimizing operating system overhead. Operating system utilities will be greatly expanded by means of the development of various command lan-

- **ADVANCED DISK SUPPORT (ACCESS AND TRANSFER)**
- **EXPANSION TO LOOSELY COUPLED NETWORK FACILITIES**
- **GATEWAYS TO LOW SPEED NETWORKS**
- **GREATLY INCREASED INTERACTIVE SUPPORT**
- **AUTOMATIC DEMAND PAGING HARDWARE SUPPORT**
- **INTELLIGENT FILE ACCESSES**
- **MULTIPLE ACCESS PRODUCTIVITY IMPROVEMENTS**
- **UNIVERSAL OPERATING SYSTEM WITH COMMAND LANGUAGE SHELLS**

Figure 10. Future possibilities CYBER 200

guage "shells" as UNIX, NOS and VSOS. Operation system efficiency will be greatly increased by additional hardware support functions which will improve queue, dequeue, request, accept, reject, wait, post, link heap, stack, etc., all along the lines of current developments in the theory of operating systems.

REFERENCES

1. C.J. PURCELL. *Using the CYBER 205*, Conference on Vector and Parellel Processors in Computational Science, Chester College, 25-28 August 1981.
2. *CDC CYBER 200 Model 205 Computer System: Hardware Reference Manual*, Publication 60256020, Control Data Corporation, Arden Hills, MN 55112, USA.

(Edited by JvL)

An Overview of the *Amoeba* Distributed Operating System

A.S Tanenbaum
Free University, Amsterdam

S.J. Mullender
Centre for Mathematics and Computer Science, Amsterdam

Fifth generation computer systems will use large numbers of processors to achieve high performance. In this paper a capability-based operating system designed for this environment is discussed. Capability-based operating systems have traditionally required large, complex kernels to manage the use of capabilities. In our proposal, capability management is done entirely by user programs without giving up any of the protection aspects normally associated with capabilities. The basic idea is to use one-way functions and encryption to protect sensitive information. Various aspects of the proposed system are discussed.

1980 Mathematics Subject Classification: 68A05, 68B20.
1982 CR Categories: C.2.2, C.2.4, D.4.4, D.4.6.
Keywords & Phrases: distributed operating systems, capabilities, connectionless protocols, transaction-oriented protocols, protection, accounting, file systems, service model.
Note: This paper has been submitted for publication elsewhere.

1. Introduction

Fifth generation computers must be fast, reliable, and flexible. One way to achieve these goals is to build them out of a small number of basic modules that can be assembled together to realize machines of various sizes. The use of multiple modules can make the machines not only fast, but also achieve a substantial amount of fault tolerance. The system architecture and software for such machines are described below.

1.1. System Architecture

The price of processors and memory is decreasing at an incredible rate. Extrapolating from the current trend, it is likely that a single board containing a powerful CPU, a substantial fraction of a megabyte of memory, and a fast network interface will be available for a manufacturing cost of less than $100 in 1990. Our intention is therefore to do research on the architecture and software of machines built up of a large number of such modules.

In particular, we envision three classes of machines: (1) personal computers consisting of a high-quality bit-map display and a few processor-memory modules; (2) departmental machines consisting of hundreds of such modules; and (3) large mainframes consisting of thousands of them. The primary difference between these machines is the number of modules, rather than the type of the modules. In principle, any of these machines can be gracefully increased in size to improve performance by adding new modules or decreased in size to allow removal and repair of defective modules. The software running on the various machines should be in essence identical. Furthermore, it should be possible to connect different machines together to form even larger machines and to partition existing machines into disjoint pieces when necessary, all in a way transparent to the user level software.

This model is superior to the oft-proposed "Personal Computer Model," in a number of ways. In the personal computer model, each user has a dedicated minicomputer, complete with disks, in his office or at home. Unfortunately, when people work together on large projects, having numerous local file systems can lead to multiple, inconsistent copies of many programs. Also, the noise generated by disks in every office, and the maintenance problems generated by having machines spread all over many buildings can be annoying.

Furthermore, computer usage is very bursty: most of the time the user does not need any computing power, but once in a while he may need a very large amount of computing power for a short time (e.g., when recompiling a program consisting of 100 files after changing a basic shared declaration). The fifth generation computer we propose is especially well suited to bursty computation. When a user has a heavy computation to do, an appropriate number of processor-memory modules are temporarily assigned to him. When the computation is completed, they are returned to the idle pool for use by other users.

1.2. System Software

A machine of the type described above requires radically different system software than existing machines. Not only must the operating system effectively use and manage a very large number of processors, but the communication and protection aspects are very different from those of existing systems.

Traditional networks and distributed systems are based on the concept of two processes or processors communicating via connections. The connections are typically managed by a hierarchy of complex protocols, usually leading to

complex software and extreme inefficiency. (An effective transfer rate of 0.1 megabit/sec over a 10 megabit/sec local network, which is only 1% utilization, is frequently barely achievable.)

We reject this traditional approach of viewing a distributed system as a collection of discrete processes communicating via multilayer (e.g., ISO) protocols, not only because it is inefficient, but because it puts too much emphasis on specific processes, and by inference, on processors. Instead we propose to base the software design on a different conceptual model – the object model. In this model, the system deals with abstract objects, each of which has some set of abstract operations that can be performed on it.

Associated with each object are one or more "capabilities" [1] which are used to control access to the object, both in terms of who may use the object and what operations he may perform on it. At the user level, the basic system primitive is performing an operation on an object, rather than such things as establishing connections, sending and receiving messages, and closing connections. For example, a typical object is the file, with operations to read and write portions of it.

The object model is well-known in the programming languages community under the name of "abstract data type" [5]. This model is especially well-suited to a distributed system because in many cases an abstract data type can be implemented on one of the processor-memory modules described above. When a user process executes one of the visible functions in an abstract data type, the system arranges for the necessary underlying message transport from the user's machine to that of the abstract data type and back. The header of the message can specify which operation is to be performed on which object. This arrangement gives a very clear separation between users and objects, and makes it impossible for a user to directly inspect the representation of an abstract data type by bypassing the functional interface.

A major advantage of the object or abstract data type model is that the semantics are inherently location independent. The concept of performing an operation on an object does not require the user to be aware of where objects are located or how the communication is actually implemented. This property gives the system the possibility of moving objects around to position them close to where they are frequently used. Furthermore, the issue of how many processes are involved in carrying out an operation, and where they are located is also hidden from the user.

It is frequently convenient to *implement* the object model in terms of clients (users) who send messages to services. A service is defined by a set of commands and responses. Each service is handled by one or more server processes that accept messages from clients, carry out the required work, and send back replies. The design of these servers and the design of the protocols they use form an important part of the system software of our proposed fifth generation computers.

As an example of the problems that must be solved, consider a file server. Among other design issues that must be dealt with are how and where information is stored, how and when it is moved, how it is backed up, how concurrent reads and writes are controlled, how local caches are maintained, how information is named, and how accounting and protection are accomplished. Furthermore, the internal structure of the service must be designed: how many server processes are there, where are they located, how and when do they communicate, what happens when one of them fails, how is a server process organized internally for both reliability and high performance, and so on. Analogous questions arise for all the other servers that comprise the basic system software.

2. Communication Primitives and Protocols

In the literature about computer networks, one finds much discussion of the ISO OSI reference model [12] these days. It is our belief that the price that must be paid in terms of complexity and performance in order to achieve an "open" system in the ISO sense is much too high, so we have developed a much simpler set of communication primitives, which we will now describe.

2.1. Transaction vs. Stream Communication

Most distributed systems have a connection mechanism that is based on the idea of two processes going to some effort to set up a connection, using the connection, and then tearing it down. The assumption is that a connection will be used for a stream of information so long that the overhead needed to set it up and tear it down are basically negligible. Most streams will consist of a file of one kind or another – a source program, a binary program, an input file, and so on. To see how long the average file is, we have conducted some measurements on the UNIX† system used in our department by the faculty and staff for research (no students, thus). The results of these measurements show that 34% of all files are less than 512 bytes, 52% are less than 1K bytes, 67% are less than 2K bytes, 79% are less than 4K bytes, 88% are less than 8K bytes, and 94% are less than 16K bytes.

The above considerations have led us to a different approach [8]. With packets of even 2K bytes, two thirds of all files fit into a single packet. Consequently, it is much simpler to adopt a "Request-Reply" or "Transaction" style of communication, in which the basic primitive is the client sending a request to a server and the server sending a reply back to the client. The client uses trans and the server getreq and putrep. Trans sends a request, and blocks until a reply is received. Getreq blocks the server until a request is received, which can then be processed, after which a reply can be sent using putrep. Each request-reply pair is completely self-contained, and independent of any other ones that

† UNIX is a Trademark of Bell Laboratories.

may previously been sent. In other words, no concept of a "connection" exists. Not only is this conceptually much more appropriate for use in an operating system, but it is much simpler to implement than a complex 7-layer protocol, not to mention offering lower delay. Henceforth we will refer to a request-reply pair as a *transaction*, which is not to be confused with transactions with a data base.

2.2. Basic Communication Protocol

Instead of a 7-layer protocol, we effectively have a 4-layer protocol. The bottom layer is the Physical Layer, and deals with the electrical, mechanical and similar aspects of the network hardware. The next layer is the Port Layer, and deals with the location of services, the transport of (32K byte) datagrams (packets whose delivery is not guaranteed) from source to destination and enforces the protection mechanism, which will be discussed in the next section. On top of this we have a layer that deals with the reliable transport of bounded length (32K byte) requests and replies between client and server. We have called this layer the Transaction Layer. The final layer has to do with the semantics of the requests and replies, for example, given that one can talk to the file server, what commands does it understand. The bottom three layers (Physical, Port and Transaction) are implemented by the kernel and hardware; only the Transaction Layer interface is visible to users.

Since systems of the kind we are describing will use high-speed, highly reliable local networks, few, if any, of the complex mechanisms designed for flow- and error-control in long-haul networks are useful here. Among other things, a simple stop-and-wait protocol is sufficient. The main function of the Transaction Layer is to provide an end-to-end *message* service built on top of the underlying *datagram* service, the main difference being that the former uses timers and acknowledgements to guarantee delivery whereas the latter does not.

The Transaction Layer protocol is straightforward. When the client does a trans, a packet, or sequence of packets, containing the request is sent to the server, the client is blocked, and a timer is started (inside the Transaction Layer). If the server does not acknowledge receipt of the request packet before the timer expires (usually by sending the reply, but in some special cases by sending a separate acknowledgement packet), the Transaction Layer retransmits the packet again and restarts the timer. When the reply finally comes in, the client sends back an acknowledgement (possibly piggybacked onto the next request packet) to allow the server to release any resources, such as buffers, that were acquired for this transaction. Under normal circumstances, reading a long file, for example, consists of the sequence

From client: request for block 0
From server: here is block 0
From client: acknowledgement for block 0 and request for block 1
From server: here is block 1

etc.

The protocol can handle the situation of a server crashing and being rebooted quite easily since each request contains the capability for the file to be read and the position in the file to start reading. Between requests, the server has no "activation record" or other table entry whose loss during a crash causes the server to forget which files were open, etc., because no concept of an open file or a current position in a file exists on the server's side. Each new request is completely self-contained. Of course for efficiency reasons, a server may keep a cache of frequently accessed i-nodes, file blocks etc., but these are not essential and their loss during a crash will merely slow the server down slightly while they are being dynamically refreshed after a reboot.

2.3. The Port Layer

The Port Layer is responsible for the speedy transmission of 32K byte datagrams. The Port Layer need only do this reasonably reliably, and does not have to make an effort to guarantee the correct delivery of every datagram. This is the responsibility of the Transaction Layer. Our results show that this approach leads to significantly higher transmission speeds, due to simpler protocols.

Theoretically, very high speeds are achievable in modern local-area networks. A typical speed for DMA transfers is 1 byte/μsec, and the typical transmission speed of a 10 Mbit local-area network is also 1 byte/μsec. Since DMA transfer and network transfer cannot overlap, but DMA at the destination host *can* overlap with the DMA of the next packet at the source host, an upper bound for the transfer rate of a typical local-area network is 500,000 bytes/sec point-to-point.

In practise, however, speeds of 100,000 bytes per second between user processes have rarely been achieved. Obviously, to achieve higher transmission rates, the overhead of the protocol must be kept very low indeed, while an effort must be made to overlap DMAs at both communicating parties. To achieve this, we have chosen a large datagram size for the Port Layer, which has to split up the datagrams into small packets that the network hardware can cope with. This approach allows the implementor of the Port Layer to exploit the possibilities that the hardware has to offer to achieve an efficient stream of packets.

Our implementation of the Port Layer interfaces to a 10 Mbit token ring that allows *scatter-gather*; that is, a packet can be sent to or from the interface in several DMA transfers, and then transmitted over the network separately. This allows us to do two important things to speed up the protocol. First, when a packet is received, the header can be inspected separately, so the protocol can decide where in memory the packet must go. The protocol driver can then transfer the packet directly from the interface to the right place in memory, without having to copy it. A copy loop would halve the transmission speed. Second, the separation of DMA and transmission allows the driver to prepare a

transmission by doing the DMA. The transmission can then be initiated immediately when the signal is received that the receiver is ready. In our implementation of the Port Layer, these considerations have resulted in the protocol that will now be described.

The transmitter begins by transferring and sending the first 2K of the datagram to be transmitted (2K is the maximum packet size allowed by the hardware). Immediately after the transmission is complete, the DMA for the next 2K bytes is started, but they are not yet transmitted. In the mean time, the receiver is interrupted by the arrival of the first packet. It extracts the header, examines it and decides where the body of the packet should go. Then the body of the packet is transferred from the interface to its final location in memory. While this is being done, the receiver prepares a tiny *acknowledgement* packet to tell the transmitter it is prepared for the next packet. As soon as the DMA transfer of the previous packet has finished, this acknowledgement is sent back to the transmitter. When the transmitter receives it, the transfer of the next packet to the interface will have finished, so it can then be sent immediately. This sequence is continued until the whole datagram is transmitted.

2.4. The Transaction Layer

It is the responsibility of the Transaction Layer to guarantee the arrival of requests and replies. The Transaction Layer makes use of the Port Layer and timers to achieve this.

The interface to the transaction layer basically consists of three calls, one for clients, and two for servers. All calls use a small datastructure, called Mref, which contains a pointer to a small fixed-size out-of-band buffer for the transmission of commands and parameters to the server, a pointer to the main body of data to be transferred, and the length of the main body of data (0 to 32768), as follows:

```
typedef struct Mref {
        char     *M_oob;
        char     *M_buf;
        unsigned  M_len;
} Mref;

typedef struct Cap {
        Port      C_port;           /* 6-byte port */
        char      C_private[10];    /* 10-byte private */
} Cap;    /* capability */
```

The client, in order to do a transaction calls

```
trans(cap, req, rep);
        Cap *cap;
        Mref *req, *rep;
```

The server receives requests and sends replies with

```
getreq(port, cap, req);
        Port *port;
        Cap *cap;
        Mref *req;

putrep(rep);
        Mref *rep;
```

In principle, the Transaction Layer works as follows: When a client calls trans, the Transaction Layer generates a *reply-port* to enable the server to send a reply. The server port is deduced from the capability; the first 48 bits of the capability for an object identify the service that controls the object. The request is then sent, using put, and a *retransmission timer* is started.

The server, which previously had made a call to getreq, receives the request; the capability is filled in, and the received message is put in the buffers referred to by req. As soon as the request is received, the server's Transaction Layer starts a *piggyback timer*. When the server has not sent a reply before this timer expires, a separate acknowledgement is sent to put the client at ease, and stop its retransmission timer. When the server sends a reply to the client the same thing happens, more or less, with the role of client and server reversed. When a client makes a sequence of transactions with a single server, a subsequent request will acknowledge receipt of the previous reply.

The client maintains one more timer, the *crash timer*. This timer is set when the server's acknowledgement to a request has been received, and is used to detect server crashes. Whenever this timer expires, the client sends an "are you still alive?" packet to the server, to which the server replies with an acknowledgement.

When transactions occur quickly, one after the other, no extra acknowledgements are sent at all. Only when transactions take a long time (say, longer than a minute), acknowledgements are sent, and when transactions take much longer than that (say, ten minutes) then "are you still alive" messages begin to be sent.

2.5. Timer Management

If the timers are started and stopped in exactly the way described above, the Transaction Layer would become unacceptably slow. Per (quick) transaction, two retransmission timers and two piggyback timers would have to be started and stopped, eight timer actions altogether.

There is a much more efficient way of dealing with timers, one that makes use of a *sweep algorithm*. This algorithm does not implement very accurate timers, but accuracy of the timer intervals is not very important to the correct and efficient operation of the protocol.

The sweep algorithm is run every N clock ticks. N must be chosen such that N ticks is about the minimum timer interval needed (the piggyback timer interval). Whenever the algorithm is called, it makes a sweep over all outstanding

transactions. If the state of a transaction has changed, the new state is recorded. If it has not changed, a counter is incremented, telling for how long the state has remained the same. If the (state, counter) combination has reached a certain value, the sweep algorithm carries out the appropriate actions, usually sending an acknowledgement, retransmitting a message, or aborting a transaction.

Because this algorithm is used there is no code needed in the transaction code itself, reducing the overhead of the Transaction Layer significantly. In this way, the code executed in the Transaction Layer is optimised for the normal case (no errors).

2.6. Blocking vs. Non-Blocking Transaction Primitives

Most services need to be able to handle multiple requests from different clients simultaneously. It therefore seems natural to implement non-blocking calls for interprocess communication, as this will allow a service to react to events in the order they occur. When blocking communication calls are used, a server is forced to wait for the specific event that unblocks the call.

Because it is rather difficult to write correct code for a process which has to handle multiple flows of control indeterministically, the ***Amoeba*** system provides the concept of *tasks*, sharing an address space. A number of tasks in one address space forms a *cluster*, and specific rules govern the scheduling of tasks within a cluster: only one task can run at a time, and a task runs until it voluntarily relinquishes control (e.g., on trans and getreq calls).

A server can thus easily be structured as a collection of co-operating tasks, each task handling one request. This model has greatly simplified the structure of services, as each task making up the server cluster now has a single thread of execution. The model also obviated the need for non-blocking transaction calls, with their complicated (and slow) extra interface for handling interrupts.

2.7. Results

Two versions of the algorithm have now been implemented. The one described has been implemented on the ***Amoeba*** distributed operating system, and achieves over 300,000 bytes a second from user process to user process (using M68000s and a Pronet* ring). It is now being implemented under UNIX where we expect to obtain more than 200,000 bytes/sec, assuming the communicating processes are not swapped.

An older version of the protocol, using 2K byte datagrams, now gets 90,000 bytes/sec across the network between two VAX-750s running a normal load of work, without causing a significant load on the system itself.

Several services, implemented under UNIX, are using the Transaction Layer interface, and it is our experience that these services are easy to design and that

* PRONET is a trademark of Proteon Associates, Inc.

they work efficiently.

3. Ports and Capabilities

3.1. Ports

Every service has one or more *ports* [7] to which client processes can send messages to contact the service. Ports consist of large numbers, typically 48 bits, which are known only to the server processes that comprise the service, and to the service's clients. For a public service, such as the system file service, the port will be generally made known to all users. The ports used by an ordinary user process will, in general, be kept secret. Knowledge of a port is taken by the system as prima facie evidence that the sender has a right to communicate with the service. Of course the service is not required to carry out work for clients just because they know the port, for example, the public file service may refuse to read or write files for clients lacking account numbers, appropriate authorization, etc.

Although the port mechanism provides a convenient way to provide partial authentication of clients ("if you know the port, you may at least talk to the service"), it does not deal with the authentication of servers. The basic primitive operations offered by the system are `trans`, `putreq` and `getrep`. Since everyone knows the port of the file server, as an example, how does one insure that malicious users do not execute `getreq`s on the file server's port, in effect impersonating the file server to the rest of the system?

One approach is to have all ports manipulated by kernels that are presumed trustworthy and are supposed to know who may `getreq` from which port. We reject this strategy because some machines, e.g., personal computers connected to larger multimodule systems may not be trustworthy, and also because we believe that by making the kernel as small as possible, we can enhance the reliability of the system as a whole. Instead, we have chosen a different solution that can be implemented in either hardware or software. First we will describe the hardware solution; later we will describe the software solution.

In the hardware solution, we need to place a small interface box, which we call an F-box (Function-box) between each processor module and the network. The most logical place to put it is on the VLSI chip that is used to interface to the network. Alternatively, it can be put on a small printed circuit board inside the wall socket through which personal computers attach to the network. In those cases where the processors have user mode and kernel mode and a trusted operating system running in kernel mode, it can also be put into operating system software. In any event, we assume that somehow or other all packets entering and leaving every processor undergo a simple transformation that users cannot bypass.

The transformation works like this. Each port is really a pair of ports, P, and G, related by: $P = F(G)$, where F is a (publicly-known) one-way function

[14, 10, 3] performed by the F-box. The one-way function has the property that given G it is a straightforward computation to find P, but that given P, finding G is so difficult that the only approach is to try every possible G to see which one produces P. If P and G contain sufficient bits, this approach can be made to take millions of years on the world's largest supercomputer, thus making it effectively impossible to find G given only P. Note that a one-way function differs from a cryptographic transformation in the sense that the latter must have an inverse to be useful, but the former has been carefully chosen so that no inverse can be found.

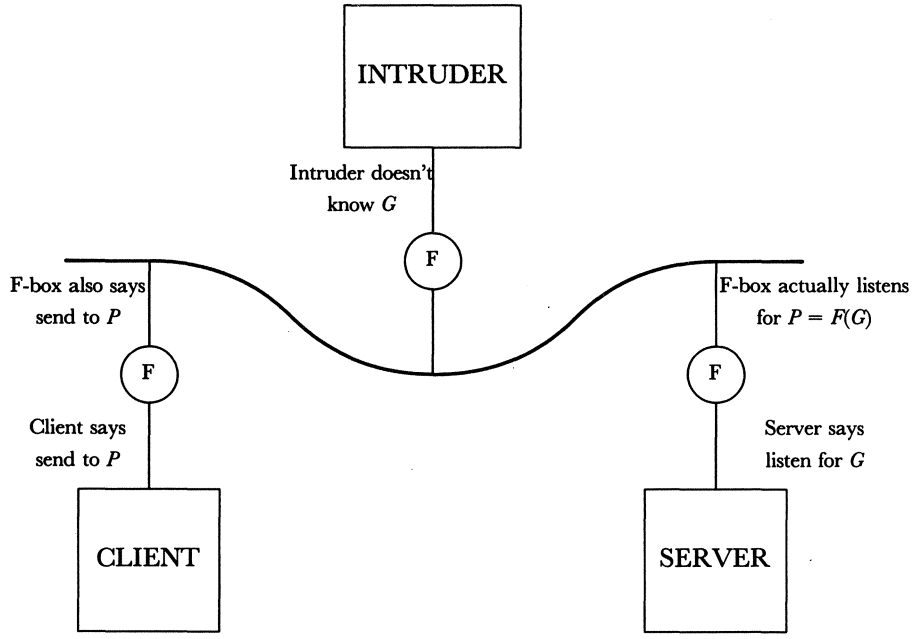


FIGURE 1.

Using the one-way F-box, the server authentication can be handled in a simple way, illustrated in FIGURE 1. Each server chooses a get-port, G, and computes the corresponding put-port, P. The get-port is kept secret; the put-port is distributed to potential clients or in the case of public servers, is published. When the server is ready to accept client requests, it does a `getreq(G, cap, req)`. The F-box then computes $P = F(G)$ and waits for packets containing P to arrive. When one arrives, it is given to the appropriate process. To send a packet to the server, the client merely does `trans(cap, req, rep)`, where the *port* field of `cap` is set to P. This will cause a datagram to be sent by the local F-box with P in the destination-port field of the header. The F-box on the

sender's side does not perform any transformation on the P field of the outgoing packet.

Now let us consider the system from an intruder's point of view. To impersonate a server, the intruder must do `getreq(`G`, ··· )`. However, G is a well-kept secret, and is never transmitted on the network, Since we have assumed that G cannot be deduced from P (the one-way property of F) and that the intruder cannot circumvent the F-box, he cannot intercept packets not intended for him. Replies from the server to the client are protected the same way, only with the client's Transaction Layer picking a get-port for the reply, say, G', and including $P' = F(G')$ in the request packet.

The presence of the F-box makes it easy to implement digital signatures for still further authentication, if that is desired. To do so, each client chooses a random signature, S, and publishes $F(S)$. The F-box must be designed to work as follows. Each packet presented to the F-box contains three special header fields: destination (P), reply (G'), and signature (S). The F-box applies the one-way function to the second and third of these, transmitting the three ports as: P, $F(G')$, and $F(S)$, respectively. The first is used by the receiver's F-box to admit only packets for which the corresponding `getreq` has been done, the second is used as the put-port for the reply, and the third can be used to authenticate the sender, since only the true owner of the signature will know what number to put in the third field to insure that the publicly-known $F(S)$ comes out.

It is important to note that the F-box arrangement merely provides a simple *mechanism* for implementing security and protection, but gives operating system designers considerable latitude for choosing various *policies*. The mechanism is sufficiently flexible and general that it should be possible to put it into hardware with precluding many as-yet-unthought-of operating systems to be designed in the future.

3.2. Capabilities

In any object-based system, a mechanism is needed to keep track of which processes may access which objects and in what way. The normal way is to associate a capability with each object, with bits in the capability indicating which operations the holder of the capability may perform. In a distributed system this mechanism should itself be distributed, that is, not centralized in a single monolithic "capability manager." In our proposed scheme, each object is managed by some service, which is a user (as opposed to kernel) program, and which understands the capabilities for its objects.

A capability typically consists of four fields, as illustrated in FIGURE 2:

1. The put-port of the service that manages the object
2. An Object Number meaningful only to the service managing the object
3. A Rights Field, which contains a 1 bit for each permitted operation
4. A Random Number for protecting each object

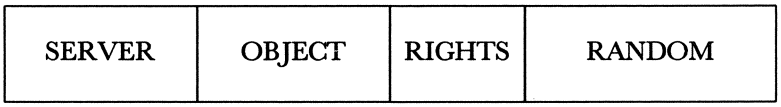


FIGURE 2.

The basic model of how capabilities are used can be illustrated by a simple example: a client wishes to create a file using the file service, write some data into the file, and then give another client permission to read (but not modify) the file just written. To start with, the client sends a message to the file service's put-port specifying that a file is to be created. The request might contain a file name, account number and similar attributes, depending on the exact nature of the file service. The server would then pick a random number, store this number in its object table, and insert it into the newly-formed object capability. The reply would contain this capability for the newly created (empty) file.

To write the file, the client would send a message containing the capability and some data. When the `write` request arrived at the file server process, the server would normally use the object number contained in the capability as as index into its tables to locate the object. For a UNIX like file server, the object number would be the i-node number, which could be used to locate the i-node.

Several object protection systems are possible using this framework. In the simplest one, the server merely compares the random number in the file table (put there by the server when the object was created) to the one contained in the capability. If they agree, the capability is assumed to be genuine, and all operations on the file are allowed. This system is easy to implement, but does not distinguish between `read`, `write`, `delete`, and other operations that may be performed on objects.

However, it can easily be modified to provide that distinction. In the modified version, when a file (object) is created, the random number chosen and stored in the file table is used as an encryption/decryption key. The capability is built up by taking the Rights Field (e.g., 8 bits), which is initially all 1s indicating that all operations are legal, and the Random Number Field (e.g., 56 bits), which contains a known constant, say, 0, and treating them as a single number. This number is then encrypted by the key just stored in the file table, and the result put into the newly minted capability in the combined Rights-Random Field. When the capability is returned for use, the server uses the object number (not encrypted) to find the file table and hence the encryption/decryption key. If the result of decrypting the capability leads to the known constant in the Random Number Field, the capability is almost assuredly valid, and the Rights Field can be believed. Clearly, an encryption function that mixes the bits thoroughly is required to ensure that tampering with the

Rights Field also affects the known constant. Exclusive or'ing a constant with the concatenated Rights and Random fields will not do.

When this modified protection system is used, the owner of the object can easily give an exact copy of the capability to another process by just sending it the bit pattern, but to pass, say, read-only access, is harder. To accomplish this task, the process must send the capability back to the server along with a bit mask and a request to fabricate a new capability whose Rights Field is the Boolean-and of the Rights Field in the capability and the bit mask. By choosing the bit mask carefully, the capability owner can mask out any operations that the recipient is not permitted to carry out.

This modified system works well except that it requires going back to the server every time a sub-capability with fewer rights is needed. We have devised yet another protection system that does not have this drawback. This third scheme requires the use of a set of N commutative one-way functions, $F_0, F_1, \cdots, F_{N-1}$ corresponding to the N rights present in the Rights Field. When an object is created, the server chooses a random number and puts it in both the file table and the Random Number Field, just as in the first scheme presented. It also sets all the Rights Field bits to 1.

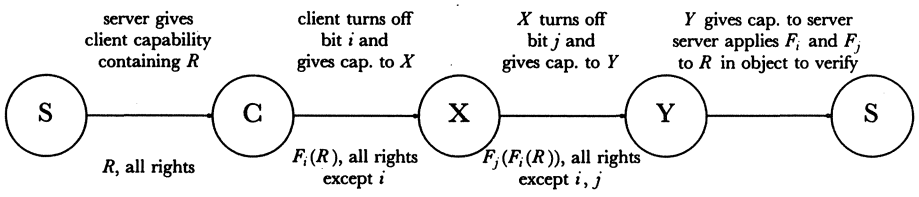


FIGURE 3.

A client can delete permission k from a capability by replacing the random number, R, with $F_k(R)$ and turning off the corresponding bit in the Rights Field. When a capability comes into the server to be used, the server fetches the original random number from the file table, looks at the Rights Field, and applies the functions corresponding to the deleted rights to it. If the result agrees with the number present in the capability, then the capability is accepted as genuine, otherwise it is rejected. The mechanism is illustrated in FIGURE 3. Note that although the Rights Field is not encrypted, it is pointless for a client to tamper with it, since the server will detect that immediately. In theory at least, the Rights Field is not even needed, since the server could try all 2^N combinations of the functions to see if any worked. Its presence merely speeds up the checking. It should also be clear why the functions must be commutative — it does not matter in what order the bits in the Rights Field were turned off.

The organization of capabilities and objects discussed above has the interesting property that although no central record is kept of who has which capabilities, it is easy to retract existing capabilities. All that the owner of an object need do is ask the server to change the random number stored in the file table. Obviously this operation must be protected with a bit in the Rights Field, but if it succeeds, all existing capabilities are instantly invalidated.

3.3. Protection without F-Boxes

Earlier we said that protection could also be achieved without F-boxes. It is slightly more complicated, since it uses both conventional and public-key encryption, but it is still quite usable. The basic idea underlying the method is the fact that in nearly all networks an intruder can forge nearly all parts of a packet being sent except the source address, which is supplied by the network interface hardware. To take advantage of this property, imagine a (possibly symmetric) conceptual matrix of conventional (e.g., DES) encryption keys, with the rows being labeled by source machine and the columns by destination machine. Thus the matrix selects a unique key for encrypting the *capabilities* in any packet. The data need not be encrypted, although that is also possible if needed.

Each machine is assumed to know its row and column of the matrix, and nothing else (how this will be achieved will be discussed shortly). With this arrangement, intruder I can easily capture packets from client C to server S, but attempts to "play them back" to the server will fail because the server will see the source machine as I (assumed unforgeable) and use element M_{IS} as the decryption key instead of the correct M_{CS}. No matter what the intruder does, he cannot trick the server into using a decryption key that decrypts the capabilities to make sense, that is, to contain random numbers that agree with those stored in the file tables.

To avoid having to run the encryption/decryption algorithm frequently, all machines can maintain a hashed cache of capabilities that they have been using frequently. Clients will hash their caches on the unencrypted capabilities in the form of triples: (unencrypted capability, destination, encrypted capability), whereas servers will hash theirs in the form of triples: (encrypted capability, source, unencrypted capability).

To set up the matrix initially, the following procedure can be used. A public server, such as a file server, makes its put-port and a public encryption key known to the whole world. When a new machine joins the network (e.g., after a crash or upon initial system boot), it sends a broadcast message announcing its presence. Suppose, for example, the file server has just come up, and must (1) prove that it is the file server to other processes, and (2) establish the conventional keys used for encrypting capabilities in both directions.

A client machine, C, which receives the broadcast from the alleged file server, F, picks a new conventional encryption key, K, for use in subsequent C to F

traffic and sends it to F encrypted with F's public key. F then decrypts K and replies to C by sending a packet containing both K and a newly chosen conventional key to be used for reverse traffic. This packet is encrypted both with K itself and with the inverse of F's public key, so C can use K and F's public key to decrypt it. If the decrypted packet contains K, C can be sure that the other conventional key was indeed generated by the owner of F's public key, thus convincing C that he is indeed talking to the file server. Both of the above-mentioned conditions have now been fulfilled, so normal communication can now take place. Note that the use of different conventional keys after each reboot make it impossible for an intruder to fool anyone by playing back old packets.

4. The Amoeba File System

The file system has been designed to be highly modular, both to enhance reliability and to provide a convenient testbed for doing research on distributed file systems. It consists of three completely independent pieces: the block service, the file service, and the directory service. In short, the block service provides commands to read and write raw disk blocks. As far as it is concerned, no two blocks are related in any way, that is, it has no concept of a file or other aggregation of blocks. The file service uses the block service to build up files with various properties. Finally, the directory service provides a mapping of symbolic names onto object capabilities.

4.1. Block Service

The block service is responsible for managing raw disk storage. It provides an object-oriented interface to the outside world to relieve file servers from having to understand the details of how disks work. The principle operations it performs are:

- `allocate` a block, write data into it, and return a capability to the block
- given a capability for a block, `free` the block
- given a capability for a block, `read` and return the data contained in it
- given a capability for a block and some data, `write` the data into the block
- given a capability for a block and a key, `lock` or unlock the block

These primitives provide a convenient object-oriented interface for file servers to use. In fact, any client who is unsatisfied [11,13] with the standard file system can use these operations to construct his own.

The first four operations of `allocate`, `free`, `read`, and `write` hardly need much comment. The fifth one provides a way for clients to lock individual blocks. Although this mechanism is crude, it forms a sufficient basis for clients (e.g., file systems) to construct more elaborate locking schemes, should they so desire. One other operation is worth noting. The data within a block is entirely under the control of the processes possessing capabilities for it, but we expect that most file servers will use a small portion of the data for redundancy

purposes. For example, a file server might use the first 32 bits of data to contain a file number, and the next 32 bits to contain a relative block number within the file. The block server supports an operation `recovery`, in which the client provides the account number it uses in `allocate` operations and requests a list of all capabilities on the whole disk containing this account number. (The block server stores the account number for each block in a place not accessible to clients.) Although `recovery` is a very expensive operation, in effect requiring a search of the entire disk, armed with all the capabilities returned, a file server that lost all of its internal tables in a crash could use the first 64 bits of each block to rebuild its entire file list from scratch.

4.2. File Service

The purpose of splitting the block service and file service is to make it easy to provide a multiplicity of different file services for different applications. One such file service that we envision is one that supports flat files with no locking, in other words, the UNIX model of a file as a linear sequence of bytes with no internal structure and essentially no concurrency control. This model is quite straightforward and will therefore not be discussed here further.

A more elaborate file service with explicit version and concurrency control for a multiuser environment will be described instead [6]. This file service is designed to support data base services, but it itself is just an ordinary, albeit slightly advanced, file service. The basic model behind this file service is that a file is a time-ordered sequence of versions, each version being a snapshot of the file made at a moment determined by a client. At any instant, exactly one version of the file is the *current version*. To use a file, a client sends a message to a file server process containing a file capability and a request to create a new, private version of the current version. The server returns a capability for this new version, which acts like it is a block for block copy of the current version made at the instant of creation. In other words, no matter what other changes may happen to the file while the client is using his private version, none of them are visible to him. Only changes he makes himself are visible.

Of course, for implementation efficiency, the file is not really copied block for block. What actually happens is that when a version is created, a table of pointers (capabilities) to all the file's blocks is created. The capability granted to the client for the new version actually refers to this version table rather than the file itself. Whenever the client reads a block from the file, a bit is set in the version table to indicate that the corresponding block has been read. When a block is modified in the version, a new block is allocated using the block server, the new block replaces the original one, and its capability is inserted into the version table. A bit indicating that the block is a new one rather than an original is also set. This mechanism is sometimes called "copy on write."

Versions that have been created and modified by a client are called *uncommitted versions*. At a particular moment, the current version may have several

(different) uncommitted versions derived from it in use by different clients. When a client is finished modifying his private version, he can ask the file server to *commit* his version, that is, make it the current version instead of the then current version. If the version from which the to-be-committed version was derived is still current at the time of the commit, the commit succeeds and becomes the new current version.

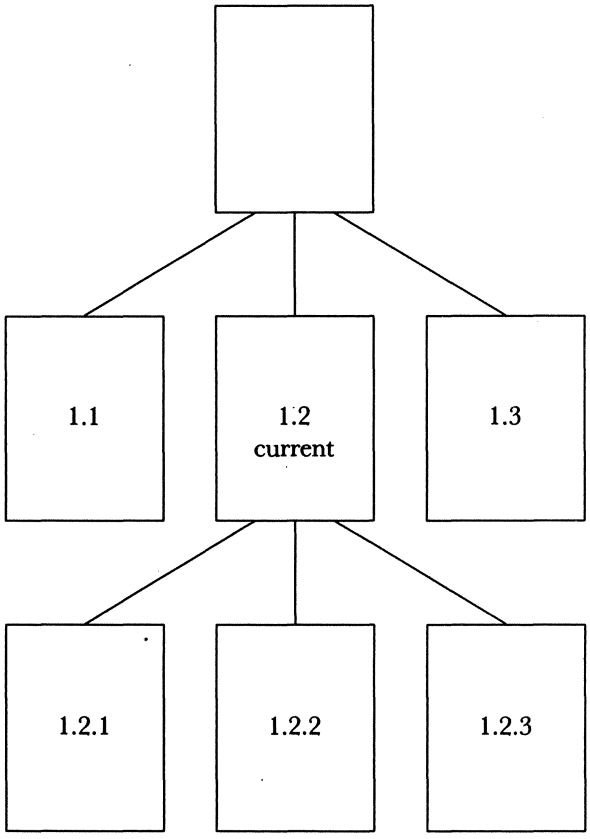


FIGURE 4.

As an example, suppose version 1 is initially the current version, with various clients creating private versions 1.1, 1.2, and 1.3 based on it. If version 1.2 is the first to commit, it wins and 1.2 becomes the new current version, as illustrated in FIGURE 4. Subsequent requests by other clients to create a version will result in versions 1.2.1, 1.2.2, and 1.2.3, all initially copies of 1.2.

The fun begins when the owner of version 1.3 now tries to commit. Version 1, on which it is based, is no longer the current version, so a problem arises. To

see how this should be handled, we must introduce a concept from the data base world, *serializability* [2,9]. Two updates to a file are said to be serializable if the net result is either the same as if they were run sequentially in either order. As a simple example, consider a two character file initially containing "ab." Client 1 wants to write a "c" into the first character, wait a while, and then write a "d" into the second character. Client 2 wants to write an "e" into the first character, wait a while, and then write an "f" into the second character. If 1 runs first we get "cd"; if 2 runs first we get "ef." Both of these are legal results, since the file server cannot dictate when the users run. However, its job is to prevent final configurations of "cf" or "de," both of which result from interleaving the requests. If a client locks the file before starting, does all its work, and then unlocks the file, the result will always be either "cd" or "ef," but never "cf" or "de." What we are trying to do is accomplish the same goal without using locking.

The idea behind not locking is that most updates, even on the same file, do not affect the same parts of the file, and hence do not conflict. For example, changes to an airline reservation data base for flights from San Francisco to Los Angeles do not conflict with changes for flights from Amsterdam to London. The strategy behind our commit mechanism is to let everyone make and modify versions at will, with a check for serializability when a commit is attempted. This mechanism has been proposed for data base systems [4], but as far as we know, not for file systems.

The serializability check is straightforward. If a version to be committed, A, is based on the version that is still current, B, it is serializable and the commit succeeds. If it is not, a check must be made to see if all of the blocks belonging to A that the client has read are the same in the current version as they were in the version from which A was derived. If so, the previous commit or commits only changed blocks that the client trying to commit A was not using, so there is no problem and the commit can succeed.

If, however, some blocks have been changed, modifications that A's owner has made may be based on data that are now obsolete, so the commit must be refused, but a list is returned to A's owner of blocks that caused conflicts, that is, blocks marked "read" in A and marked "written" in the current version (or any of its ancestors up to the version on which A is based). At this point, A's owner can make a new version and start all over again. Our assumption is that this event is very unlikely, and that is occasional occurrence is a price worth paying for not having locking, deadlocks, and the delays associated with waiting for locks.

4.3. Directory Service

Because it is frequently inconvenient to deal with long binary bit strings such as capabilities, a directory service is needed to provide symbolic naming. The directory service's task is to manage directories, each of which contains a collection of (ASCII name, capability) pairs. The principal operation on a directory object is for a client to present a capability for a directory and an ASCII name, and request the directory service to look up and return the capability associated with the ASCII name. The inverse operation is to store an (ASCII name, capability) pair in a directory whose capability is presented.

5. PROCESS MANAGEMENT

Like any other operating system, this one must also have a way to manage processes. In our design, processes are created and managed by the process service, which consists of three major subsystems, the generic server, the process server, and the boot server.

5.1. Generic Server

The idea behind the generic server is that much of the time a user wants a certain program to be run, but does not care about where it is run or on which CPU type. For example, a user might have a Pascal program to be compiled, and wants a Pascal compiler that produces, say, Motorola 68000 code. However, he does not care whether the compiler itself runs on a 68000, a VAX or any other CPU. We speak of this as a generic Pascal compiler.

The generic server's job is to locate a suitable hardware/software combination and start it up. This can be done by maintaining internal tables of locations where the appropriate service is likely to be located. By sending a message to the chosen service, the generic server can see if the corresponding server is currently available and willing to take on the offered work. If so, it can begin; if not, the generic server can broadcast a request for bids to see if someone else can be located. If no willing server exists, the generic server will have to cause one to be created by invoking the process server.

5.2. Process Server

The process server's job is to take a process descriptor sent to it, locate a free processor, and send sufficient information to the processor to allow the processor to run. The process descriptor must contain at least the following information:

1. The CPU type desired.
2. A capability for the binary file to be executed.
3. Capabilities for process environment.
4. Accounting information.

The CPU type and binary file capability are obvious. The third item has to do with things like the file descriptors and environment strings in UNIX. When a UNIX process is started up, it inherits certain parameters from its parent, among

these are usually file descriptors for standard input, output, and diagnostic, and possibly other files as well. In our design, a process can inherit capabilities for standard input, standard output, and standard diagnostic, as well as other ones. By using these, one can implement UNIX pipes and filters easily, as well as more general mechanisms (e.g., passing capabilities to third parties, storing them in files for later use, etc.).

Another area that the process service must deal with is scheduling. It must allocate processes to processors, and possibly control migration and swapping among processors as well. By introducing the concept of a "process image" which contains all the information necessary to run a process (e.g., its memory, registers, capabilities, etc.) it becomes straightforward to handle process migration and swapping in a unified way. When a process is swapped out to a disk somewhere, there is no need to have it swapped back to the same machine that it originated on.

5.3. Boot Service

Many services must achieve high availability. Our approach to this issue is using fault tolerance, rather than fault intolerance. In the former, one expects hardware and software to fail, and makes provision for dealing with it; in the latter, one assumes that they are perfect and that no such provision need be made. Since many services are faced with the same problem: how to provide high availability in the face of occasional crashes, we have abstracted out a common part of the crash recovery mechanism and put it into a separate service, the boot service.

Any service that wants to provide a continuous availability can register with the boot service. Such registration entails providing a polling message to send the service periodically, the expect reply, the polling frequency, and a prescription of what to do in case of failure. The boot service then sends the polling message to the service at the requested frequency. As long as the service continues to send the appropriate reply, all is well and the boot service has nothing else to do.

However, if the service fails to reply properly, or fails to reply at all within an agreed upon time interval, the boot service declares the service to be out-of-order, and goes to the process service to start up a new version of it. Of course, the boot service itself must not crash, but it consists of a number of server processes that constantly check each other, and if need be, replace sick members with healthy ones.

6. RESOURCE MANAGEMENT

In keeping with our general philosophy of making the system kernel as small as possible, we have devised a way to put the resource control and accounting outside the kernel. Furthermore, a clear distinction is made between policy and mechanism, so that subsystem designers can implement their own policies with

the standard mechanisms.

Traditionally, accounting was used by the management of a computer center to levy charges for the use of the computer center's resources: CPU time, file space, lineprinter paper. This method worked quite well in the past, when hardware resources were expensive compared to the software used. Nowadays, hardware is cheap, software expensive. However, in the traditional approach there is usually no possibility to bill users for the use of a particular piece of software, or to have one user bill another for using his services.

Additionally, distributed systems need not be under control of one centralized management any more; private, personal computers can be plugged into the network and both use and offer services to the rest of the network. The accounting mechanisms in a distributed systems must be able to handle this new view on operating systems and allow any user that sets up a service to gather information about who uses his service.

6.1. Bank Service

The bank service is the heart of the resource management mechanism. It implements an object called a "bank account" with operations to transfer virtual money between accounts and to inspect the status of accounts. Bank accounts come in two varieties: individual and business. Most users of the system will just have one individual account containing all their virtual money. This money is used to pay for CPU time, disk blocks, typesetter pages, and all other resources for which the service owning the resource decides to levy a charge.

Business accounts are used by services to keep track of who has paid them and how much. Each business account has a subaccount for each registered client. When a client transfers money from his individual account to the service's business account, the money transferred is kept in the subaccount for that client, so the service can later ascertain each client's balance. As an example of how this mechanism works, a file service could charge for each disk block written, deducting some amount from the client's balance. When the balance reached zero, no more blocks could be written. Large advance payments and simple caching strategies can reduce the number of messages sent to a small number.

Another aspect of the bank service is its maintenance of multiple currencies. It can keep track of say, virtual dollars, virtual yen, virtual guilders and other virtual currencies, with or without the possibility of conversion among them. This feature makes it easy for subsystem designers to create new currencies and control how they are allocated among the subsystems users.

6.2. Accounting Policies

The bank service described above allows different subsystems to have different accounting policies. For example, a file or block service could decide to use either a buy-sell or a rental model for accounting. In the former, whenever a block was allocated to a client, the client's account with the service would be

debited by the cost of one block. When the block was freed, the account would be credited. This scheme provides a way to implement absolute limits (quotas) on resource use. In the latter model, the client is charged for rental of blocks at a rate of X units per kiloblock-second or block-month or something else. In this model, virtual money is constantly flowing from the clients to the servers, in which case clients need some form of income to keep them going. The policy about how income is generated and dispensed is determined by the owner of the currency in question, and is outside the scope of the bank server.

7. Summary

This paper has discussed a model for a fifth generation computer system architecture and its operating system. The operating system is based on the use of objects protected by sparse capabilities. An outline of some of the key services has been given, notably the block, file, directory, generic, process, boot and bank services.

References

[1] Dennis, J. B. and Horn, E. C. van, "Programming Semantics for Multiprogrammed Computations," *Comm. ACM*, vol. 9, no. 3, pp.143-155, March 1966.

[2] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The Notions of Consistency and Predicate Locks in a Database Operating System," *Comm. ACM*, vol. 19, no. 11, pp.624-633, November 1976.

[3] Evans, A., Kantrowitz, W., and Weiss, E., "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Comm. ACM*, vol. 17, no. 8, pp.437-442, August 1974.

[4] Kung, H. T. and Robinson, J. T., "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp.213-226, June 1981.

[5] Liskov, B. and Zilles, S., "Programming With Abstract Data Types," *SIGPLAN Notices*, vol. 9, pp.50-59, April 1974.

[6] Mullender, S.J. and Tanenbaum, A.S., "A Distributed File Server Based on Optimistic Concurrency Control", IR-80, Vrije Universiteit, Amsterdam, November 1982.

[7] Mullender, S.J. and Tanenbaum, A.S., "Protection and Resource Control in Distributed Operating Systems", IR-79 (to appear in Computer Networks), Vrije Universiteit, Amsterdam, August 1982.

[8] Mullender, S. J., Renesse, R. van, and Tanenbaum, A. S., "A Transaction-Oriented Transport Protocol", Report CS-R84XX, Centre for Mathematics and Computer Science (CWI), Amsterdam, November 1984.

[9] Papadimitriou, C. H., "Serializability of Concurrent Updates," *J. ACM*, vol. 26, no. 4, pp.631-653, October 1979.

[10] Purdy, G. B., "A High Security Log-in Procedure," *Comm. ACM*, vol. 17, no. 8, pp.442-445, August 1974.

[11] Stonebraker, M., "Operating System Support for Database Management," *Comm. ACM*, vol. 24, no. 7, pp.412-418, July 1981.

[12] Tanenbaum, A.S., "The ISO-OSI reference model", IR-71, Vrije Universiteit, Amsterdam, 1981.

[13] Tanenbaum, A. S. and Mullender, S. J., "Operating System Requirements for Distributed Data Base Systems," pp. 105-114 in Distributed Data Bases, ed. H. J. Schneider, North-Holland Publishing Co. (1982).

[14] Wilkes, M. V., *Time-Sharing Computer Systems*. New York:American Elsevier, 1968.

Trace Theory and the Design of Concurrent Computations

M. Rem
University of Technology, Eindhoven

ABSTRACT

After a brief introduction to trace theory, programs are discussed that consist of communicating subprograms. Trace structures are used to characterize the concurrent computations that may be evoked under control of these programs. In a number of examples the design of such programs from their formal specifications is discussed.

1. INTRODUCTION

In this article we discuss the design of concurrent computations. Concurrent computations may be looked upon as computations brought about by compositions of mutually communicating machines. It is well-known that finite-state machines can be characterized by regular expressions (MINSKY [4]). Thus, compositions of machines may be characterized by compositions of regular expressions. If these compositions are nonrecursive the ensuing machines are still finite-state machines. We want to discuss recursive compositions as well, thus leaving the realm of finite-state machines.

We extend the theory of regular expressions to make it more suitable for our purposes. This extended theory is called trace theory. Based on trace theory we introduce a program notation to express the programs under control of which the concurrent computations may take place. How exactly these computations may be evoked, i.e. how our notation may be implemented — though an interesting subject — falls outside the scope of this article. A VLSI implementation for a subset of the programs in our notation is presented in VAN DE SNEPSCHEUT [7].

While we know of a number of good design techniques for sequential computations (DIJKSTRA [2], GRIES [3]), the same is not true for concurrent

computations. Trace theory provides a mechanism that may be used to advantage when designing programs for concurrent computations. In order to illustrate this and to investigate the suitability of our approach for program synthesis, we discuss a number of example computations. In each example we try to derive, as systematically as we can, the program from the specification of the computation. Although programming is difficult and the derivation of programs from specifications may not be expected to become an automatism, the examples we present do identify some effective techniques for program synthesis.

2. TRACE THEORY

This section contains a brief and sometimes informal introduction to trace theory. For a more comprehensive treatment the reader is referred to REM, VAN DE SNEPSCHEUT & UDDING [5], UDDING [6], VAN DE SNEPSCHEUT [8].

An example of a very simple machine, or component as we like to call them, is the binary semaphore (DIJKSTRA [1]). A binary semaphore can be involved in two events only: P-operations and V-operations. These events must occur alternately and the first occurrence, if any, must be a V-operation. We formalize this description by saying that it has an alphabet of two symbols, p and v say, and that its behaviour is characterized by the regular set generated by the expression $(vp)^* + (vp)^*v$. The latter is the set of all finite-length alternations of v and p that do not start with a p . We shall characterize every component by such a combination of an alphabet of symbols and a set of finite-length sequences of symbols.

A *trace structure* T is a pair $\langle \underline{a}T , \underline{t}T \rangle$, in which $\underline{a}T$ is a finite set of symbols and $\underline{t}T \subseteq (\underline{a}T)^*$. A^* denotes, as usual, the set of all finite-length sequences of elements of A , including the empty sequence ε . Finite-length sequences of symbols are called *traces*. $\underline{a}T$ is called the *alphabet* of T and $\underline{t}T$ its *trace set*.

The trace structure captures all possible "behaviours" of a component. An initial segment of a possible behaviour is, of course, a possible behaviour as well. Therefore, our components will have prefix-closed trace structures. With the definition

$$PREF(T) = \langle \underline{a}T , \{t \mid (\exists u: u \in (\underline{a}T)^*: tu \in \underline{t}T)\} \rangle$$

a trace structure T is called *prefix-closed* if PREF(T) = T .

We wish to compose components into larger components. If trace structures characterize behaviours of components, how then can we compose them to characterize "joint behaviours"? Consider two trace structures T_0 and T_1 with $\underline{a}T_0 = \{x,y\}$ and $\underline{a}T_1 = \{x,z\}$. Let $\underline{t}T_0$ be the set generated by the regular expression $(xy)^*(\varepsilon + x)$ and $\underline{t}T_1$ the set generated by $(xz)^*(\varepsilon + x)$. Every trace in the composite trace set must be in accordance with both regular expressions, i.e. the symbols x and y in it must alternate and the symbols x and z in it must alternate. Phrased differently: for every trace in the composite its projection on $\{x,y\}$ must be in $\underline{t}T_0$ and its projection on $\{x,z\}$ in $\underline{t}T_1$. This way of composing is called weaving. The *weave* of two trace structures T_0 and T_1, denoted as $T_0 \underline{w} T_1$, is defined by

$$T_0 \underline{w} T_1 = \langle \underline{a}T_0 \cup \underline{a}T_1 , \{t \mid t\lceil(\underline{a}T_0) \in tT_0 \wedge t\lceil(\underline{a}T_1) \in \underline{t}T_1\} \rangle$$

In the above $t\lceil A$ denotes the *projection* of trace t on alphabet A. It is defined as follows.

$$\begin{aligned} \varepsilon\lceil A &= \varepsilon \\ (ta)\lceil A &= (t\lceil A)a \quad \text{if} \quad a \in A \\ (ta)\lceil A &= t\lceil A \quad \text{if} \quad a \notin A \end{aligned}$$

For a trace structure T, $T\lceil A$ denotes the trace structure $\langle \underline{a}T \cap A , \{t\lceil A \mid t \in \underline{t}T\} \rangle$. In the example above the trace set of $T_0 \underline{w} T_1$ is the set generated by the regular expression

$$(xyz + xzy)^*(\varepsilon + x + xy + xz)$$

The weaving operation expresses "joint behaviour". We have found this operation so useful that we have added it (in our program notation) to the operators that are traditionally used to form regular expressions. (The weave of two regular sets is again a regular set.) Weaving, however, is not the operation we have in mind to express composition of components. The symbols that are common to the different trace structures, as the x in our example, serve as a synchronization and communication means between the components. We wish to hide this "internal traffic" from the trace structure of the composite. For that reason we introduce a second composition operation, called blending, which is weaving followed by the elimination of common symbols. The *blend* of two trace structures T_0 and T_1, denoted as $T_0 \underline{b} T_1$, is defined by

$$T_0 \underline{b} T_1 = (T_0 \underline{w} T_1)\lceil (\underline{a}T_0 \div \underline{a}T_1)$$

where $\div$ stands for symmetric set difference, i.e. $A \div B = (A \cup B) \setminus (A \cap B)$. (Symmetric set difference is associative.) In the example given earlier the trace set of $T_0 \underline{b} T_1$ is the set generated by the regular expression

$$(yz + zy)^*(\varepsilon + y + z)$$

Weaving is associative, but blending is not. If $\underline{a}T_0 \cap \underline{a}T_1 \cap \underline{a}T_2 = \emptyset$ we have, however,

$$(T_0 \underline{b} T_1) \underline{b} T_2 = T_0 \underline{b} (T_1 \underline{b} T_2)$$

Whenever employing the blending operation, we will see to it that each symbol occurs in at most two alphabets of the constituting trace structures. Under this restriction blending is associative. Weaving and blending have $\langle \emptyset , \{\varepsilon\} \rangle$ as a unity.

We can introduce a partial order $\leq$ on the set of all trace structures: $T_0 \leq T_1$ means that $\underline{a}T_0 = \underline{a}T_1 \wedge \underline{t}T_0 \subseteq \underline{t}T_1$. Weaving and blending are monotonic, i.e. if $T_0 \leq T_1$ then, for any trace structure U, $T_0 \underline{w} U \leq T_1 \underline{w} U$ and $T_0 \underline{b} U \leq T_1 \underline{b} U$. As a consequence, recursive equations involving weaving and blending have a least fixpoint.

We discuss a few trace structures. The first one is $SEM_i(x,y)$, in which i is a natural number and x and y are two distinct symbols. It is defined by

$$\underline{a}SEM_i(x,y) = \{x,y\}$$
$$\underline{t}SEM_i(x,y) = \{t \in \{x,y\}^* \mid (\forall t_0,t_1 : t = t_0t_1 : 0 \leq t_0\underline{N}x - t_0\underline{N}y \leq i)\}$$

in which $t\underline{N}x$ stands for the number of occurrences of symbol x in trace t. The trace structures T_0 and T_1 in our earlier example were $SEM_1(x,y)$ and $SEM_1(x,z)$ respectively. Notice that SEM is ascending in its subscript, i.e. for all i, $i \geq 0$,

$$SEM_i(x,y) \leq SEM_{i+1}(x,y)$$

A generalization of SEM is $SYNC$. The trace structure $SYNC_{i,k}(x,y)$ is defined by

$$\underline{a}SYNC_{i,k}(x,y) = \{x,y\}$$
$$\underline{t}SYNC_{i,k}(x,y) = \{t \in \{x,y\}^* \mid (\forall t_0,t_1 : t = t_0t_1 : -k \leq t_0\underline{N}x - t_0\underline{N}y \leq i)\}$$

Notice that $SEM_i(x,y) = SYNC_{i,0}(x,y)$. $SYNC$ is ascending in both subscripts.

An example of a trace structure that is not regular, i.e. of a trace structure whose trace set is not a regular set, is DEL. $DEL(x,y)$ is the

union of the ascending sequence $(SEM_i(x,y))_{i=0}^{\infty}$. It is defined by

$$\underline{a}DEL(x,y) = \{x,y\}$$
$$\underline{t}DEL(x,y) = \{t \in \{x,y\}^* \mid (\forall t_0,t_1: t = t_0t_1: t_0 \underline{N} x \geq t_0 \underline{N} y)\}$$

We formulate a number of properties for these trace structures. For $i+j \geq 1 \wedge k+l \geq 1$

(2.1) $$SYNC_{i,j}(x,y) \ \underline{b} \ SYNC_{k,l}(y,z) = SYNC_{i+k,j+l}(x,z)$$

and, hence, for $i \geq 1 \wedge k \geq 1$

(2.2) $$SEM_i(x,y) \ \underline{b} \ SEM_k(y,z) = SEM_{i+k}(x,z)$$

Furthermore,

(2.3) $$DEL(x,z) = DEL(x,y) \ \underline{b} \ SEM_1(y,z)$$
$$= SEM_1(x,y) \ \underline{b} \ DEL(y,z)$$

3. A PROGRAM NOTATION BASED ON TRACE THEORY

In this section we introduce the program notation we use for the representation of components. The components we discuss in this article are fully characterized by their trace structures. Thus far we have introduced trace structures by giving regular expressions. For regular trace structures we can indeed do so, but our notation for regular expressions differs slightly from the standard way. Rather than just juxtaposing terms we use the semicolon as the concatenation operator. We, furthermore, use the vertical bar instead of the plus for the union. As an additional operator we have the comma, which denotes weaving. We call such expressions *commands*. Examples of commands are

$(x ; y)^*$
$(x , y)^*$
$(x \mid y)^*$
$(x ; y)^* , (x ; z)^*$

The same commands written as standard regular expressions would be

$(xy)^*$
$(xy + yx)^*$
$(x + y)^*$
$(xyz + xzy)^*$

Of the three dyadic operators (comma, semicolon, bar) the comma has the

highest priority and the bar the lowest.

If S is a command, TR(S) denotes its associated trace structure. The alphabet of TR(S) consists of all symbols occurring in S . Just a command by itself is not a complete program. The simplest form a program can have is

com C(A): S moc

In the above S is a command and A a list of symbols such that $\underline{a}TR(S) = A$. The text represents a component called C whose trace structure TR(C) is, by definition, PREF(TR(S)) . Examples of such components are

com $sem_1(x,y)$: $(x\ ;\ y)^*$ moc

com $sync_{1,1}(x,y)$: $(x\ ,\ y)^*$ moc

com $sem_2(x,y)$: $x\ ;\ (x\ ,\ y)^*$ moc

The reader is encouraged to check that the programs above indeed have as their trace structures $SEM_1(x,y)$, $SYNC_{1,1}(x,y)$, and $SEM_2(x,y)$ respectively.

In general a component will be composed of subcomponents. Such a component is represented by a program text of the following form.

com C(A): sub s_0: C_0, ... , s_{n-1}: C_{n-1}
$a_0 = b_0$, ... , $a_{m-1} = b_{m-1}$
S

moc

Component C has n subcomponents named s_0 , ... , s_{n-1} . Subcomponent s_i is said to be of type C_i . C_i is a component. The next line of the program text contains the equalities. We will come to them shortly. S is again a command. The trace structure associated with C is, by definition, the blend of $n+1$ trace structures:

$$TR(C) = PREF(TR(S)) \ \underline{b}\ s_0.TR(C_0)\ \underline{b}\ \ldots\ \underline{b}\ s_{n-1}.TR(C_{n-1}) \tag{3.1}$$

If T is a trace structure then s.T denotes the trace structure T in which each symbol $a \in \underline{a}T$ is (both in $\underline{a}T$ and in $\underline{t}T$) replaced by s.a (read "s its a"). Thus the n trace structures $s_i.TR(C_i)$ have disjoint alphabets. Let B denote the union of the $n+1$ (disjoint) alphabets involved:

$$B = A \cup \underline{a}(s_0.TR(C_0)) \cup \ldots \cup \underline{a}(s_{n-1}.TR(C_{n-1}))$$

We require that

- $\underline{a}TR(S) \subseteq B$;
- in each equality $a_j = b_j$ the symbols a_j and b_j belong to two different alphabets of those constituting B ;
- each symbol in B occurs either in exactly one equality or in $\underline{a}TR(S)$.

An equality $a_j = b_j$ expresses that the two symbols a_j and b_j should be treated as the same symbol. Thus the equalities make the alphabets constituting B nondisjoint, thereby increasing the amount of synchronization between the $n+1$ trace structures involved. Due to the restrictions above we have $\underline{a}TR(C) = A$. It is, furthermore, allowed to omit the command S in the program text. In that case $TR(S) = \langle \emptyset , \{\varepsilon\} \rangle$ is understood.

Let component sem_1 be defined as earlier. Consider the following example of a program.

```
com sem4(x,y): sub s0 , s1: sem1
                  s0.y = s1.x
                  (x ; s0.x)* , (s1.y ; y)*
moc
```

("s0 , s1: sem_1" is short for "s0: sem_1 , s1: sem_1" .) We show how trace xxxx may be obtained from traces of the trace structures of the subcomponents and of the command:

```
S:   x  s0.x  x        s0.x  x  s1.y        s0.x  x
s0:     s0.x     s0.y  s0.x        s0.y  s0.x
s1:              s1.x           s1.y  s1.x
```

The top line is a trace of $PREF(TR(S))$, S denoting the command of sem_4 . The next two lines are traces of $s0.TR(sem_1)$ and $s1.TR(sem_1)$ respectively. Equal symbols that are cancelled by blending have been placed vertically under each other. The trace that remains is xxxx . Notice that the only possible next symbol is y .

We now discuss the example more formally. According to VAN DE SNEPSCHEUT [8] PREF distributes through $\underline{w}$ for trace structures with disjoint alphabets: if $\underline{a}T_0 \cap \underline{a}T_1 = \emptyset$

$$PREF(T_0 \ \underline{w} \ T_1) = PREF(T_0) \ \underline{w} \ PREF(T_1)$$
$$= PREF(T_0) \ \underline{b} \ PREF(T_1)$$

Therefore,

$$PREF(TR(S)) = SEM_1(x , s0.x) \ \underline{b} \ SEM_1(s1.y , y)$$

Hence, by (3.1),

$$\begin{aligned} TR(sem_4) &= SEM_1(x\,,\,s0.x)\ \underline{b}\ SEM_1(s1.y\,,\,y) \\ &\quad \underline{b}\ s0.TR(sem_1)\ \underline{b}\ s1.TR(sem_1) \\ &= SEM_1(x\,,\,s0.x)\ \underline{b}\ SEM_1(s1.y\,,\,y) \\ &\quad \underline{b}\ SEM_1(s0.x\,,\,s0.y)\ \underline{b}\ SEM_1(s1.x\,,\,s1.y) \end{aligned}$$

Using $s0.y = s1.x$ and (2.2) we find

$$TR(sem_4) = SEM_4(x,y)$$

Components having $SEM_i(x,y)$, for $i \geq 1$, as their trace structures may be represented in the following way. Let sem_1 be defined as earlier, and let for $i \geq 1$ sem_{i+1} be given as follows.

com $sem_{i+1}(x,y)$: sub s: sem_i
((x|s.y) ; (y|s.x))*

moc

Without proof we mention that

$$PREF(TR(S))\ \underline{b}\ SEM_i(s.x\,,\,s.y) = SEM_{i+1}(x,y)$$

in which S denotes the command of sem_{i+1}. Equation (3.1) reads for component sem_{i+1}

$$TR(sem_{i+1}) = PREF(TR(S))\ \underline{b}\ s.TR(sem_i)$$

Hence, by induction we obtain that $TR(sem_{i+1}) = SEM_{i+1}(x,y)$.

An interesting generalization of this example is the following one.

com del(x,y): sub s: del
((x|s.y) ; (y|s.x))*

moc

With S denoting the command of component del equation (3.1) now reads

$$TR(del) = PREF(TR(S))\ \underline{b}\ s.TR(del)$$

This is a recursive equation in TR(del). Since blending is monotonic, (3.1) has a least solution. If all commands involved have nonempty trace sets (3.1) also has a least nonempty solution. By definition, the latter is the trace structure of the component. According to UDDING [6] this yields in our example $TR(del) = DEL(x,y)$. Since blending is continuous from below, this solution may be obtained as the limit of the ascending sequence $(T_i)_{i=0}^{\infty}$ defined by

$$\begin{aligned} T_0 &= \langle \{x,y\}\ ,\ \{\varepsilon\} \rangle \\ T_{i+1} &= PREF(TR(S))\ \underline{b}\ s.TR(T_i) \end{aligned}$$

Then $T_i = SEM_i(x,y)$. This example demonstrates how recursion allows the representation of nonregular trace structures.

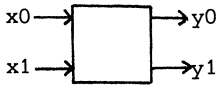
The final example of this section may be appreciated as an i-place one-bit buffer. For $i = 1$ it is the following component.

$$\underline{com}\ bqueue_1(x0,x1,y0,y1):\ (x0\ ;\ y0\ |\ x1\ ;\ y1)^*\ \underline{moc}$$

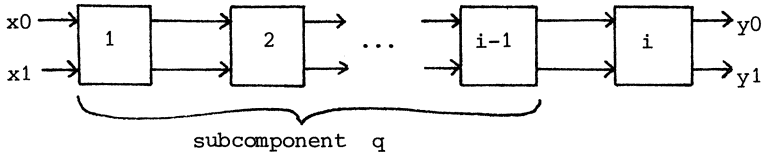
When reading "input of value 0 " , "input of value 1 " , "output of value 0 " , and "output of value 1 " for the symbols x0 , x1 , y0 , and y1 respectively, it becomes clear why we may refer to $bqueue_1$ as a one-place one-bit buffer. For $i > 1$ we propose

$$\begin{aligned}&\underline{com}\ bqueue_i(x0,x1,y0,y1):\ \underline{sub}\ q:\ bqueue_{i-1}\\ &\qquad\qquad\qquad\qquad x0 = q.x0\ ,\ x1 = q.x1\\ &\qquad\qquad\qquad\qquad (q.y0\ ;\ y0\ |\ q.y1\ ;\ y1)^*\\ &\underline{moc}\end{aligned}$$

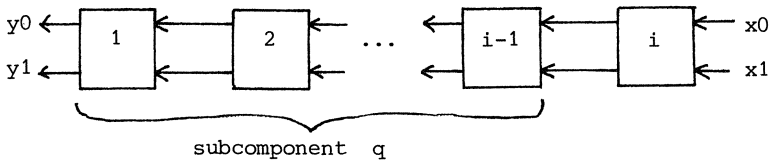
It consists of an (i-1)-place buffer q and a 1-place buffer between the outputs (q.y0 , q.y1) of q and the outputs (y0 , y1) of the component. Drawing a 1-place buffer between inputs (x0,x1) and outputs (y0,y1) as



we may depict component $bqueue_i$ as



A number j ($1 \le j \le i$) in a box indicates that this 1-place buffer is brought about by the command of $bqueue_j$. If we replace the equalities by "q.y0 = y0 , q.y1 = y1" and the command by $(x0\ ;\ q.x0\ |\ x1\ ;\ q.x1)^*$ we obtain the following drawing.



4. FROM SPECIFICATIONS TO PROGRAMS

In the preceding sections we have introduced a program notation for concurrent computations and a formalism for expressing the effects of these computations. We now turn to the question how one could invent such programs. Given a characterization of the computation intended, how can one synthesize a program under the control of which the computation intended may be evoked? Such a characterization is usually referred to as a functional specification or just a specification. Phrased in terms of trace theory, our question becomes: "Given a specification of a trace structure, how can we find a program that expresses a component with that trace structure?". This leads us first to the question what kind of specifications we have in mind. In Section 2 we have encountered our first specifications, viz. those of the trace structures SEM , SYNC , and DEL . In these specifications we used the "counting operation" $t\underline{N}x$. This has turned out to be a way of specifying that, if applicable, is well-suited to program derivation. As an example we discuss the construction of a bag of binary values.

4.1. A BAG OF BINARY VALUES

A bag of binary values is a component in which we can store an arbitrary number of binary values. Each value stored can also be deleted again. Storage of a value 0 or 1 is denoted by the symbol x0 or x1 respectively. Deletion of a value 0 or 1 is denoted by y0 or y1 respectively. The four symbols x0 , x1 , y0 , y1 constitute the alphabet of the component bag . Of course, for each binary value the number of values deleted should not exceed the number of values stored. This gives rise to the following specification.

$$(4.1) \qquad t: \quad t\underline{N}x0 \geq t\underline{N}y0 \;\wedge\; t\underline{N}x1 \geq t\underline{N}y1$$

By the above specification we mean that the trace structure of bag is the greatest prefix-closed trace structure for which all traces t in its trace set satisfy (4.1).

According to (4.1) a bag is just a combination of DEL(x0,y0) and DEL(x1,y1) :

```
com bag(x0,x1,y0,y1):
    sub d0 , d1: del
    x0 = d0.x , y0 = d0.y
    x1 = d1.x , y1 = d1.y
moc
```

By (3.1)

$$TR(bag) = \langle \emptyset \ , \ \{\varepsilon\} \rangle \ \underline{b} \ DEL(d0.x \ , d0.y) \ \underline{b} \ DEL(d1.x \ , d1.y)$$
$$= DEL(x0,y0) \ \underline{b} \ DEL(x1,y1)$$
$$= DEL(x0,y0) \ \underline{w} \ DEL(x1,y1)$$

According to (2.3)

$$DEL(x0,y0) = SEM_1(x0 \ , d0.x) \ \underline{b} \ DEL(d0.x \ , d0.y) \ \underline{b} \ SEM_1(d0.y \ , y0)$$

We may, consequently, replace the equalities by a command expressing the weave (and hence, by the disjointness of their alphabets, the blend) of the appropriate SEM_1 's:

```
com bag(x0,x1,y0,y1):
    sub d0 , d1: del
    (x0 ; d0.x)* , (d0.y ; y0)* ,
    (x1 ; d1.x)* , (d1.y ; y1)*
moc
```

Such a replacement of equalities by repetitions in the command is a technique we use when we want to accommodate additional constraints in the specification. We demonstrate this in the following example, in which two of the four equalities are replaced by repetitions.

4.2. A SORTER OF BINARY VALUES

A sorter is a bag with the additional constraint that in deletions only the least value contained in the bag may be deleted. Such a component is sometimes referred to as a priority queue. This gives rise to the following specification for a binary sorter.

$$\begin{aligned} t: \quad & tNx0 \geq tNy0 \\ & \wedge \ tNx1 \geq tNy1 \\ & \wedge \ (\forall s : t = s\,y1 : \ sNx0 = sNy0) \end{aligned} \tag{4.2}$$

Since the first two conjuncts express that a sorter is a bag, we start with a program for a bag and then manipulate it so as to have it satisfy the third

conjunct of the specification as well.

A program for a bag was presented in Section 4.1. According to the third conjunct of (4.2) we need to restrict the selection of y1 . In order to determine that the trace s thus far selected satisfies $sNx0 = sNy0$, we replace component d0 by a component "dal" , which is a del that signals, by a symbol e , when this equality holds. The alphabet of component dal is {x,y,e} and its specification is

$$t: \quad tNx \geq tNy$$
$$\wedge\ (\forall\, s : t = s\,e : \quad sNx = sNy)$$

Of course, we obtain the program for dal from that for del , which reads

```
com del(x,y): sub s: del
                  ((x | s.y) ; (y | s.x))*
moc
```

An invariant of the repetition is that the trace t thus far selected satisfies

$$tNx + tNs.y = tNy + tNs.x$$

i.e.

$$tNx - tNy = tNs.x - tNs.y$$

and, hence,

$$(4.3) \qquad (tNx = tNy) \equiv (tNs.x = tNs.y)$$

At the end of each step of the repetition the equality of the number of x 's and y 's for this component may thus be concluded from the same equality for the subcomponent. We shall add $(s.e\ ;\ e^*)^*$ at the end of the repeated expression. At the semicolon we have

$$tNx - tNy = 1 + tNs.x - tNs.y \geq 1$$

and symbol e must not be selected. Of course, e may be selected prior to the repetition. We thus arrive at the following program for component dal .

```
com dal(x,y,e): sub s: dal
                    e* ; ((x | s.y) ; (y | s.x) ; (s.e ; e*)*)*
moc
```

We change in the component bag the type of subcomponent d0 from del into dal , by which it remains a bag. But now we can see to the observance of the third conjunct of (4.2). In order to guarantee that the

trace s thus far selected satisfies $s\underline{N}x0 = s\underline{N}y0$ the selection of y1 is preceded by the selection of d0.e , yielding $(d0.e\ ;\ d1.y\ ;\ y1)^*$. We must prevent that between d0.e and y1 the equality

$$t\underline{N}x0 = t\underline{N}y0 = t\underline{N}d0.x = t\underline{N}d0.y$$

is destroyed. This could happen only if x0 is selected. Therefore, we make the selections of "x0 ; d0.x" and "d0.e ; d1.y ; y1" mutually exclusive:

```
com sorter(x0,x1,y0,y1):
    sub d0: dal, d1: del
    y0 = d0.y , x1 = d1.x
    (x0 ; d0.x | d0.e ; d1.y ; y1)*
moc
```

4.3. A COUNTER

The counter we consider records an integer value (negative values included). Its initial value is 0 and it can be incremented by 1 and decremented by 1 . The latter two operations are denoted by the symbols u and d respectively. Its value cannot be inspected, but we can inquire whether it is 0 . The alphabet of the component is {u,d,z} and its specification reads

$$t:\ (\forall\, s:\ t = s\,z:\ \ s\underline{N}u = s\underline{N}d)$$

We again employ the technique of translating a count for the component into one for the subcomponent c . We do so by having as our command a repetition that maintains as an invariant for the trace t thus far selected

$$t\underline{N}u - t\underline{N}d = t\underline{N}c.u - t\underline{N}c.d$$

while selecting *any* sequence of u 's and d 's:

$$((u \mid c.d)\ ,\ (d \mid c.u))^*$$

The selection of z should then be preceded by the selection of c.z and it should exclude "(u | c.d) , (d | c.u)":

```
com counter(u,d,z):
    sub c: counter
    z* ; ((u | c.d) , (d | c.u) | c.z ; z*)*
moc
```

4.4. A BOUNDED BAG

We now design a bag that can store at most k binary values. Its specification reads

$$\begin{aligned} t: \quad & \underline{tN}x0 \geq \underline{tN}y0 \\ & \wedge\ \underline{tN}x1 \geq \underline{tN}y1 \\ & \wedge\ (\underline{tN}x0 + \underline{tN}x1) - (\underline{tN}y0 + \underline{tN}y1) \leq k \end{aligned}$$

This is equivalent to

$$\begin{aligned} t: \quad & 0 \leq \underline{tN}x0 - \underline{tN}y0 \leq k \\ & \wedge\ 0 \leq \underline{tN}x1 - \underline{tN}y1 \leq k \\ & \wedge\ 0 \leq (\underline{tN}x0 + \underline{tN}x1) - (\underline{tN}y0 + \underline{tN}y1) \leq k \end{aligned}$$

Each conjunct expresses a SEM_k. Hence, the behaviour of the component is the joint behaviour of three SEM_k's. We express joint behaviour by weaving. So in a sense we are translating each " $\wedge$ " in the specification into a " , " in the program text.

For $k = 1$ we find

```
com bbag1(x0,x1,y0,y1):   (x0 ; y0)*
                        , (x1 ; y1)*
                        , ((x0|x1) ; (y0|y1))*
moc
```

This may, by algebraic manipulation, be simplified to

```
com bbag1(x0,x1,y0,y1):  (x0 ; y0 | x1 ; y1)*  moc
```

which equals $bqueue_1$.

For $k > 1$ we use three subcomponents of type sem_k:

```
com bbagk(x0,x1,y0,y1):
    sub b0,b1,b01: semk
      (b0.x ; x0 | b0.y ; y0)*
    , (b1.x ; x1 | b1.y ; y1)*
    , (b01.x ; (x0|x1) | b01.y ; (y0|y1))*
moc
```

After a selection of $x0$ the top line of the command guarantees that the trace t thus far selected satisfies

$$\underline{tN}x0 = \underline{tN}b0.x \ \wedge\ \underline{tN}y0 = \underline{tN}b0.y$$

Since $b0$ is of type sem_k, we have

$$0 \leq tNb0.x - tNb0.y \leq k$$

Hence, the first conjunct of the specification is satisfied. A similar reasoning applies to the selection of other symbols in the alphabet of $bbag_k$ and to the other conjuncts of the specification.

Also this component may be changed into a bounded sorter. This can be done in a way similar to the unbounded case.

4.5. A STACK AND A QUEUE

Thus far we were able to express the specifications by just using the counting operator N. That led to more or less straightforward derivations of program texts. In this section we discuss two examples of more complicated specifications.

The first one is an unbounded stack of binary values. Consider the language generated by the following production rule.

$$S ::= (x0\ S\ y0 \mid x1\ S\ y1)^*$$

The trace set of the stack is the set of all prefixes of sentences in this language. Any such trace t satisfies

$$tNx0 \geq tNy0$$
$$\wedge\ tNx1 \geq tNy1$$

A stack is, therefore, some combination of DEL(x0,y0) and DEL(x1,y1). Inspired by the program for component del we propose the following program text.

```
com stack(x0,x1,y0,y1):
    sub s: stack
    ( (x0 | s.y0) ; (y0 | s.x0)
    | (x1 | s.y1) ; (y1 | s.x1)
    )*
moc
```

An invariant of the repetition is that every trace t thus far selected satisfies

$$tNx0 - tNy0 = tNs.x0 - tNs.y0 \qquad (4.4)$$
$$\wedge\ tNx1 - tNy1 = tNs.x1 - tNs.y1$$

This may, somewhat loosely, be formulated as: subcomponent s contains all binary values stored in the component. At a semicolon, however, the appro-

priate left-hand side is 1 greater than its right-hand side: there is one more value in the component than in the subcomponent. This value is consequently either output or, if a next input follows, transferred to the subcomponent.

We now turn to our next and last example: an unbounded queue of binary values. It has as its specification:

$$
\begin{aligned}
t:\quad & t\underline{N}x0 \geq t\underline{N}y0 \\
& \wedge\ t\underline{N}x1 \geq t\underline{N}y1 \\
& \wedge\ (t\lceil\{y0,y1\})^{y0,y1}_{x0,x1} \text{ is a prefix of } t\lceil\{x0,x1\}
\end{aligned}
\tag{4.5}
$$

in which $(t)^{y0,y1}_{x0,x1}$ denotes trace t with the symbols y0 and y1 replaced by x0 and x1 respectively. The third conjunct of (4.5) expresses that the sequence of values output is a prefix of the sequence of values input. The first two conjuncts show that, just like the stack, the queue is a combination of two DEL 's . The difficulty is that the value to be output is not simply the last one received.

We again maintain (4.4) as an invariant. During one step of the repetition we store the next value to be output and produce that output. This value is either the last one received (this case occurs only if the component was empty upon reception of that value) or the value longest residing in the subcomponent. We, consequently, employ again a subcomponent of type dal so that, by interrogating whether the component is empty, we can distinguish between these two cases.

```
com queue(x0,x1,y0,y1):
    sub s: queue, d: dal
    (d.x ; (x0|x1) | (y0|y1) ; d.y)* ,
    ( (d.e ; x0 | s.y0) ; (x0 ; s.x0 | x1 ; s.x1)* ; y0
    | (d.e ; x1 | s.y1) ; (x0 ; s.x0 | x1 ; s.x1)* ; y1
    )*
moc
```

Because of the structure of the first part of the command we have for any trace t thus far selected

$$
\begin{aligned}
& t\underline{N}d.x \geq t\underline{N}x0 + t\underline{N}x1 \\
\wedge\ & t\underline{N}y0 + t\underline{N}y1 \geq t\underline{N}d.y
\end{aligned}
\tag{4.6}
$$

Since subcomponent s is a queue , it satisfies (4.5). Hence,

$$\begin{aligned} & \underline{tN}s.x0 - \underline{tN}s.y0 \geq 0 \\ \wedge\ & \underline{tN}s.x1 - \underline{tN}s.y1 \geq 0 \end{aligned}$$

This combined with (4.4) yields

$$\underline{tN}x0 + \underline{tN}x1 \geq \underline{tN}y0 + \underline{tN}y1$$

By the above and (4.6) we obtain

$$(\underline{tN}d.x = \underline{tN}d.y) \Rightarrow (\underline{tN}x0 + \underline{tN}x1 = \underline{tN}y0 + \underline{tN}y1)$$

Therefore, when $d.e$ is selected the component is indeed empty.

The second part of the command consists of two alternatives: storage and production of value 0 or of value 1. Between storage and production arbitrary many inputs may occur; these are, by means of

$$(x0\ ;\ s.x0 \mid x1\ ;\ s.x1)^*$$

transferred to the suncomponent.

The stack and the queue are two examples of problems with specifications that do not just consist of applications of the counting operator. As a result, the techniques for program synthesis employed earlier were less applicable in these examples. By — sometimes formally, sometimes informally — reasoning about the programs proposed we may have become convinced of their correctness. But in these two problems we did not succeed in systematically deriving the program texts from their specifications.

5. CONCLUSIONS

We have addressed two issues: how concurrent computations can be based on trace theory, and the amenability of such an approach to program synthesis. In a number of examples we were quite successful at deriving the programs from their specifications. The reason for this is probably the suitability of trace structures to algebraic manipulation.

We recall some of the techniques for program synthesis we have encountered. If a specification consists of a number of conjuncts we may try to meet these conjuncts in succession. Thus, if we design a sorter we may start with a bag. Sometimes the program for the conjunctive specification may be obtained by weaving the program parts that correspond to the conjuncts. The bounded bag is a nice example of this technique. Sometimes the accommodation of additional conjuncts requires the replacement of equalities by repetitions in the command. The conversion of a bag into a sorter

contains an example of this. With specifications involving the counting operator it is often a good technique to introduce a repetition that maintains as an invariant the same count for the subcomponent. We have encountered a number of examples of this. Two problems, the stack and the queue, have been discussed in which we were less successful in deriving systematically the programs from the specifications.

6. ACKNOWLEDGEMENTS

Acknowledgements are due to Jan L.A. van de Snepscheut, Jan Tijmen Udding, and the other members of the Eindhoven VLSI Club.

7. REFERENCES

[1] DIJKSTRA, EDSGER W. *Cooperating Sequential Processes*, In: Programming Languages (F. Genuys,ed.), Academic Press, 1968, 43 - 112.

[2] DIJKSTRA, EDSGER W. *A Discipline of Programming*, Prentice-Hall, 1976.

[3] GRIES, DAVID. *The Science of Programming*, Springer-Verlag, 1981.

[4] MINSKY, MARVIN L. *Computation: Finite and Infinite Machines*, Prentice-Hall, 1972.

[5] REM, MARTIN, JAN L.A. VAN DE SNEPSCHEUT & JAN TIJMEN UDDING. *Trace Theory and the Definition of Hierarchical Components*, In: Third Caltech Conference on VLSI (Randal Bryant, ed.), Computer Science Press, 1983, 225 - 239.

[6] UDDING, JAN TIJMEN. *On Recursively Defined Sets of Traces*, Memorandum JTU0a, Dept. of Math. & Computing Science, Eindhoven Univ. of Technology, 1983.

[7] VAN DE SNEPSCHEUT, JAN L.A. *Deriving Circuits from Programs*, In: Third Caltech Conference on VLSI (Randal Bryant, ed.), Computer Science Press, 1983, 241 - 256.

[8] VAN DE SNEPSCHEUT, JAN L.A. *Trace Theory and VLSI Design*, Doctoral Dissertation, Eindhoven University of Technology, 1983.

The Second Machine Class: Models of Parallelism

P. van Emde Boas
University of Amsterdam

1. INTRODUCTION

Computation theory knows a large variety of models of computing devices, or formal calculi for effective computation. This divergence has not led to a large variety of computability theories, due to the basic observation that within each formalism, each computation can be simulated some way or another by any of the other device types.

A similar situation holds for the foundations of complexity theory, which has established itself as one of the prime pilars on which the building of computer science as we know it today is founded. There exist various models of computing devices like Turing machines (in various variants), random access machines (RAM's) with or without the possibility of modifying the program, reference machines, and several less known ones. Each of these models can be equipped with a reasonable measure for computation time and storage use, such that the following *Invariance Thesis* [36] can be made to be holding true:

> For each machine M_i of one type having running time T_i and storage use S_i one can find in any other type of machinery a simulating device M'_j which simulates M_i with polynomially bounded overhead in time and constant factor overhead in storage.

The bounds on the overhead are more formally expressed by: if T'_j and S'_j are the run time and storage use of device M'_j in the other formalism, then for some suitable chosen constants c and c' one has $T'_j(n) \leqslant c.(T_i(n))^c$

resp. $s'_j(n) \leqslant c'.S_i(n)$. The symbol n is used in this context to describe the length of the input of the computation. (Clearly c and c' are independent of n).

The important consequence of this fortunate state of the world is that, although the precise time and space bounds for solving concrete problems are highly dependent of the machine model used, a few fundamental complexity classes, which happen to be closed under the overhead factors mentioned above, are machine independent. These classes form the well known hierarchy:

LOGSPACE $\subset$ NLOGSPACE $\subset$ P $\subset$ NP $\subset$ PSPACE = NPSPACE $\subset$ EXPTIME $\subset$ NEXPTIME $\subset$

where each inclusion hides the open problem of whether the inclusion is proper or not.

I assume that the reader is familiar with the machine models and complexity classes mentioned above; see references [13,18,22,34,35].

The family of machine models which can be used when defining the above classes together represent a "reasonable" model for the concept of sequential computation. It was to be expected that a similar family of models would be invented for parallel computation. As it turns out this second machine class has even a wider variety of device types than the first one, where the parallelism can be made either fully explicit like in the case of the SIMDAG model of Goldschlager [15], it can be hidden in some tree of possible computations as in the alternation model [4], or it can be made completely invisible in the form of a sequential computation on unreasonably long objects, as done in the multiplication RAM's [17].

Rather than by proving bounds on the relative simulation efficiencies by which these models simulate each other, the members of the second machine class are joined together in a family by the so-called *parallel computation thesis*. This thesis expresses the fact that for these machines a sequence of fundamental complexity classes can be defined as well. We indicate this sequence by prefixing the classes with the indicant // for parallelism: ★

//LOGSPACE $\subset$ //NLOGSPACE $\subset$ //PTIME $\subset$ //NPTIME $\subset$ //PSPACE $\subset$ //NPSPACE $\subset$

The sequence is moreover nothing but a shifted version of the sequence for the standard class of sequential devices. In fact one has with some minor exceptions

★ In the sequel // will be replaced by some indicant for the particular model being considered.

//LOGSPACE = P, //NLOGSPACE = NP, //PTIME = PSPACE, //NPTIME = NPSPACE, etc. As a consequence, by Savitch' theorem that PSPACE = NPSPACE [28] one obtains //PTIME = //NPTIME .

In this survey I will concentrate on the equality //PTIME = PSPACE . I will present several of the members of the second machine class in some detail, and try to indicate the outlines of a proof of the above equality for the case of these models. For the other relations and models I have to refer to the extensive literature on this subject, since it seems quite feasible to give a one semester course on the details of this area of complexity theory.

It is reasonable to ask wether the parallel computation thesis, as formulated above, is correct or not [2] . If believed as an assertion on all possible models of parallel computers the thesis is probably false (unless PSPACE = EXPTIME). Instead I use the parallel computation thesis as a characteristic feature deliminating the second machine class. This is similar to the way the standard machine class is deliminated by requiring the invariance thesis to be valid. Machine models for which the parallel computation thesis might be false then can be gathered into an even more powerful third machine class.

2. VARIATIONS ON A THEME COMPOSED BY SAVITCH

Before investigating the unknown realm of the second machine class, we should be fully informed on the corresponding part of the picture in the world of sequential computation : the class PSPACE . Which problems are likely to belong to PSPACE, what is the background of Savitch' theorem that PSPACE = NPSPACE , and what are the standard complete problems for PSPACE?

By definition a problem (which as a mathematical object is nothing but a set of strings or a language) belongs to the class PSPACE provided it can be recognized by some device (e.g., a multitape Turing machine) which has the property that input x is recognized using no more space than $k.|x|^k$ for some constant k independent of x ; here $|x|$ denotes the length of input x .

An elementary counting argument, on the number of possible Turing machine configurations with a bounded number of tape cells being used, learns that

the computation accepting x cannot be longer than some exponential quantity like $2^{c.|x|^k}$. We can consider a huge graph, whose nodes are the configurations of this Turing machine on input x which satisfy the space bound, and whose edges are the individual computation steps between a configuration and its successor in the computation (if defined). If the Turing machine is deterministic then this graph has nodes with outdegree at most one; it becomes the union of a number of components, each component being a single sink, or a cycle, with in-trees converging into the sink or cycle (disregarding computation steps where the space bound gets violated).

It is not difficult to modify the formalism in such a way that the accepting configuration of a computation becomes unique. Membership of the input x in the language can now be translated into the question, whether the initial configuration belongs to the component of the unique accepting sink.

A similar translation is possible for nondeterministic computations as well. In this case the nodes in the graph have outdegree which may be larger than one, but without loss of generality one may assume that the maximal outdegree is two (binary choices only). Membership in the language now amounts to the question whether there exists a path in the graph from the initial node to the final node. Since such a path may be assumed to be cycle free (otherwise there exists a shorter path) the length of such a path is bounded by the number of nodes in the graph.

The existence of such a path can be determined by a transitive closure computation. Under normal circumstances this would be an infeasible job, due to the exponential size of the graph involved. Still it is exactly this transitive closure algorithm which is the algorithm performed by various machines in the second machine class, when recognizing a member of PSPACE or NPSPACE in polynomial time.

For better understanding we must be aware of the following facts relevant to this graph:

Although the graph itself is exponential its nodes, i.e. the configurations of the Turing machine involved, are polynomially bounded; they can therefore be written down in polynomial time. We may even assume that they are encoded by simple binary numbers of polynomially bounded length. The coding used is such that it is easy to decide whether some node is a successor of another node. So the edges of the graph are recognizable in polynomial time as well.

Next we consider the following transitive closure algorithm:

Input: A matrix M of size 2^k by 2^k such that M[i,j] = 1 iff i = j or if there exists an edge form node i to node j , and 0 otherwise

Output: The transitive closure M of the input matrix, stored in the same array M .

Algorithm:

```
for p to k do
  for all i,j in parallel do
      M[i,j] := ∃k[ M[i,k] = 1 and M[k,j] = 1 ]
                                od
            od ;
```

If we have $2^{3.k}$ processors available then the inner loop and the evaluation of the existentially quantified expression can be performed in time O(1). So the entire computation requires time O(k) . Now in our application k stand for the length of the binary numbers used in the encoding of the graph representing the given Turing machine computation; this length is, as we have seen, polynomially bounded by the length of the input. So our transitive closure algorithm uses polynomial time.

Other algorithmic implementations of the transitive closure algorithm can be used for obtaining other interesting theoretical results. For example, one may consider the following recursive version:

```
Proc path = (int p, node i,j)bool :
if p = 0 then i = j or i succ j # there is an edge from i to j #
         else
           bool found:= false ;
           for node n from 0 to 2^k-1 while not found do
             found := found or ( path(p-1,i,n) and path(p-1,n,j) )
           od ;
           found
fi ;
```

Existence of a path between the initial configuration init and the accepting configuration accept now is computed by evaluating path(k,init,accept) . The recursion depth equals k where 2^k is the size of the graph; this value of k therefore is proportional to the space used by the (nondeterministic) Turing machine used for accepting our set in

PSPACE . Note that the same k also measures the space needed for writing down the arguments of the recursive procedure. So this recursive procedure, which is a deterministic computation, requires space for k stack frames each of size $O(k)$, so its total space requirements are $O(k^2)$. This construction is the main ingredient of the well known theorem of Savitch:

THEOREM (Savitch, 1970 [28]) If the language L is recognized by some nondeterministic Turing machine in space $S(n) \geq \log(n)$ then it can be recognised by some deterministic Turingmachine in space $S^2(n)$. As a consequence PSPACE = NPSPACE .

By another simple transformation of the above transitive closure algorithm we can establish the PSPACE completeness of the problem QUANTIFIED BOOLEAN FORMULAS (QBF) [24] .

QUANTIFIED BOOLEAN FORMULAS:
INSTANCE: A formula of the form $Q_1x_1Q_2x_2\ldots Q_nx_n[\ P(x_1,\ldots,x_n)\ ]$, where each Q_i equals $\exists$ or $\forall$, and where $P(x_1,\ldots,x_n)$ is a formula in the propositional calculus in the boolean variables $x_1,\ldots,x_n$.
QUESTION: Is this formula true ?

Remember that the graph which we are considering consists of space bounded configurations of some Turing machine accepting our given language in PSPACE . Each configuration can be encoded by a binary number of length k for some value of k bounded by some polynomial in the input length. Such a bit string can also be seen as a truth value assignment to a sequence of k boolean variables. From the proof of Cook's theorem, establishing the NP-completeness of SATISFIABILITY (see e.g. [5] or [13]) one can infer that there exists a propositional formula of length $O(k)$ in $2.k$ propositional variabeles P_0 (k for i and k for j) such that $P_0(x_1\ldots x_k y_1\ldots y_k)$ $\underline{\text{iff}}$ $i = j$ or i $\underline{\text{succ}}$ j .
One can now define by induction formulas $P_p(x_1\ldots x_k y_1\ldots y_k)$ expressing the existence of a path from i to j of length $\leq 2^p$. A naive description would P_p have containing two copies of P_{p-1} but a standard trick from complexity theory allows us to reduce the number of occurrences of P_{p-1} in P_p to 1 :

$$P_p(x_1\ldots x_k y_1\ldots y_k) = \exists z_i\ldots z_k[\forall u_1\ldots u_k v_1\ldots v_k\ [$$
$$(\ (u_i..u_k=x_1..x_k\ \underline{\text{and}}\ v_1..v_k=z_1..z_k)\ \underline{\text{or}}\ (u_1..u_k=z_1..z_k\ \underline{\text{and}}\ v_1..v_k=y_1..y_k)\ )$$
$$\underline{\text{imp}}\ P_{p-1}(u_1\ldots u_k v_1\ldots v_k)\ ]]$$

Substituting for $x_1 \ldots x_k$ and $y_1 \ldots y_k$ the codings of the initial resp. final configuration of our machine in $P_k(x_1 \ldots x_k y_1 \ldots y_k)$ we obtain a closed Quantified Boolean Formula , whose validity expresses the existence of an accepting computation. From this observation and from the fact that it is not hard to see that P_k is a formula of length $O(k^2)$ in $O(k^2)$ variables, one concludes that QBF is PSPACE complete.

One can consider SATISFIABILITY to be a type of a solitaire game: the player has to search for a truth assignment to the variables which makes the given formula true. Cook's theorem shows that it is in fact a rather difficult kind of solitaire game. Similarly QBF can be seen to be a two person perfect information game. Two players, Elias and Alice, in turn choose truth values for the variables quantified by $\exists$ and $\forall$ respectively in the order of the nesting of the quantifiers. Elias tries to establish the truth of the formula whereas Alice tries to prove that the given formula is false. If the formula indeed is true then Elias has a winning strategy; otherwise it is a win for Alice.

The proof of PSPACE-completeness of QBF has inspired various researchers in complexity theory to investigate the complexity of endgame analysis of various (generalizations) of real life games.

A useful intermediate game is GENERALIZED GEOGRAPHY [33] ; from there one can reach HEX, CHECKERS, GO (provided some termination rule is given forcing the game to terminate fast enough), and even the HEX game on the traditional hexagonal board ; see [8,10,21,27,33] .
For people interested what has happened to CHESS : this royal game is not in the list since its endgame analysis turns out to be even more complex than PSPACE [10] .

The reader should not obtain the impression from the preceding remarks that in general solitaire games are at worst NP-hard ; in fact deciding whether a given directed acyclic graph can be pebbled using a given number of black pebbles is another example of a PSPACE complete problem [14] . This problem occupies a rather exceptional position in the zoo of PSPACE complete problems, for most animals in this collection either allow some direct encoding of space bounded computations (like e.g. Reif's generalized mover's problem [26]) or they show the alternating behavior of a two person game.

It should therefore be no surprise at all that "alternation" in fact forms the fundamental concept in one of the machine models in the second machine class.

3. RANDOM ACCESS MACHINES WITH UNBOUNDED PARALLELISM

A model based on a RAM with unbounded parallelism has been described by L.M. Goldschlager [15] . There is a main RAM called the CPU and a possibly infinite sequence of subordinate RAM processors, called $PPU_0,\ldots,PPU_m$,..... . These processors all operate on a common memory of (unbounded) RAM registers.

The CPU is an ordinary RAM with the usual load instructions (numeric, direct and indirect), store instructions (direct and indirect), arithmetic instructions (addition, substraction, parallel bitwise boolean operations, and division by 2), accept and reject, and a conditional jump. It also can broadcast an instruction to all PPU's which then execute this instruction all simultaneously.

The PPU's have beside the access to the main memory also some private registers. Their instruction code includes numeric, direct and indirect loading from private memory, indirect loading from global memory, direct and indirect store to private memory, indirect store to global memory and the same arithmetic instructions as the CPU . There is no accept or conditional jump for the PPU's . Some form of conditional instruction is needed, however, in order to make the proofs correct. For this reason the instruction which writes in global memory is made a conditional one.

Clearly it is unwise to have a model where some computation step involves an infinite amount of work due to the fact that an infinite number of PPU's execute some instruction simultaneously. Therefore some mechanism is needed to keep the PPU's, except for a finite set, inactive during an instruction. This mechanism is obtained by giving each PPU access to its index number, which is stored in some local register called its signature : this register is denoted SIG . The contents of SIG can be read by a PPU but not be overwritten.

Each instruction broadcasted by the CPU contains as a mandatory parameter a register, whose value is used as an upper bound for the PPU's which will perform this instruction: PPU_k will execute the instruction in case its signature (equal to k) is less than or equal to this upper bound. Otherwise it will disregard the instruction.

The signatures are also used to solve the problem of simultaneous writes in global memory. If more than one processor writes in the same register the PPU with the lowest signature will find its value stored after the instruction; the other ones were overruled. For the effect of such a rule in real world life the reader is referred to [38] .

This model, called the SIMDAG, can be understood as the extreme of a SIMD machine. Each PPU executes the same instruction but due to the local data and the indirect addressing different PPU's can access different parts of global memory in the same instruction.

The crucial result which establishes the SIMDAG as a member of the second machine class can be formulated as follows (Th. 2.1 and 2.2 in [15]):

THEOREM: Let $T(n)$ be a constructible time bound $\geq \log(n)$. Then $NSPACE(T(n)) \subset SIMDAG\text{-}TIME(T(n)) \subset SPACE(T^2(n))$. Consequently $PSPACE = SIMDAG\text{-}PTIME$.

The first inclusion is proved by implementing the transitive closure algorithm from section 2 in its parallel version. The implementation runs in two stages. Let $\langle i,j \rangle$ be a pairing function from $\mathbb{N} \times \mathbb{N}$ onto $\mathbb{N}$. Again numbers of length $c.T(n) =: K$ represent configurations of a Turing machine with space bound $T(n)$. During the first stage of the algorithm $PPU_{\langle i,j \rangle}$ is instructed to compute the matrix element $M[i,j] = 1$ if $i = j$ or i succ j . Next for K iterations $PPU_{\langle i,\langle j,k \rangle\rangle}$ inspects the matrix elements $M[i,k]$ and $M[k,j]$ and puts a 1 in $M[i,j]$ if it finds a 1 in both. This step of the algorithm requires the possibility of a conditional instruction in the PPU , since otherwise it seems to be impossible that the positive information of the 1 found by some $PPU_{\langle i,\langle j,k \rangle\rangle}$ is not overruled by the negative 0 from some $PPU_{\langle i,\langle j,k' \rangle\rangle}$ with $k' < k$.

The CPU now can decide to accept at the end of the computation by inspecting the matrix element M[init, accept] . It is not difficult to see that both stages of the transitive closure algorithm require time $O(T(n))$. This proves the first inclusion.

The second inclusion is established by showing how to simulate a SIMDAG on a Turing machine. Note that the number of PPU's invoked during a computation may grow exponential in $T(n)$. However, all values manipulated have lengths linear in $T(n)$.

Note that the contents both of the local registers of each PPU and of the global registers at time t are completely determined by their contents at time t-1 . Working out the rules for each instruction (where the parallel write turns out to be the most problematic one) one obtains a complicated recursive procedure expressing the content of register i at time t denoted WORD(i,t) resp. LOCAL(i,t,k) depending on whether it is a global register or a register of PPU_k , in terms of WORD(i',t-1) and LOCAL(i',t-1,k') for various i' and k' . Some of dependencies may involve a nonconstant number of previous values like in the case of a write into global memory instruction executed by a number of PPU's , but in those cases a simple iterative scheme for evaluating this recursion is available.

An analysis along these lines learns that the functions WORD(i,t) and LOCAL(i,t,k) are mutually recursive, but can be evaluated for time t by loading no more than a constant times t values and pending recursive calls in stackframes; so the memory requirements are bounded by T(n).T(n). For the simulation it is necessary that the same chain of instructions is executed time and again. In order to obtain correct information on the instruction executed at time t the simulation begins by writing the entire trace of computation on a worktape. This trace is certified during the computation as being correct (if not the next trace is attempted) . In this way a simulation is obtained which even works for nondeterministic SIMDAG's. The simulator can accept after certifying a trace which ends by performing the accept instruction. This proves the second inclusion and the theorem. It should be noted that the proof for the second inclusion with some minor modifications, will be repeated for some other models in the sequel.

4. MODELS WITH PARALLEL RECURSION

The recursive transitive closure algorithm on which the Savitch theorem is based clearly is perfectly implementable on a machine model which supports recursive procedures. Now it is well known that using some form of a stack implementation traditional RAM's and Turing machines support recursion, but there is hardly any gain in efficiency compared to sequential computation. The cause of this lack of gain is the fact that recursive calls do not run in parallel on these models. Moreover, a machine, after having performed a recursive call, has to wait for the called

procedure to terminate before it can compute any further.

The members of the second machine class based upon parallel recursion accommodate to this defect by allowing the calling procedure to go on, after having performed the call. The recursive call is performed on a newly created or activated copy of a device isomorphic to the calling device. The calling device can use its power to perform further recursive calls. Using a tree structure of recursive calls a single machine therefore can activate in linear time an exponential number of subordinate devices, all working in parallel.

The most intricate part of setting up such a model is the choice of communication and parameter passing mechanisms. One possibility, related to the SIMDAG model discussed before, would be to use communication via global memory. This choice, however, turns out to be not the correct one - the resulting machine would become so powerful that it exceeds the second machine class. I will return to this issue at the end of this survey. Instead one considers communication and parameter passing to be a local activity, to be performed under the responsibility of the called procedure and the calling process, inaccessible to anyone else.

In order to be more specific I will go somewhat deeper in the details of the PRAM model introduced by Savitch and Stimson [30,31] . A k-PRAM is a version of a RAM which can issue upto k recursive calls in parallel. When calling a recursive subordinate machine it passes (a bounded number of) parameters by loading these values in the first registers of the recursively called machine. This subordinate machine then starts computing by executing its initial instruction; due to the presence of parameters its logical "task" can be different for different calls. The subordinate machine can perform recursive calls on its turn, thereby creating further offspring.

When the recursive call is completed the subordinate machine terminates computation by loading its answer into a special purpose channel register which can be read by its parent (it cannot be written by the parent). This represents the only way the offspring can inform the parent that it has completed computing. The parent can either ask for the value stored in the channel register, in which case the parent gets suspended upto the time the offspring is terminated if the situation arises that the answer asked for is not yet available; or the parent can inquire about the status of its son by performing a conditional instruction, having the

definedness of the channel register as a condition. Using this feature a machine can create two recursive sons, each processing some modification of the same task, and accept the answer provided by the son who happens to be the first in obtaining this answer. The same feature allows a parent to terminate or abort computation before all its descendants have terminated.

The k-PRAM has beside the instructions needed for performing recursive calls, reading channel registers and testing for termination of offspring, an instruction set containing the traditional RAM instructions for literal, direct and indirect loading, direct and indirect storing, and a conditional jump. Its arithmetic consists of addition and substraction only.

Savitch and Stimson show that the deterministic version of the PRAM satisfies PRAM-PTIME = PSPACE , provided the bound on the number of recursive calls performed in parallel k is at least 2 . From the preceeding comments it is not difficult to see why this equality holds.

To see that PSPACE $\subset$ PRAM-PTIME one just should convince oneself that the recursive version of the transitive closure algorithm can be implemented on a k-PRAM whenever $k \geqslant 2$, in such a way that the recursive calls are performed in parallel. This would however yield an algorithm which still violates the polynomial time bound due to the fact that the intermediate node n introduced when performing the call path(p,i,j) by the recursive calls path(p-1,i,n) and path(p-1,n,j) has to cycle through 2^k possible values. A possible solution is to use a nondeterministic PRAM version for the proof of this inclusion, establishing at a later time that the deterministic and nondeterministic versions of the PRAM have equal PTIME classes. This turns out to require a nontrivial proof.

A more direct solution is to have the machine which is assigned to perform the call path(p,i,j) creating a tree of 2^k offspring machines, each being given a different guess for an intermediate node n ; once n is fixed the two recursive calls path(p-1,i,n) and path(p-1,n,j) are performed. This transformation increases the recursion depth from k to k^2 . The time needed by each machine can be bounded as being linear in the size of the arguments dealt with. Hence the total time used by the simulation becomes of order k^3 . Remembering that k was proportional to the space used by the original PSPACE bounded Turing machine one

obtains the required inclusion.

For the above simulation there is no problem concerning nonterminating behavior of the machines created during the simulation. Once k has been evaluated the ultimate running time is known.

This is rather different from the situation where one wants to simulate an arbitrary $T(n)$-time bounded nondeterministic k-PRAM by a deterministic k-PRAM with polynomial overhead in time. The basic trick is the same as in the above simulation. Instead of performing a single nondeterministic call a tree of $2^{T(n)}$ deterministic simulations is initiated, each being equipped with a different oracle of length $T(n)$ which tells it how to choose whenever a nondeterministic choice arise. In this general case there arise problems due to the fact that different choices of the oracle may lead to different answers appearing in the channel registers, and it is no longer clear which of these answers is the one which should be forwarded to the parents. Neither is it clear what to do about machines which have not yet terminated. How does one implement the test-on-termination instruction issued by a father if its offspring which represents a single son actually consists of $2^{T(n)}$ descendants ?

These and other problems are solved in the paper by Savitch and Stimson; the final result shows that for well behaved $T(n)$ a deterministic 2-PRAM can simulate a $T(n)$-Time-bounded k-PRAM in time $O(T^6(n))$; for a polynomial overhead factor in machine model theory the exponent 6 is unusually high.

The proof of the inclusion PRAM-PTIME $\subset$ PSPACE is a straightforward sequential implementation of a parallel algorithm. Clearly the recursion depth is bounded by the running time of the PRAM. So if we know for sure that all calls terminate the standard stack implementation of recursion will yield a RAM computation whose space requirements are $O(T^3(n))$ for a $T(n)$-time-bounded PRAM . The exponent 3 is the rough upper bound based on the argument that in time $T(n)$ the PRAM can create copies upto recursion depth at most $T(n)$, each consuming at most $T(n)$ registers, filling them with values of length at most $O(T(n))$.

The condition that all calls terminate is easy to satisfy by equipping all copies with a clock initialized at creation time, which will shut off the copy at time $T(n)$, reporting failure to its parent. Such clocks are needed anyhow, due to the fact that the father computes on after having created offspring - how otherwise can one determine in the

sequential simulation which one of two sons was the first to terminate?

For more details on the rather intricate simulations I refer to the paper [31] . Although the basic ideas of the simulations seem to be simple the details are complex due to the fact that the model allows for rather ill-behaved machines, in particular for the nondeterministic variant. It should be mentioned also that a similar recursive machine model can be based upon the standard Turing machine model rather than on a RAM; Savitch has worked out such a model in [28] .

5. MACHINES BASED ON ALTERNATION

The concept of alternation has produced the family within the second machine class with the "cleanest" theoretical behavior. It was discovered independently by Stockmeyer & Chandra [3] and Kozen [20] ; these authors together composed a paper [4] for the JACM . It should be mentioned also that the discovery of this concept lead to the authors receiving an Outstanding Inovation Award by IBM - one of the few occasions where this award was granted to a purely theoretical result.

As was the case with the models based upon parallel recursion, alternating models can be built on almost every machine. In fact an alternating machine is nothing but a nondeterministic machine with a modified mode of acceptance. In a deterministic machine there is just one computation, and the machine accepts iff this one computation is accepting. In the nondeterministic mode there is a tree of possible computations and the machine accepts as soon as there exists an accepting path in the tree. One could therefore investigate the situation where acception occurs iff all possible paths in the tree accept. One even can go further by interleaving both modes of accepting (universal and existential modes), and this is the essence of the alternation concept.

An alternating machine is a nondeterministic device, the states of which are labeled either accepting, rejecting, negating, universal or existential. Accepting and rejecting states are terminal - they have no successor states. A negating state should have exactly one successor state, whereas universal and existential states should have one or more successor states.

If one considers alternating Turing machines, then the state of such a machine clearly corresponds to the entire configuration of reading heads,

input head and worktape contents. The label (existential, universal, etc.) is determined completely by the finite control state in this configuration.

Given some input the alternating device determines a unique tree in configurations, the paths in which represent possible computations. The root of the tree is the initial configuration, whereas the sons of some configuration are its successor states (without loss of generality each configuration has at most two descendants). Accepting and rejecting configurations are the leaves.

Each node in the computation tree is assigned a quality from the set { accept, reject, indef } where the quality indef is needed for dealing with infinite branches in the tree. The quality of an accepting (rejecting) leaf is accept (reject). For internal nodes the rules for determining the quality are as follows. The quality of a negating node is accept (reject) iff the quality of its unique descendant is reject (accept). The quality of an existential node is accept (reject) if one of its descendants has quality accept (all its descendants have quality reject). The quality of a universal node is accept (reject) if all its descendants have quality accept (one of its descendants has quality reject). All nodes whose quality cannot be decided based upon the above rules have quality indef.

It is clear that for a finite computation tree the label indef occurs nowhere. But also in the case that the tree contains infinite branches it is possible that the indef quality fails to propagate upwards, since the indef quality at some node is canceled by the accept (reject) quality of its brother in the case of an existential (universal) node.

The input of the computation is accepted iff the root obtains the quality accept.

The rules determining the quality of the nodes in the tree can be considered to be a recursive procedure for evaluating this quality. As a consequence this evaluation can be performed by evaluating the limit of a sequence of partial evaluations. In the first partial evaluation only the leaves obtain their quality. Next each existential node with an accept son obtains the quality accept, each existential node with all its sons being reject becomes reject, etc. If the root obtains a

quality at all it obtains one after finitely many iterations. As a consequence the quality of the root, if defined, depends on a finite subtree only, from which one infers that alternating machines accept recursively enumerable sets only.

The time consumed by a computation of an alternating machine is the maximal length of a path in a finite subtree determining the quality of the root. The space consumed by this computation is the maximal space of any configuration in this finite subtree. Based upon this definition one can obtain the space or time bounded complexity classes for alternating machines.

In the sequel we will restrict ourselves to alternating Turing machines. In the first place we have the equality A-PTIME = PSPACE . The proof of this equality is not difficult compared to what we have seen before. To establish the inclusion PSPACE $\subset$ A-PTIME it suffices to show that the PSPACE complete problem QBF can be accepted in polynomial time on an alternating Turing machine, but this is almost trivial: the machine can try out all 2^m truth value assignments for the m quantified variables by making a nondeterministic choice for each of the variables in turn. If the quantifier is existential the choice will be performed by an existential node, and if the quantifier is universal a universal node will make the choice. If all variables have obtained a value the formula is evaluated. It is clear that such an alternating Turing machine will accept QBF in quadratic time.

The inclusion A-PTIME $\subset$ PSPACE is obtained as follows. Without loss of generality one can assume that the PTIME bounded alternating Turing machine has terminating computations only. As a consequence the quality of the root of a tree of computations can be decided by a simple tree traversal, and a stack implementation for this traversal will use no more space than the square of the running time of the given alternating Turing machine.

It is interesting to have a look at the equalities A-LOGSPACE = P and A-PSPACE = EXPTIME as well. These equations are based upon the fact that alternating space is equivalent to deterministic time of the next exponental level, i.e., $\text{A-SPACE}(S(n)) = \bigcup_{c>0} \text{DTIME}(c^{S(n)})$.

A space bound on an alternating machine determines an exponential bound on the number of possible configurations. By making a list of all these configurations together with their labels (as determined by the

embedded state of the finite control) we can initiate a computation which will determine the quality of some configuration as soon as one has sufficient information on the quality of its (at most two) sons. By a repeated scan over the entire list, during which either a new quality is determined or it becomes clear that no further qualities can be determined at all, one can determine the quality of the initial configuration on the given input. The entire process takes time quadratic in the number of configurations. This proves the inclusion $A\text{-}SPACE(S(n)) \subset \bigcup_{c>0} DTIME(c^{S(n)})$.

The converse inclusion is based upon the following construction. Let M be a T(n) - time bounded deterministic Turing machine. We must show how an alternating Turing machine can accept L(M) in space O(log(T(n))). Consider an accepting computation of M . We represent this computation in the usual way by giving a complete time-space diagram of the computation. This diagram has the property that its correctness can be certified by purely local conditions: the symbols in row i are completely determined by the three symbols in row i-1 located directly above this symbol. The contents of row 0 are nothing but the initial configuration of the computation simulated. Finally the computation accepts iff its bottom row contains somewhere an occurrence of the accepting state symbol.
An alternating machine can now guess where this accepting state symbol occurs in the diagram; next it certifies this occurrence by guessing the three symbols above it (using existential choices) and certifying each of these three symbols (using universal configurations); moreover the machine certifies whether the three symbols are consistent with the symbol to be certified according to the program of M .

The storage required for specifying this process consists of space for storing row and column numbers (both of which are bounded by T(n)) and the space for storing the symbols to be certified (space O(1)) . This shows that the entire simulation requires space O(log(T(n))) .

The alternating machines provide also a simple representation of the polynomial time hierarchy [37] ; the layers in this hierarchy can be obtained by bounding the number of alternations along a computation path in the tree. The type of a class corresponds with the label of the initial state. Everything is fully natural.

The naturalness of the alternation concept has lead to a large number of applications of these devices for establishing upper and lower bounds for the complexity of concrete problems; reference [4] already mentions

applications varying between propositional dynamic logic, combinatorial games and decision complexity bounds for fragments of mathematics. It is clear that the alternation concept should be included in the standard introduction to complexity theory presented in the curriculum for computer science students.

6. MACHINES WHICH MANIPULATE HUGE DATA IN UNIT TIME

The final clan of machines in the second machine class which I want to discuss consists of RAM-like devices with powerful instructions where time is measured using uniform time measure : each instruction is being charged one unit of time independent of the size of the values manipulated. The outstanding representatives of this group of machines are the vector machines as introduced by Pratt & Stockmeyer [25] , and the RAM's with multiplication and division as invented by Hartmanis & Simon [16 , 17] .

As proposed originally these machines require both the parallel bit manipulation instructions (and , or , xor , not performed bitwise on very large numbers) , and instructions which allow for the creation of exponentially large numbers in polynomial time. In the vector machine these huge numbers are obtained by shift instructions where the distance of the shift is read from a scalar register; the model allows for scalar registers only the use of standard arithmetic like + and - . Hartmanis and Simon produce these large numbers by allowing for multiplications and divisions in unit time. Since shifts can be obtained by multiplications or divisions by pure powers of 2 it is directly clear that the multiplication - division RAM's can simulate the vector machines with constant factor overhead in time.

In fact one can forsake the right shift performed by a division ; instead of shifting one register content k bits to the right, all other registers are shifted k bits to the left ! This observation yields the consequence that multiplication in the context of the presence of bitwise logical instructions suffices for playing this game. This is the MRAM model described in [17] . It has been only recently established that having available both multiplication and division one can do without the parallel bitwise logical instructions; see Bertoni, Mauri & Sabadini [1] .

In order to establish the connection between these powerful RAM models and the class PSPACE we have to show two simulations. First we must show how to accept all PSPACE sets in polynomial time using the extended arithmetic of the model. It suffices to show this for the PSPACE-complete problem QBF . Consider therefore an arbitrary closed quantified boolean expression $Q_1x_1Q_2x_2\ldots Q_kx_kP(x_1,\ldots,x_k)$ where the length of the propositional expression P called m , can be assumed to exceed k .

First we indicate how our RAM in polynomial time can compute a single number which, if considered as a bit string, consists of the binary representations of the first 2^k numbers each separated by a block of m zeros. As a side product a "mask" is obtained consisting of 2^k ones with k-m-1 zeros inbetween each pair of ones. This mask together with its translates can be used to and out of the number obtained the 2^k corresponding digits in each of the binary representations. If we consider these binary representations to be the list of all possible truth value assignments to the k boolean variables from the given expression, we see that the mask enables us to evaluate in parallel each variable for all assignments in unit time.

Consider the following program fragment (where p = k+m, the block length):

```
mask:= 1    ; mult:= 2^p ;  reps := 0 ;
for i to k   do  x_i:= 1  od  ;
for j to k   do
  for  i  to  k  do
    if  i ≠ j  then  x_i  :=  x_i.(1+mult) fi
  od;
  mask :=  mask.(1+mult) ;  mult:= mult.mult
od ;
for  i  to  k  do   reps := reps + 2^(i-1).x_i  od
```

After executing this fragment mask is the intended mask, whereas reps contains the 2^k binary representations on a row separated by m 0-s .

Using similar tricks, involving the bitwise and and or instruction it now is possible to write in the block of zeros a copy of the propositional formula P with the truth values substituted for the variables in each block. Next, using masks and small shifts, one can

replace every occurrence of a logical connective in P by the truth value computed by this connective in the block. Finally this will lead to the parallel evaluation of P for all possible truth assignments. Another use of the mask will enable us to extract these 2^k resulting values.

In the third stage of our algorithm we will fold these answers together into a single answer. For i from k down to 1 the list of 2^i remaining answers is split into two equal parts, the two parts are aligned by another multiplication with some power of 2 and the two strings are merged using a parallel and (or) depending whether $Q_i = \forall$ ($Q_i = \exists$) . After this folding operations the final result is the single bit which remains.

The above proof is a simplification of the proof given both by Stockmeyer & Pratt or the similar proof by Hartmanis & Simon : these original proofs give a direct simulation of the transitive closure algorithm from section 2 . A related shortcut is present in [1] .

The second simulation we must provide is the simulation in polynomial space on a Turing machine of one of these powerful RAM's. Again the fact that the RAM is polynomial time bounded is used in setting up a recursive definition for the value of register i at time t in terms of register values at time $t-1$. Note however that in the present case the register values itself become so huge that they no longer can be written in polynomial space★. Instead the recursive definition involves a third parameter: the bit position . So one writes down a recursive expression for the function FIND(i,t,p) representing the value of bit p in register i at time t . The fastest growing function which can be computed on this model is the function obtained by squaring a fixed register at every instruction, i.e., the function c^{2^t} . This shows that after t steps the largest register content has at most exponential size $2^t.c'$ so the index of a bit position has length proportional to t . It can therefore be written down in polynomial space.

One of the more difficult operations to implement in this bitwise recursive expression is the multiplication; the value of bit i in the product

★ By a standard trick we can use a block of consecutive registers only, and therefore the register addresses remain bounded.

depends of the first i bits in both arguments (due to the carry) so in order to evaluate it one must in fact perform the entire multiplication of both arguments modulo 2^{i+1}. Don't ask for the final running time of the simulation. As with the SIMDAG model one must repeat the same instructions over and over again. To guide the process the entire trace of the computation is written down on a separate worktape, and the simulation will certify this trace by establishing that all conditional jumps are properly performed. If not the next trace is attempted. So again the simulation in the deterministic case is not essentially different from the one for the nondeterministic case.

7. BEYOND THE SECOND MACHINE CLASS: THE NONDETERMINISTIC MIMD-RAM

In their STOC 10 paper Fortune & Wyllie [9] describe a variant of a RAM model with full parallel recursion which somehow has a structure obtained by combining some elements of the SIMDAG and the PRAM as described in sections 3 and 4. Since this machine is capable of setting up a large number of copies of itself, all operating on private memory and a joint global memory (as is the case with the SIMDAG), but not restricted to execute the same instruction in all active copies at the same time (as is the case with the PRAM), I will refer to this model under the name MIMD-RAM. This is not the name used by the authors - they refer to their machine under the name P-RAM but in the present paper this would be misleading. The MIMD-RAM is a synchronous model. In the MIMD-RAM there is an infinite set of registers used as global memory. Each individual processor has its private set of infinitely many local registers. Each machine has the usual RAM instructions for loading and storing (literal, direct and indirect), and the usual conditional jumps. The arithmetic is restricted to addition and subtraction. The parallelism is activated by a FORK instruction, which creates a new copy of the machine which starts by executing the initial instruction. Information is passed by initializing the accumulator of the newly created machine with the value of the accumulator of the machine copy performing the FORK instruction. This enables the machine both to pass parameters and to inform the newly created copy of the precise task it should perform. Information is passed back by writing in global memory. Simultaneous reading from global memory is permitted. A simultaneous write is considered to yield an error condition - if such a write occurs the entire computation

jams and rejects. The machine accepts iff the oldest copy halts with a 1 in the accumulator. The READ instruction is massaged in such a way that sublinear running times become feasible; for details of this trick I refer to the full paper.

The deterministic version of this machine is a fair member of the second machine class. One has MIMD-RAM-PTIME = PSPACE . The inclusion PSPACE $\subset$ MIMD-RAM-PTIME can be shown by simulating a 2-PRAM computation. One only needs to simulate the channels by global memory registers, and this is not difficult - just make sure that different channels are simulated in different registers. One can also perform a computation which determines in k steps the 2^k-th power of the transition relation in the graph of all space bounded configurations of a Turing machine, where k is linear in the space bound.

The converse inclusion is proved by obtaining a recursive expression for the contents of register i of machine j at time t . Since in t steps at most 2^t machines are activated and values of size at most t are computed, all arguments for this recursive procedure can be written down in polynomial space. As before the depth of the recursion is bounded by t as well. The only problem presented by this model is that it is no longer possible to write down a complete trace of the entire computation since each of the exponentially many copies can perform a different sequence of instructions. For that reason at each instance in the recursion the instruction executed is guessed nondeterministically, creating the obligation to certify that indeed the instruction guessed is the right one. Since the machine simulated is deterministic one can prove by induction that for every machine at every time there exists just a single certifiable guess for the instruction performed. So also in the situation which is inevitable that each guess is made over and over again (and has to be certified every time anew) the guesses will be made in a consistent manner. Clearly the recursive expression for this machine will be far more complicated than for the simulations we have seen before.

It seems that the nondeterministic variant of the MIMD-RAM is more powerful than the deterministic one. Fortune & Wyllie show that one has NP = NMIMD-RAM-LOGTIME and NEXPTIME = NMIMD-RAM-PTIME .

It is not difficult to show that with an exponential overhead in time an ordinary Turing machine can guess a complete computation record of some

NMIMD-RAM computation, write it down on a worktape and certify its correctness. This suffices for providing the inclusions in the direction indicated by NMIMD-RAM-LOGTIME $\subset$ NP . In order to understand the reverse inclusion we have to show how to accept some NP-complete problem in logarithmic time on an NMIMD-RAM . For this problem we take the problem BOUNDED TILING (called SQUARE TILING (GP13) by Garey & Johnson [13]):

BOUNDED TILING:

INSTANCE: A finite set of tiles (squares with colours given on their edges) T , and an N by N square with a given colouring of the 4N edges on the border.

QUESTION: Is it possible to tile the N by N square with copies of the tiles in T (without rotations or reflections) such that each pair of adjacent tiles have matching colours, and such that the tiles adjacent to the border have colours matching the given colouring of the border?

Given an instance of BOUNDED TILING the NMIMD-RAM will first (using its modified input convention) extract the value of N from the input in logarithmic time (note that the instance actually encodes N in unitary notation). Next it will create N^2 copies of itself, each representing a square in the N by N square. This can be done in time log(N) since in constant time each machine can create two new ones. Each of these copies will guess the tile by which its corresponding square will be tiled (the only nondeterministic step in the entire computation). Finally these guesses will be written in global memory (each machine having its own register) and next each machine will verify whether its edge colours match with the ones of its neighbours. If all verifications succeed this positive information is collected by having each father in the tree writing an acknowledging value in his own register in global memory, after having verified that his two sons (if activated) have done so before. Collecting the positive information upwards in the tree again requires time O(log(N)) .

Since the inclusion PSPACE $\subset$ NEXPTIME is unknown to be proper the above result yields no certified proof that the NMIMD-RAM indeed exceeds the second machine class by being to powerful. It is however certain that these machines are more powerful than the standard class since NP $\neq$ NEXPTIME .

A similar behavior is shown by the LPRAM model introduced by W. Savitch [32] . This model is a hybrid of the k-PRAM and the MRAM , since it combines recursive parallelism with vector instructions which manipulate huge objects in single registers. The model satisfies the equalities LPRAM-NLOGTIME = NP and LPRAM-PTIME = PSPACE . The first equality makes it plausible that the model does not belong to the second machine class.

Finally I should mention in this section a recent note by Norbert Blum [2] . In this note Blum attacks the parallel computation axiom by providing a model for which one has $NP \subseteq$ //-LOGTIME . His model resembles the SIMDAG ; however the convention for simultaneous writes is different. In fact Blum considers two distinct modes for dealing with simultaneous writes. In the note the key observation is that the detection of a path of length $T(n)$ in the configuration graph requires $O(\log(T(n)))$ iterations of the transitive closure algorithm. However, in order to obtain the time bound $O(\log(T(n)))$ the initial transition matrix must be created in time $O(\log(T(n)))$ as well, and this issue is not discussed in the note at all.

If we also have a space bound $S(n)$ for the Turing machine computation to be simulated, using the standard arithmetic in the RAM a time bound $\Omega(S(n))$ is required for encoding/decoding $S(n)$-bounded configurations by bit strings. Blum suggested in a private correspondence several methods for reducing this time to $O(\log(S(n)))$ by distributing the pattern matching required for detecting a possible transition over $O(S(n))$ processors, but again it is difficult to see how this partitioning can be done, unless the arithmetic of the machine is extended by rather mild shift and masking instructions. A final possibility is to extend the hardware in order that processor $P_{<i,<j,k>>}$ should have direct access to bit k in P_i and P_j .

Notwithstanding Blum's belief that "this is not a great extension of the machine model", these ideas seem to indicate that the borderline between the second and the third machine class is rather hard to locate exactly.

8. PARALLEL MACHINES WITH POLYNOMIAL BOUNDS ON THE AMOUNT OF HARDWARE USED

It has been observed by many authors whose papers we have discussed in the preceding sections, that the power of the members of the second machine class is rooted in the possibility of activating an exponential amount of hardware in polynomial time. For example, a multiplication RAM with a polynomial bound on the length of the values used during the computation can be simulated in polynomial time by an ordinary Turing machine. Similarly, a k-PRAM which is restricted to use no more than a polynomial number of registers altogether can be simulated in polynomial time by an ordinary RAM (Corr. 52. in [31]). For the NMIMD-RAM one has the result that the combination of a polynomial time bound and a polynomial bound on the total number of registers reduces the power to PSPACE (Th. in 3 in [9]).

Still the above restrictions do not yield a finite bound on the hardware used by the components in the parallel computer, since each RAM register still can store an arbitrary large integer. The real restriction to finite components is enforced by considering parallel computers whose components are finite automata, which some way or another work together in performing a computation.

An example of a machine with finite components is the alternating finite automaton as considered in [4] . It is shown that these devices accept only regular languages; the gain is a doubly exponential blowdown in the minimal number of states needed for constructing the automaton (see section 5 in [4]) .

Goldschlager [15] introduces the model called conglomerate. This is a machine composed of finite controls with each control having k input channels and a single output channel. The topology of the network of components is described by a connection function $f:\{1,\ldots,k\}^* \to \mathbb{N} \cup \{\perp\}$. This function is defined inductively by:

$f(\varepsilon) = 0$ (the index of the root-component);
$f(sa) = j$ iff for some k $f(s) = k$ and M_j's output channel is connected to input port a of M_k;
$f(sa) = \perp$ iff no such k exists.

Clearly one can encode hideous complex sets by using highly complex or even undecidable connection functions. One should therefore restrict oneself to connection functions with low complexity.

Goldschlager shows that conglomerates with a PSPACE connection function which are PTIME bounded recognize sets in PSPACE only. On the other hand it turns out to be possible to simulate a SIMDAG with quadratic overhead on a conglomerate with a LOGSPACE (universal) connection function, so all of PSPACE can be accepted in this way. This shows that these devices composed from finite controls form a genuine member of the second machine class. There exists moreover a LOGSPACE computable universal connection function with the property that the conglomerate based on this connection function can simulate every other conglomerate with a most reasonable overhead (which depends on the parallel time required for computing the connection function of the simulated conglomerate), see Th. 5.1 in [15] .

The AGGREGATE which was introduced by Dymond & Cook [7] is a related model of a parallel machine consisting of finite components. It shares with the reference machine [35] the power of modifying its topology during the computation. On the other hand it resembles a logical circuit in the sense that it is intended to work correctly for inputs of a fixed length only. Two relevant measures are time and hardware (number of components). The authors investigate a number of connections between these measures and the standard measures of time, space and reversals for the ordinary Turing machines. From circuit theory the issue of uniform circuit families vs. nonuniform families is inherited, further complicating the theory. The paper investigates classes defined by three simultaneous resource bounds. A detailed discussion would be far beyond the scope of this survey paper, so I refer to the literature.

Finally, I would like to mention Galil & Paul's work on universal interconnection patterns for parallel networks [12] (this theory is being extended by Fr. Meyer auf der Heide [23]), and Cook's survey paper on concrete problems which are recognized in POLYLOG time using a polynomially bounded number of processors [6]. The latter paper reports research on two complexity classes which deal with parallelism and which are becoming quite popular. These classes are:

NC : the languages recognizable by uniform logical circuits of polynomial size and polylog depth ;

SC : the languages recognizable by sequential devices simultaneously in polynomial time and polylog space .

Both of these classes are subsets of P , with all further reasonable questions on equalities or inclusions being unknown. So a more detailed introduction to these classes is outside the scope of this survey.

9. REFERENCES

[1] A. BERTONI, G.MAURI & N. SABADINI, *A characterization of the class of functions computable in polynomial time on random access machines*, Proc. STOC 13 (1983) 168-176.

[2] N. BLUM, *A note on the "Parallel Computation Thesis"*, Inf. Proc. Letters 17 (1983) 203-205.

[3] A.K. CHANDRA & L. STOCKMEYER, *Alternation*, Proc. FOCS 17 (1976) 98-108.

[4] A.K. CHANDRA, D.C. KOZEN & L.J. STOCKMEYER, *Alternation*, JACM 28 (1981) 114-133.

[5] S.A. COOK, *The complexity of theorem proving procedures*, Proc. STOC 3 (1971), 151-158.

[6] S.A. COOK, *The classification of problems which have fast parallel algorithms*, Proc. FCT'83; Springer LCS 158 (1983) 78-93.

[7] P.W. DYMOND & S.A. COOK, *Hardware complexity and parallel computation*, Proc. FOCS 21 (1980) 360-372.

[8] S. EVEN & R.E. TARJAN, *A combinatorial problem which is complete in polynomial space*, JACM 23 (1976) 710-719.

[9] S. FORTUNE & J. WYLLIE, *Parallelism in random access machines*, Proc. STOC 10 (1978) 114-118.

[10] A.S. FRAENKEL, M.R. GAREY, D.S. JOHNSON, T. SCHAEFER & Y. YESHA, *The complexity of checkers on an $N \times N$ board - preliminary report*, Proc. FOCS 19 (1978) 55-64.

[11] A.S. FRAENKEL & D. LICHTENSTEIN, *Computing a perfect strategy for $n \times n$ chess requires time exponential in n* , Proc. ICALP 8 (1981), Springer LCS 115 278-293.

[12] Z. GALIL & W.J. PAUL, *An efficient general-purpose parallel computer*, JACM 30 (1983) 360-387.

[13] M.S. GAREY & D.S. JOHNSON, *Computers and Intractability, a Guide to the Theory of NP-Completeness*, Freeman, San Fransisco, 1979.

[14] J.R. GILBERT, T. LENGAUER & R.E. TARJAN, *The pebbling problem is complete in polynomial space*, SICOMP 9 (1980) 513-524.

[15] L.M. GOLDSCHLAGER, *A universal interconnection pattern for parallel computers*, JACM 29 (1982) 1073-1086.

[16] J. HARTMANIS & J. SIMON, *On the power of multiplication in random access machines*, Proc. SWAT 15 (1974) 13-23.

[17] J. HARTMANIS & J. SIMON, *On the structure of feasible computations*, in M. RUBINOFF & M.C. YOVITS eds., *Advances in Computers* 14, Acad. Press 1976, pp. 1-43.

[18] J.E. HOPCROFT & J.D. ULLMAN, *Introduction to Automata Theory, Languages and Computaion*, Addison Wesley 1979.

[19] D.S. JOHNSON, *The NP-Completeness Column: An Ongoing Guide*, J. of Algorithms, quarterly, since Dec. 1981.

[20] D. Kozen, *On parallelism in Turing machines*, Proc. FOCS 17 (1976) 89-97.

[21] D. LICHTENSTEIN & M. SIPSER, *Go is PSPACE-hard*, Proc. FOCS 19 (1978) 48-54.

[22] M. MACHTEY & P. YOUNG, *An Introduction to the General Theory of Algorithms*, Theory of computation series, North-Holland, New York 1978.

[23] F. MEYER AUF DER HEIDE, *Efficiency of universal parallel computers*, Acta Informatica 19 (1983) 269-296.

[24] A.R. MEYER & L.J. STOCKMEYER, *The equivalence problem for regular expressions with squaring requires exponential time*, Proc. SWAT 13 (1972) 125-129.

[25] V.R. PRATT & L.J. STOCKMEYER, *A characterization of the power of vector machines*, JCSS 12 (1976) 198-221.

[26] J.H. REIF, *Complexity of the mover's problem and generalizations - extended abstract*, Proc. FOCS 20 (1979) 421-227.

[27] S. REISCH, *HEX ist PSPACE-volständig*, Acta Informatica 15 (1981) 167-191.

[28] W.J. SAVITCH, *Relations between deterministic and nondeterministic tape complexities*, JCSS 4 (1970), 177-192.

[29] W.J. SAVITCH, *Recursive Turing machines*, Intern. J. Comput. Math. 6 (1977) 3-31.

[30] W.J. SAVITCH, *The influence of the machine model on computational complexity*, in J.K. LENSTRA, A.H.G. RINNOOY & P. VAN EMDE BOAS eds. *Interfaces between Computer Science and Operations Research*, Math. Centre Tracts 99 (1978) 3-32.

[31] W.J. SAVITCH & M.J. STIMSON, *Time bounded random access machines with parallel processing*, JACM 26 (1979) 103-118.

[32] W.J. SAVITCH, *Parallel random access machines with powerful instruction sets*, Math. Syst. Theory 15 (1982) 191-210.

[33] T.J. SCHAEFFER, *Complexity of some two-person perfect-information games*, JCSS 16 (1978) 185-225.

[34] A. SCHÖNHAGE, *On the power of random access machines*, Proc. ICALP 6 (1979) Springer LCS 71 (1979) 520-529.

[35] A. SCHÖNHAGE, *Storage modification machines*, SICOMP 9 (1980) 490-508.

[36] C. SLOT & P. VAN EMDE BOAS, *On tape versus core; an application of space efficient hash functions to the invariance of space*, Proc. STOC 16 (1984) to appear.

[37] L.J. STOCKMEYER, *The polynomial-time hierarchy*, TCS 3 (1976) 1-22.

[38] M. TOONDER, *Tom Poes en de Krakers* (in Dutch), Reprint, Oberon, The Hague (1978).

An Introduction to Parallelism in Combinatorial Optimization

G.A.P. Kindervater, J.K. Lenstra
Centre for Mathematics and Computer Science, Amsterdam

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

1980 Mathematics Subject Classification: 90Cxx, 68A05, 68C25, 68Exx.
Key Words & Phrases: parallel computer, computational complexity, polylog parallel algorithm, sorting, shortest paths, scheduling, log space completeness for $\mathcal{P}$, linear programming, dynamic programming, knapsack, branch and bound, traveling salesman.
Note: This paper will also be published in *Discrete Applied Mathematics*.

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

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

The organization of the paper is as follows.

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

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

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

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

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

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

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

1. Machine models

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

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

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

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

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

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

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

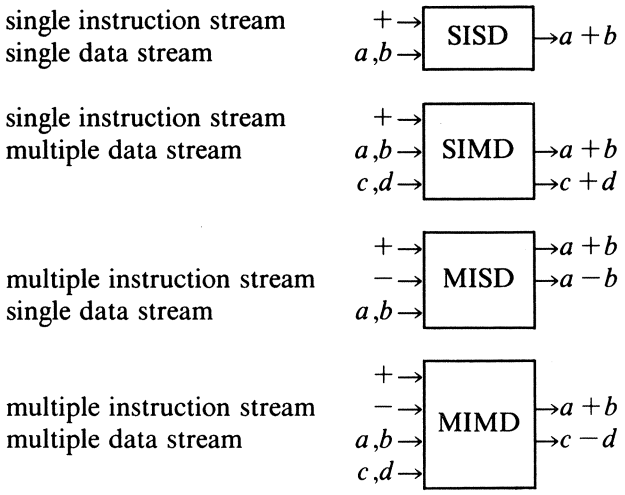


FIGURE 1. The classification of Flynn.

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

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

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

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

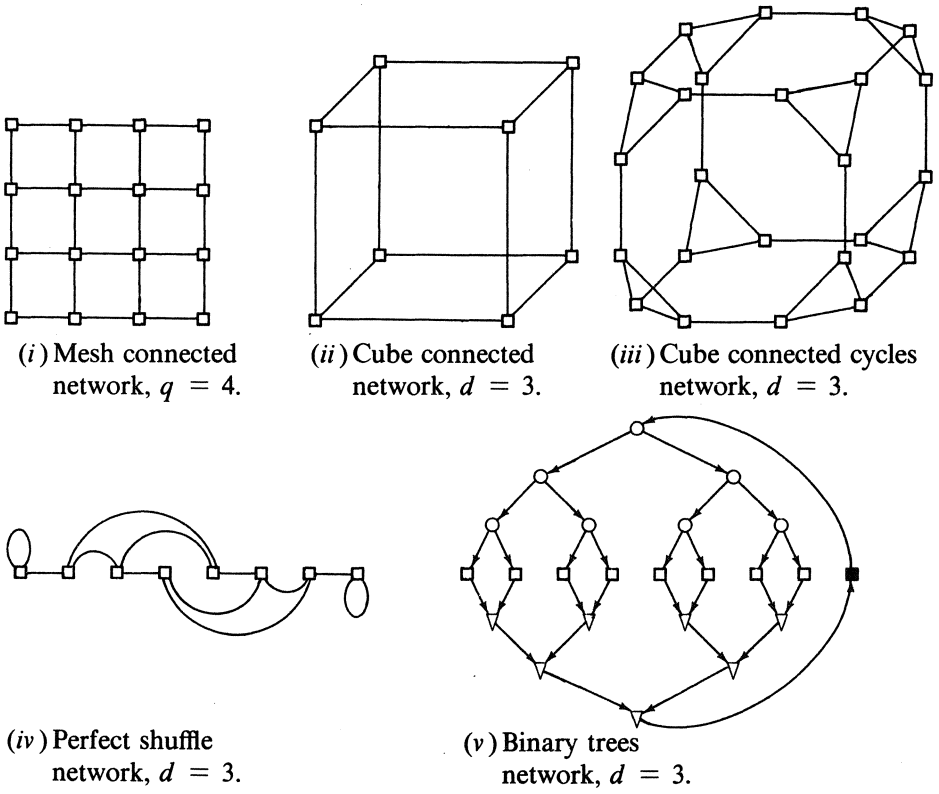


FIGURE 2. Five interconnection networks.

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

(*i*) *Two-dimensional mesh connected* network [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 $d_2 = 2(q-1) = \Theta(\sqrt{p})$.

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

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

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

(*v*) *Binary trees* network [Bentley & Kung 1979]. There are $p = 3\cdot2^d-2$

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

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

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

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

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

EXAMPLE 3. *Membership testing.* Given a set S of n elements and an element e, one can test whether $e \in S$ in $O(\log n)$ time on a binary trees network with

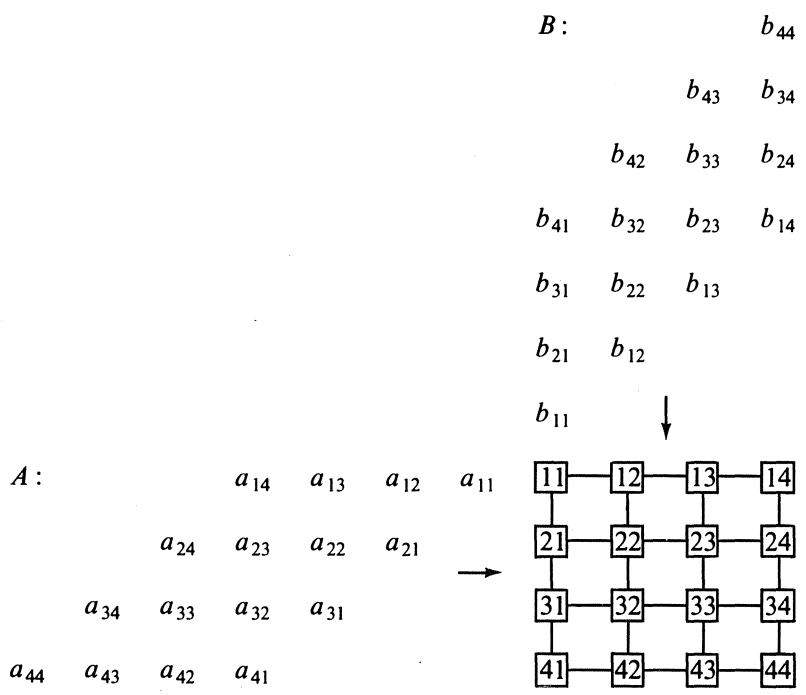


FIGURE 3. Matrix multiplication on a mesh connected network.

$d = \lceil \log n \rceil$. Denote the processors corresponding to the common leaves by P_i ($i = 1,...,2^d$) and suppose that P_i stores the ith element e_i of S ($i \leqslant n$). It takes d steps for the processors in the top tree to send e down, one step for the P_i's to check whether $e_i = e$, and d steps for the processors in the bottom tree to compute the disjunction of the results.

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

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

EXAMPLE 4. *Minimum spanning tree* [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; 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 $O(1)$.

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

2. Complexity theory

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

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

models and several 'well-behaved' time bounds; see [Van Emde Boas 1985] for a survey.

The parallel computation thesis holds, for example, in the case that the machine model is a PRAM (Parallel Random Access Machine) and $T(n) = n^{O(1)}$ (i.e., a polynomial function of problem size). The PRAM is a synchronized machine with an unbounded number of processors and a shared memory, which allows simultaneous reads from the same memory location but disallows simultaneous writes into the same memory location. The computation starts with one processor activated; at any step, an active processor can do a standard operation or activate another processor; and the computation stops when the initial processor halts.

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

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

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

We have thus arrived at a distinction within $\mathcal{P}$ between the 'very easy'

problems, which can be solved in polylog parallel time, and the 'not so easy' ones, for which a dramatic speedup due to parallelism is unlikely.

The picture of the PRAM model as sketched above is in need of some qualification. The model is theoretically very useful, but its unbounded parallelism is hardly realistic. The reader will have no difficulty in verifying that a PRAM is able to activate a superpolynomial number of processors in subpolynomial time. If a polynomial time bound is considered reasonable, then certainly a polynomial bound on the number of processors should be imposed. It is a trivial observation, however, that the class of problems solvable if both bounds are respected is simply equal to $\mathcal{P}$. Within this more reasonable model, hard problems remain as hard as they were without parallelism.

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

3. Polylog parallel algorithms

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

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

In the PIDGIN ALGOL procedures in this section, we write

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

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

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

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

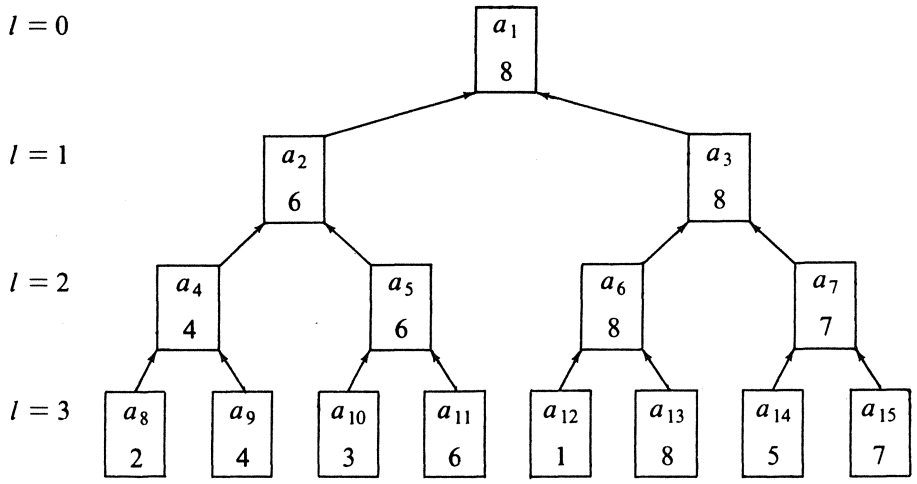


FIGURE 4. Maximum finding: an instance with $n = 8$.

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

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

EXAMPLE 6. *Partial sums* [Dekel & Sahni 1983a]. Given n numbers $a_n, a_{n+1}, \ldots, a_{2n-1}$ with $n = 2^m$, one wishes to find the partial sums $a_n + \ldots + a_{n+j}$ 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 5. In the first phase, represented by the solid arrows, the sum of the a_j's is calculated in the same way as their maximum was calculated in Example 5. Note that the a-value corresponding to a non leaf node is set equal to the sum of all a-values corresponding to the leaves descending from that node. In the second phase, represented by the dotted arrows, each parent node sends a b-value (starting with $b_1 = a_1$) to its children: the right child receives the same value, the left one receives that value minus the a-value of his brother. The b-value of a certain node is therefore equal to the sum of all a-values of the nodes of the same generation, except

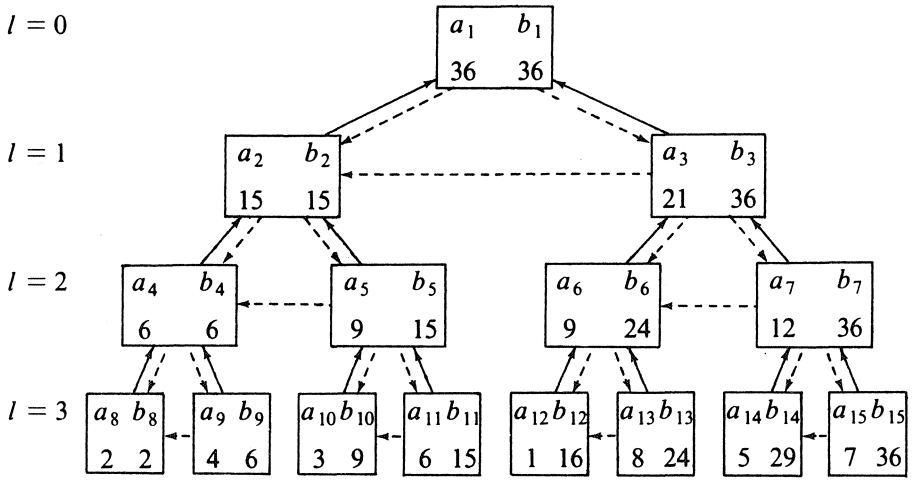


FIGURE 5. Partial sums: an instance with $n = 8$.

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.

EXAMPLE 7. *Sorting* [Muller & Preparata 1975]. Given n numbers $a_1,\ldots,a_n$, one wishes to renumber them such that $a_1 \leqslant \ldots \leqslant a_n$. 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 Example 6);

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

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

EXAMPLE 8. *Shortest paths* [Dekel, Nassimi & Sahni 1981]. Given a complete directed graph with vertex set $\{1,\ldots,n\}$ and a length c_{ij} for each arc (i,j), one wishes to find the shortest path lengths for all pairs of vertices. In [Lawler

1976] an algorithm is given which requires $O(n^3 \log n)$ time. It is based on matrix multiplication. Let $d_{ij}^{(l)}$ denote the length of a shortest path from vertex i to vertex j, containing no more than l arcs. Since a path from vertex i to vertex j consisting of at most $2l$ arcs can be split into two paths of no more than l arcs each, we have that $d_{ij}^{(2l)} = \min_{k \in \{1,\ldots,n\}} \{d_{ik}^{(l)} + d_{kj}^{(l)}\}$. Taking into account that a shortest path, 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 Example 5 with maximization replaced by minimization yields an algorithm which requires $O(\log^2 n)$ time and $O(n^3/\log n)$ processors, with a processor utilization of $O(1)$.

EXAMPLE 9. *Preemptive scheduling* [Dekel & Sahni 1983b]. Given m machines M_i $(i = 1,\ldots,m)$ and n jobs J_j, each with a processing time p_j $(j = 1,\ldots,n)$, one wishes to find a preemptive schedule of minimum length. A preemptive schedule assigns to each J_j a number of triples (M_i,s,t), where $1 \leqslant i \leqslant m$ 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* from [McNaughton 1959]. The algorithm first computes a value t^* which is an obvious lower bound on the minimum schedule length. It then constructs a schedule of length t^* by considering the jobs in an arbitrary order and scheduling them in the m periods $(0,t^*)$, carrying over the part of a job that does not fit at the end of the period on M_i to the beginning of the period on M_{i+1}. More formally:

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

An example is given in Figure 6. There are two global parameters that are updated sequentially as the job index j increases: the starting time s and the machine index i of J_j. We can calculate all starting times and machine indices simultaneously in logarithmic time, using the parallel procedures for finding the maximum and the partial sums from Examples 5 and 6 as subroutines:

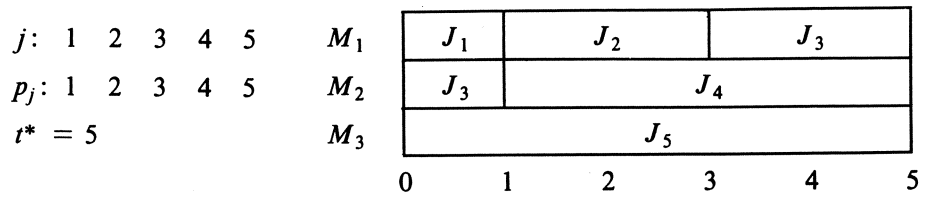


FIGURE 6. Preemptive scheduling: an instance with $m = 3$ and $n = 5$.

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

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

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

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

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

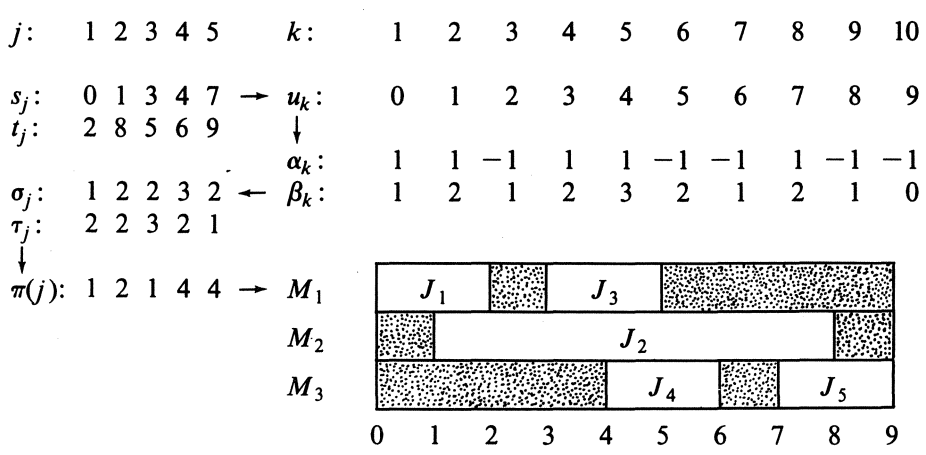


FIGURE 7. Scheduling fixed jobs: an instance with $n = 5$.

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

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

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

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

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

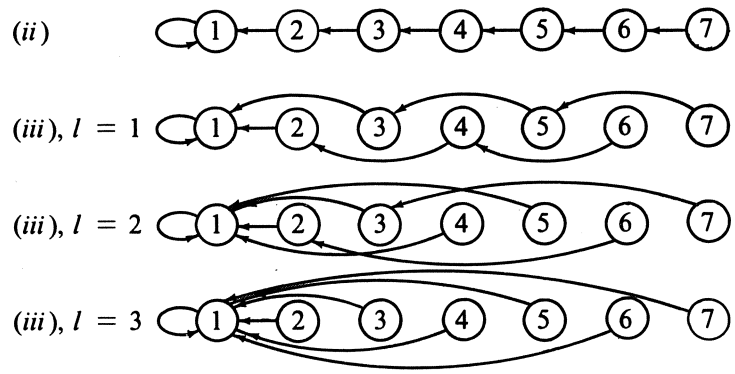


FIGURE 8. Scheduling fixed jobs: finding the first preceding job on the same machine.

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

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

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

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

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

4. LOG SPACE COMPLETENESS FOR $\mathcal{P}$

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

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

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

Goldschlager extended this result to the cases of *monotone* circuits, which have only AND and OR gates, and *planar* circuits, which have a cross free

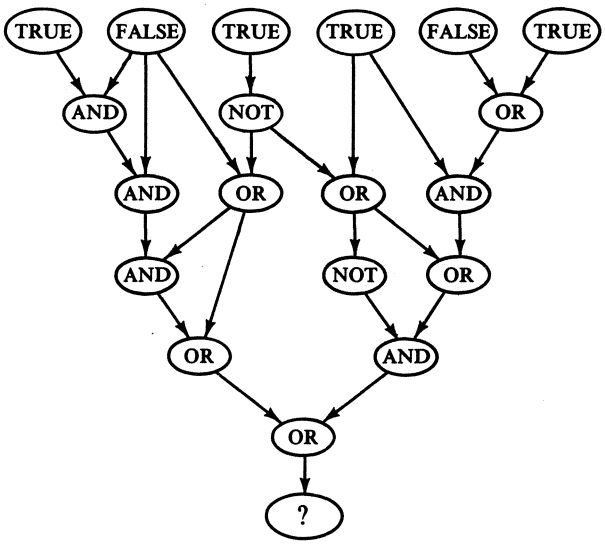


FIGURE 9. A logical circuit.

planar embedding, by giving log space transformations from the circuit value problem.

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

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

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

if gate j is	then we introduce the equations and inequalities
· an input gate with value TRUE,	· $x_j = 1$,
· an input gate with value FALSE,	· $x_j = 0$,
· an AND gate with inputs from gates h and i,	· $x_j \leqslant x_h$, $x_j \leqslant x_i$, $x_j \geqslant 0$, $x_j \geqslant x_h + x_i - 1$,
· 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 log space complete for $\mathcal{P}$ if all coefficients are equal to -1, 0 or 1, and each row and column of the constraint matrix contains at most three entries.

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

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

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

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

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

5. ENUMERATIVE METHODS

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

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

Branch and bound methods generate search trees in which each node has to deal with a subset of the solution set. Since the instructions performed at a node very much depend on the particular subset associated with that node, it is more appropriate to implement these methods in a distributed fashion on asynchronous MIMD machines. An initial analysis of distributed branch and bound, in which the processors communicate only to broadcast new solution values or to redistribute the remaining work load, is given in [El-Dessouki & Huen 1980]. In a sequential branch and bound algorithm, the subproblems to be examined are given a priority and from among the generated subproblems the one with the highest priority is selected next. In a parallel implementation, it depends on the number of processors which subproblems are available and thus how the tree is searched. One can construct examples in which p processors together are slower than a single processor, or more than p times as fast. These anomalies are analyzed in [Burton, Huntbach, McKeown & Rayward-Smith 1983; Lai & Sahni 1984] and illustrated on the traveling salesman problem in Example 15.

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

It is convenient to introduce the notation

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

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

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

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

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

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

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

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

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

ALGORITHM KS2
par $[1 \leq j \leq n]$ **par** $[0 \leq z < a_j]$ $C(j,j,z) \leftarrow 0$,
$[a_j \leq z \leq b]$ $C(j,j,z) \leftarrow c_j$;
for $l \leftarrow 1$ **to** $\log n$ **do**
$k \leftarrow 2^l$,
par $[0 \leq j < n/k]$ **par** $[0 \leq z \leq b]$ $C(jk+1,jk+k,z)$
$\leftarrow \max_{0 \leq y \leq z}\{C(jk+1,jk+\frac{1}{2}k,z-y)+C(jk+\frac{1}{2}k+1,jk+k,y)\}$.

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

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

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

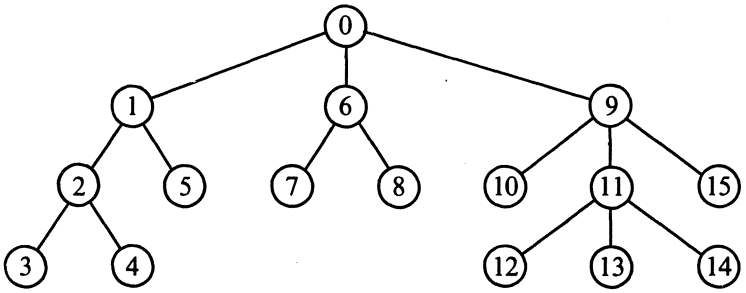
(*iii*) Apply Algorithm KS2, starting with g groups rather than with n items: $O(\log g \log b)$ time, $O(gb^2/\log b)$ processors.

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

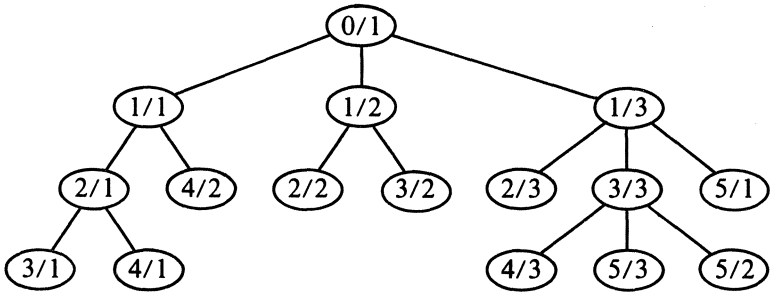
EXAMPLE 15. *Traveling salesman* [Pruul 1975]. Given a complete graph with n vertices and a weight for each edge, one wishes to find a Hamiltonian cycle (i.e., a cycle passing through each vertex exactly once) of minimum total weight.

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

Pruul designed a parallel version of this method for an asynchronous MIMD machine. Each processor performs its own depth first search; when it encounters a node that has already been selected by another processor, it



(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 10. Depth first tree search.

selects in the subtree rooted by that node an unexamined node at the highest level. Figure 10(b) illustrates the process.

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

REFERENCES

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

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

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

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

F.W. BURTON, M.M. HUNTBACH, G.P. MCKEOWN, V.J. RAYWARD-SMITH

(1983). *Parallelism in Branch-and-Bound Algorithms*, Report CSA/3/1983, University of East Anglia, Norwich.

J. CASTI, M. RICHARDSON, R. LARSON (1973). Dynamic programming and parallel computers. *J. Optim. Theory Appl. 12*, 423-438.

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

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

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

E. DEKEL, D. NASSIMI, 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.

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

O.I. EL-DESSOUKI, W.H. HUEN (1980). Distributed enumeration on between computers. *IEEE Trans. Comput. C-29*, 818-825. Note: in the title, read 'network' for 'between'.

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

M.R. GAREY, 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.

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.

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

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

R.M. KARP, E. UPFAL, A. WIGDERSON (1985). Constructing a perfect matching is in Random NC. *Proc. 17th Annual ACM Symp. Theory of Computing*, 22-

32.
L.G. KHACHIAN (1979). A polynomial algorithm in linear programming. *Soviet Math. Dokl. 20*, 191-194.
G.A.P. KINDERVATER, J.K. LENSTRA (1985). Parallel algorithms. M. O'hEIGEARTAIGH, J.K. LENSTRA, A.H.G. RINNOOY KAN (eds.). *Combinatorial Optimization: Annotated Bibliographies*, Wiley, Chichester, Ch. 8.
R.E. LADNER (1975). The circuit value problem is log space complete for *P*. *SIGACT News 7.1*, 18-20.
T.-H. LAI, S. SAHNI (1984). Anomalies in parallel branch-and-bound algorithms. *Comm. ACM 27*, 594-602.
E.L. LAWLER (1976). *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York.
E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN, D.B. SHMOYS (eds.) (1985). *The Traveling Salesman Problem: a Guided Tour of Combinatorial Optimization*, Wiley, Chichester.
R. MCNAUGHTON (1959). Scheduling with deadlines and loss functions. *Management Sci. 6*, 1-12.
N. MEGIDDO (1982). *Poly-log Parallel Algorithms for LP with an Application to Exploding Flying Objects*, Unpublished manuscript.
D.E. MULLER, F.P. PREPARATA (1975). Bounds to complexities of networks for sorting and for switching. *J. Assoc. Comput. Mach. 22*, 195-201.
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.
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.
S.H. UNGER (1958). A computer oriented toward spatial problems. *Proc. IRE 46*, 1744-1750.
L.G. VALIANT (1982). Reducibility by algebraic projections. *Enseign. Math. (2) 28*, 253-268.
P. VAN EMDE BOAS (1985). The second machine class: models of parallelism. *This Volume*, 133-161.

Addresses of Authors

P. van Emde Boas
IIW/FVI
University of Amsterdam
Roetersstraat 15
1018 WB Amsterdam

D.J. Evans
Department of Computer Studies
University of Technology
Loughborough
Leicestershire LE11 3TU
U.K.

G.A.P. Kindervater
Centre for Mathematics and
Computer Science
P.O. Box 4079
1009 AB Amsterdam

J. van Leeuwen
Department of Computer Science
University of Utrecht
P.O. Box 80.012
3508 TA Utrecht

J.K. Lenstra
Centre for Mathematics and
Computer Science
P.O. Box 4079
1009 AB Amsterdam

S.J. Mullender
Centre for Mathematics and
Computer Science
P.O. Box 4079
1009 AB Amsterdam

C.J. Purcell
Control Data Corporation
4290 Fernwood Avenue
St. Paul, MN 55112
U.S.A.

M. Rem
Department of Mathematics and
Computer Science
University of Technology
P.O. Box 513
5600 MB Eindhoven

A.S. Tanenbaum
Department of Mathematics and
Computer Science
Free University
P.O. Box 7161
1007 MC Amsterdam

H.A. van der Vorst
Department of Mathematics and
Computer Science
University of Technology
P.O. Box 356
2600 AJ Delft

MC SYLLABI

1.1 F. Göbel, J. van de Lune. *Leergang besliskunde, deel 1: wiskundige basiskennis.* 1965.

1.2 J. Hemelrijk, J. Kriens. *Leergang besliskunde, deel 2: kansberekening.* 1965.

1.3 J. Hemelrijk, J. Kriens. *Leergang besliskunde, deel 3: statistiek.* 1966.

1.4 G. de Leve, W. Molenaar. *Leergang besliskunde, deel 4: Markovketens en wachttijden.* 1966.

1.5 J. Kriens, G. de Leve. *Leergang besliskunde, deel 5: inleiding tot de mathematische besliskunde.* 1966.

1.6a B. Dorhout, J. Kriens. *Leergang besliskunde, deel 6a: wiskundige programmering 1.* 1968.

1.6b B. Dorhout, J. Kriens, J.Th. van Lieshout. *Leergang besliskunde, deel 6b: wiskundige programmering 2.* 1977.

1.7a G. de Leve. *Leergang besliskunde, deel 7a: dynamische programmering 1.* 1968.

1.7b G. de Leve, H.C. Tijms. *Leergang besliskunde, deel 7b: dynamische programmering 2.* 1970.

1.7c G. de Leve, H.C. Tijms. *Leergang besliskunde, deel 7c: dynamische programmering 3.* 1971.

1.8 J. Kriens, F. Göbel, W. Molenaar. *Leergang besliskunde, deel 8: minimaxmethode, netwerkplanning, simulatie.* 1968.

2.1 G.J.R. Förch, P.J. van der Houwen, R.P. van de Riet. *Colloquium stabiliteit van differentieschema's, deel 1.* 1967.

2.2 L. Dekker, T.J. Dekker, P.J. van der Houwen, M.N. Spijker. *Colloquium stabiliteit van differentieschema's, deel 2.* 1968.

3.1 H.A. Lauwerier. *Randwaardeproblemen, deel 1.* 1967.

3.2 H.A. Lauwerier. *Randwaardeproblemen, deel 2.* 1968.

3.3 H.A. Lauwerier. *Randwaardeproblemen, deel 3.* 1968.

4 H.A. Lauwerier. *Representaties van groepen.* 1968.

5 J.H. van Lint, J.J. Seidel, P.C. Baayen. *Colloquium discrete wiskunde.* 1968.

6 K.K. Koksma. *Cursus ALGOL 60.* 1969.

7.1 *Colloquium moderne rekenmachines, deel 1.* 1969.

7.2 *Colloquium moderne rekenmachines, deel 2.* 1969.

8 H. Bavinck, J. Grasman. *Relaxatietrillingen.* 1969.

9.1 T.M.T. Coolen, G.J.R. Förch, E.M. de Jager, H.G.J. Pijls. *Colloquium elliptische differentiaalvergelijkingen, deel 1.* 1970.

9.2 W.P. van den Brink, T.M.T. Coolen, B. Dijkhuis, P.P.N. de Groen, P.J. van der Houwen, E.M. de Jager, N.M. Temme, R.J. de Vogelaere. *Colloquium elliptische differentiaalvergelijkingen, deel 2.* 1970.

10 J. Fabius, W.R. van Zwet. *Grondbegrippen van de waarschijnlijkheidsrekening.* 1970.

11 H. Bart, M.A. Kaashoek. H.G.J. Pijls, W.J. de Schipper, J. de Vries. *Colloquium halfalgebra's en positieve operatoren.* 1971.

12 T.J. Dekker. *Numerieke algebra.* 1971.

13 F.E.J. Kruseman Aretz. *Programmeren voor rekenautomaten; de MC ALGOL 60 vertaler voor de EL X8.* 1971.

14 H. Bavinck, W. Gautschi, G.M. Willems. *Colloquium approximatietheorie.* 1971.

15.1 T.J. Dekker, P.W. Hemker, P.J. van der Houwen. *Colloquium stijve differentiaalvergelijkingen, deel 1.* 1972.

15.2 P.A. Beentjes, K. Dekker, H.C. Hemker, S.P.N. van Kampen, G.M. Willems. *Colloquium stijve differentiaalvergelijkingen, deel 2.* 1973.

15.3 P.A. Beentjes, K. Dekker, P.W. Hemker, M. van Veldhuizen. *Colloquium stijve differentiaalvergelijkingen, deel 3.* 1975.

16.1 L. Geurts. *Cursus programmeren, deel 1: de elementen van het programmeren.* 1973.

16.2 L. Geurts. *Cursus programmeren, deel 2: de programmeertaal ALGOL 60.* 1973.

17.1 P.S. Stobbe. *Lineaire algebra, deel 1.* 1973.

17.2 P.S. Stobbe. *Lineaire algebra, deel 2.* 1973.

17.3 N.M. Temme. *Lineaire algebra, deel 3.* 1976.

18 F. van der Blij, H. Freudenthal, J.J. de Iongh, J.J. Seidel, A. van Wijngaarden. *Een kwart eeuw wiskunde 1946-1971, syllabus van de vakantiecursus 1971.* 1973.

19 A. Hordijk, R. Potharst, J.Th. Runnenburg. *Optimaal stoppen van Markovketens.* 1973.

20 T.M.T. Coolen, P.W. Hemker, P.J. van der Houwen, E. Slagt. *ALGOL 60 procedures voor begin- en randwaardeproblemen.* 1976.

21 J.W. de Bakker (red.). *Colloquium programmacorrectheid.* 1975.

22 R. Helmers, J. Oosterhoff, F.H. Ruymgaart, M.C.A. van Zuylen. *Asymptotische methoden in de toetsingstheorie; toepassingen van naburigheid.* 1976.

23.1 J.W. de Roever (red.). *Colloquium onderwerpen uit de biomathematica, deel 1.* 1976.

23.2 J.W. de Roever (red.). *Colloquium onderwerpen uit de biomathematica, deel 2.* 1977.

24.1 P.J. van der Houwen. *Numerieke integratie van differentiaalvergelijkingen, deel 1: eenstapsmethoden.* 1974.

25 *Colloquium structuur van programmeertalen.* 1976.

26.1 N.M. Temme (ed.). *Nonlinear analysis, volume 1.* 1976.

26.2 N.M. Temme (ed.). *Nonlinear analysis, volume 2.* 1976.

27 M. Bakker, P.W. Hemker, P.J. van der Houwen, S.J. Polak, M. van Veldhuizen. *Colloquium discretiseringsmethoden.* 1976.

28 O. Diekmann, N.M. Temme (eds.). *Nonlinear diffusion problems.* 1976.

29.1 J.C.P. Bus (red.). *Colloquium numerieke programmatuur, deel 1A, deel 1B.* 1976.

29.2 H.J.J. te Riele (red.). *Colloquium numerieke programmatuur, deel 2.* 1977.

30 J. Heering, P. Klint (red.). *Colloquium programmeeromgevingen.* 1983.

31 J.H. van Lint (red.). *Inleiding in de coderingstheorie.* 1976.

32 L. Geurts (red.). *Colloquium bedrijfsystemen.* 1976.

33 P.J. van der Houwen. *Berekening van waterstanden in zeeën en rivieren.* 1977.

34 J. Hemelrijk. *Oriënterende cursus mathematische statistiek.* 1977.

35 P.J.W. ten Hagen (red.). *Colloquium computer graphics.* 1978.

36 J.M. Aarts, J. de Vries. *Colloquium topologische dynamische systemen.* 1977.

37 J.C. van Vliet (red.). *Colloquium capita datastructuren.* 1978.

38.1 T.H. Koornwinder (ed.). *Representations of locally compact groups with applications, part I.* 1979.

38.2 T.H. Koornwinder (ed.). *Representations of locally compact groups with applications, part II.* 1979.

39 O.J. Vrieze, G.L. Wanrooy. *Colloquium stochastische spelen.* 1978.

40 J. van Tiel. *Convexe analyse.* 1979.

41 H.J.J. te Riele (ed.). *Colloquium numerical treatment of integral equations.* 1979.

42 J.C. van Vliet (red.). *Colloquium capita implementatie van programmeertalen.* 1980.

43 A.M. Cohen, H.A. Wilbrink. *Eindige groepen (een inleidende cursus).* 1980.

44 J.G. Verwer (ed.). *Colloquium numerical solution of partial differential equations.* 1980.

45 P. Klint (red.). *Colloquium hogere programmeertalen en computerarchitectuur.* 1980.

46.1 P.M.G. Apers (red.). *Colloquium databankorganisatie, deel 1.* 1981.

46.2 P.G.M. Apers (red.). *Colloquium databankorganisatie, deel 2.* 1981.

47.1 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60: general information and indices.* 1981.

47.2 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60, vol. 1: elementary procedures; vol. 2: algebraic evaluations.* 1981.

47.3 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60, vol. 3A: linear algebra, part I.* 1981.

47.4 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60, vol. 3B: linear algebra, part II.* 1981.

47.5 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60, vol. 4: analytical evaluations; vol. 5A: analytical problems, part I.* 1981.

47.6 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60, vol. 5B: analytical problems, part II.* 1981.

47.7 P.W. Hemker (ed.). *NUMAL, numerical procedures in ALGOL 60, vol. 6: special functions and constants; vol. 7: interpolation and approximation.* 1981.

48.1 P.M.B. Vitányi, J. van Leeuwen, P. van Emde Boas (red.). *Colloquium complexiteit en algoritmen, deel 1.* 1982.

48.2 P.M.B. Vitányi, J. van Leeuwen, P. van Emde Boas (red.). *Colloquium complexiteit en algoritmen, deel 2.* 1982.

49 T.H. Koornwinder (ed.). *The structure of real semisimple Lie groups.* 1982.

50 H. Nijmeijer. *Inleiding systeemtheorie.* 1982.

51 P.J. Hoogendoorn (red.). *Cursus cryptografie.* 1983.

CWI SYLLABI

1 Vacantiecursus 1984 *Hewet - plus wiskunde.* 1984.

2 E.M. de Jager, H.G.J. Pijls (eds.). *Proceedings Seminar 1981-1982. Mathematical structures in field theories.* 1984.

3 W.C.M. Kallenberg, et al. *Testing statistical hypotheses: worked solutions.* 1984.

4 J.G. Verwer (ed.). *Colloquium topics in applied numerical analysis, volume 1.* 1984.

5 J.G. Verwer (ed.). *Colloquium topics in applied numerical analysis, volume 2.* 1984.

6 P.J.M. Bongaarts, J.N. Buur, E.A. de Kerf, R. Martini, H.G.J. Pijls, J.W. de Roever. *Proceedings Seminar 1982-1983. Mathematical structures in field theories.* 1985.

7 Vacantiecursus 1985 *Variatierekening.* 1985.

8 G.M. Tuynman. *Proceedings Seminar 1983-1985. Mathematical structures in field theories, Vol.1 Geometric quantization.* 1985.

9 J. van Leeuwen, J.K. Lenstra (eds.). *Parallel computers and computations.* 1985.