vakgroep informatica R.U. Utrecht

PARALLEL COMPUTERS AND ALGORITHMS

J. van Leeuwen

RUU-CS-83-13

September 1983

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6
Postbus 80.012
3508 TA Utrecht
Telefoon 030–53 1454
The Netherlands

PARALLEL COMPUTERS AND ALGORITHMS

J. van Leeuwen

Technical Report RUU-CS-83-13

September 1983

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
Utrecht, the Netherlands

This paper was presented in the Colloquium on "Parallel computers and algorithms" held at the University of Utrecht, Fall 1983.

PARALLEL COMPUTERS AND ALGORITHMS

J. van Leeuwen

Department of Computer Science, University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht, the Netherlands

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 sise scientific problems can be solved. We describe some of the algorithmic principles that underly many parallel and distributed 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 speed-ups 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]). 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 inclu-

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

ding 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, 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"

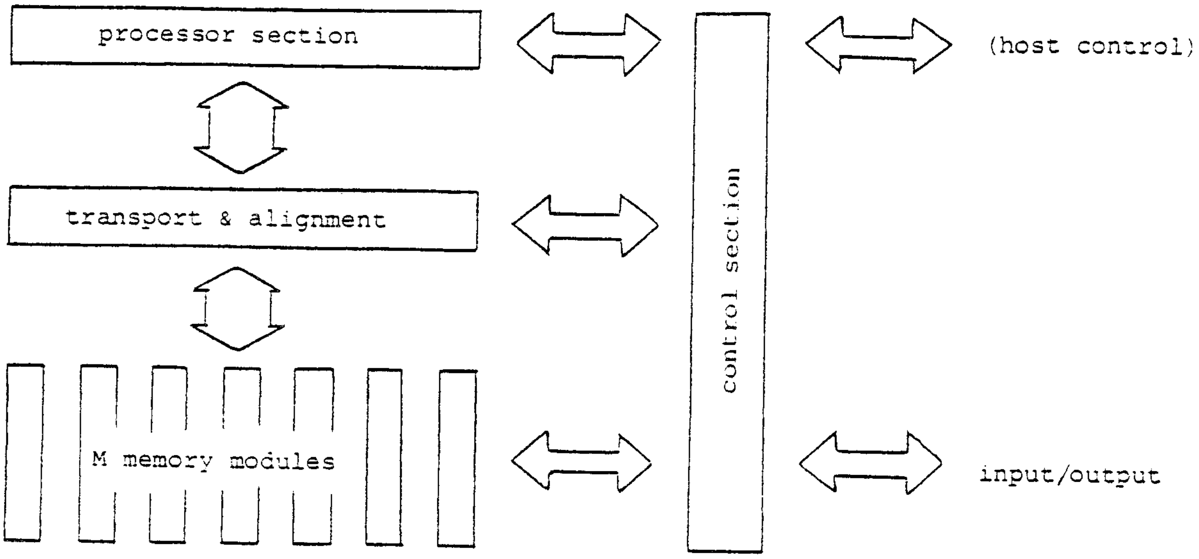


Figure 2. Global block diagram of a parallel computer

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 for the CYBER 205, M = 16 for the CRAY-1) whereas in other design 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 idealised 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) vectorised 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 pattern 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 $b \neq 0$ 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, $B \neq 0$, 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 $\alpha, \beta$}
 $\delta := 0$;
 repeat
     while $b$ even do begin $b := b$ div $2$; $\delta := \delta + 1$ end;
     if $\delta \geq 0$ then begin swap$(a, b)$; $\delta := -\delta$ end;
     if $(a+b) \bmod 4 = 0$ then $b := (a+b)$ div $2$ else $b := (a-b)$ div $2$;
```

```
    until b = 0 ;
```

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.

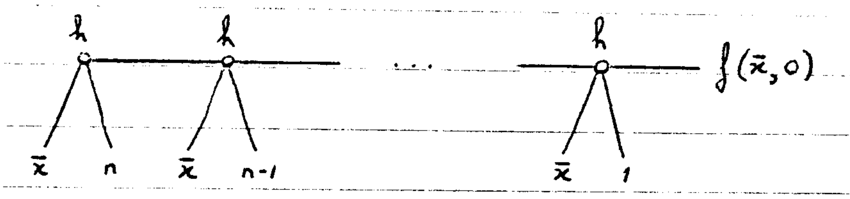
Fortunately it is not always this tricky to come up with a fast(er) parallel method. We shall discuss a number of important paradigms and the underlying techniques. We assume that processors are available in unlimited supply.

The best known examples of parallelism probably are the computations of x^n and of $a_0 + \cdots + a_{n-1}$, 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 0 to log n. The true effect of parallelism is that in the same time-bound one can compute the values $\{x, x^2, x^3, \ldots, x^n\}$ and $\{a_0, a_0+a_1, a_0+a_1+a_2, \ldots, \sum_{i=0}^{n-1} a_i\}$. There are several ways to see this. Consider a function $f(\bar{x}, n)$ defined as follows:

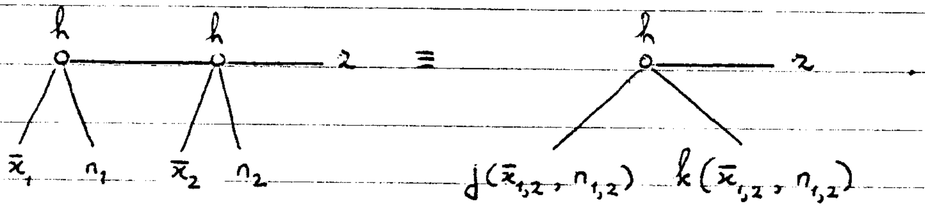
$$f(\bar{x}, 0) = g(\bar{x})$$
$$f(\bar{x}, n) = h(\bar{x}, n, f(\bar{x}, n-1)) \quad \text{for } n > 0$$

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:

$f(\bar{x}, n) =$



Definition. A function $h(\bar{x}, n, z)$ is called 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 $\{f(\bar{x},0), f(\bar{x},1), \dots, f(\bar{x},n)\}$ 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 $n \times n$ tridiagonal matrix (assuming it exists) can be computed in $O(\log n)$ time, using n processors.

Proof.

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

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

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

, then the following recursions are obtained:

$$m_i = e_i/u_{i-1} \quad (2 \le i \le n) \qquad \text{and} \qquad u_1 = d_1$$
$$u_i = d_i - e_i f_{i-1}/u_{i-1} \quad (2 \le i \le n).$$

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

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 $n \times n$-matrix with bandwith $m+1$. Then

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

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

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}. Alltogether 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 $n \times n$-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 $O(n^{\alpha+1/2}/\log^2 n)$ 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 $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ 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 $p(x) = q(x) \cdot x^{(n+1)/2} + r(x)$ 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 [37] 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. 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] has made the following observation:

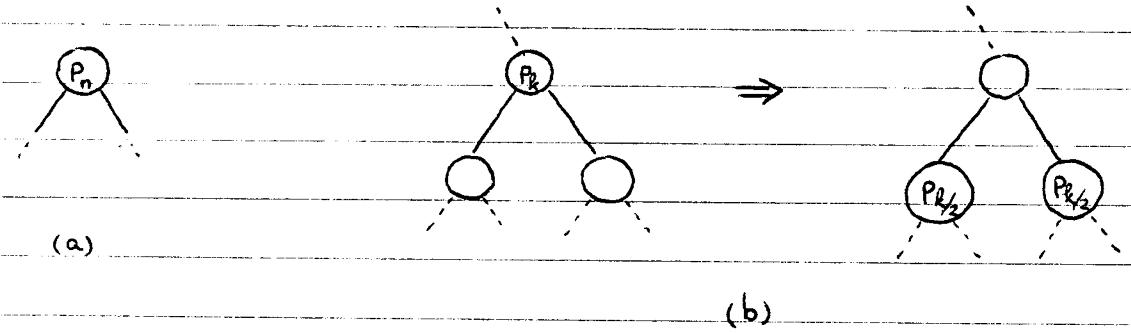


figure 4

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 algorithms 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 an $n \times n$ 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 "vectorising" existing software, but there are other applications too. An important example is the problem of computing the product $C = A \cdot B$ of two $n \times n$ matrices, which can be described by the n^2 expressions C_{ik} $(1 \le i,k \le n)$ with $C_{ik} = \sum_{j=1}^{n} A_{ij} \cdot B_{jk}$ or pictorially as

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

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:

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

$(1 \le k \le n)$. 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 vector multiplications and $n \cdot (n-1)$ 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 bandwidth). 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

$$\begin{array}{c|ccccccccc|c}
x_0 & x_1 & x_2 & x_3 & x_4 & & \cdots & & & x_{2^k-1} & x_{2^k}\\
e_1 & d_1 & f_1 & & & & & & & & \\
 & e_2 & d_2 & f_2 & & & & & & & \\
 & & e_3 & d_3 & f_3 & & & & & & \\
 & & & e_4 & d_4 & f_4 & & & & & \\
 & & & & & \ddots & \ddots & \ddots & & & \\
 & & & & & & & e_{2^k-2} & d_{2^k-2} & f_{2^k-2} & \\
 & & & & & & & & e_{2^k-1} & d_{2^k-1} & f_{2^k-1}
\end{array}$$

where e_1 and f_{2^k-1} are added for consistency and use the convention that $x_0 = x_{2^k} = 0$. By a sweep using the odd-numbered equations we zero the e and f coefficients in the even-numbered equations, to obtain a system of the form

$$\begin{array}{c|ccccccccc|c}
x_0 & x_1 & x_2 & x_3 & x_4 & & \cdots & & & x_{2^k-1} & x_{2^k}\\
e_1 & d_1' & f_1 & & & & & & & & \\
e_1' & 0 & d_2' & 0 & f_2' & & & & & & \\
 & & e_3 & d_3' & f_3 & & & & & & \\
 & & e_4' & 0 & d_4' & 0 & f_4' & & & & \\
 & & & & & \ddots & \ddots & \ddots & & & \\
 & & & & & & e_{2^k-2}' & 0 & d_{2^k-2}' & 0 & f_{2^k-2}' \\
 & & & & & & & & e_{2^k-1} & d_{2^k-1}' & f_{2^k-1}
\end{array}$$

Now observe that if we have the values of $x_0, x_2, x_4, \cdots$ then the values of $x_1, x_3, \cdots$ follow in one further step from the odd-numbered equations. But the even-numbered equations form a tridiagonal system on the $x_0, x_2, x_4, \cdots$ separately and we can continue recursively until a single equation in x_0, x_{2^k-1} and x_{2^k} remains (assuming the algorithm nowhere degenerates). Since x_0 and x_{2^k} were defined 0 we can solve for x_{2k-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 $n \times n$ 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 $(2 \le i \le n)$ and assume that after eliminating x_{j-1} $(j \ge 2)$ the P_i with $i \ge j$ have the value $l_{i1}x_1 + \dots + l_{ij-1}x_{j-1}$ 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 single block in P_{i+1}'s store. P_{i+1} starts as soon as 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 a 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.

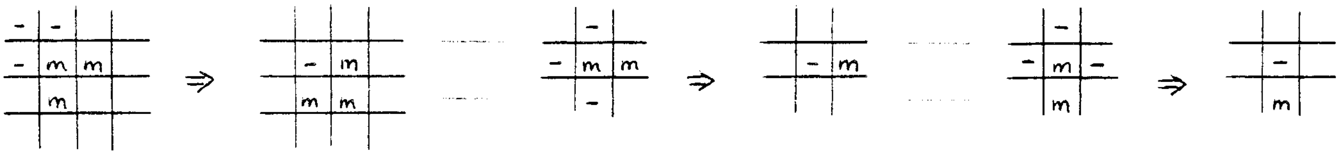
Valiant [59] has shown that $O(\log n . \log\log n)$ parallel comparisons are sufficient to sort. An excellent survey of parallel sorting algorithms was given by Friedland [14].

An eighth technique for obtaining parallel methods is often found in graph algorithms and is known as <u>collapsing</u>. 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 0 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].

<u>Theorem</u> 2.15 Let some of the processors of an $n \times n$ processor array be marked. Connectivity of the marked processors can be recognized in $O(n)$ steps.

<u>Proof</u>.

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 $n \times n \times n$ 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 algorithm. (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 $\log \mathrm{Deg}(f)$ 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

$$x_0 = y$$
$$x_{i+1} = \varphi(x_i)$$

with φ rational.

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

<u>Theorem</u> 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.

<u>Proof</u>.

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 $\deg \Phi = (\deg \varphi)^n$, 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 $x_0 = a$, $x_{i+1} = \frac{1}{2}(x_i + a/x_i)$ cannot be sped up by more than a constant factor, nc matter how many processors are supplied.

In section 2 we have always assumed that processors are available in unlimited supply ("<u>unbounded parallelism</u>"). 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]).

<u>Theorem</u> 3.3 Assume a computation consisting of a total of b operations can be carried out in time t using unbounded parallelism. Then the computation can be carried out with p processors in approxima-

tely $t + {}^{(b-t)}/_p$ steps.

Proof.

Suppose s_i operations are performed in parallel during step i $(1 \le i \le t)$, 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 $\sum_1^t \lceil {}^{s_i}/_p \rceil \le \sum_1^t {}^{(s_i+p-1)}/_p =$ $= (1 - {}^1/_p)\, t + {}^1/_p \cdot \sum_1^t s_i = t + {}^{(b-t)}/_p$. □

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 the stages of parallel activity. We assume that nodes have in-degree 0 (inputs) or, say, 2. An algorithm representation of this kind is called a <u>circuit</u>. If the algorithm 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 circuit-depth is an adequate measure of "parallel time". We show this using a simple result due to Greenberg et. al. [17].

<u>Theorem</u> 3.4 A circuit of size s and depth d can be "evaluated" in time $O(d)$ using $\lceil {}^s/_d \rceil$ processors.

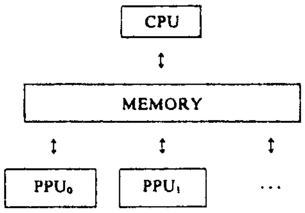
Proof.

Consider the circuit and define S_i to be the set of nodes that are i edges away from the farthest input node $(0 \le i \le d)$. Clearly S_0 consists of the input nodes, and the nodes of S_i can be evaluated once the nodes in $\bigcup_0^{i-1} S_j$ are. Thus the circuit can be "evaluated" in the order $S_0, S_1, \cdots$. Evaluation of the nodes in S_i takes $\lceil |S_i|/p \rceil$ time using p processors. The entire computation takes $\sum_0^d \lceil |S_i|/p \rceil \leq$ $\leq \sum_0^d (|S_i|+p-1)/p = (1-1/p)\,d + 1/p \sum_0^d |S_i| = d + (s-d)/p$ 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 sise 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 that 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 4) and encompasses many earlier models. All pro-

figure 4

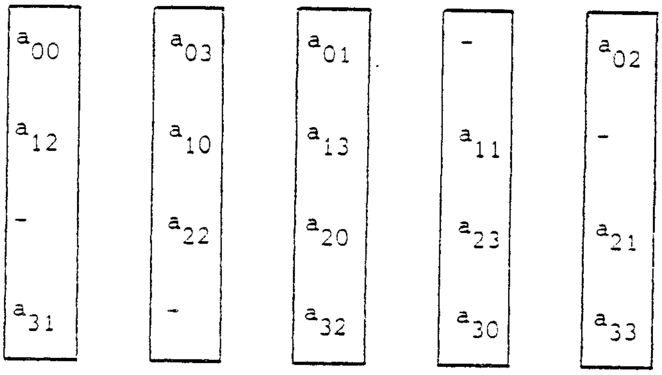


cessors have a full RAM-instruction set (no multiplication primitive), the CPU "contains" the program and occasionally broadcasts "parallel" instructions to all PPU's. Each PPU has its own index stored in a special signature register (which thus provides a way to distinguish or mark processors). Simultaneous 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:

<u>Theorem</u> 3.5 For every SIMDAG-computable function $T(n) \geq \log n$ one has $\bigcup_k \text{SIMDAG-TIME}(T^k(n)) = \bigcup_k \text{SPACE}(T^k(n))$.

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 <u>banks</u> 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 all 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 an $N \times N$ 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 5 (with $N = 4$ and $M = 5$) alleviates these difficulties at least for rows, columns, and forward and backward diagonals.



a_{00}	a_{03}	a_{01}	-	a_{02}
a_{12}	a_{10}	a_{13}	a_{11}	-
-	a_{22}	a_{20}	a_{23}	a_{21}
a_{31}	-	a_{32}	a_{30}	a_{33}

figure 5. Storing a 4×4 matrix into 5 memory banks.

Any storage scheme s that maps the elements of an $N \times N$ 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

<u>Theorem</u> 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 $N \times N$ 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.

<u>Definition</u>. A d-ordered k-vector is a vector of k elements whose i^{th} logical element $(0 \le i < k)$ 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])

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

<u>Proof</u>.

$\Rightarrow$. Consider a d-ordered k-vector and assume it can be accessed conflict-free. It means that for all $0 \le i_1, i_2 < k$, $i_1 \neq i_2$, we have $c + di_1 \not\equiv c + di_2 \pmod M$ hence $d.i \not\equiv 0 \pmod M$ for every $0 < i < k$. 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 $s(i,j) = ai + bj \pmod M$ for storing an $N \times N$ 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 $N \times N$ 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 & Wijshoff [61].

In SIMD-type architectures the processors (or perhaps even the processors and the memories) are connected in an <u>interconnection network</u> and yet another component is added to the problem of realising 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 <u>the routing problem</u> 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).

<u>Theorem</u> 3.11 Every network that realises all connections between N processors must have $\Omega(N \log N)$ switches.

<u>Proof</u>.

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 settings. 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 $a = a_{n-1}a_{n-2}\cdots a_0$ to address $b = b_{n-1}b_{n-2}\cdots b_0$ (with $0 \leq a, b < N$). 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 <u>the omega network</u> that has received most attention.

Define the "shuffle" as the mapping σ defined by $\sigma(a_{n-1}a_{n-2}\cdots a_0) = a_{n-2}\cdots a_0 a_{n-1}$ 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 or does an exchange e defined by $e(a_{n-1}a_{n-2}\cdots a_1 a_0) = a_{n-1}a_{n-2}\cdots a_1\bar{a}_0$, 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 6 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:

<u>Theorem</u> 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 $\{f_i\}_{0 \leq i < n}$

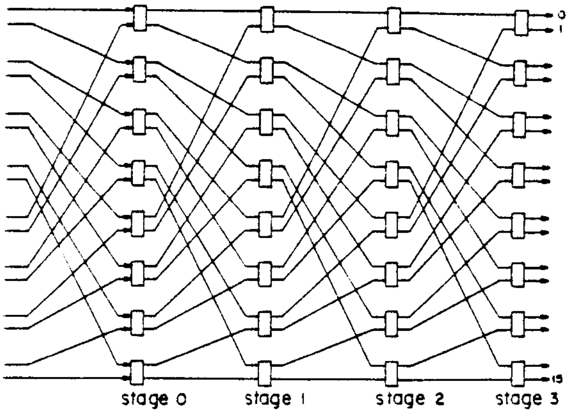


figure 6. The omega network for $N=16$.

of n-1 variables such that for all $0 \le i < n$ bit b_i of b can be expressed as $b_i = a_i \oplus f_i(b_{n-1}, \cdots, b_{i+1}, a_{i-1}, \cdots, a_0)$.

($\oplus$ 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 $\rho(a_{n-1}a_{n-2}\cdots a_0) = a_0 \cdots a_{n-2}a_{n-1}$.

<u>Theorem</u> 3.12

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

(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 $\le 3 \log N$ 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 <u>indirect binary n-cube</u>. Define the i^{th} order "butterfly" permutation ($1 \le i \le n$) 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 ($1 \le i \le n$) and an inverse shuffle at the end. The indirect binary n-cube for the case $N=16$ ($n=4$) is shown in figure 7. 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]:

<u>Theorem</u> 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

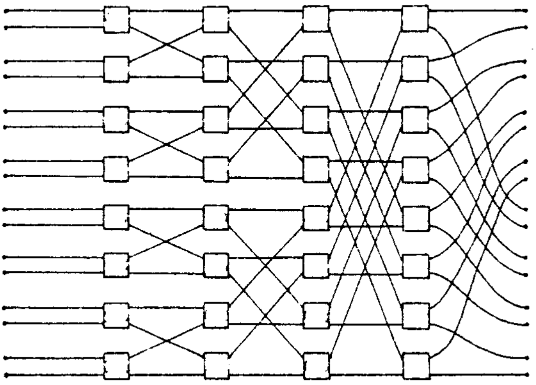


figure 7. The indirect binary n-cube for N=16

from the structure of the network (see figure 7), 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: $(c_0, \cdots, c_{N-1}) \longrightarrow (X_0, \cdots, X_{N-1})$ with $X_s = \sum_0^{N-1} c_k \omega^{sk}$ for $0 \le s \le N-1$, ω a primitive N^{th} root of unity

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 $\{1, \omega, \omega^2, \cdots, \omega^{N-1}\}$. Write $p(x) = \sum_0^{\frac{N}{2}-1} c_{2i} x^{2i} + x \cdot \sum_0^{\frac{N}{2}-1} c_{2i+1} x^{2i} = p_1(x^2) + x \cdot p_2(x^2)$, 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 compute both $p(\omega^j)$ and $p(\omega^{\frac{N}{2}+j}) = p(-\omega^j)$.

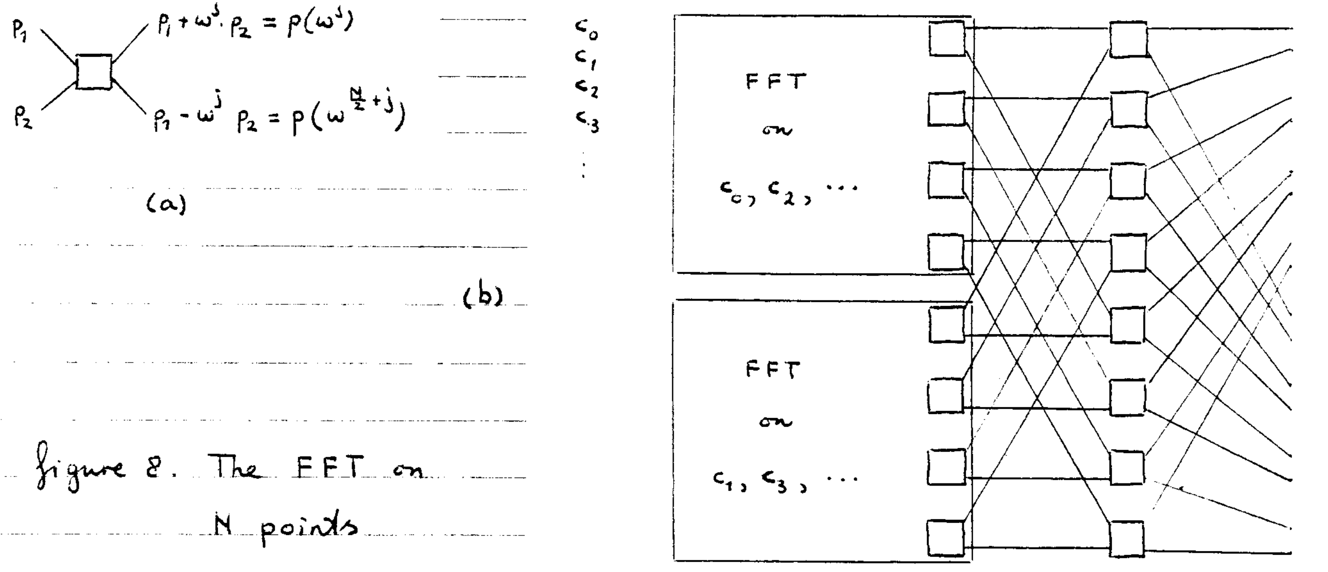

p_1
p_2
$p_1 + \omega^j \cdot p_2 = p(\omega^j)$
$p_1 - \omega^j\ p_2 = p(\omega^{\frac{N}{2}+j})$
(a)
c_0
c_1
c_2
c_3
⋮
FFT on $c_0, c_2, \cdots$
FFT on $c_1, c_3, \cdots$
(b)

figure 8. The FFT on N points

Using processing elements as shown in figure 8(a) the N-points FFT can be computed with a network of the recursive structure shown in figure 8(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 (in progress), Dept of Computer Science, University of Utrecht, Utrecht, 1983.

[3] Borodin, A., On relating time and space 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 inversion 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 bibliography, 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, Proc. 11th Annual ACM Symp. Theory of Computing, 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, Carnegie-Mellon 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., Generalised 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 tridiagonal linear system of equations, J. ACM 20 (1973) 27-38.

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

[57] Sweet, R.A., A cyclic reduction algorithm for solving block-tridiagonal 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.