Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

H.J.J. te Riele

Applications of supercomputers in mathematics

Department of Numerical Mathematics Note NM-N8502 October

APPLICATIONS OF SUPERCOMPUTERS IN MATHEMATICS

by

Herman J.J. te Riele
Centre for Mathematics and Computer Science
Kruislaan 413
1098 SJ Amsterdam
The Netherlands

Course notes for the Short Course:
SUPERCOMPUTERS IN SCIENCE AND ENGINEERING
15 - 16 October 1985
Southampton, England

These course notes provide a concise survey of the role played by vector and parallel processors in the solution of problems in computational mathematics. Some vectorization and parallelization techniques are discussed. Many examples illuminate the discussion.

Report NM-N8502
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1. INTRODUCTION

The advent of extremely powerful computers like the CRAY-1 in 1976 and the CYBER 205 in 1981 has strongly stimulated the interest of scientists and engineers in finding ways to (re-)organize their algorithms such that these computers can solve their problems with maximal performance. One could say that pipelining and parallelism in hardware has added a new dimension to algorithm design and analysis.

The CRAY-1, CYBER 205, FACOM VP-200, HITACH S810 are examples of so-called pipelined or vector processors, which perform in an optimal way when they operate on long vectors of data. Reformulation of algorithms in terms of (long) vectors is called vectorization. In the past few years parallel computers, i.e., computers with more than one central processor which can operate independently and concurrently, have appeared (like the CRAY X-MP and the Denelcor HEP). For these machines it is important to parallelize algorithms, i.e., to trace (possibly different) subprocesses which can be executed independently.

Vectorization and parallelization are techniques which, of course, have much in common (cf. section 2.3). Therefore, we will only distinguish between the two concepts (and between vector and parallel computers) if this is necessary in the given context.

In the above development, one may perceive two trends: one is a tendency to optimize algorithms for a particular vector or parallel computer by exploiting specific hardware and software features of the machine; the other is to adapt and implement algorithms in such a way that the resulting software is portable and can be auto-vectorized by a good compiler. At this moment it is difficult to judge which approach is to be preferred: rapid developments in parallel hardware and in its prize-performance ratio, and lack of standards in programming tools for vector and parallel computers are factors which make a definite choice difficult, if not impossible. Maybe the best choice at this moment is to "divide-and-conquer": develop portable software and if the performance obtained with auto-vectorization on a given machine is unsatisfactory, try to optimize the software for the given machine.

The solution of many problems in mathematics and physical sciences requires heavy computations. The corresponding algorithms can often be formulated in terms of operations on vectors and in terms of a (small or large) number of independent subcomputations. In particular, algorithms from numerical linear algebra, which operate on vectors and matrices, play a crucial role in many computational problems. A bibliography from Bochum (F.R.G.) ([BRS]) entitled: "Parallel Computing" illustrates the rapid developments: the total number of references in the second edition of September 1983 is 5161, against 2610 in the first edition of June 1982. Table 1 gives the "top ten" list of subjects from this bibliography.

Table 1
The top ten subjects from [BRS]

Subject	Number of references
Computer Architecture	1779
Algorithms	1177
Numerical Algorithms	680
Multiprocessors	592
Vector Computer	515
Networks	390
Image Processing	338
Complexity	324
Linear Algebra	303
Programming Languages	238

Here, hardware subjects show the highest scores (architecture, multi-processor, vector computer). This is reflected in the kind of subjects covered by a number of recent conferences on supercomputers and applications ([DR], [EM]). Table 2 presents a list of main subjects from mathematics and physical sciences with their scores in the Bochum Bibliography.

Table 2
Mathematical and Physical Sciences subjects from [BRS]

Mathematics	#	Physical Sciences	#
Matrix Algorithms	192	Fluid Dynamics	69
PDEs	116	Pattern Recognition	68
FFT	111	Transonic Flow	35
Graph Theory	87	Air Traffic Control	33
Arithmetic Expressions	82	Potential equation	31
Iterative Methods	73	Radar Control, Systems, Data Processing	24
Sorting	60	Poisson equation	22
Optimization	57	Weather Forecast	20
Matrix Multiplication	53	Monte Carlo Method	18
Sparse Matrices	47	Ballistic Missile Defense	14
Tridiagonal Matrices	41	Navier-Stokes equation	8
ODEs	22	Nuclear Physics	8
Direct Methods	17	Quantum Chemistry	5
Runge Kutta Methods	4		

In Section 2 of these course notes some general concepts concerning supercomputers and parallelism will be treated. Section 3 discusses a number of applications in mathematics and Section 4 treats important vectorization and parallelization techniques employed in these applications.

Excellent surveys on vector and parallel computers and algorithms are: [HE], [HJ], [MI], [OV], [SA], [SCHE], [VL] and [ZA].

2. SOME GENERAL CONCEPTS CONCERNING VECTOR AND PARALLEL COMPUTING

2.1 Some definitions

The speed of vector and parallel computers is often expressed in MFLOPS: the number of Million FLOating Point operations per Second. If a vector or parallel computer has a clock cycle time of c nanoseconds (e.g., c = 12.5 for the CRAY-1 and c = 20 for the CYBER 205), and if one result per cycle is produced (which is usually the case for the operations +, - and *), then the speed is 1000/c MFLOPS. In certain cases operations can be chained or linked such that two results per cycle can be produced, which gives a speed of 2000/c MFLOPS. However, these performances are difficult to reach in practice since there is always some overhead which decreases these figures. When we compare MFLOPS-speeds of different computers, we should be aware that different computers usually have different clock cycle times. For example, when two computers with different cycle times show the same MFLOPS-speed for some problem, then apparently the computer with the smallest clock cycle time shows the largest overhead.

When for a given problem we compare the CPU-times of a serial and a parallel or vector computer, then the speed-up S is defined as the quotient Ts/Tp where Ts is the serial and Tp the parallel CPU-time. According to Stone ([ST]), for a parallel processor with p processors typical speed-up ratios are the following:

S	Examples of Algorithms with this speed-up
kp	Matrix computations, mesh calculations
kp/log(p)	Sorting, tridiagonal linear systems, linear recurrence relations, polynonomial evaluation
klog(p)	searching
k (independent of p)	Certain nonlinear recurrence relations, certain compiler processes

Here, k is a machine-dependent constant, independent of p.

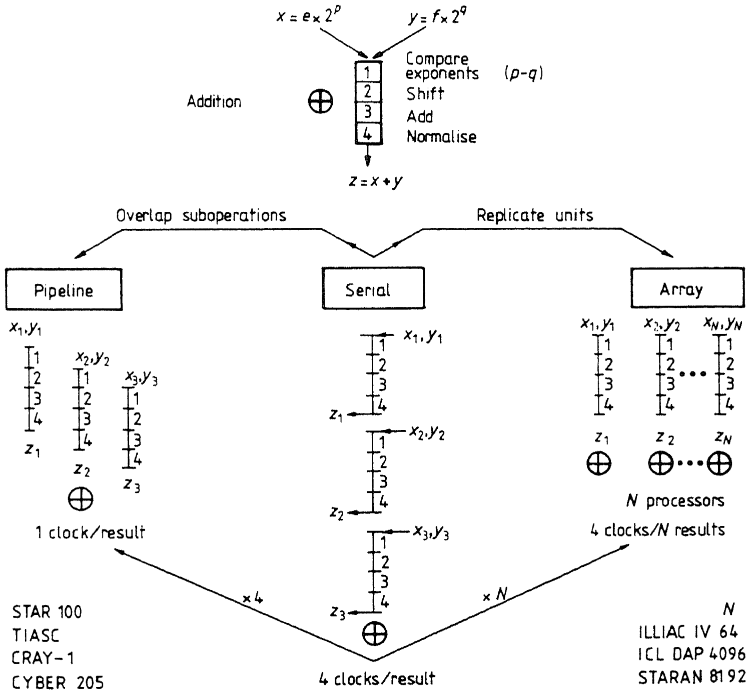
If S_p is the speed-up for p processors, then the <u>efficiency</u> E_p is defined as the quotient S_p/p. E_p measures how busy the parallel processors are during the computation. The longer the processors are idle, or carry out extra calculations introduced through the parrallelisation of the algortihm, the smaller becomes E_p .

On the various architectures the arithmetic operations may be executed in three different modes, viz. serial, pipelined and parallel. Consider, for example, the problem of adding two floating-point vectors $x = (x_i)$ and $y = (y_i)$, to obtain the sum vector $z = (z_i)$ $(i=1,2,\dots,n)$, where $z_i = x_i + y_i$. The operation of adding a pair x_i, y_i may be divided into four sub-operations, viz.,

(1) compare the exponents,
(2) shift,
(3) add mantissae, and
(4) normalize.

Figure 1 exemplifies the three different modes (derived from [HJ]).

<u>Figure 1</u>
Comparison of serial, pipelined and array architectures



It may be interesting to remark here that many supercomputers (like the CYBER 205) have both a vector and a scalar processor which may operate concurrently on different data. We have not yet seen any applications which exploit this feature of supercomputers.

2.2 Classification

Attempts have been made to arrange the various computer designs in classes. The simplest scheme is due to Flynn ([FL]): single(S) and multiple(M) streams of instructions(I) and data(D) are distinguished. This gives four possibilities: SISD, SIMD, MISD and MIMD. SISD is the classical von Neumann model: a single instruction stream operates on a single stream of data. SIMD is the class to which array processors and pipelined computers belong: all the processors interpret the same instructions and execute them on different data. The MISD class may be argued to be empty (cf. [SCHE, p. 121]). The MIMD class is the multi-processor version of the SIMD class: all processors interpret different instructions and operate on different data. For the four classes, Table 3 gives examples of machines and, schematically, examples of operations which can be executed at the same time.

Table 3
Flynn's classification

type	operations which can be executed at the same time	examples
SIMD	a + b	conventional von Neumann
SIMD	a + b, c + d	processor array (ICL/DAP, ILLIAC IV) pipelined processor (CRAY-1, CYBER 205)
MISD	a + b, a * b	
MIMD	a + b, c * d	multi-processors (CRAY X-MP, HEP)

A problem in this classification scheme are the pipelined processors. Usually, they are placed in the SIMD class although, strictly spoken, the instructions on different data are not executed at the same time; rather, each clock cycle one result is delivered from the input data stream(s).

A classification of parallel computers based on how computations proceed and how components in the architecture interact, is given in Table 4 (derived from [BO]). Other taxonomies of computers are given by Shore ([SH]) and by Schwartz ([SCHW]).

Table 4
Computer architectures and their underlying computational model

Model of Computation	Corresponding Computer Architecture
A. Sequential control on scalar data	A1. von Neumann-type computers A2. Multifunction CPU A3. Pipelined computers
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

2.3 Algorithm parallelism

It is customary ([HJ]) to define, at any stage of an algorithm, the degree of parallelism of that algorithm as the number of independent operations that can be performed in parallel, that is to say concurrently or simultaneously. On a pipelined computer the data would be interpreted as vectors and the operation would be performed on one vector. The parallelism is then the same as the vector length.
On a processor array the data for each operation are allocated to different processing elements of the array and the operations on all elements are performed at the same time. The parallelism is then the number of data elements being operated upon in parallel in this way. The degree of parallelism may remain constant during the different steps of the algorithm, or it may vary from step to step.

Usually, there are two ways to analyse algorithms for use on vector and parallel procesors:

1. Try to find, in a given algorithm, as many as possible independent subprocesses;

2. Devise a new algorithm with as many as possible independent subprocesses.

The following scheme suggests which type of computer is suitable, depending on whether the algorithm can be divided into few/many equal/different subprocesses.

$$\begin{tikzcd}
 & & \text{many} \arrow[r, dashed] & \text{pipelined} \\
 & \text{equal} \arrow[ur] \arrow[dr] & & \\
 & & \text{few} \arrow[r, dashed] & \text{parallel, serial} \\
\text{subprocesses} \arrow[uur] \arrow[ddr] & & & \\
 & & \text{many} \arrow[r, dashed] & \text{parallel} \\
 & \text{different} \arrow[ur] \arrow[dr] & & \\
 & & \text{few} \arrow[r, dashed] & \text{parallel, serial}
\end{tikzcd}$$

Various techniques for vectorization and parallelization are known, like recursive doubling, cyclic reduction, divide-and-conquer, pipelining and broadcasting. In fact, there is some overlap in these techniques. In Section 4 we shall explain the two most important ones, viz., recursive doubling and cyclic reduction. An interesting survey of many techniques, aimed at a theoretical analysis of parallel algorithms, was presented recently by Van Leeuwen ([VL]).

2.4 Organization of data

In algorithms for parallel processing, the organization and dynamic arrangement of the data play a decisive role. Let us consider a very simple example of a SIMD processor with three processors P_1, P_2 and P_3, each of which has access to three storage locations. Suppose that the elements of a 3 x 3 matrix $A = (a_{ij})$ are stored in their "natural" order, as shown below:

P_1	P_2	P_3
a_{11}	a_{12}	a_{13}
a_{21}	a_{22}	a_{23}
a_{31}	a_{32}	a_{33}

Here, we assume that P_1 has access to a_{11}, a_{21} and a_{31} and P_2 and P_3 to the second and third columns of A, respectively. However, P_1 does not have access to the second and third column, and so on. Then, parallel operation is possible on the rows and main diagonals of A, but not on the columns of A. However, the following skew arrangement enables us to operate also on the columns:

P_1	P_2	P_3
a_{11}	a_{12}	a_{13}
a_{23}	a_{21}	a_{22}
a_{32}	a_{33}	a_{31}

Some general results concerning conflict free storage access in array processors are given in [SCHE].

A related, notorious problem, called memory bank conflict, may rise because of the presence of a so-called memory bank cycle time, which means that when loading an element from one memory bank, it is not possible to load another element from that same bank in the next few, e.g., three, clock cycles. For example, suppose we have an 8-bank machine and a vector is stored in the memory as follows: the elements with index $8m + n$, $0 \leqslant n \leqslant 7$, are stored in bank number n. Then, if we need the elements with indices 0, 1, 2, ... there will be no memory bank conflict and the speed of loading is one vector element per cycle. However, if we would need the elements with indices 0, 4, 8, ... there is a memory bank conflict and the speed of loading will be one element per two cycles. If we need the elements with indices 0, 8, 16, ... the loading speed will only be one element per three cycles. A remedy against memory bank conflicts would be to store the elements in some skewed order. Of course, the best way to do this depends very much on the particular problem at hand.

2.5 Numerical stability

Not much is known yet about stability, rounding errors and error propagation in parallel algorithms. In some cases, it appears that parallel processing leads to numerically inferior results, but this is not always the case. The following example shows how a parallel version of a simple algorithm actually yields better stability results than the serial version.

Consider the sum $S_N := \sum_{k=1}^{N} a_k$, where, for simplicity, we take $N = 2^n$. The serial algorithm for finding S_N reads as follows:

$$S_0 := 0, \quad S_k := S_{k-1} + a_k, \; k = 1,2,\ldots,N.$$

If the mantissa of the floating point numbers has s binary places, then the machine approximation S'_N of S_N satisfies the inequality:

$$|S_N - S'_N| < 2^{-s} aN(N+1),$$

where $a = \max_k |a_k|$.

A parallel version of this algorithm reads as follows:

$$S_{0i} := a_i, \; i=1,2,\ldots,N$$

$$S_{ki} := S_{k-1,2i-1} + S_{k-1,2i}, \; k=1,2,\ldots,n; \; i=1,2,\ldots,2^{n-k}$$

$$S_N := S_{n1}.$$

Here, estimation of the overall error yields:

$$|S_N - S'_N| < 2^{-s+1} aN\log_2 N,$$

which improves the serial $O(N^2)$-upperbound to $O(N\log N)$.

For this parallel algorithm, it is not difficult to compute, for a given number p of parallel processors, the speed-up S_p and efficiency E_p . For N=8 and p=2, 3 and 4, the results are given below.

p	S_p	E_p
4	7/3	7/12
3	7/4	7/12
2	7/4	7/8

3. APPLICATIONS

In the Bochum Bibliography [BRS], many fields of mathematics are mentioned in connection with parallel computing (cf. Table 2). Here, we shall discuss a number of important examples.

3.1 Solution of systems of linear equations

An excellent survey of parallel linear algebra algorithms and their complexity is given by Heller ([HE]). He treats the following subjects:

* linear systems
  - general dense matrices
  - triangular systems
  - tridiagonal systems
  - block tridiagonal and band systems
  - sparse matrices

* eigenvalues

Presently, much research is carried out on vector and parallel algorithms in numerical linear algebra. We mention a few groups:

* Van der Vorst (Delft, The Netherlands)
* Dekker, Hoffmann (Amsterdam, The Netherlands)
* Axelsson (Nijmegen, the Netherlands)
* Evans (Loughborough, UK)
* Young (Houston, Texas, USA)
* Dongarra (Argonne, Illinois, USA)
* Sameh (Urbana, Illinois, USA)

Here, we shall briefly describe an algorithm for solving linear dense systems of equations on a CYBER 205, as presented by Hoffmann ([HO]). First some notational conventions: lower case greek letters denote real scalars, lower case roman letters denote vectors and upper case letters stand for matrices. The j-th column of the matrix A is given as $a_{\circ j}$ and the i-th row as $a_{i \circ}$. The non-zero part of a column or row of a triangular matrix is indicated by writing a bar above the character which denotes the column or row. The order of a matrix is denoted by n. The algorithm used is the well-known Gaussian elimination process which is equivalent to the factorization of the

coefficient matrix A into A = LDU (apart from pivoting). Here, U is an upper triangular, L a lower triangular and D a diagonal matrix, whose elements are denoted by υ_{ij}, λ_{ij} and δ_i , respectively.
The elements α_{ij} of A satisfy the equations:

$$\alpha_{ij} = \sum_{k=1}^{\min(i,j)} \lambda_{ik}\,\delta_k\,\upsilon_{kj} \quad ; \qquad i,j \in \{1,2,\dots,n\}.$$

In the following algorithm the matrices L and U (and the information for D) are built up in the location of A which should be clear from the notation. The choice of the diagonal elements of L, D and U is left open yet.

```
for k = 1(1)n do
begin determine $q \in \{k,\dots,n\}$ : $|\alpha_{kq}| = \max_{k\le j\le n} |\alpha_{kj}|$   { search for maximum }

      $a_{\cdot q} \Leftrightarrow a_{\circ k}$                                  { interchange two length n columns }

      $\alpha_{kk} \leftarrow \left\{\begin{matrix}\upsilon_{kk}\\ \delta_k\\ \lambda_{kk}\end{matrix}\right\} = ?$                    { choose diagonal normalization, store }

      $\bar{a}_{\cdot k} \leftarrow \bar{\ell}_{\cdot k} = \bar{a}_{\circ k}/(\delta_k\upsilon_{kk})$         { calculate k-th col of $\bar{L}$ and store }

      for j = k+1(1)n do
      begin $\alpha_{kj} \leftarrow \upsilon_{kj} = \alpha_{kj}/(\delta_k\lambda_{kk})$        { next element in $u_{k.}$ and store }

            $a_{\cdot j} \leftarrow a_{\cdot j} - (\upsilon_{kj}\delta_k)\bar{\ell}_{\circ k}$            { update next column of A }

      end
end
```

When we study this algorithm, it may be clear that the choice $\lambda_{kk} = \upsilon_{kk} = \delta_k^{-1}$ gives optimal results, since it makes the calculations of $\bar{\ell}_{\circ k}$ and υ_{kj} trivial. The description of the algorithm then becomes much shorter, especially if the introduction of the names for L, D and U is eliminated:

```
for k = 1(1)n do
begin { maximum search and column interchange }
      for j = k+1(1)n do
            $a_{\cdot j} \leftarrow a_{\cdot j} - (\alpha_{kj}/\alpha_{kk})\bar{a}_{\cdot k}$
end
```

On a 1-pipe CYBER 205, Hoffmann obtained the following MFLOPS-speeds for various values of the order n of the matrix A:

n =	25	50	100	200	400
speed in MFLOPS	7.3	15.5	28.4	46.1	63.6

In [HO] many more experiments are reported and a comparison is made with standard routines for solving dense linear systems from the program libraries LINPACK, NAG and QQLIB. The above algorithm gives the smallest CPU-time.

Dongarra and Hewitt ([DH]) have implemented dense linear algebra algorithms on a CRAY X-MP-4 using multitasking and obtained a MFLOPS-speed of more than 700. They remark that a system of equations of order 1000 can now be factored and solved in less than one second!

3.2 Expressions: evaluating a polynomial

Given a real number x_0 and a polynomial

$P(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_n x^n$,

the well-known rule of Horner for computing $P(x_0)$ reads as follows:

$b_n := a_n$,

$b_j := a_j + x_0 b_{j+1}$, $j = n-1\ (-1)\ 0$,

$P(x_0) := b_0$.

If we would have 2 processors, able to work in parallel, we could write P(x) as:

$P(x) = a_0 + a_2 x^2 + a_4 x^4 + \dots$ (even powers)

$+ x(a_1 + a_3 x^2 + \dots)$ (odd powers).

The first and the second processor could then compute the even and the odd powers sum, respectively, as follows:

$$\begin{tikzcd}
\underline{\text{first processor}} & & \underline{\text{second processor}} \\
b_n := a_n & & b_{n-1} := a_{n-1} \\
b_{n-2} := a_{n-2} + x_0^2 b_n & & b_{n-3} := a_{n-3} + x_0^2 b_{n-1} \\
\vdots \arrow[r] & b_0 & \vdots \arrow[l] \\
& b_1 \arrow[d] & \\
& P(x_0) := b_0 + b_1 x_0 &
\end{tikzcd}$$

This process can be generalized for many processor systems.

3.3 ODEs

Let us consider the scalar ordinary differential equation

$y' = f(x,y), \quad x > 0, \quad y(0) = y_0.$

At first sight, it seems that there is little scope for parallelism in solving (scalar) ODEs, since the usual integration methods are essentially sequential. However, there exist parallel versions of serial predictor-corrector methods (cf. [ML]). We will describe one of them in some detail. Fix a mesh size h and let $x_n := (n-1)h$, $n=1,2,...$, and let y_n be an approximation to the solution y at x_n. Then one serial predictor-corrector scheme is the following:

$$y^p_{n+1} := y^c_n + (h/2)[3f^c_n - f^c_{n-1}],$$

$$y^c_{n+1} := y^c_n + (h/2)[f^p_{n+1} + f^c_n],$$

where y^p_n and y^c_n are predicted and corrected values of y_n, respectively, and f^c_n and f^p_n represent $f(x_n, y^c_n)$ and $f(x_n, y^p_n)$, respectively. The sequel of computations is shown in Figure 2a where the upper line represents the process for y^p_n and f^p_n and the lower line for y^c_n and f^c_n. The sequence of computations here is: --> y^p_{n+1} --> f^p_{n+1} --> y^c_{n+1} --> f^c_{n+1} --> and the computational front is indicated by the dotted line. This process is essentially sequential. For the alternative pair of predictor-corrector formulas

$$y^p_{n+1} := y^c_{n-1} + 2hf^p_n ,$$
$$y^c_n := y^c_{n-1} + (h/2)[f^p_n + f^c_{n-1}],$$

the computational process may be divided into two concurrent parts:

$$\longrightarrow y^p_{n+1} \longrightarrow f^p_{n+1} \longrightarrow$$
$$\longrightarrow y^c_n \longrightarrow f^c_n \longrightarrow$$

which can be processed in parallel, since the computational front is now skewed. See Figure 2b. This kind of parallelization has been extended to many ($\gg 2$) processors and to other algorithms like the Runge-Kutta method.

Figure 2a
A serial predictor-corrector scheme

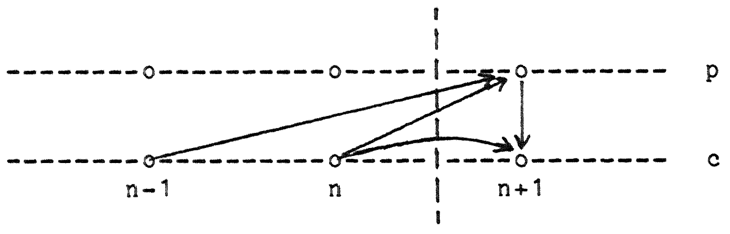
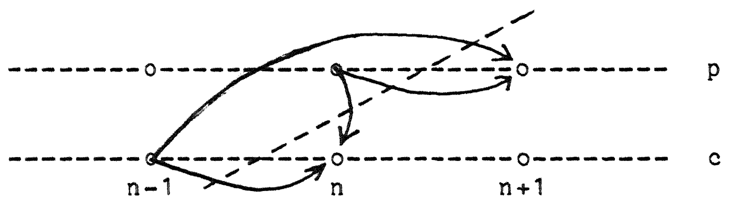


Figure 2b
A parallel predictor-corrector scheme



3.4 PDEs

There is an extensive survey paper by Ortega and Voigt ([OV]) which surveys the present status of numerical methods for partial differential equations on vector and parallel processors, together with a discussion of applications in fluid dynamics (Navier-Stokes equations, potential equation, reservoir simulation, numerical weather prediction), structural analysis, acoustic wave propagation, plasma physics, design of VLSI devices, molecular dynamics, etc, Generally spoken, discretization of place variables in PDEs may yield large systems of ODEs or of (non)linear equations. For vector and parallel machines, it seems that explicit methods for the former and iterative methods for the latter are more suitable than their respective counterparts: implicit and direct methods.

We mention here a few groups of people working on "vector and parallel" software:

Schonauer, 3D - problems;

Barkai, Brandt Hackbusch, Trottenberg, Stuben Hemker, Wesseling	multigrid methods;

Stelling, Wubs, Shallow water equations.

etc. etc.

3.5 FFT

The discrete Fourier Transform of a vector $a = (a_j)$, $j=0,1,\dots,N-1$, is given by

$$b_i = \sum_{j=0}^{N-1} \omega^{ij} a_j\ , \quad i = 0\ (1)\ N-1, \qquad \omega = \exp(2\pi i/N).$$

This is a complex matrix-vector multiplication requiring $O(N^2)$ operations. However, in the Fast Fourier Transform the matrix $\Omega = (\omega^{ij})$ is factorized in (assuming, for simplicity, $N = 2^{n+1}$) $\log N = n + 1$ very simple matrices, and the cumulative product is computed. The number of operations required now is $O(N \log N)$. The FFT algorithm can be described as follows:

for $r = 0\ (1)\ N-1$, $k = 0\ (1)\ n$, let

$$r := [r_0 r_1 \dots r_n] = \sum_{j=0}^{n} r_{n-j}\ 2^j\ , \quad r_i = 0 \text{ or } 1,$$

$$f(r,k) := [r_0 \dots r_{k-1}\ 0\ r_{k+1} \dots r_n],$$

$$h(r,k) := [r_0 \dots r_{k-1}\ 1\ r_{k+1} \dots r_n],$$

$$g(r,k) := [r_k r_{k-1} \dots r_0\ 0 \dots 0],$$

$$rev(r) := [r_n r_{n-1} \dots r_0] \quad (= g(r,n)).$$

```
FFT:
    $z_i := \omega^i$ (i = 0 (1) N-1 ),
    $c_i := a_i$ (i = 0 (1) N-1 ),
    for k:=0 step 1 until n do
        $c_i := c_{f(i,k)} + z_{g(i,k)} \cdot c_{h(i,k)}$    ( i = 0 (1) N-1 ),
    od,
    $b_i := c_{rev(i)}$    ( i = 0 (1) N-1 ).
```

It should be observed that either $f(r,k) = r$ and $h(r,k) = r + 2^{n-k}$ or $f(r,k) = r - 2^{n-k}$ and $h(r,k) = r$, so the movements for c within do-loop are well-structured. One should be careful in order to avoi memory bank conflicts. The book by Hockney and Jesshope ([HJ]) prov an excellent discussion of parallel aspects of the FFT and of other discrete transforms. Recent work on vectorizing the FFT ([FO], [KL7 [SWA], [WA]) indicates that the efficiency increases with the numbe transforms.

3.6 Number Theory

In the last decade, methods for factorization of positive intege have attracted much attention, partly, because of the discovery that the security of certain cryptographic systems depends on the difficu of the decomposition of integers into prime factors (cf. [RI]). Factorization methods like the quadratic sieve method and Lenstra's recent elliptic curve factorization method have certain features by which these algorithms may be very suitable for implementation on v and parallel computers. As an example, we mention one of the steps quadratic sieve methods, viz., to compute, modulo a given number N, the product of a large number M of integers with values between 0 a The scalar FORTRAN version of this step reads as follows:
(here, it is assumed that the square of N can still be represented an integer, and the integers to be multiplied are stored in the ar

```
      INTEGER N, M, PROD, A(M)
      PROD = 1
      DO 1 I = 1, M
         PROD = MOD( PROD * A(I), N )
    1 CONTINUE
```

A vector version of this step has been implemented on a 1-pipe CYBER 205 and looks as folows:

```
      INTEGER N, M, PROD, A(M), B(M/2), C(M/2), K, K2
      REAL    REVN
      REVN = 1.0 / N
      K = M
    1 K2 = K / 2
      B(1; K2) = A(1; K2) * A(K2+1; K2)
      C(1; K2) = B(1; K2) * REVN
      A(1; K2) = B(1; K2) - N * VINT(C(1;K2); K2)
      IF (2*K2 .EQ. K) THEN
         K = K2
      ELSE
         A(K2+1) = A(K)
         K = K2 + 1
      END IF
      IF (K .GT. 1) GOTO 1
      PROD = A(1)
```

The technique used here is a form of recursive doubling (cf. Section 4.1): two vectors which consist of the first and the second half part of A are multiplied (modulo N); the result is stored in the first half of A. This multiplication and storage step are repeatedly applied on A, where in each step the length of A is halved.

The REAL REVN (=1/N) is used because vector multiplication is much cheaper than vector division (on a 1-pipe CYBER 205, multiplication of two vectors of length N requires 52 + N clock cycles and division 80 + 25N/4).

Vector syntax is used. E.g., A(I;J) is the vector consisting of A(I), A(I+1), ..., A(I+J-1). The vector function VINT computes the integral parts of all the elements of its vector argument.

Some timings are given below.

N	scalar version	vector version
10,000	0.016 sec.	0.002 sec.
50,000		0.009 sec.(28 MFLOPS)

3.7 Numerical verification of the Riemann Hypothesis

The Riemann Hypothesis says that all the complex zeros of the Riemann zeta function $\zeta(s)$ have real part 1/2. This is a famous 125 year old statement of Riemann, which has resisted up till now the efforts of the best mathematicians to prove or disprove it. In order to verify the Riemann Hypothesis numerically, it is necessary to know, for many (hundreds of millions) values of t, the sign of the following real-valued function:

$$Z(t) = 2 \sum_{k=1}^{m} k^{-\frac{1}{2}} \cos[t.\log(k) - \theta(t)] + R_m(t),$$

where $m = \lfloor (t/2\pi)^{\frac{1}{2}} \rfloor$. The time needed to compute $\theta(t)$ and $R_m(t)$ is negligible compared with the total time needed for Z(t).

Three versions to compute Z(t) on a 1-pipe CYBER 205 have been developed : a half, a normal and a double precision version. With the first very fast version about 99% of the values of Z(t) could be computed with certainty. With the second version, about 99% of the remaining values could be determined with certainty. The double precision version was accurate enough to cover all the remaining values. The half precision version gained a speed of about 134 MOPS (Million Operations per Second), the normal precision version about 57 MOPS, so that the CYBER 205 turned out to be extremely suitable to solve this problem (reasons: pipelining, different precisions possible, possibility to link operations, e.g., constructs like |A(I)|*B + |C(I)| require 1 clock cycle in a vector call on the CYBER 205; this corresponds to a speed of 400 million operations per second!). Details may be found in [RWL], [WR] and [LRW].

4. VECTORIZATION AND PARALLELIZATION TECHNIQUES

Quite a number of techniques are known for generating parallel algorithms. One important distinction should be made in this respect: the number of available processors is limited or not. The latter case is interesting from a theoretical point of view, yielding results like: a non-singular n x n matrix can be inverted in $O(\log^2 n)$ time, using $O(n^4)$ processors ([CS]). The former case is more practical, since it is usually concerned with a particular parallel processor with a given number of processing elements, or a pipelined processor with fixed characteristics like clock cycle time, start up time, memory bank cycle time.

In this Section examples of recursive doubling and cyclic reduction techniques will be treated. Moreover, techniques for matrix-vector and matrix-matrix multiplication will be discussed. Finally, some results will be given of implementation and optimization on a CYBER 205 of a set of standard matrix-vector subroutines (so-called Extended BLAS).

A number of examples given in this Section are derived from a lecture by H.A. van der Vorst presented at the Colloquium 'Numerical Aspects of Vector and Parallel Processing' on the meeting of Sept. 27, 1985 which was held in Amsterdam.

4.1 Recursive doubling

Recursive doubling is a powerful method of generating parallel algorithms. The basic idea is to repeatedly separate each computation into two independent parts of equal complexity which can then be computed in parallel. For example,

$$\sum_{i=1}^{N} a_i = \left(\sum_{i=1}^{n-1} a_i \right) + \left(\sum_{i=n}^{N} a_i \right), \quad n = \lceil N/2 \rceil ,$$

and by further application of this splitting, the sum can be computed in $\lceil \log N \rceil$ steps using N/2 processors. This may be implemented on a pipelined processor as follows (here, like in Section 3.6, we use vector-syntax for the CYBER 205):

```
WHILE N > 1 DO
   M = ( N+1 ) / 2
   A( 1; M ) = A( 1; M ) + A ( M+1; N-M )
   N = M
OD
```

If an addition of two vectors of length N on a pipelined computer takes a + bN clock cycles (a is the start-up time), and if scalar addition takes c clock cycles, then the times needed for the sequential algorithm and for the parallel version are approximately cN and $a.\log_2 N + bN$ cycles. Comparing these two times, we can compute the approximate turning point for which the parallel version becomes faster than the sequential. Some examples are given in Table 5.

Table 5
Turning point from which parallel addition algorithm runs faster than sequential version

a + bN: number of clock cycles needed for vector addition (length N)
c : number of clock cycles needed to add two scalars

Computer	a	b	c	value of N for which $a.\log_2 N + bN \approx cN$
CRAY-1	30	3	6	59
p-pipe CYBER 205	51	1/p	5	81 for p=1, 69 for p=2, 65 for p=4
CRAY X-MP	30	1		
FUJITSU VP-100	30	2		

Recursive doubling is applicable in a large number of instances. Table 6 is taken from [ST1]. Theoretically, most of the recurrences mentioned there can be computed in O(log N) time if O(N) processors are available. However, actual implementation (like the one given above) is needed to show the real benefit obtainable with this technique.

Table 6
Functions suitable for recursive doubling

Function	Description
$X_i = X_{i-1} + a_i$	Sum the elements of a vector
$X_i = X_{i-1} * a_i$	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
$X_i = a_i X_{i-1} + b_i X_{i-2} + \ldots$	Any order linear recurrence, homogeneous or inhomogeneous
$X_i = (a_i X_{i-1} + b_i)/(c_i X_{i-1} + d_i)$	First order rational fraction recurrence
$X_i = a_i + b_i /X_{i-1}$	Special case of first order rational fraction
$X_i = \mathrm{sqrt}(X_{i-1}^2 + a_i^2)$	Vector norm

Another example of recursive doubling occurs in the solution of bidiagonal linear systems Ax = b, where

$$A = \begin{bmatrix} 1 & & & & \\ a_2 & 1 & & & \\ & a_3 & 1 & & \\ & & \ddots & \ddots & \\ & & & a_N & 1 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_N \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_N \end{bmatrix}$$

The standard solution method is:

$x_1 := b_1$,

$x_i := b_i - a_i * x_{i-1}$, $i = 2,3,\ldots,N$.

Some (scalar) improvement can be obtained by loop-unrolling (cf. [VK, section 3.8]). A recursive doubling technique for solving the bidiagonal linear system can be described as follows. Left-multiplication by the matrix $-A + 2I$ yields the equation $A'x = b'$ where

$$A' = \begin{bmatrix} 1 & & & & & & \\ 0 & 1 & & & & & \\ a'_3 & 0 & 1 & & & & \\ & a'_4 & 0 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & a'_N & 0 & 1 \end{bmatrix},$$

with obvious values of a'_i $(i=3,4,\dots,N)$ and b'_i $(i=1,2,\dots,N)$.
Next, we left-multiply with the matrix $-A' + 2I$ to obtain the equation $A''x=b''$, where

$$A'' = \begin{bmatrix} 1 & & & & & & & \\ 0 & 1 & & & & & & \\ 0 & 0 & 1 & & & & & \\ 0 & 0 & 0 & 1 & & & & \\ a''_5 & 0 & 0 & 0 & 1 & & & \\ & a''_6 & 0 & 0 & 0 & 1 & & \\ & & a''_7 & 0 & 0 & 0 & 1 & \\ & & & \ddots & \ddots & \ddots & \ddots & \ddots \end{bmatrix}$$

Repeating this process at most $\lceil \log_2 N \rceil$ steps eliminates all the unknowns and yields the solution in the vector b. On a parallel processor with N processing elements, this would yield the solution in about $\log_2 N$ time-steps. However, the total number of operations in this parallel algorithm is much larger than that in the serial version, and, even though the operations now are all vector operations, the actual performance on a vector computer (like the CRAY-1) is worse than the loop-unrolled version. Speeds of 2 - 5 MFLOPS are reported for the parallel version implemented on pipelined computers. For the loop-unrolled scalar version [VK] report speeds of 8.0 MFLOPS on a CRAY-1 and 7.8 on a CYBER 205 (1- and 2-pipe) with N=5000. Very recently, J. Schlichting of CDC and the CWI managed to reach a speed of 12 MFLOPS with N=3000.

4.2 Cyclic reduction

It seems that the technique of cyclic reduction was first used to solve tridiagonal equations by Hockney in 1965 (in collaboration with Golub; cf. [HJ, p.286]). We illustrate this technique with the bidiagonal equation of the previous section. Eliminating x_1 from the second equation, x_3 from the fourth, and so on, we obtain the system

$$\begin{bmatrix} 1 & & & \\ a'_4 & 1 & & \\ & a'_6 & 1 & \\ & & \ddots & \ddots \end{bmatrix} \begin{bmatrix} x_2 \\ x_4 \\ x_6 \\ \vdots \end{bmatrix} = \begin{bmatrix} b'_2 \\ b'_4 \\ b'_6 \\ \vdots \end{bmatrix},$$

where $a'_{2i} = - a_{2i} * a_{2i-1}$, $b'_{2i} = b_{2i} - a_{2i} * x_{2i-1}$.
This process can be repeated, if suitable, and on a parallel processor with N processors, this algorithm needs about $\log_2 N$ steps.
In [VK] experiments with this algorithm on pipelined computers like the CRAY-1 and the CYBER 205 are reported. On the CRAY-1 speeds close to 12 MFLOPS (for N=5000) were obtained, and 9 MFLOPS on a 2-pipe CYBER 205.

4.3 Matrix-vector and matrix-matrix multiplication

The usual method of computing the matrix-vector product y := Ax is by taking innerproducts of rows of A with x:

$$y_i := \sum_{j=1}^{N} a_{ij} x_j, \quad i = 1,2,\ldots,N.$$

Implementing this method on a CRAY-1, a speed of 53 MFLOPS is obtained (N=300) For N=200 and 201, the speeds are 37 and 49 MFLOPS, respectively (this is due to memory bank conflicts). On the CYBER 205, however, there is a so-called stride problem which means that elements have to be loaded from memory which are not stored in contiguous locations (arrays in Fortran are stored column-wise). The speed obtained on a 1-pipe CYBER 205 is 5.7 MFLOPS. However, by reordering the computations column-wise:

$$y := x_1 a_{\circ 1} + x_2 a_{\bullet 2} + \dots + x_N a_{\circ N},$$

where $a_{\circ i}$ is the i-th column of the matrix A, the speed obtained on a 1-pipe CYBER 205 is 66 MFLOPS and on a 2-pipe CYBER 205 106 MFLOPS!

If, however, we would have to compute $y := A' x$, the inner product version should be used on the CYBER 205. If the matrix A is symmetric and if only the upper (or lower) part is available in storage, a combination of the two methods mentioned above should be used on the CYBER 205.

For band matrices, the picture is quite different (cf. [MRK]). Consider, for example, the tridiagonal band-matrix product

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \cdot \\ \cdot \\ \cdot \\ y_N \end{bmatrix} := \begin{bmatrix} a_{11} & a_{12} & & & & \\ a_{21} & a_{22} & a_{23} & & & \\ & a_{32} & a_{33} & a_{34} & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & \cdot \\ & & & & \cdot & \cdot \quad \cdot \\ & & & & a_{N,N-1} & a_{NN} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ \cdot \\ x_N \end{bmatrix} .$$

In order to save space, band matrices are usually stored in rectangular arrays such that the non-zero diagonals are stored in rows or column of the array. The above multiplication can be executed very efficiently on a vector computer by expressing the product as a sum of three vector-vector multiplications:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \cdot \\ \cdot \\ \cdot \\ y_{N-1} \\ y_N \end{bmatrix} := \begin{bmatrix} a_{21} \\ a_{32} \\ \cdot \\ \cdot \\ \cdot \\ a_{N-1,N-2} \\ a_{N,N-1} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_{N-2} \\ x_{N-1} \end{bmatrix} + \begin{bmatrix} a_{11} \\ a_{22} \\ a_{33} \\ \cdot \\ \cdot \\ \cdot \\ a_{N-1,N-1} \\ a_{NN} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ \cdot \\ x_{N-1} \\ x_N \end{bmatrix} + \begin{bmatrix} a_{12} \\ a_{23} \\ a_{34} \\ \cdot \\ \cdot \\ \cdot \\ a_{N-1,N} \end{bmatrix} * \begin{bmatrix} x_2 \\ x_3 \\ x_4 \\ \cdot \\ \cdot \\ \cdot \\ x_N \end{bmatrix}$$

Hence, the time for this expression is essentially 5N clock cycles (3N for the multiplication and 2N for the addition of the vectors), if we neglect start-up times and if the diagonals of the matrix are stored column-wise.
If there are diagonals with constant value, the number of clock cycles can be decreased by N for each constant diagonal (except the last) by using chaining or linked triads.

The ideas described here can be extended to matrix-matrix multiplication. For example, in [DGK] six possible permutations of the three loop indices in matrix-matrix multiplication programming are described.
This gives rise to six possible implementations of matrix multiplication. Each implementation has quite different memory access patterns, which will have an important impact on the performance of the algorithms on a given vector or parallel processor. Also cf. [MRK].

4.4 Extended BLAS

Recently, Dongarra et al. ([DDHH]) have proposed a standard set of routine for matrix-vector multiplication, rank-1 and rank-2 updates, and inversion of triangular systems of equations, called the set of Extended BLAS (Basic Linear Algebra Subroutines). This extends the existing set of BLAS, which are standard routines for operations on vectors ([LHKK]).
The extended BLAS routines will become available in Fortran 77. Besides that the proposers hope that efficient implementations will become available on a

wide range of computer architectures. At the CWI an implementation of the extended BLAS on the CYBER 205 is being developed. Here we will give a short description of the contents of the set of extended BLAS, and present some timings obtained on the CYBER 205.

Three types of basic operations (MV, R1/R2 and TR) are proposed:

a) Matrix vector (MV) products of the form

$$y := \alpha Ax + y, \text{ and } \quad y := \alpha A'x + y$$

where α is a scalar, x and y are vectors and A is a matrix, and

$$x := Tx \text{ and } \quad x := T'x,$$

where T is an upper or lower triangular matrix.

b) Rank-one (R1) and rank-two (R2) updates of the form

$$A := \alpha xy' + A \text{ and } \quad A := \alpha xy' + \alpha yx' + A.$$

c) Solution of triangular equations (IV) of the form

$$x := T^{-1}x,$$

where T is an upper or lower non-singular triangular matrix.

The subroutines have a name which consists of five characters. The first character is an S (indicating real versions; other possibilities are C for complex, D for double precision). Characters two and three denote the kind of matrix involved and the final two character denote the type of operation. There are sixteen subroutines, marked by an * below.

	operation			
type of matrix	MV	R1	R2	IV
GE general matrix	*	*		
GB general band	*			
SY symmetric	*	*	*	
SP symmetric packed	*	*	*	
SB symmetric band	*			
TR triangular	*			*
TP triangular packed	*			*
TB triangular band	*			*

The following table gives timings in milliseconds of the 16 subroutines. The timings of the packed matrix versions are the same as those of the unpacked versions, and are omitted. Two timings per routine are given: that of the Fortran 77 version and that of the 1-pipe CYBER 205 optimized version. The final column gives MFLOPS-speeds of the optimized versions. The order of the matrices used was 500 for full matrices and 30000 for band matrices. In the case of band matrices, the number of non-zero diagonals was: 2 upper and 3 lower in the general case, 2 in the symmetric case and 5 in the triangular case.

Subroutine	Fortran 77	CYBER 205 Optimized	MFLOPS-speed of Opt.
SGEMV	7	7	75
SGBMV	107	15	26
SSYMV	58	8	63
SSBMV	184	12	27
STRMV	4	4	63
STBMV	98	14	24
SGER1	7	7	75
SSYR1	4	4	60
SSYR2	8.5	7.8	64
STRIV		6	62
STBIV		230	2

The general matrix routines all run with a speed which comes quite close to the optimal speed of 100 MFLOPS, obtainable for general matrix multiplication. In the band matrix case, the speeds are negatively influenced by the row-wise storage convention for the diagonals: this requires gathering of the diagonal elements. If column-wise storage would be allowed, then the MFLOPS-speeds could be multiplied by a factor of at least 1.6, which would bring these speeds reasonably close to the optimum of 50 MFLOPS, obtainable for band matrix operations.

5. References

[BO] A.P.W. Bohm, Dataflow computation, CWI Tract 6, Centre for Mathematics and Computer Science, Amsterdam, 1983.

[BRS] U. Bernutat-Buchmann, D. Rudolph and K.-H. Schlosser, Parallel Computing I, Eine Bibliographie, Bochumer Schriften zur Parallelen Datenverarbeitung 1, Computing Centre, Bochum, second ed., Sept. 1983.

[CS] L. Csanky, Fast parallel matrix inversion algorithms, SIAM J. Comput. 5(1976)618-623.

[DDHH] J.J. Dongarra, J. Du Croz, S. Hammarling and R.J. Hanson, A proposal for an extended set of Fortran Basic Linear Algebra Subprograms, Techn. Memo. No. 41, Argonne National Lab., Argonne, Ill. 60439, Dec. 1984.

[DH] J.J. Dongarra and T. Hewitt, Implementing dense linear algebra algorithms using multitasking on the CRAY X-MP-4 (or approaching the gigaflop), Techn. Memo No. 55, Argonne National Laboratory, Argonne, Aug. 1985.

[DGK] J.J. Dongarra, F.G. Gustavson and A. Karp, Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, SIAM Review 26(1984)91-112.

[DR] I.S. Duff and J.K. Reid(eds.), Vector and Parallel Processors in Computational Science, Proceedings of the second International Conference in VPPCS, Oxford August 1984, North-Holland, Amsterdam, 1985.

[EM] A.H.L. Emmen(ed.), Supercomputer Applications, Proc. of the International Supercomputer Applications Symposium, Amsterdam Nov. 1984, North-Holland, Amsterdam, 1985.

[FL] M.J. Flynn, Very high speed computing systems, Proc. IEEE 14(1966) 1901-1909.

[FO] B. Fornberg, A Vector Implementation of the Fast fourier Transform Algorithm, Math. Comp. 36(1981)189-191.

[HE] D. Heller, A survey of parallel algorithms in numerical linear algebra, SIAM Review 20(1978)740-777.

[HJ] R.W. Hockney and C.R. Jesshope, Parallel Computers, Adam Hilger Ltd, Bristol, 1981.

[HO] W. Hoffmann, Gaussian elimination algorithms on a vector computer, Rept. 85-10, Dept. of Math., Univ. of Amsterdam, June 1985.

[KL79] D.G. Korn and J.L. Lambiotte, Jr., Computing the Fast Fourier Transform on a Vector Computer, Math. Comp. 33(1979)977-992.

[KL83] G.A.P. Kindervater and J.K. Lenstra, Parallel algorithms in combinatorial optimization: an annotated bibliography, Report BW 189 /83, Centre for Mathematics and Computer Science, Aug. 1983.

[LHKK] C. Lawson, R. Hanson, D. Kincaid and F. Krogh, Basic Linear Algebra Subprograms for Fortran usage, ACM Trans. on Math. Software 5(1979) 308-323.

[LRW] J. van de Lune, H.J.J. te Riele and D.T. Winter, On the zeros of the Riemann zeta function in the critical strip. IV, Report NM-8515, Centre for Mathematics and Computer Science, Amsterdam, June 1985 (to appear in Math. Comp.).

[MI] W.L. Miranker, A survey of parallelism in numerical analysis, SIAM Review 13(1971)524-547.

[ML] W.L. Miranker and W.M. Liniger, Parallel methods for the numerical integration of ODEs, Math. Comp. 21(1967)303-320.

[MRK] Matrix multiplication by diagonals on a vector/parallel processor, Information Process. Letters 5(1976)41-45.

[OV] J.M. Ortega and R.G. Voigt, Solution of partial differential equations on vector and parallel computers, ICASE Rept. No. 85-1, NASA Langley Research Center, Hampton, Virginia 23665, Jan. 1985.

N.K. Madsen, G.H. Rodrigue and J.I. Karush,

[RI] H. Riesel, Prime numbers and computer factorization, Birkhauser 1985.

[RMK] G.H. Rodrigue, N.K. Madsen and J.I. Karush, Odd-even reduction for banded linear equations, JACM 26(1979)72-81.

[RWL] H.J.J. te Riele, D.T. Winter and J. van de Lune, Numerical verification of the Riemann hypothesis on the CYBER 205, pp. 33-38 in [EM].

[SA] A.H. Sameh, Numerical parallel algorithms - A survey, pp. 207-228 in: D.J. Kuck et al.(eds.), High speed computer and algorithm organization, Acad. Press, 1977.

[SCHE] U. Schendel, Introduction to numerical methods for parallel computers (translated from German), Ellis Horwood Ltd, Chichester, 1984.

[SCHW] J. Schwartz, A Taxonomic Table of Parallel Computers, based on 55 designs, Ultra computer Note No. 69, Courant Institute, New York Univ., 1983.

[SH] J.E. Shore, Second thoughts on parallel processing, Comput. Elect. Eng. 1(1973)95-109.

[ST1] H.S. Stone, An efficient parallel algorithm for the solution of a tridiagonal linear system of equations, JACM 20(1973)27-38.

[ST2] H.S. Stone, Problems of parallel computation, pp. 1-16 in [TR].

[SWA] P.N. Swartztrauber, FFT algorithms for vector computers, Parallel Computing 1(1984)45-63.

[SWE] R.A. Sweet, A cyclic reduction algorithm for solving block-tridiagonal systems of arbitrary dimension, SIAM J. Numer. Anal. 14(1977)706-719.

[TR] J.F. Traub(ed.), Complexity of sequential and parallel numerical algorithms, Acad. Press, 1973.

[VK] H.A. van der Vorst and J.M. van Kats, The performance of some linear algebra algorithms in Fortran on CRAY-1 and CYBER 205 supercomputers, Techn. Rept. TR-17, ACCU, Utrecht, 1984.

[VL] J. van Leeuwen, Parallel computers and algorithms, Rept. RUU-CS-83-13, RU Utrecht, Sept. 1983.

[WA] H.H. Wang, On vectorizing the Fast fourier Transform, BIT 20(1980)233-243.

[WR] D.T. Winter and H.J.J. te Riele, Optimization of a program for the verification of the Riemann hypothesis, Supercomputer, 5(1985)29-32.

[ZA] V. Zakharov, Parallelism and array processing, IEEE Trans. Comp. C-33 (1984)45-78.