Chapter 5

Vectorisation and Parallelisation of Algorithms

Contributed by Walter Hoffmann, Walter Lioen,
Margreet Louter, Nicolay Petkov, Herman te Riele,
Ben Sommeijer, Henk van der Vorst, and Dik Winter

5.1 General

In this chapter vectorisation and parallelisation of various algorithms will be discussed. In Section 5.1, general concepts concerning performance and data organisation are treated, followed by the introduction by examples of the widely-used vectorisation/parallelisation techniques *recursive doubling, cyclic reduction, divide-and-conquer, and domain decomposition.* In Section 5.2 we will discuss seven representative classes of numerical algorithms and in Section 5.3 two representative classes of non-numerical algorithms. These algorithms play a role in various application areas. Section 5.4 concludes this chapter with the description of some systolic algorithms.

5.1.1 Definitions

The *speed* of vector and parallel computers is often expressed in *Mflop/s*: the number of Million floating point operations per second. If a vector or parallel processor has a clock cycle time of c nanoseconds (e.g. $c = 6$ for the Cray Y/MP) and if one result per clock cycle is produced (which is usually the case for the operations $+$, $-$, and $*$), then the speed is $1000/c$ Mflop/s. However, such a

peak performance is difficult to reach in practice because of all kinds of overhead involved.

When we compare the execution times of a serial and a vector/parallel computer, the *speed-up* S_p is defined as the quotient T_1/T_p, where T_1 is the serial, and T_p the vector/parallel execution time. On a parallel computer with p independent processors we would hope that $T_p \sim T_1/p$, i.e., we aim for a linear speed-up $S_p = \mathcal{O}(p)$. This is not always the case, since it may be impossible to use all p processors effectively. In the following table we give typical speed-ups for a parallel machine with p processors [38]. Here, k is a machine-dependent constant, independent of p.

S_p	algorithms with this speed-up
kp	matrix computations, mesh calculations
$kp/\log p$	sorting, tridiagonal systems, linear recurrence relations, polynomial evaluation
$k \log p$	searching
k	certain non-linear recurrence relations

The *efficiency* 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 by the parallelisation of the algorithm, the smaller E_p.

On various architectures, the arithmetic operations may be executed in three different modes, viz., *scalar*, *parallel*, and *vector-* or *pipelined*. Consider, for example, the addition of two vectors of floating-point $z_i = x_i + y_i$, $(i = 1, \ldots, n)$. The operation of adding a pair of floating-point numbers x_i, y_i may be divided into four sub-operations (this is a somewhat simplified model), viz.,

- compare the exponents,
- shift,
- add mantissae, and
- normalise.

Suppose that each sub-operation takes one clock cycle.
In *scalar* mode the additions of the components of the vectors x and y are done one after the other, so that the computation of the vector z takes $4n$ clock cycles.
In *parallel* mode, if p processors are available, p components of the vectors can be added at the same time, so that the total time for z is about $4n/p$ clock cycles (assuming $n \geq p$).
The way a *vector* computer operates can be compared with the principle of an assembly-line: the four *sub-operations* can be carried out in parallel. This means that, after a start-up time of three clock cycles, during each subsequent

clock cycle the four different sub-operations are carried out concurrently on four different components of the vector z. The following table gives, for the four different sub-operations, the components of the vector z which are treated during clock cycles $1, 2, \ldots$. So after clock cycle 4, z_1 is ready, after clock cycle 5, z_2, and so on. It follows that the total time to compute z in vector mode is $n + 3$ clock cycles.

clock cycle	compare	shift	add	normalise	just finished
1	z_1				
2	z_2	z_1			
3	z_3	z_2	z_1		
4	z_4	z_3	z_2	z_1	
5	z_5	z_4	z_3	z_2	z_1
6	z_6	z_5	z_4	z_3	z_2
.	.	.	.	.	.

Comparing the three different modes, we see that in vector mode a speed-up of about a factor of 4 can be reached compared with scalar mode, and in parallel mode a speed-up of a factor p compared with scalar mode.

Obviously, if we can express our algorithm in terms of long vectors, or in terms of independent parts which can be executed concurrently on different processors, we can obtain a considerable time gain in the execution of our job. *Vectorisation* and *parallelisation* are the terms used to describe these activities.

5.1.2 Performance, Amdahl's law

Vector and parallel computers generally can show a much higher performance than scalar computers for algorithms that are sufficiently well vectorisable and/or parallelisable. Various techniques for vectorisation and parallelisation are discussed in Section 5.1.4. Here we shall discuss some performance issues. Usually, not all our computations can be vectorised and/or parallelised. Suppose that 75% of the operations in our job are vectorisable or parallelisable and that 25% is not, so that the latter part of the job has to be executed in scalar mode. Then, consequently, the speed-up factor we can obtain by vectorisation/parallelisation is bounded above by 4, no matter how well we are able to vectorise/parallelise. Amdahl's law quantifies this phenomenon as follows [41].

Suppose a certain algorithm requires N serial flops (floating-point operations), and suppose that a positive fraction α of them, i.e., αN flops can be executed with a vector speed of v Mflop/s on a given vector computer. The remainder is done with scalar speed, say s Mflop/s, where s is much smaller than v. For the total execution time T we get

$$T = \frac{\alpha N}{v} + \frac{(1-\alpha)N}{s} = N\left(\frac{\alpha}{v} + \frac{1-\alpha}{s}\right) \quad \mu\text{sec.}$$

This time is always larger than $N(1-\alpha)/s$ μsec, no matter how large we manage to choose v. For the speed R of the algorithm (the number of flops divided by the time in μsec.) we obtain

$$R = 1 \Big/ \left(\frac{\alpha}{v} + \frac{1-\alpha}{s} \right) \quad \text{Mflop/s.}$$

This is *Amdahl's law* in simplified form. For the Cray-1, we have $v = 150$ and $s = 5$. For $\alpha = 0$ we get $R = 5$. In order to double this speed we need $\alpha = 15/29$, and to get a speed-up of 10 we need $\alpha = 27/29$. In general we can conclude from this that in order to get a good performance the fraction of the computational work which is vectorisable should be close to 1.

For a parallel computer suppose that the fraction α of the computation is parallelisable and that the fraction $1-\alpha$ is "essentially serial", i.e., not amenable to any speed-up on a parallel machine. Then we would expect

$$T_p = (1-\alpha)T_1 + \alpha T_1/p$$

so for the overall speed-up we find

$$S_p = \frac{1}{1-\alpha+\alpha/p} \leq \frac{1}{1-\alpha},$$

i.e., the speed-up is *bounded*, and *not* a linear function of p. This could be used as an argument against parallelisation. However, what it shows is that the speed-up is bounded as we increase the number of processors for a *fixed* problem. In practice, it is more likely that we want to solve larger problems as the number of processors in our computer increases, because the wish to solve large problems is a primary motivation for building large parallel machines. Let N be a measure for the problem size. For many problems it is reasonable to assume that the essentially scalar fraction decreases as the inverse of the problem size, so that $1-\alpha \leq k/N$ for some constant k. Suppose furthermore that N increases at least linearly with p, with the same constant k, i.e., $N \geq kp$. Then it follows that $(1-\alpha)p \leq 1$ so that

$$S_p = \frac{p}{(1-\alpha)p+\alpha} \geq \frac{p}{1+\alpha} \geq \frac{p}{2}.$$

Thus we get a linear speed-up, with efficiency $E_p \geq \frac{1}{2}$.

5.1.3 Data organisation

In algorithms for parallel processing, the organisation and the dynamic arrangement of the data can play a decisive role. Let us consider an extremely simplified example of an MIMD computer 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 3×3 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}

So, P_1 has access to a_{11}, a_{21}, and a_{31}, and P_2 and P_3 to the second and third column 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 the main diagonal 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 by [32]

A related more realistic problem, a so-called memory bank conflict, may rise due to the fact that memories in large vector computers are split up in banks, which have a so-called bank latency time. This means that when an element is loaded from a memory bank, it is not possible to load another element from the same bank in the next few clock cycles. For example, suppose we have an 8-bank machine and a vector is stored in memory as follows: the elements with index $8m + n$, $0 \leq n \leq 7$, are stored in bank number n. Suppose the bank latency time is three clock cycles. Then, if we need the elements with indices $0, 1, 2, \ldots$ there will be no conflict and the speed of loading is one vector element per clock cycle. However, if we need the elements with indices $0, 4, 8, \ldots$ there will be a bank conflict and the speed of loading will be two elements per three cycles. If we need the elements with indices $0, 8, 16, \ldots$ the loading speed will only be one element per three clock cycles. A remedy against such conflicts would be to store the elements in some skewed order. Of course, the optimal storage strategy depends very much on the particular problem at hand.

5.1.4 Techniques: recursive doubling, cyclic reduction, divide-and-conquer, domain decomposition

Quite a number of techniques are known for generating vector and parallel algorithms. One important distinction should be made in this respect: the number of available processors in a parallel computer is limited or not. The latter assumption occurs in theoretical studies which yield results like: a non-singular $n \times n$ matrix can be inverted in $\mathcal{O}(\log^2 n)$ time, by using $\mathcal{O}(n^4)$ processors [6]. The former case is more practical, since it is usually concerned with a particular processor with a given number of parallel processing elements, or a pipelined processor with fixed characteristics like clock cycle time, start-up time, memory

bank cycle time. In this section we discuss the techniques of recursive doubling, cyclic reduction, divide-and-conquer for realistic computer architectures.

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

$$\sum_{i=1}^{N} a_i = \sum_{i=1}^{n} a_i + \sum_{i=n+1}^{N} a_i, \quad n = \lfloor N/2 \rfloor,$$

and by further application of this splitting, the sum can be computed in $\lceil \log N \rceil$ steps using $\lceil N/2 \rceil$ processors. For a vector computer, this may be implemented in Fortran 90 as follows (`A(N1:N2)` is the vector consisting of the N2−N1+1 array elements `A(N1), A(N1+1), ... , A(N2)`).

```
WHILE n > 1 DO
   m1 = n/2
   m2 = (n + 1)/2
   a(1:m1) = a(1:m1) + a(m2+1:n)
   n = m2
END DO
```

If an addition of two vectors of length N on a vector computer takes $a + bN$ clock cycles (a is the start-up time), and if scalar addition takes s clock cycles, then the times for the sequential algorithm and for the parallel version are approximately sN and $a \log_2 N + bN$ cycles, respectively. Comparing these two times we can compute the approximate turning point for which the parallel version becomes faster than the sequential one.

Recursive doubling is applicable to a large number of instances on shared-memory systems. The table below is taken from [38]. Theoretically, most of the recurrences mentioned there can be computed in $\mathcal{O}(\log N)$ time if $\mathcal{O}(N)$ processors are available. However, actual implementation is needed to show the real gain obtainable with this technique.

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
$X_i = (a_i X_{i-1} + b_i)/(c_i X_{i-1} + d_i)$	first order rational fraction recurrence
$X_i = a_i + b_i / X_{i-1}$	special case of the above recursion
$X_i = \sqrt{(X_{i-1}^2 + a_i^2)}$	vector norm

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

$$A = \begin{pmatrix} 1 & & & & & & \\ a_2 & 1 & & & & & \\ & a_3 & 1 & & & & \\ & & \cdot & \cdot & & & \\ & & & \cdot & \cdot & & \\ & & & & a_{N-1} & 1 & \\ & & & & & a_N & 1 \end{pmatrix}, \; x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ x_{N-1} \\ x_N \end{pmatrix}, \; b = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \cdot \\ \cdot \\ b_{N-1} \\ b_N \end{pmatrix}.$$

This system arises in the solution of tridiagonal systems of equations (cf. Section 5.2.1). The standard solution method is given by the linear recursion:

$$\begin{aligned} &x_1 = b_1, \\ &x_i = b_i - a_i * x_{i-1}, \quad i = 2, 3, \ldots, N. \end{aligned}$$

Some (scalar) improvement can be obtained by loop-unrolling, but this recursion *as it stands* clearly is not vectorisable neither parallelisable.

A recursive doubling technique for solving this system reads as follows. Left multiplication by the matrix $-A + 2I$ yields the equation $A'x = b'$ where

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

with obvious values of a'_i $(i = 3, 4, \ldots, N)$ and b'_i $(i = 1, 2, \ldots, N)$. In fact this step amounts to replacing x_{i-1} in the right-hand-side of the expression for x_i by the same expression for x_{i-1} (for $i = 2, 3, \ldots, N$), so that now x_i is expressed in x_{i-2}. With this recursion we can, starting from x_1, generate the values of $x_3, x_5, \ldots$, and starting from x_2 $(= b_2 - a_2 b_1)$ we can *independently*, hence in parallel, generate $x_4, x_6, \ldots$. So this can be done on a two-processor machine. For a four-processor machine, we again left-multiply with the matrix $-A' + 2I$

to obtain the equation $A''x = b''$, where

$$A'' = \begin{pmatrix} 1 & & & & & & & \\ 0 & 1 & & & & & & \\ 0 & 0 & 1 & & & & & \\ 0 & 0 & 0 & 1 & & & & \\ a''_5 & 0 & 0 & 0 & 1 & & & \\ & a''_6 & 0 & 0 & 0 & 1 & & \\ & & a''_7 & 0 & 0 & 0 & 1 & \\ & & & \cdot & \cdot & \cdot & \cdot & \cdot \\ & & & & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

and now we can generate the four sequences $x_i, x_{4+i}, x_{8+i}, \ldots$, for $i = 1, 2, 3, 4$ in parallel. Repeating this process at most $\lceil \log_2 N \rceil$ times 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 still worse than the loop-unrolled version. Only for special parallel computers this technique may be profitable.

The idea of *cyclic reduction* is closely related to recursive doubling, but this technique is more amenable to vectorisation. For simplicity, assume that N is even. The above substitution trick is applied now only to the variables with *odd* index. This yields, for $i = 1, 2, \ldots, N/2$,

$$\begin{aligned} x_{2i} &= b_{2i} - a_{2i}x_{2i-1} \\ &= b_{2i} - a_{2i}(b_{2i-1} - a_{2i-1}x_{2i-2}) \\ &= \tilde{b}_{2i} - \tilde{a}_{2i}x_{2i-2} \end{aligned}$$

and we now have a recursion for the variables with even index and the original recursion has been halved at the expense of some substitution work. Computing the coefficients $\tilde{b}_{2i}$ and $\tilde{a}_{2i}$ requires the vector operations

$$\begin{matrix} \tilde{b}_j = b_j - a_j b_{j-1} \\ \tilde{a}_j = -a_j a_{j-1} \end{matrix} \quad , j = 2, 4, \ldots, N.$$

Next, the recursion of length $N/2$ has to be executed on the variables with even index. This is halve the original scalar work. Finally, the variables with *odd* index can be computed from those with even index with the vector operation

$$x_i = b_i - a_i x_{i-1}, \quad i = 3, 5, \ldots, N-1.$$

This step of halving the scalar work can be repeated if more than two processors are available, each time reducing the number of scalar operations at the expense of increasing the number of vector operations. Van der Vorst [41] has analysed

these (and other) techniques for solving bidiagonal systems on various vector computers.

We will illustrate *divide-and-conquer* techniques by an algorithm of Cuppen [7] for computing eigenvalues of a symmetric tridiagonal matrix T which we assume to be $2n \times 2n$. Write it as a sum

$$T = \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} + \rho x x^{\mathrm{T}}$$

of a block diagonal matrix with tridiagonal blocks T_1 and T_2, and a rank-1 matrix $\rho x x^{\mathrm{T}}$ which is non-zero only in the four entries at the intersection of rows and columns n and $n+1$. Now we can compute the eigendecompositions $T_1 = Q_1 \Lambda_1 Q_1^{\mathrm{T}}$ and $T_2 = Q_2 \Lambda_2 Q_2^{\mathrm{T}}$ in parallel on two processors. This yields the partial eigendecomposition for T

$$\begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} \cdot \left(\begin{bmatrix} \Lambda_1 & 0 \\ 0 & \Lambda_2 \end{bmatrix} + \rho z z^{\mathrm{T}} \right) \cdot \begin{bmatrix} Q_1^{\mathrm{T}} & 0 \\ 0 & Q_2^{\mathrm{T}} \end{bmatrix}$$

where $z = \mathrm{diag}(Q_1^{\mathrm{T}}, Q_2^{\mathrm{T}})x$. So to compute the *complete* eigendecomposition of T, we need to compute the eigendecomposition $Q \Lambda Q^{\mathrm{T}}$ of the matrix $\mathrm{diag}(\Lambda_1, \Lambda_2) + \rho z z^{\mathrm{T}}$ which is a diagonal matrix plus a rank-1 update. Fortunately, this can be done very rapidly (for details, see [7]) and we finally have found the full eigendecomposition

$$T = (\mathrm{diag}(Q_1, Q_2)Q) \Lambda (\mathrm{diag}(Q_1, Q_2)Q)^{\mathrm{T}}.$$

If four processors are available, this algorithm can be applied recursively to find the eigendecompositions of T_1 and T_2, and so on. For details, see [7].

Domain decomposition almost speaks for itself: a computational domain, e.g. on which a partial differential equation has to be solved, is split into subdomains and each subdomain is assigned to a processor. Each processor solves the computational subproblem on its own subdomain and next the solutions found have to be "matched" on the boundaries of the subdomains. Often some iterations have to be performed in order to let the different solutions converge on overlapping boundaries. One critical problem is to split up the computational domain in such a way that all processors get equally large computational tasks to perform on their respective subdomains.

5.2 Numerical algorithms

5.2.1 Tridiagonal systems

The usual sequential method to solve a tridiagonal system $Ax = b$ is to decompose the matrix A into the product of a lower bidiagonal matrix L and an upper bidiagonal matrix U, and then solve $Ly = b$ in a forward sweep, followed by solving $Ux = y$ in a backward sweep. Both sweeps are linear recursions, for which one could resort to one of the techniques of recursive doubling or cyclic reduction, as described in Section 5.1.4. The performance obtainable in this way is not very spectacular. Better results can be obtained if *many* independent tridiagonal systems have to be solved at the same time. This occurs, for example, in the numerical solution of the 3-D shallow water equations [15].

Exercise 5.1 Design an efficient algorithm to solve the $n \times n$ unit lower bidiagonal systems $A^{(k)}x^{(k)} = b^{(k)}, k = 1, \ldots, m$, where $m \gg n$, on a vector computer. □

Another divide-and-conquer type technique for solving tridiagonal systems was studied by Wang [43]; it is a "partition" method. The system is first partitioned into subsystems, after which elimination proceeds simultaneously on all subsystems by elementary row transformations until finally A is diagonalised. We will illustrate it by an example of a tridiagonal 8×8 system $Ax = b$, where

$$A = \left[\begin{array}{cccc|cccc} a_1 & d_1 & & & & & & \\ c_2 & a_2 & d_2 & & & & & \\ & c_3 & a_3 & d_3 & & & & \\ & & c_4 & a_4 & d_4 & & & \\ \hline & & & c_5 & a_5 & d_5 & & \\ & & & & c_6 & a_6 & d_6 & \\ & & & & & c_7 & a_7 & d_7 \\ & & & & & & c_8 & a_8 \end{array}\right]$$

Figure 5.1: *The original tridiagonal matrix A*

We assume we have two parallel processors.

(a) Partition the matrix into 4×4 block tridiagonal form, as shown in Figure 5.1.

(b) (Elimination of the lower diagonals in the diagonal blocks.) Eliminate c_2, c_6 simultaneously on the two processors, then eliminate c_3, c_7 simultaneously, followed by c_4, c_8. The matrix is now triangular except for the fourth column. The f's shown in Figure 5.2 are non-zero fill-ins created during the elimination.

(c) (Elimination of the upper diagonals in the diagonal blocks, except for d_3 and d_7.) Next, eliminate d_2, d_6 simultaneously, followed by d_1, d_5, and then d_4. This leaves us with a diagonal matrix, except for the fourth and eighth columns, as shown in Figure 5.2. The g's are fill-ins created during this step.

(d) The matrix is triangularised by the elimination of c_5, f_6, f_7, f_8.

(e) The matrix is diagonalised by the elimination of d_7, g_6, g_5, g_4; d_3, g_2, g_1.

$$\left[\begin{array}{ccccc|ccccc}
a_1 & & & g_1 & & & & & \\
 & a_2 & & g_2 & & & & & \\
 & & a_3 & d_3 & & & & & \\
 & & & a_4 & & & & & g_4 \\
- & - & - & - & + & - & - & - & - \\
 & & & c_5 & & a_5 & & & g_5 \\
 & & & f_6 & & & a_6 & & g_6 \\
 & & & f_7 & & & & a_7 & d_7 \\
 & & & f_8 & & & & & a_8
\end{array}\right]$$

Figure 5.2: *The matrix A after elimination of the lower (step (b)) and upper diagonals (step (c)) in the diagonal blocks.*

Exercise 5.2 Describe the Wang algorithm for a matrix of order n, partitioned in $p \times p$-blocks, assuming that n is divisible by p with $\frac{n}{p} = k \geq 2$, and that k processors are available. □

In [42] Van der Vorst and Dekker have studied suitable variants of the method of Wang for different vector computers and they have presented upper bounds for their performance and some actually observed performances in a Fortran environment.

5.2.2 Blocked algorithms

Introduction

For architectures having a memory hierarchy, the ratio of floating-point operations to data movement is in general not high enough to make efficient reuse of data that reside in cache or local memory. Therefore, it is often preferable to partition matrices into blocks and to perform the computation by matrix-matrix operations on the blocks. This approach, however, requires a different organisation of the computations. In this section we describe two different algorithms for the Cholesky decomposition, one based on column partition and one based

on a block partition. For both algorithms an implementation is given written in terms of the BLAS.

The BLAS, a set of basic linear algebra subprograms (introduced in Chapter 4), has become an important tool to obtain high performance on supercomputers while preserving program portability. Nowadays, optimised machine-specific implementations of the BLAS are provided by vendors of high-performance computers, as part of their standard run-time libraries. As a consequence, programs coded in terms of the BLAS can achieve near-peak performance on many high-performance computers, although they are still written in portable Fortran 77. Moreover, by using the BLAS it is possible to exploit parallelism in a transparent way suited for all kinds of machine architectures. Optimised versions of the BLAS have been installed not only on most supercomputers, but also on mainframes and workstations. So, when you prepare a code on your workstation which is meant to run on a super, you can already start by using the BLAS and have a good chance to achieve a high degree of efficiency on your workstation, too.

In the Appendix we list a Level 2 BLAS implementation of the Cholesky decomposition and an implementation written in terms of Level 3 BLAS kernels. As mentioned before, the performance of the Level 2 BLAS operations is limited by the rate of data movement between different levels of memory. The Level 2 BLAS performs only $\mathcal{O}(n^2)$ floating-point operations on $\mathcal{O}(n^2)$ data, whereas the Level 3 BLAS performs $\mathcal{O}(n^3)$ operations on $\mathcal{O}(n^2)$ data. Performance expressed in Mflop/s achieved for both implementations on an Alliant FX/4 is listed. The optimal blocksize is machine-dependent.

Cholesky decomposition

The Cholesky factorisation of a positive definite symmetric matrix A is given by

$$A = LL^{\mathrm{T}}$$

where L is a lower triangular matrix. Since A is positive definite, pivoting is not necessary to ensure or improve numerical stability. The general step for the Cholesky factorisation looks like

```
for ···
    for ···
        for ···
            a_ij = a_ij − l_ik · a_kj    (l_ik = a_ik/a_kk)
```

and we are free to place i, j, and k in the outer, middle and inner loop. This yields six different forms, viz., ijk, ikj, jik, jki, kij and kji, which all have the same number of floating-point operations but different data access and updating patterns.

At first sight it is not easy to recognise how we can write this factorisation in terms of the BLAS. Therefore, we describe the decomposition as follows

$$\begin{bmatrix} A_{11} & & \\ A_{21} & A_{22} & \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11}^{\mathrm{T}} & L_{21}^{\mathrm{T}} & L_{31}^{\mathrm{T}} \\ & L_{22}^{\mathrm{T}} & L_{32}^{\mathrm{T}} \\ & & L_{33}^{\mathrm{T}} \end{bmatrix}$$

Suppose L_{11}, L_{21} and L_{31} were already calculated in previous steps, then L_{22} and L_{32} can be obtained by multiplying L and L^{T} and comparing corresponding elements in $L \cdot L^{\mathrm{T}}$ and A:

$$\begin{aligned} L_{21}L_{21}^{\mathrm{T}} + L_{22}L_{22}^{\mathrm{T}} &= A_{22} \\ L_{31}L_{21}^{\mathrm{T}} + L_{32}L_{22}^{\mathrm{T}} &= A_{32} \end{aligned} \tag{5.1}$$

Note that we have not mentioned yet how the matrix A has been partitioned. Two cases can be distinguished:

- $(A_{22}, A_{32})^{\mathrm{T}}$ is a single column vector. In that case one column at a time is updated.
- $(A_{22}, A_{32})^{\mathrm{T}}$ is a submatrix of nb columns and those nb columns are updated simultaneously.

The first case can be described in terms of Level 2 BLAS, the second one in terms of the Level 3 BLAS as is illustrated below. All BLAS subroutines used have prefix `S_`, which means that they work with `REAL` or `SINGLE PRECISION` numbers. Similar versions with prefix `D_` for `DOUBLE PRECISION` numbers are available.

Level 2 BLAS implementation

Suppose in previous steps the elements of L_{11}, L_{21} and L_{31} have already been calculated. In case A_{22} and L_{22} are single elements we proceed as follows:

- compute the innerproduct (by calling the Level 1 BLAS function `SDOT`) $L_{21}L_{21}^{\mathrm{T}}$ and subtract this value from A_{22};
- next, since L_{22} is a single element, L_{22} becomes the square root of this value.

From the second equation of (5.1) we compute L_{32} as follows:

- compute the matrix-vector product $L_{31}L_{21}^{\mathrm{T}}$ and subtract the result from A_{32} by means of the BLAS-2 subroutine `SGEMV`;
- divide the elements of vector L_{32} by the value of L_{22}, by using `SSCAL`.

The main part of the computation is performed by the Level 2 BLAS subroutine `SGEMV`. At each step one column is updated, but whether the jik or the jki-variant is applied depends on the vendor implementation of `SGEMV`, which, at its turn, is dependent on the machine architecture.

Level 3 BLAS implementation

Again we suppose that L_{11}, L_{21} and L_{31} have been calculated in previous steps. We describe how to update a block of nb columns. For that purpose A and L are subdivided, such that A_{22} is a symmetric matrix of order nb and L_{22} is a lower triangular matrix of the same order. From the first equation of (5.1) we now obtain L_{22} as follows:

– compute $A'_{22} \leftarrow A_{22} - L_{21}.L_{21}^{\mathrm{T}}$, which corresponds to a symmetric rank-k update. The operation can be performed by calling the BLAS subroutine `SSYRK`.

– next, solve L_{22} from $L_{22}.L_{22}^{\mathrm{T}} = A'_{22}$, i.e., perform the Cholesky decomposition on a block matrix of order nb. This can be carried out by calling, for instance, the Level 2 BLAS implementation discussed above.

The second equation of (5.1) delivers L_{32} as follows:

– call the Level 3 BLAS subroutine `SGEMM` to multiply the matrix L_{31} by L_{21}^{T} and to subtract the result from A_{32}: $A'_{32} \leftarrow A_{32} - L_{31} \cdot L_{21}^{\mathrm{T}}$.

– next, solve L_{32} from $L_{32} \cdot L_{22}^{\mathrm{T}} = A'_{32}$, where L_{22}^{T} is an upper triangular matrix, by calling the BLAS 3 subroutine `STRSM`.

Performance of blocked and non-blocked algorithm

It turns out that the performance of the blocked version is better than the non-blocked version on a wide range of high-performance vector and parallel machines. As an example we give some figures for the Alliant FX/4:

n	non-blocked	$nb = 32$	$nb = 64$
100	4.91	5.44	5.10
200	7.67	10.62	9.46
300	8.14	12.70	12.20
400	8.14	13.83	11.17
500	8.36	14.86	14.01

Performance in Mflop/s of the Cholesky decomposition on an Alliant FX/4.

Exercise 5.3 Compute the number of Mflop/s of a blocked and non-blocked code for the Cholesky decomposition for several matrix orders and several nb values. The original LAPACK codes (non-blocked `SPOTF2.F` and blocked `SPOTRF.F`) can be obtained by sending an E-mail to `netlib@ornl.gov` with the body:

```
send spotf2.f from lapack
```

```
send spotrf.f from lapack
```

You will obtain the sources (and some auxiliary sources which are called by SPOTF2 and SPOTRF). □

Appendix

Here, we list Level 2 and 3 BLAS implementations for matrices that are stored in the lower part of the array A. The sources have been taken from the LAPACK subroutine SPOTF2 and SPOTRF. The description of the specific calls to BLAS is briefly commented.

```
      SUBROUTINE SLLT2( n, a, lda , info )
*
*     Level 2 BLAS version (non-blocked)
*
*     .. Scalar Arguments ..
      INTEGER            info, lda, n
*     ..
*     .. Array Arguments ..
      REAL               a(lda, *)
*     ..
*
*  Purpose
*  =======
*
*  SLLT2  computes  the Cholesky  factorisation  of  a  real   symmetric
*  positive  definite  matrix  stored in  the lower part of the array a.
*
*     .. Parameters ..
      REAL               one, zero
      PARAMETER          (one = 1.0E+0, zero = 0.0E+0)
*     ..
*     .. Local Scalars ..
      INTEGER            j
*     ..
*     .. External BLAS Functions and Subroutines ..
      REAL               SDOT
      EXTERNAL           SDOT
      EXTERNAL           SGEMV, SSCAL
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          SQRT
*     ..
*     .. Executable Statements ..
*
*     First compute L(1,1) and test for non-positive-definiteness.
```

```
*
      IF( a(1,1) .LE. zero ) GO TO 30
      a(1,1) = SQRT( a(1,1) )
*
      DO 10 j = 2, n
*
*        Update elements j:n of column j-1.
C .....................................................................
C BLAS   SGEMV computes the matrix-vector product
C BLAS
C BLAS       a[j:n,j-1] - a[j:n,1:j-2] * Transpose(a[j-1,1:j-2])
C BLAS
C BLAS   SGEMV stores the result into a[j:n,j-1].
C .....................................................................
*
         CALL SGEMV( 'Transpose', n-j+1, j-2, -one, a(j, 1), lda,
     +               a(j-1, 1), lda, one, a(j, j-1), 1 )
*
*        Scale elements j:n of column j-1.
C .....................................................................
C BLAS   SSCAL multiplies vector a[j:n,j-1] by the reciprocal of the
C BLAS   diagonal element a[j-1,j-1].
C .....................................................................
*
         CALL SSCAL( n-j+1, one/a(j-1, j-1), a(j, j-1), 1 )
*
*        Update a(j,j).
C .....................................................................
C BLAS   SDOT computes the innerproduct of a[j,1:j-1] x a[j,1:j-1].
C .....................................................................
*
         a(j,j) = a(j,j) -
     +               SDOT( j-1, a(j, 1), lda, a(j, 1), lda )
*
*        Compute L(j,j) and test for non-positive-definiteness.
*
         IF( a(j,j) .LE. zero )
     +      GO TO 30
         a(j,j) = SQRT( a(j,j) )
   10 CONTINUE
      GO TO 40
*
   30 CONTINUE
      info = j
*
   40 RETURN
*
```

```
*     End of SLLT2
*
      END
```

The source of SLLTB has been derived from the LAPACK subroutine SPOTRF.

```
      SUBROUTINE SLLTB( n, nb, a, lda, info )
*
*     Level 3 BLAS version (blocked)
*
*     .. Scalar Arguments ..
      INTEGER            info, lda, n, nb
*     ..
*     .. Array Arguments ..
      REAL               a(lda, *)
*     ..
*
*  Purpose
*  =======
*
*  SLLTB  computes  the  Cholesky  factorisation  of  a  real  symmetric
*  positive  definite matrix stored  in the  lower part of  the array a.
*
*     .. Parameters ..
      REAL               one
      PARAMETER          ( one = 1.0E+0 )
*     ..
*     .. Local Scalars ..
      INTEGER            j, jb
*     ..
*     .. External Subroutines ..
      EXTERNAL           SGEMM, SLLT2, SSYRK, STRSM
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MIN
*     ..
*     .. Executable Statements ..
*
*     Factor  the  first  diagonal block  a[1:jb,1:jb]  by  calling  the
*     non-blocked factorisation subroutine SLLT2.
*
      j  = 1
      jb = MIN( nb,n )
      CALL SLLT2( jb, a(1,1), lda, info )
      IF( info .NE. 0 ) GO TO 30
*
*     Do for each successive blocked column ...
*
```

```
      DO 20 j = nb + 1, n, nb
*
*        Update subdiagonal block.
C ....................................................................
C BLAS   SGEMM computes the matrix-matrix product
C BLAS
C BLAS       a[j:n,j-nb:j-1] -
C BLAS       a[j:n,1:j-nb-1] * Transpose(a[j-nb:j-1,1:j-nb-1])
C BLAS
C BLAS   SGEMM stores the result into a[j:n,j-nb:j-1].
C ....................................................................
*
         CALL SGEMM( 'No transpose', 'Transpose', n-j+1, nb, j-nb-1,
     +               -one, a(j, 1), lda, a(j-nb, 1), lda, one,
     +               a(j,j-nb), lda )
*
*        Compute subdiagonal block of L.
C ....................................................................
C BLAS   STRSM solves the triangular system
C BLAS
C BLAS      L * Transpose(a[j-nb:j-1,j-nb:j-1]) = a[j:n,j-nb:j-1]
C BLAS   STRSM stores the result l into a[j:n,j-nb:j-1]
C ....................................................................
*
         CALL STRSM( 'Right', 'Lower', 'Transpose', 'Non-unit',
     +               n-j+1, nb, one, a(j-nb, j-nb), lda,
     +               a(j, j-nb), lda )
*
*        Update diagonal block.
C ....................................................................
C BLAS   SSYRK computes the symmetric rank-k update
C BLAS
C BLAS      a[j:j+jb-1,j:j+jb-1] -
C BLAS      a[j:j+jb-1,1:j-1] * Tranpose(a[j:j+jb-1,1:j-1])
C BLAS
C BLAS   SSYRK stores the result L into a[j:j-jb+1,j:j+jb-1]
C ....................................................................
*
         jb = MIN( nb, n-j+1 )
         CALL SSYRK( 'Lower', 'No transpose', jb, j-1, -one,
     +               a(j,1), lda, one, a(j, j), lda )
*
*        Factorise  diagonal block a[j:j+jb-1,j:j+jb-1] by  calling  the
*        non-blocked factorisation subroutine SLLT2.
*
         CALL SLLT2( jb, a(j,j), lda, info )
         IF( info .NE. 0 ) GO TO 30
```

```
   20 CONTINUE
      GO TO 40
*
   30 CONTINUE
      info = info + j - 1
*
   40 CONTINUE
      RETURN
*
*     End of SLLTB
*
      END
```

5.2.3 Sparse matrix operations

What is a sparse matrix?

A sparse matrix is a matrix for which only a very small fraction of the elements is non-zero. In general such a matrix has only a few non-zero elements in each row (in many cases less than ten non-zero elements in a matrix which has an order of tens of thousands or even more). It is clear that it would be a waste of memory to devote large amounts of storage to the zero elements. For that reason many ways have been devised to store such matrices. The actual method chosen depends on the pattern, if any, of the non-zero elements, and on the operations needed. On the other hand you do not want to perform operations on such matrices where a large number of zero elements becomes non-zero (so-called fill-in).

General sparse matrices

The most general kind of sparse matrix is one where a randomly chosen element has a small (but non-zero) probability of being non-zero. This means that there is no pattern to be discerned in the non-zero elements; they are randomly placed within the matrix. There are a few models for storage layout in this case:

1. A one-dimensional array containing the non-zero data plus two one-dimensional index arrays that contain the actual row c.q. column indices of those elements. In its general form this version is not very useful; it is difficult to find an explicit element of the matrix (given row and column number). Moreover, finding whether an element is zero means a pass through the non-zero elements to see that the element in question is missing. The storage can be specialised a bit by storing the non-zero elements of a row (or column) consecutively in the array and providing additional index arrays that point to the start and/or end of these rows (columns). This however will cause fill-in and a rearrangement of many elements.

2. A series of one-dimensional data and index arrays, one each for every row of the matrix. This has the advantage over the first model that it is easy to trace rows of the matrix and, if the arrays are large enough, fill-in is handled on a row-by-row basis.

3. Of course we could use a similar storage by column with similar advantages and disadvantages.

4. We can follow a different approach by using linked lists. For instance we can create a linked list of all the non-zero elements of a row for each row of the matrix. In this case a non-zero element is presented by a node containing the value and a pointer to the next non-zero value of the row. Clearly, fill-in is easily handled by just inserting the new element in the list. A disadvantage is (at least in some programming languages) that some form of memory management is necessary. An additional feature is the use of doubly linked lists, where each node not only contains a pointer to the next non-zero element, but also to the previous non-zero element.

5. Indeed, also here the linked lists can be columnwise rather than rowwise.

6. The previous storage versions can be combined by making nodes not only point to next (and possibly previous) non-zero elements in the same row, but also in the same column. A main advantage of this method is that it makes it very easy to follow both rows and columns of a matrix.

Vectorisation of some operations is possible with storage methods 2 and 3 (and method 1 if non-zero elements of the same row or column are kept together). For example, assume the following (Fortran) declarations (here `MATDIM` is the actual order of the matrix and `MATFRAC` is the largest number of non-zero elements in a row, for storage method 2):

```
REAL elem(matdim, matfrac)
INTEGER index(matdim, matfrac), inonz(matdim)
```

The value of `elem(i,j)` is the actual value of matrix element `i, index(i,j)`. Moreover, `inonz(matdim)` contains the number of non-zero elements in a particular row. To evaluate the innerproduct of row `ir` of the matrix with a (given) vector `V` the following loop will do:

```
prod = 0.0
DO i = 1, inonz(ir)
   prod = elem(ir,i) * v(index(ir,i)) + prod
ENDDO
```

If the processor has "gather" hardware, this loop can be completely vectorised. On the other hand, standard "SAXPY" operations (where the multiple of a particular column of the matrix is added to a vector) are not simple; for such operations storage method 3 would be preferred, but in that case innerproducts

are very difficult. If you need many matrix-vector products , the way you want to do it (using saxpys or innerproducts) more or less dictates the storage method to be chosen. (As and aside: in Fortran you want of course to interchange the indices of `INDEX` and `ELEM`, so that the stride becomes 1.)

Storage methods 4 to 6 are never vectorizable because accessing a row (or column) means "pointer-chasing", i.e., only after you have retrieved a particular element you know where the next one will be. However, with the current crop of fast scalar processors, these storage methods will in most cases not lead to worse performance than what can be achieved by other methods. This means that if you want to parallelise operations on a series of fast scalar processors, storage methods 4 and 6 might very well be preferable. (Of course, if you want to parallelise, you want to keep the linked lists sorted according to row and/or column index.)

A set of operations for these storage methods (in particular method 2 and 3) have been proposed by Dodson and Lewis[9]. A major disadvantage of this paper is that it only defines BLAS-1 type operations, but as shown above, some of these are only useful if the row-storage method is used while others are only useful if the column-storage method is used.

Most research on these matrices is in ways to find a set of row and/or column interchanges such that the pattern of non-zero elements becomes more tractable. When this is possible, more straightforward techniques can be used to manipulate such matrices. But this is still on-going research.

Diagonal matrices

The situation is much simpler if we look at sparse matrices where only a limited number of diagonals (sub- and super-diagonals) contains non-zero elements. Devoting a row or a column of a two-dimensional array to each non-zero (sub/super-) diagonal of the matrix can in general solve most problems, and if the number of non-zero diagonals is very small, a number of one-dimensional arrays can be used. As an example, take a matrix of order N^2, where only the main diagonal, the first sub- and super-diagonal and the N-th sub- and super-diagonal contain non-zero elements. We might consider the following declaration in Fortran 77:

```
REAL elem(nsqu, -2:2)
```

where `elem(i,0)` is the main diagonal element `(i,i)`; `elem(i,1)` is the super-diagonal element `(i,i+1)`, for `i` $\leq$ `nsqu-1`; `elem(i,2)` is the super-diagonal element `(i,i+n)`, for `i` $\leq$ `nsqu-n`; and similarly, `elem(i,-1)` is the sub-diagonal element `(i,i-1)`, for `i` $>$ 1 and `elem(i,-2)` is the sub-diagonal `(i,i-n)`, for `i` $>$ `n`. To calculate the product `w` of this matrix with a vector `v` we need the following (five) loops:

```
DO i = 1, nsqu
   w(i) = v(i) * elem(i,0)
ENDDO
```

```
DO i = 1, nsqu-1
   w(i) = w(i) + v(i+1) * elem(i,1)
ENDDO
DO i = 1, nsqu-n
   w(i) = w(i) + v(i+n) * elem(i,2)
ENDDO
DO i = 1+1, nsqu
   w(i) = w(i) + v(i-1) * elem(i,-1)
ENDDO
DO i = n+1, nsqu
   w(i) = w(i) + v(i-n) * elem(i,-2)
ENDDO
```

(Check out that this indeed will give the matrix-vector product.)

All of these operations are easily vectorizable (they are all "SAXPY"s); moreover, all operations have unit stride on all arrays involved (if you use Fortran). You can also code the same matrix multiplication using innerproducts but in that case the stride across the matrix is non-unit, and there will be many short loops.

We could also store the diagonals in rows of the array; but in that case matrix-vector product with `SAXPY` will result in a non-unit stride, while the (shorter) vector inner-product-operation will have a unit stride. Alas, storing the diagonals in rows is the method chosen in both LINPACK and LAPACK!

Exercise 5.4 For the six storage methods proposed, some indication is given on how the operations can be vectorised. Check exactly how a matrix-vector product can be vectorised using those six methods (using innerproducts or SAXPY) and whether GATHER/SCATTER operations are needed. □

Exercise 5.5 Which storage methods allow simple distribution of the data across a number of processors, such that a matrix-vector product can be calculated without intermediate communication. Here simple distribution means few communications and no data rearrangement. □

5.2.4 Solving linear equations by direct methods

The solution of a system of linear equations can be computed by direct or by iterative methods. The choice between the two is determined by the expected time for the calculations and the expected total time for data access to solve the system at hand. In general we could say that direct methods are used for dense matrices that are not extremely large.

We will not deal with iterative methods in this section. In all direct methods the matrix is factored as a product of so-called simple matrices; a matrix is simple if a linear system with such a matrix can be solved with $\mathcal{O}(n^2)$ arithmetical operations. Examples of simple matrices are triangular matrices (both upper- and lower triangular), unitary matrices and elementary matrices, which

are matrices of the form $(I + gh^{\mathrm{T}})$ – in the real case – with g and h satisfying $h^{\mathrm{T}}g = 0$.

Triangular factorisation

The best known and most applied direct methods are the triangular factorisation methods. In these methods matrix A is factored as the product of a lower- and an uppertriangular matrix, L and U, respectively. For numerical stability it is necessary in general that the rows of matrix A are permuted. The appropriate permutation is determined during the factorisation process and applied in such a way that the final result holds as if this permutation had been known in advance.

In formula we have

$$PA = LU,$$

where $L_{ij} = 0$ for $i < j$, $U_{ij} = 0$ for $i > j$ and P is a permutation matrix. The solution of $Ax = b$ is now calculated by solving for y the triangular system $Ly = Pb$ (forward substitution) followed by solving for x the triangular system $Ux = y$ (backward substitution), according to the following equivalencies:

$$(Ax = b) \Leftrightarrow (PAx = Pb) \Leftrightarrow (LUx = Pb) \Leftrightarrow (Ly = Pb \wedge Ux = y).$$

The many variants of LU-factorisation that do exist, can be divided into three classes: the *innerproducts* variants, the *middleproducts* variants and the *outerproducts* variants.

The advantages and disadvantages of the various types are determined by the architecture of the machine and the type of data storage that is used for the matrix.

I Innerproducts variants.

Once a choice has been made for the way the diagonal elements of L and U are scaled, the remaining elements of L and U are determined consecutively from equating the elements of A:

$$A_{ij} = \sum_{k=1}^{\min(i,j)} L_{ik} \times U_{kj}.$$

LU-factorisation codes belonging to this class are seldom used on vector- and parallel computers.

II Middleproducts variants.

From the many schemes that belong to this class, we take one that is oriented columnwise. The algorithm that we will describe is called a "left looking algorithm" for reasons that become clear from the description and the illustration in Figure 5.3.

Assume that the first k columns of L and U have been calculated. The $(k+1)$-st columns of U and L are calculated from the following relations:

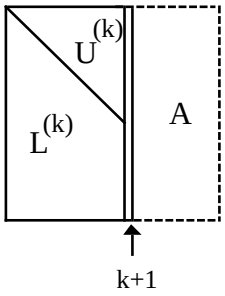


Figure 5.3: *Left looking algorithm of middleproduct variant of LU factorisation*

1. $L_{[1:k,1:k]}U_{[1:k,k+1]} = A_{[1:k,k+1]}$.
2. $U_{k+1,k+1} = 1$ (by choice of normalisation).
   The $(k+1)$-st column of L is determined in two steps. First a temporary version of $L_{[k+1:n,k+1]}$ is determined from the following relation:
3. $L_{[k+1:n,1:k]}U_{[1:k,k+1]} + L_{[k+1:n,k+1]} = A_{[k+1:n,k+1]}$.
4. Determine the index p (say) of an element with maximal size in $L_{[k+1:n,k+1]}$.
5. Interchange rows p and $(k+1)$ of the $(n-k) \times (k+1)$ submatrix $L_{[k+1:n,1:k+1]}$; the same rows in the remaining part of matrix A are also to be interchanged. Information for this interchanging is stored.

BLAS-2 routines can be used in steps 1 and 3, step 1 requiring the solution of a linear system with a triangular matrix and step 3 a matrix-vector multiplication.

After $n-1$ steps the factorisation is accomplished. All row interchanges must be recorded in order to perform all appropriate interchanges on the elements of the right-hand side vector. The solution vector of the original system can then be computed by forward- and backward substitution with the triangular systems as was described earlier. For these substitutions an appropriate BLAS-2 routine can be used.

III Outerproducts variants.

The factorisation schemes in this class use updates of the entire part of the remaining matrix. The "classical" Gaussian elimination scheme is a member of this class. A representative example is described below; it is called a "right looking algorithm", for reasons that will become clear from the description and the illustration in Figure 5.4.

Assume that the first k columns of L and the first k rows of U have already been calculated and that the remaining part of the matrix has

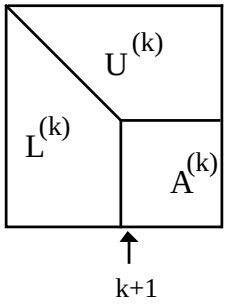


Figure 5.4: *Right looking algorithm of outerproduct variant of LU factorisation.*

been updated in each step as is defined in the algorithm. The $(k+1)^{\text{st}}$ column of L and the the $(k+1)^{\text{st}}$ row of U are calculated from the following relations:

1. Determine the index p (say) of an element with maximal size in $A_{[k+1:n,k+1]}$.
2. Interchange rows p and $(k+1)$ of the $(n-k)\times k$ submatrix $L_{[k+1:n,1:k]}$; the same rows in the remaining part of matrix A must also be interchanged. Information for this interchanging is stored.
3. $U_{[k+1,k+1:n]} = A_{[k+1,k+1:n]}$.
4. $L_{k+1,k+1} = 1$ (by choice of normalisation).
5. $L_{[k+2:n,k+1]} = A_{[k+2:n,k+1]}/U_{k+1,k+1}$.
6. $A_{[k+2:n,k+2:n]} := A_{[k+2:n,k+2:n]} - L_{[k+2:n,k+1]} \times U_{[k+1,k+2:n]}$.

A BLAS-2 routine can be used in step 6 which requires a rank-1 matrix update. After $n-1$ steps the factorisation is accomplished.
All row interchanges must be recorded in order to perform all appropriate interchanges on the elements of the right-hand side vector. The solution vector of the original system can then be computed by forward- and backward substitution with the triangular systems as was described before.

Note that in outerproduct variant we made a choice for the diagonal elements of L being equal to 1, which differs from the earlier innerproduct variant where we chose the diagonal elements of U being equal to 1.

Hierarchical algorithms

The algorithms from the three classes above may not be optimal for multiprocessor computers. Better performance can be attained with block versions of the LU-factorisation algorithms such that BLAS-3 subroutines can be applied.

In that case the matrix is divided in blocks of size $s \times s$. The optimal value of parameter s is determined by the architecture of the specific computer. It depends on a variety of machine characteristics such as the ratio of the time for a floating-point operation to the time for fetching or storing such a number, the size of the cache memory, the size of the registers and so on. (If s is not a divisor of n, smaller rectangular blocks do occur along the border of the matrix, but if n is large enough, this has no measurable effect on the performance of the algorithm).

Block algorithms deal with submatrices where possible and with elements where necessary, therefore they are called hierarchical algorithms; they can be derived from the algorithms that belong to the earlier defined three classes of LU-factorisation algorithms. As examples we will describe the block versions of the two algorithms that were described earlier in classes II and III.

IIB A block middleproduct variant in a left looking version.
Suppose that the first $(k-1) \times s$ columns of L and U have already been calculated and stored in the corresponding parts of A. The next s columns of L and U are calculated from the definition of multiplication of matrices in partitioned form as is illustrated by Figure 5.5.

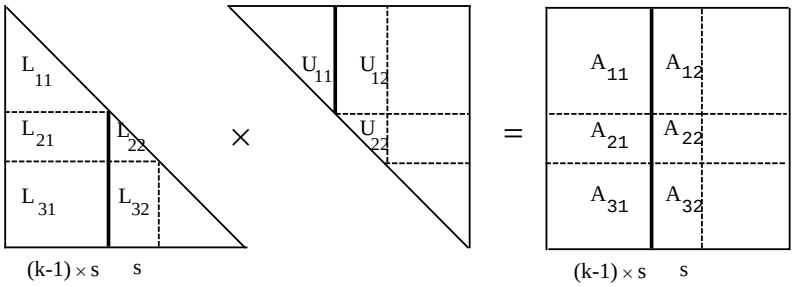


Figure 5.5: *A block middleproduct variant in a left looking version.*

For the columns of L and U that are to be calculated the following relations hold:

$$\begin{aligned} L_{11}U_{12} &= A_{12}, \\ L_{21}U_{12} + L_{22}U_{22} &= A_{22}, \\ L_{31}U_{12} + L_{32}U_{22} &= A_{32}. \end{aligned}$$

From this we find:

1. Determine U_{12} by solving the linear system $L_{11}U_{12} = A_{12}$;
   (Solution of a system with multiple right-hand sides and a triangular coefficient matrix.)
2. Perform the matrix update $\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} := \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} U_{12}$;
   (Multiplication and subtraction of matrices.)

3. Calculate an LU-factorisation with partial pivoting by row interchanges of the rectangular matrix $\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix}$; this is a straightforward generalisation of the LU factorisation of a square matrix for which one of the earlier algorithms can be applied. This yields L_{22}, L_{32} and U_{22} holding: $\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22}$.

After $\lceil n/s \rceil$ "block-steps" an LU factorisation is calculated.

In steps 1 and 2 BLAS-3 routines can be applied for the indicated computations.

IIIb A block outerproduct variant in a right looking version.
Suppose that the first $(k-1) \times s$ columns of L and the first $(k-1) \times s$ rows of U have already been calculated and stored in the corresponding parts of A, and that the lower right-hand square part of A has been updated as defined in the following algorithm. The next s columns of L and s rows of U are calculated from the definition of matrix multiplication as illustrated in Figure 5.6:

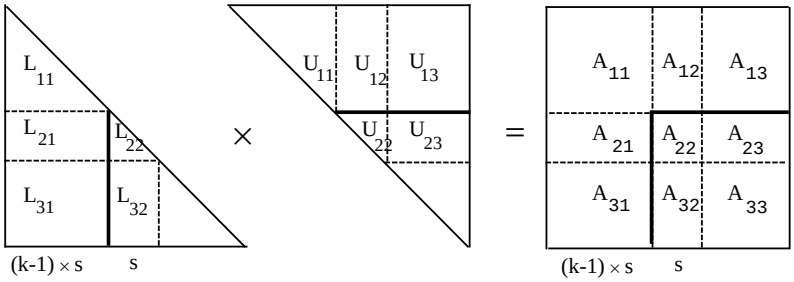


Figure 5.6: *A block outerproduct variant in a right looking version.*

1. Calculate an LU factorisation with partial pivoting by row interchanges of the rectangular matrix $\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix}$ yielding L_{22}, L_{32} and U_{22} such that $\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22}$;
2. Determine U_{23} by solving the linear system $L_{22}U_{23} = A_{23}$;
(Solution of a system with multiple right-hand sides and a triangular coefficient matrix.)
3. Perform matrix update: $A_{33} := A_{33} - L_{32}U_{23}$.
(Multiplication and subtraction of matrices.)

Also in this case we find that an LU factorisation is calculated after $\lceil n/s \rceil$ "block-steps". BLAS-3 routines can be applied in steps 2 and 3 for the computations indicated.

Considerations for use on parallel platforms

"Which algorithm for the LU factorisation should be used on a given parallel computer system?"

This question cannot be answered in general terms. The choice depends on the hierarchy that is present in the memory of the given system. We will try to illustrate this with some examples.

At one end of the spectrum we find the multiprocessor systems with distributed memory where each of the processors own a part of the total memory. Also a cluster of workstations connected to a network can be viewed as belonging to this class. The essential property is that all processors have access to all of the memory, but the access time to the private memory is negligible in comparison with the access time to the memory of other processors.

At the other end of the spectrum we find the multiprocessor systems with shared memory. All processors share one, usually large, memory. To increase the overall speed of such a computer system, a cache memory is linked between the processors and the main memory. Data transfer between processors and cache memory is an order of magnitude faster than between cache and main memory. In general the processors do have registers which can be viewed as small private memory with negligible access time.

In all variants of LU factorisation, the elements of L and U are written over the original elements of matrix A, or at least this feature is offered as a possibility. So the total process of LU factorisation can be seen as a way of transforming the original matrix elements of A into the corresponding elements of either L or U. All computations, apart from those that are needed for pivoting, are such that a matrix element is replaced by or augmented with some arithmetical function of other matrix elements. The total number of floating-point operations that is needed in all variants is $\frac{2}{3}n^3 + cn^2$ for small value of c. The difference between the variants is in the number of data accesses and the order in which the data is accessed.

In distributed-memory computers, the matrix element that is subject to change, should remain in the local memory of the processor that performs that particular computation as much as possible for an efficient implementation. Only the elements that are needed in the arithmetical function and which are not private to the memory of that processor should be transported.

In shared-memory computers none of the data is local to none of the processors; so in each computation all occurring matrix elements, either on the left-hand side or on the right-hand side of an assignment, need to be transported to a processor. Here the important issue is to optimise the use of the cache memory in such a way that elements that are used in different compu-

tations can reside in the cache until they are needed in the next computation as much as possible. In this respect the use of BLAS level-3 routines is very advantageous; the size of the submatrices must be tailored to the size of the cache memory.

To be more specific, the outerproducts variants III and III B are very efficient on distributed-memory systems. In each step of the algorithm under III, only elements of the newly determined column of L and row of U need to be broadcasted to other processors where they can be used for updating the appropriate matrix elements in the lower right-hand corner. In this way only $2(n-k)$ elements are broadcasted in each step. This yields in total n^2-n elements to be broadcasted for the whole algorithm. If the network in the distributed-memory system has a high latency (resulting in a large start-up time for interprocessor communication), then the use of block algorithms enables the data to be sent in larger chunks which is much more efficient. In that case the use of the algorithm described under III B is to be advised.

The middleproduct variant under II B gives rise to an efficient implementation for a shared-memory system. As has been mentioned before, the proper size of the blocks, (parameter s in the description of the algorithm), depends on the size of the cache memory in relation to the size of the matrix.

Conclusion

For shared-memory parallel computers, reliable Fortran 77 implementations of algorithms for solving linear algebra problems are available in LAPACK and can be obtained via NETLIB. The routines in LAPACK do exploit BLAS-3 possibilities. For efficient use on a given shared-memory computer it is necessary that the appropriate BLAS-3 routines are implemented efficiently.

For distributed-memory parallel computers, matters are more complicated. The efficiency of an implementation depends on the distribution of the matrix elements over the processors, the topology of the network and its latency. These issues influence the structure of a parallel program via the use of communication primitives. At the moment of this writing (April 1993) various developments can be recognised in an effort to reach international standardisation in this area.

Exercise 5.6 In rare cases the partial pivoting strategy as used in the algorithms described under II, III, IIB and IIIB is not numerically stable. As a consequence the elements of U become much larger in modulus than the elements of the given matrix A. This will result in large errors. An effectual remedy is the use of complete pivoting. According to this strategy rows and columns must be interchanged such that in step k of Gaussian elimination the (k,k)-diagonal element has maximal modulus among all elements in the remaining lower right-hand corner submatrix of size $(n-k+1)\times(n-k+1)$.

Discuss whether or not it is possible to apply the complete pivoting strategy in the four described variants of Gaussian elimination. □

Exercise 5.7 A variant of Gauß-Jordan elimination has become known under the name Gauß-Huard algorithm. It resembles Gauß-Jordan in the fact that application of the algorithm results in transforming the given matrix into the identity matrix. This is accomplished by eliminating both the elements under and above the diagonal of the matrix in combination with appropriate scaling.

The difference between Gauß-Huard and Gauß-Jordan lies in the order in which this elimination is performed. In the k^{th} step of Gauß-Huard an already created identity matrix of size $(k-1) \times (k-1)$ in the upper left-hand corner is extended to a matrix of size $k \times k$ as follows.

i) The first $(k-1)$ elements of row k are eliminated by rows 1 to $k-1$.

ii) A maximal element in modulus among the remaining elements in row k is brought into diagonal position by interchanging the two relevant columns of the matrix.

iii) The elements of the k^{th} row are divided by this pivotal diagonal element so that a new value 1 is created at the diagonal.

iv) The elements in the k^{th} column above the diagonal are eliminated by the k^{th} row.

Note that all elements in the matrix below the k^{th} row remain unchanged. Verify that the number of floating-point operations for evaluating this algorithm is $(2/3)n^3 + \mathcal{O}(n^2)$ like in standard Gaussian elimination. Note the difference with the total of $n^3 + \mathcal{O}(n^2)$, which is the number of floating-point operations that is needed in the Gauß-Jordan algorithm. □

5.2.5 Iterative methods for linear systems

Many large scale computational problems lead to the problem of solving a large linear system $Ax = b$, where A is a square nonsingular matrix of order n. Most often, the matrix A is sparse, which means that only few elements per row are non-zero, and usually only these non-zeros are stored along with pointers. This sparse structure is important, since it permits to store very large systems, but in many relevant cases much of the sparsity is lost if the system is solved by a direct solution method, like, e.g., Gaussian elimination. Loss of sparsity leads to an increase in computer memory requirements, and it may also lead to unacceptable computer time requirements, since we have to proceed the calculations with the new non-zero elements as well.

An alternative in such circumstances is offered by iterative methods. The idea is that we try to improve an approximation to the solution x with only little work and with modest memory resources. This is typically done by replacing the given problem $Ax = b$ by some other problem $Kz = d$, which is much easier to solve. The standard approach is to write A as:

$$A = K - R. \tag{5.2}$$

The original system can then be reformulated as

$$Kx = b + Rx = b + (K - A)x,$$

and this system would have given the solution if we knew the right-hand side. The basic iteration scheme follows from computing the right-hand side with a current approximation, x_i say, and to solve the system for x_{i+1}:

$$Kx_{i+1} = b + (K - A)x_i,$$

or

$$Kx_{i+1} = Kx_i + (b - Ax_i).$$

This basic iteration can be further simplified as

$$x_{i+1} = x_i + K^{-1}r_i, \tag{5.3}$$

where $r_i = b - Ax_i$ denotes the residual vector, which tells us how well the current approximate solution x_i satisfies the equation $Ax = b$. It should be stressed that, although we have used the notation K^{-1}, we almost never invert K explicitly. Instead, when we need the vector $y = K^{-1}r_i$, we compute it by solving y from $Ky = r_i$.

For our further discussion on iterative methods we will assume that the matrix A has been split as

$$A = I - (I - A). \tag{5.4}$$

This is no loss of generality, since the more general splitting $A = K - R$ can be rewritten as the splitting $B = I - (I - B)$ for the matrix $B = K^{-1}A$. In this case we say that the matrix A has been preconditioned with K. The iteration in (5.3) then reads as

$$x_{i+1} = x_i + r_i, \tag{5.5}$$

which is the standard Richardson iteration method.
For this standard iteration method it follows that

$$x_{i+1} = x_0 + r_0 + r_1 + r_2 + ... + r_i. \tag{5.6}$$

Multiplying the standard iteration (5.5) with $-A$, and adding b at both sides, gives

$$b - Ax_{i+1} = b - Ax_i - Ar_i,$$

or

$$r_{i+1} = (I - A)r_i = (I - A)^{i+1}r_0. \tag{5.7}$$

In order to keep our formulas as simple as possible, we will further assume that $x_0 = 0$. This too does not mean a loss of generality, for the given system with

$x_0 \neq 0$ can through a simple linear transformation $z = x - x_0$ be transformed to the system $Az = b - Ax_0 = \tilde{b}$ for which obviously $z_0 = 0$.

This leads to the observation that x_{i+1} can be viewed as an expression in powers of A:

$$x_{i+1} = r_0 + r_1 + ... + r_i = \sum_{j=0}^{i} (I - A)^j r_0, \tag{5.8}$$

and hence, although computed differently, the vector x_{i+1} is in the subspace spanned by the vectors $r_0, Ar_0, ..., A^i r_0$. This $(i + 1)$ dimensional subspace is called the Krylov subspace, denoted as $K^{i+1}(A; r_0)$.
Apparantly, the Richardson iteration delivers approximations that are elements of Krylov subspaces of increasing dimension.

Exercise 5.8 Show that for any $x_{i+1} \in K^{i+1}(A; r_0)$ the residual r_{i+1} can be written as

$$r_{i+1} = P_{i+1}(A)r_0, \tag{5.9}$$

where P_{i+1} is polynomial of degree $i + 1$, with the property that $P_{i+1}(0) = 1$. □

Iterative methods that generate solutions in Krylov subspaces are called Krylov subspace methods. For a good mathematical introduction to this class of successful and popular methods, see [13].

The Conjugate Gradients Method

The natural question arises whether we can find a better approximate solution x_i in the Krylov subspace. There are two approaches, one of which is to try to find the $x_i \in K^{i-1}(A; r_0)$ for which the residual r_i is orthogonal to the current subspace. This means that in a sense the subspace is explored as good as possible.

It is easy to verify that $r_1 \in \{r_0, Ar_0\}$, and hence, for the desired x_i we have that r_0 and r_1 should form an orthogonal basis for the Krylov subspace of dimension 2. We leave it as an exercise to show that if we continue in this way, then

$$\{r_0, r_1, ..., r_i\} \text{ form an orthogonal basis for } K^{i+1}(A; r_0). \tag{5.10}$$

This leads to the idea to construct an orthogonal basis for the Krylov subspace, since then each basis vector would be a multiple of a residual vector, and we may hope to find the corresponding approximations to the solution relatively easy. We will sketch briefly how this works out. For a more rigorous derivation we refer to [16].
If the matrix A is symmetric, then one can show by an induction argument that the orthogonal basis can be generated by a 3-term recurrence:

$$\tilde{\alpha}_{j+1} r_{j+1} = Ar_j - \tilde{\beta}_j r_j - \tilde{\gamma}_j r_{j-1}. \tag{5.11}$$

The constant $\tilde{\alpha}_{j+1}$ is necessary to make sure that r_{j+1} has the right length, i.e., this vector should be equal to $b - Ax_{j+1}$. The value follows from the requirement that r_{j+1} can be written as $r_{j+1} = P_{j+1}(A)r_0$, with $P_{j+1}(0) = 1$. This is a technical detail that has been taken care for in the final algorithm. The other constants follow from the ortogonality conditions.
If we define R_i as the matrix with columns $r_0, \ldots, r_{i-1}$, then the recurrence relation can be rewritten in matrix notation as

$$AR_i = R_i \begin{pmatrix} \ddots & \ddots & & & \\ \ddots & \ddots & \tilde{\gamma}_j & & \\ & \ddots & \tilde{\beta}_j & \ddots & \\ & & \tilde{\alpha}_{j+1} & \ddots & \ddots \\ & & & \ddots & \ddots \end{pmatrix} + \tilde{\alpha}_i \begin{pmatrix} 0, 0, \ldots, r_i \end{pmatrix}$$

or

$$AR_i = R_i T_i + \tilde{\alpha}_i r_i e_i^T, \tag{5.12}$$

in which T_i is an i by i tridiagonal matrix and e_i is the ith canonical vector in $\mathbb{R}^i$.

Since we are looking for a solution x_i in $K^i(A; r_0)$, that vector can be written as a combination of the basis vectors of the Krylov subspace, and hence

$$x_i = R_i y.$$

(Note that y has i components)
Further we have for the x_i that its residual is orthogonal to all columns of the matrix R_i:

$$\begin{aligned} & R_i^T(Ax_i - b) = 0 \\ \Rightarrow \quad & R_i^T A R_i y - R_i^T b = 0. \end{aligned}$$

Using equation (5.12) and the fact that r_i is orthogonal with respect to the columns of R_i, and that $r_0 = b - Ax_0 = b$, we obtain

$$R_i^T R_i T_i y = \|r_0\|_2^2 e_1$$

Since $R_i^T R_i$ is a diagonal matrix with diagonal elements $\|r_0\|_2^2$ up to $\|r_{i-1}\|_2^2$ we find the desired solution from

$$T_i y = e_1 \quad \Rightarrow \quad y \quad \Rightarrow \quad x_i = R_i y.$$

Note that so far we have only used the fact that A is symmetric and we have assumed that the matrix T_i is not singular. This opens the possibility for several suitable iterative methods, among which the Conjugate Gradients method. The

Krylov subspace method that has been derived here is known as the Lanczos method for symmetric systems [26].

Note that for some $j \leq n-1$ the construction of the orthogonal basis must terminate. In that case we have that $AR_{j+1} = R_{j+1}T_{j+1}$. Let y be the solution of the reduced system $T_{j+1}y = e_1$, and $x_{j+1} = R_{j+1}y$. Then it follows that $x_{j+1} = x$, i.e., we have arrived at the exact solution, since $Ax_{j+1} - b = AR_{j+1}y - b = R_{j+1}T_{j+1}y - b = R_{j+1}e_1 - b = 0$ (we have assumed that $x_0 = 0$).

The **Conjugate Gradients** method [19] is merely a variant on the above approach, which saves storage and computational effort. For, when solving the projected equations in the above way, we see that we have to save all columns of R_i throughout the process in order to recover the current iteration vectors x_i. This can be done cheaper. If we assume that the matrix A is in addition **positive definite** then, because of the relation

$$R_i^T A R_i = R_i^T R_i T_i,$$

we conclude that T_i can be transformed by a rowscaling matrix $R_i^T R_i$ into a positive definite symmetric tridiagonal matrix

Exercise 5.9 Prove that $R_i^T A R_i$ is positive definite. Hint: note that this matrix is an $(i+1)$ by $(i+1)$ matrix and note that by definition $(By, y) \neq 0$ for any $y \neq 0$, for a positive definite matrix B. □

This implies that T_i can be LU decomposed without any pivoting:

$$T_i = L_i U_i,$$

with L_i lower unit bidiagonal and U_i upper bidiagonal.
This property is used in the implementation of the Conjugate Gradients method. It turns out that the 3-term recurrence relation (corresponding to T_i) can be replaced by two 2-term recurrences (corresponding to the factors L_i and U_i), and that only the last residual and an update vector for the approximate solution have to be stored in memory.

The convergence of the Conjugate Gradients method depends on spectral properties of the matrix A. Therefore, one often applies the method in combination with a preconditioner $K = LL^T$, in order to improve the convergence behaviour, i.e., one applies the method to the system

$$L^{-1}AL^{-T}y = L^{-1}b, \quad \text{with} \quad x = L^{-T}y. \tag{5.13}$$

The following computational scheme solves $Ax = b$ with preconditioned CG and with preconditioner K. By a proper transformation we have reformulated the scheme so that the approximate solution x_i, and the corresponding residual (for the unpreconditioned equations) are delivered immediately.

x_0= initial guess; $r_0 = b - Ax_0$;
$p_{-1} = 0; \beta_{-1} = 0$;
Solve w_0 from $Kw_0 = r_0$;
$\rho_0 = (r_0, w_0)$
for $i = 0, 1, 2, ...$
    $p_i = w_i + \beta_{i-1} p_{i-1}$;
    $q_i = Ap_i$;
    $\alpha_i = \frac{\rho_i}{(p_i, q_i)}$
    $x_{i+1} = x_i + \alpha_i p_i$;
    $r_{i+1} = r_i - \alpha_i q_i$;
    if x_{i+1} accurate enough then quit;
    Solve w_{i+1} from $Kw_{i+1} = r_{i+1}$;
    $\rho_{i+1} = (r_{i+1}, w_{i+1})$;
    $\beta_i = \frac{\rho_{i+1}}{\rho_i}$;
end;

Note that this formulation, which is quite popular, has the advantage that the preconditioner needs not to be split into factors, and it is also avoided to backtransform solutions and residuals, as is necessary when one applies CG to $L^{-1}AL^{-T}y = L^{-1}b$.

When A is not positive definite, but still symmetric, then the reduced system $T_i y = e_1$ should be solved other than by LU-decomposition of T_i. In SYMMLQ [28] this is done by an LQ-decomposition.

Still another approach is to find an x_i in the Krylov subspace for which $\|r_i\|_2$ is minimal. This can also be done by first creating an orthogonal basis by a 3-term recurrence, and then solve the projected system in a least-squares sense. The resulting scheme is slightly more complicated than the Conjugate Gradient scheme, and is known as the MINRES method[28].

Nonsymmetric problems

There are essentially three different ways to solve linear systems with a non-symmetric non-singular matrix, while maintaining some kind of orthogonality between the residuals:

1. Solve the normal equations $A^T Ax = A^T b$ with conjugate gradients
2. Make all the residuals explicitly orthogonal in order to have an orthogonal basis for the Krylov subspace
3. Construct a basis for the Krylov subspace by a 3-term biorthogonality relation

The first solution seems rather obvious. However, it has severe disadvantages because of the squaring of the condition number. This has as effects that the

solution is more susceptible to errors in the right-hand side and that the rate of convergence of the CG procedure is much slower than for a comparable symmetric system with a matrix with the same condition number as A. Moreover, the amount of work per iteration step, necessary for the matrix vector product, is doubled. Nevertheless, the method may be attractive in certain situations, e.g., when the systems are overdetermined or underdetermined. For details, see [29, 39].

GMRES The second approach is to form explicitly an orthonormal basis for the Krylov subspace. Since A is not symmetric we no longer have a 3-term recurrence relation for that purpose and the new basis vector has to be made explicitly orthonormal with respect to all the previous vectors:

$$v_1 = \frac{1}{\|r_0\|_2} r_0,$$

$$h_{i+1,i} v_{i+1} = A v_i - \sum_{j=1}^{i} h_{j,i} v_j.$$

As in the symmetric case this can be exploited in two different ways. The orthogonality relation can either be written as

$$AV_i = V_i H_i + h_{i+1,i} v_{i+1} e_i^T, \tag{5.14}$$

after which the projected system, with a Hessenberg matrix instead of a tridiagonal matrix as in the symmetric case, can be solved (non-symmetric CG, GENCG, FOM, Arnoldi's method), or it can be written as

$$AV_i = V_{i+1} \bar{H}_i, \tag{5.15}$$

after which the projected system, with an $i+1$ by i upper Hessenberg matrix can be solved as a least squares system. In GMRES [33] this is done by the QR method using Givens rotations in order to annihilate the subdiagonal elements in the upper Hessenberg matrix $\bar{H}_i$.
The first approach (based upon (5.14)) is similar to the conjugate gradient approach (or SYMMLQ), the second approach (based upon (5.15)) is similar to the conjugate directions method (or MINRES).

Below we give a scheme for GMRES which may be suitable to develop a computer code. It solves $Ax = b$, with a given preconditioner K. In order to restrict on memory storage, the method is restarted after each cycle of m iteration steps. This variant is usually referred to as GMRES(m).

x_0 is an initial guess;
for $j = 1, 2, ...$
    Solve r from $Kr = b - Ax_0$;

$v_1 = r/\|r\|_2$;
$s := \|r\|_2 e_1$;
for $i = 1, 2, ..., m$
  Solve w from $Kw = Av_i$;
  for $k = 1, ..., i$ orthogonalisation of w
    $h_{k,i} = (w, v_k)$; against v's, by modified
    $w = w - h_{k,i} v_k$; Gram-Schmidt process
  end k;
  $h_{i+1,i} = \|w\|_2$;
  $v_{i+1} = w/h_{i+1,i}$;
  apply $J_1, ..., J_{i-1}$ on $(h_{1,i}, ..., h_{i+1,i})$;
  construct J_i, acting on i–th and $(i+1)$–st component
  of $h_{.,i}$, such that $(i+1)$–st component of $J_i h_{.,i}$ is 0;
  $s := J_i s$;
  if $s(i+1)$ is small enough then (UPDATE$(\tilde{x}, i)$; quit);
end i;
UPDATE$(\tilde{x}, m)$;
end j;

In this scheme UPDATE$(\tilde{x}, i)$ replaces the following computations:

Compute y as the solution of $Hy = \tilde{s}$, in which
the upper i by i triangular part of H has $h_{i,j}$ as
its elements (in least squares sense if H is singular),
$\tilde{s}$ represents the first i components of s;
$\tilde{x} = x_0 + y_1 * v_1 + y_2 v_2 + ... + y_i v_i$;
s_{i+1} equals $\|b - A\tilde{x}\|_2$;
if this component is not small enough
then $x_0 = \tilde{x}$;
else quit;

Bi-Conjugate Gradients The third class of methods arises from the attempt to construct a suitable set of basis vectors for the Krylov subspace by a three-term recurrence relation as in (5.11):

$$\tilde{\alpha}_{j+1} r_{j+1} = Ar_j - \tilde{\beta}_j r_j - \tilde{\gamma}_j r_{j-1}. \tag{5.16}$$

The orthogonality of such a set of vectors can be obtained when the matrix A is symmetric. In the non-symmetric case we need that

$$(Ar_{j-1}, r_k) = (r_{j-1}, A^T r_k) = 0 \quad \text{for} \quad k < j - 2.$$

By similar arguments as in the symmetric case we conclude that (5.16) can be used to generate a basis r_0,...,r_{i-1} for $K^i(A; r_0)$, such that $r_j \perp K^{j-1}(A^T; r_0)$, or even more general,

$$r_j \perp K^{j-1}(A^T; s_0),$$

since there is no explicit need to generate the Krylov subspace for A^T with r_0 as the starting vector.
If we let the basis vectors s_j for $K^i(A^T; s_0)$ satisfy the same recurrence relation as the vectors r_j, i.e., with identical recurrence coefficients, then we see that

$$(r_k, s_j) = 0 \quad \text{for} \quad k \neq j$$

(by a simple symmetry argument).
Hence, the sets $\{r_j\}$ and $\{s_j\}$ satisfy a *biorthogonality* relation. Now we can proceed in a similar way as in the symmetric case:

$$AR_i = R_i T_i + \tilde{\alpha}_i r_i e_i^T, \tag{5.17}$$

but now we use the matrix $S_i = [s_0, s_1, ..., s_{i-1}]$ for the projection of the system

$$S_i^T (Ax_i - b) = 0,$$

or

$$S_i^T A R_i y - S_i^T b = 0.$$

Using (5.17) we find that y_i satisfies

$$S_i^T R_i T_i y = (r_0, s_0) e_1.$$

Since $S_i^T R_i$ is a diagonal matrix with diagonal elements (r_j, s_j), we find, if all these diagonal elements are non-zero, that

$$T_i y = e_1 \quad \Rightarrow \quad x_i = R_i y.$$

This method is known as the Bi-Lanczos method[26].
We see that we are in problems when a diagonal element of $S_i^T R_i$ becomes (nearly) zero: this is referred to in literature as a serious (near) breakdown. The way to get around this difficulty is the so-called look-ahead strategy [27]. Another way to avoid break-down is to restart as soon as a diagonal element gets small. Of course, this strategy looks surprisingly simple, but one should realise that at a restart the Krylov subspace, that has been built up so far, is thrown away which destroys possibilities for faster (i.e., superlinear) convergence.

As is the case for Conjugate Gradients, the LU decomposition of the tridiagonal system can be used to obtain nice short update recurrences for the approximate solution and the corresponding residual. This avoids the need to save all intermediate r and s vectors. This variant of Bi-Lanczos is usually called Bi-Conjugate Gradients, or shortly Bi-CG [12].
Of course one can in general not be sure that an LU decomposition (without pivoting) of the tridiagonal matrix T_i exists, and this may lead to a serious break-down of the Bi-CG algorithm. Note that this break-down can be avoided in the Bi-Lanczos formulation of this iterative solution scheme.

Note that for symmetric matrices Bi-Lanczos generates the same solution as Lanczos, provided that $s_0 = r_0$, and under the same condition, Bi-CG delivers the same iterands as CG, for positive definite matrices. However, the Bi-orthogonal variants do so at the cost of two matrix-vector operations per iteration step.

It is difficult to make a fair comparison between GMRES and Bi-CG. GMRES really minimises a residual, but at the cost of increasing work for keeping all residuals orthogonal and increasing demands for memory space. Bi-CG does not minimise a residual, but often it convergences about as fast as GMRES, at the cost of twice the amount of matrix-vector products per iteration step. However, the generation of the basis vectors is relatively cheap and the memory requirements are limited and modest. Several variants of Bi-CG have been proposed which increase the effectiveness of this class of methods in certain circumstances. We mention CGS [37], BiCGSTAB [40], QMR [14], and BiCGSTAB(ℓ) [34]. For popular descriptions and implementations of these methods, as well as for guidelines for their usage, see [1].

The following scheme may be used for a computer implementation of the Bi-CG method. In the scheme the equation $Ax = b$ is solved with a suitable preconditioner K.

x_0 is an initial guess; $r_0 = b - Ax_0$;
solve w_0 from $Kw_0 = r_0$;
$\tilde{r}_0$ is an arbitrary vector such that $(w_0, \tilde{r}_0) \neq 0$,
usually one chooses $\tilde{r}_0 = r_0$ or $\tilde{r}_0 = w_0$;
solve $\tilde{w}_0$ from $K^T\tilde{w}_0 = \tilde{r}_0$;
$p_{-1} = \tilde{p}_{-1} = 0; \beta_{-1} = 0; \rho_0 = (w_0, \tilde{r}_0)$;
for $i = 0, 1, 2, ...$
  $p_i = w_i + \beta_{i-1}p_{i-1}$;
  $\tilde{p}_i = \tilde{w}_i + \beta_{i-1}\tilde{p}_{i-1}$;
  $z_i = Ap_i$;
  $\alpha_i = \frac{\rho_i}{(\tilde{p}_i, z_i)}$;
  $r_{i+1} = r_i - \alpha_i z_i$;
  $\tilde{r}_{i+1} = \tilde{r}_i - \alpha_i A^T\tilde{p}_i$;
  solve w_{i+1} from $Kw_{i+1} = r_{i+1}$;
  solve $\tilde{w}_{i+1}$ from $K^T\tilde{w}_{i+1} = \tilde{r}_{i+1}$;
  $\rho_{i+1} = (\tilde{r}_{i+1}, w_{i+1})$;
  $x_{i+1} = x_i + \alpha_i p_i$;
  if x_{i+1} is accurate enough then quit;
  $\beta_i = \frac{\rho_{i+1}}{\rho_i}$
end

Parallel Aspects

In this section we discuss briefly parallel aspects of iterative methods for solving large linear systems. From the discussions it should be clear how to combine coarse-grained and fine-grained approaches, for example when implementing a method on a parallel machine with vector processors. The implementation for such machines, in particular those with shared memory, is given much attention in [10]. For a more detailed discussion of iterative methods for linear systems on parallel architectures, see [8].

Parallelism in the kernels of iterative methods The basic time-consuming computational kernels of iterative schemes are usually:

1. innerproducts,
2. vector updates,
3. matrix-vector products, like Ap_i (for some methods also $A^T p_i$),
4. preconditioning (e.g., solve for w in $Kw = r$).

The innerproducts can be easily parallelised; each processor computes the innerproduct of two segments of each vector (local inner products or LIPs). On distributed-memory machines the LIPs have to be sent to other processors in order to be reduced to the required global inner product. This step requires communication. For shared-memory machines the innerproducts can be computed in parallel without problem. If the distributed-memory system supports overlap of communication with computation, then we have to find opportunities in the algorithm to do so. In the standard formulation of most iterative schemes this is usually a major problem.
Vector updates are trivially parallelizable: each processor updates its 'own' segment.
The matrix-vector products are often easily parallelised on shared-memory machines, by splitting the matrix in strips, corresponding to the vector segments. Each processor takes care of the matrix-vector product of one strip. For distributed-memory machines there may be a problem if each processor has only a segment of the vector in its memory. Depending on the bandwidth of the matrix we may need communication for other elements of the vector, which may lead to communication problems. However, many sparse matrix problems are related to a network in which only nearby nodes are connected. In such a case it seems natural to subdivide the network, or grid, in suitable blocks and to distribute these blocks over the processors. When computing Ap_i each processor needs at most the values of p_i at some nodes in neighbouring blocks. If the number of connections to these neighbouring blocks is small compared to the number of internal nodes, then the communication time can be overlapped with computational work.

The preconditioning part is often the most problematic part in a parallel environment. Incomplete decompositions of A form a popular class of preconditionings, in the context of solving discretised PDEs. In this case the preconditioner $K = LU$, where L and U have a sparsity pattern equal or close to the sparsity pattern of the corresponding parts of A (L is lower triangular, U is upper triangular). Solving $Kw = r$ leads to solving successively $Lz = r$ and $Uw = z$. These triangular solves lead to recurrence relations which are not easily parallelised. We will now discuss a number of approaches to obtain parallelism in the preconditioning part; for more details we refer to [10].

1. *Reordering the computations.* Depending on the structure of the matrix a *frontal approach* may lead to successful parallelism. By inspecting the dependency graph one can select those elements that can be computed in parallel. For instance, if a second order PDE is discretised by the usual five-point star over a rectangular grid, then the triangular solves can be parallelised if the computation is carried out along diagonals of the grid, instead of the usual lexicographical order. For vector computers this leads to a vectorizable preconditioner. For coarse-grained parallelism this approach is insufficient. By a similar approach more parallelism can be obtained in three-dimensional situations: the so-called *hyperplane approach*. The disadvantage is that the data need to be redistributed over the processors, since the grid points, which correspond to a hyperplane in the grid, are located quite irregularly in the array. For shared memory machines this also leads to reduced performance because of indirect addressing. In general one concludes that the data dependency approach is not adequate for obtaining a suitable degree of parallelism.

2. *Reordering the unknowns.* One may also use a *colouring scheme* for reordering the unknowns, so that unknowns with the same colour are not explicitly coupled. This means that the triangular solves can be parallelised for each colour. Of course, communication is required for couplings between groups of different colours. Simple colouring schemes, like red-black ordering for the five-point discretised Poisson operator, seem to have a negative effect on the convergence behaviour. Duff and Meurant [11] have carried out numerical experiments for many different orderings, which show that the numbers of iterations may increase significantly for other than lexicographical ordering.

3. *Forced parallelism.* Parallelism can also be forced by simply neglecting couplings to unknowns residing in other processors. This is like block Jacobi preconditioning, in which the blocks may be decomposed in incomplete form. Again, this may not always reduce the overall solution time, since the effects of increased parallelism are more than undone by an increased number of iteration steps.

The problems with parallelism in the preconditioner have led to searches for other preconditioners. Often simple diagonal scaling is an adequate preconditioner and this is trivially parallelizable. Often this approach leads to a significant increase in iteration steps. Still another approach is to use polynomial preconditioning: $w = p_j(A)r$, i.e. $K^{-1} = p_j(A)$, for some suitable jth degree polynomial. This preconditioner can be implemented by forming only matrix-vector products, which, depending on the structure of A, are easier to parallelise. For p_j one often selects a Chebychev polynomial, which requires some information on the spectrum of A. Finally we point out the possibility of using the truncated Neumann series for the approximate inverse of A, or parts of L and U.

5.2.6 Parallel Runge-Kutta methods for ODEs

Introduction

In this section we will concentrate on numerical methods for the initial value problem (IVP) for the ordinary differential equation (ODE), written in the autonomous form

$$y'(t) = f(y(t)), \;\; 0 \leq t \leq T, \;\; y \in \mathbb{R}^N, \;\; f : \mathbb{R}^N \rightarrow \mathbb{R}^N. \tag{5.18}$$

Although parallel computers are available now for quite a few years, it is remarkable that the construction of parallel methods for (5.18) received only marginal attention and in fact is still in its infancy. A possible explanation may be that the integration of an IVP by a step-by-step process is sequentially in nature and thus offers little scope to exploit parallelism.

Nevertheless, there are some avenues.

(i) At first, there is the rather obvious way to distribute the various components of the system of ODEs amongst the available processors. This is especially effective in *explicit* methods, since they frequently need the evaluation of the right-hand side function f for a given vector y, so that the components of f can be evaluated independently of one another. This is called *parallelism across the problem*.

(ii) A more interesting approach, called *parallelism across the method*, is to employ the parallelism inherently available within the method. Concurrent evaluations of the vector function f for various values of its argument and the simultaneous solution of various (non-linear) systems of equations are examples of parallelism across the method. Remark that this form of parallelism is also effective in case of a scalar ODE (i.e., $N = 1$ in (5.18)), whereas parallelism across the problem aims at large N-values. Also notice that both approaches can be combined because they are more or less independent.

(iii) Still another approach, which could be termed *parallelism across the time*, is to perform a number of time steps simultaneously, thus calculating numerical approximations in many points on the t-axis in parallel. These methods belong to the class of so-called *waveform relaxation* methods. Experiments have shown that a significant speed-up can be obtained by this approach provided that the number of time steps is (very) large. In this section we will confine ourselves to parallelism across the method.

For the numerical integration of ODEs, two classes of methods are quite popular: *Runge-Kutta methods* and *linear multistep methods*. The codes based on these classes of methods have reached a high level of sophistication and perform well on sequential computers. However, since the underlying algorithms have been designed in the "sequential era", these methods do not allow for parallelism across the method. Therefore, *new* methods have been constructed, specifically designed to take profit from a parallel configuration. Here, we will concentrate on parallel methods of Runge-Kutta type; however, parallel methods of linear multistep type have been constructed as well. For these type of methods, the interested reader is referred to [4, 5, 35] and the references quoted overthere.

It should be remarked that in constructing parallel methods, it is often unavoidable to introduce some redundancy in the total volume of computational arithmetic. Hence, compared with a good sequential Runge-Kutta method, it is overambitious to expect a speed-up in the solution time with a factor s, if s processors are available.

Finally, we remark that in many of the parallel Runge-Kutta methods considered in this section, a *small* number of concurrent subtasks of *considerable* computational complexity can be distinguished. Consequently, (i) these methods are aiming at so-called "coarse-grain" parallelism and (ii) communication and synchronisation overhead will be small compared with CPU time.

Nonstiff- and stiff-problems, stability, accuracy

In the construction of new algorithms we have to distinguish between two types of IVPs, viz., those which are called *stiff*, meaning that the solution consists of both slowly and rapidly varying components (like in chemical reactions), and those which do not have this property, termed *non-stiff* (like in the orbit equations of celestial bodies and in robotics). An example of a stiff equation is $y' = -100y + 100$, $y(0) = y_0 = 2$, with exact solution $y(t) = \exp(-100t) + 1$. Between $t = 0$ and, say, $t = 0.1$ the solution rapidly decreases from $y_0 = 2$ to its limiting value 1. Beyond $t = 0.1$ the solution varies slowly, and is essentially equal to 1.

Stiff and non-stiff types of IVPs require completely different numerical treatments. Most simple are the so-called *explicit methods*, characterised by the fact that only evaluations of the right-hand side function f are needed. The simplest explicit method is the *Euler method* which defines the approximation y_{n+1} to

the solution at t_{n+1} by $y_{n+1} = y_n + hf(y_n)$, where $h = t_{n+1} - t_n$. In the case of non-stiff problems, an explicit method is an appropriate choice. However, when we have to deal with stiff problems, these methods are very inefficient, since unavoidable rounding errors and discretisation errors will be accumulated from step to step. This accumulation, called *numerical instability*, becomes more pronounced as the stiffness increases, and may easily destroy the accuracy of the numerical solution after a few time steps. There are two remedies to bound the accumulation of errors: first one could reduce the stepsize; for many stiff problems this leads to such a stringent restriction of the time step that this remedy is not feasible from a practical point of view. A better approach is to resort to *implicit methods*, which can be given much better stability properties than explicit methods. The simplest example of an implicit method is the *backward Euler method* which is given by $y_{n+1} = y_n + hf(y_{n+1})$. The difference with the explicit Euler method is that f is evaluated at y_{n+1} rather than at y_n. As a consequence, a (generally) non-linear system of equations has to be solved in each time step to find the new approximation y_{n+1}.

Another aspect which is relevant both for stiff and non-stiff solvers is the *order of accuracy*. Obviously, the accuracy depends on the stepsize; the order of accuracy of a method is p if the error in the numerical solution behaves (asymptotically) as the stepsize to the power p. Many problems, especially those with a sufficiently smooth solution, are most efficiently solved by a high-order method.

In the sequel we shall mention various methods like Gauß-Legendre and Radau methods, and various types of stability like A- and L-stability, and $A(\alpha)$- and $L(\alpha)$-stability. Since the emphasis in this section is on *parallel* aspects of Runge-Kutta methods, we shall (and need) not explain these concepts here. The interested reader is referred to [18].

Parallel Runge-Kutta methods

The general Runge-Kutta (RK) method to proceed the numerical solution of (5.18) from $t = t_n$ over a step h is given by

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i f(Y_i), \tag{5.19}$$

$$Y_i = y_n + h \sum_{j=1}^{s} a_{ij} f(Y_j), \quad i = 1, \cdots, s. \tag{5.20}$$

Here, $y_n \approx y(t_n)$; a_{ij} and b_i are the coefficients defining the RK method, and s is called the number of stages. The quantities Y_i, the stage values, can be considered as approximations to the solution y at intermediate points. An RK method is said to be explicit if $a_{ij} = 0$, $j \geq i$ (i.e., Y_i only depends on

$Y_1, \ldots, Y_{i-1}$). Otherwise, it is called an implicit RK (IRK) method. For the algorithms described here, our starting point will always be an IRK method.

A nice feature of IRK methods is that a high order of accuracy can be combined with excellent stability properties [2]. Well-known examples of such IRKs are the so-called Gauß-Legendre methods (order $2s$ and A-stable) and the Radau IIA methods (order $2s-1$ and L-stable). A serious disadvantage however, is the high cost of solving the algebraic equations (5.20) defining the stage values Y_i. Since the Y_i are coupled in general, this is a system of dimension $s \cdot N$, thus involving $\mathcal{O}((s \cdot N)^3)$ arithmetic operations. This is the main reason that IRK methods have not received great popularity to serve as the basis for efficient, production oriented software. Several remedies have been proposed to reduce the amount of linear algebra per step; however, these variants have their own disadvantages and did not succeed in turning IRK methods into widely-used integration techniques. Another more promising possibility to realise the excellent prospects that IRK methods offer, is the use of *parallel processors*.

Motivated by the starting point that parallelism across the method should also be effective for scalar ODEs, we will assume throughout that (5.18) is a *scalar* equation. This has the notational advantage that we can avoid tensor products in our formulation. However, the extension to systems of ODEs, and therefore to non-autonomous equations, is straightforward.

In describing the parallel methods, it will be convenient to use a compact notation for the RK method (5.19) and (5.20). Introducing $A = (a_{ij})$, $\boldsymbol{b} = (b_i)$, $\boldsymbol{Y} = (Y_i)$ and $\boldsymbol{e} = (1, \ldots, 1)^{\mathrm{T}}$, all of dimension s, a succinct notation of the RK method reads

$$y_{n+1} = y_n + h\boldsymbol{b}^{\mathrm{T}} f(\boldsymbol{Y}), \tag{5.21}$$

$$\boldsymbol{Y} = y_n \boldsymbol{e} + hAf(\boldsymbol{Y}), \tag{5.22}$$

where $f(\boldsymbol{v}) := (f(v_j))$, for a given vector $\boldsymbol{v} = (v_j)$. The main problem in the application of an IRK is the solution of (5.22) for the stage vector $\boldsymbol{Y}$; once this vector has been obtained, (5.21) is straightforward. A direct treatment to solve (5.22) (i.e., applying some form of modified Newton iteration) offers little scope to exploit parallelism, except for the linear algebra part, but this aspect is not discussed here. To solve $\boldsymbol{Y}$ from (5.22), we use the iteration process

$$\boldsymbol{Y}^{(j)} - hDf(\boldsymbol{Y}^{(j)}) = y_n \boldsymbol{e} + h[A-D]f(\boldsymbol{Y}^{(j-1)}), \quad j = 1, \ldots, m. \tag{5.23}$$

Here, m is the number of iterations and D is a *diagonal* matrix. This is crucial, since now, given an iterate $\boldsymbol{Y}^{(j-1)}$, each individual component $Y_i^{(j)}$ of the unknown iterate $\boldsymbol{Y}^{(j)}$ has to be solved from an implicit relation of the form

$$Y_i^{(j)} - hd_i f(Y_i^{(j)}) - \Sigma_i = 0, \quad i = 1 \ldots, s, \tag{5.24}$$

where Σ_i is the ith component of the right-hand side vector in (5.23) and d_i is the ith diagonal entry of the matrix D. Clearly, all Σ_i depend on $\boldsymbol{Y}^{(j-1)}$,

but can be computed straightforwardly (even in parallel). The bulk of the computational effort involves the solution of the s equations for the components $Y_i^{(j)}$, $i = 1, \ldots, s$. However, given the Σ_i, the equations (5.24) are *uncoupled* and can be solved in parallel. Hence, assuming that we have s processors available, each iteration in (5.23) requires *effectively* the solution of only one implicit relation of the form (5.24). This is especially advantageous in case of (large) systems of ODEs, because then each iteration in (5.23) requires effectively the solution of a system of dimension N, the ODE dimension. As a consequence, the total iteration process has the effect that the solution of one system of dimension $s \cdot N$ has been transformed into the solution of a sequence of m systems, all of dimension N. Moreover, since D is the same in all iterations, the (parallel) LU decompositions of the matrices $I - hd_i\partial f/\partial y$ can be restricted to the first iteration. Summing up, the total computational complexity of the iteration process is $\mathcal{O}(N^3+mN^2)$, whereas a direct treatment requires $\mathcal{O}(s^3N^3+Ms^2N^2)$, with M the number of (modified) Newton iterations required. Since typical s-values range from 2 to 6 and because the required number of iterations m turns out to be quite modest, we now arrive at a manageable level of arithmetic.

Henceforth, the above Parallel Diagonally-Iterated RK methods will be denoted by *PDIRK* methods.

To start the iteration (5.23), we need the initial approximation $\boldsymbol{Y}^{(0)}$. One of the possibilities to choose this vector is given by

$$\boldsymbol{Y}^{(0)} - hBf(\boldsymbol{Y}^{(0)}) = y_n\boldsymbol{e}. \tag{5.25}$$

Here, the matrix B will be chosen either zero or of *diagonal* form in order to exploit parallelism (in the same way as described for (5.23)). In the sequel, the initial approximation $\boldsymbol{Y}^{(0)}$ will be referred to as the *predictor*.

If m iterations have been performed with (5.23), then the new approximation at t_{n+1} is defined by (cf. (5.21))

$$y_{n+1} := y_n + h\boldsymbol{b}^{\mathrm{T}} f(\boldsymbol{Y}^{(m)}). \tag{5.26}$$

Once an underlying IRK method, henceforth called the *corrector*, has been selected (i.e., once $\boldsymbol{b}$ and A have been fixed in (5.21) and (5.22)), the freedom left in the iteration process (5.23, 5.24, 5.25) consists of the matrices B and D, and the number of iterations m.

With respect to the matrix D, we have several possibilities: first of all, there is the simplest choice, which sets D equal to the zero-matrix. Notice that this choice leads to an *explicit* iteration process and, consequently, the resulting scheme is only suitable for *non-stiff* equations. Choosing the "trivial" predictor $\boldsymbol{Y}^{(0)} = y_n\boldsymbol{e}$, the order behaviour of the resulting algorithm can be formulated as

Theorem 5.1 The method {(5.23) with $D = \mathrm{O}$, (5.25) with $B = \mathrm{O}$, (5.26)} is of order $\min\{p^*, m+1\}$, where p^* is the order of the corrector (5.21) and (5.22). □

Notice that this method is itself an explicit RK methods with $s \cdot m + 1$ stages. However, on a parallel machine, the *effective* number of stages equals only $m + 1$ (provided that s processors are available). This means that if the number of iterations $m \leq p^* - 1$, then we obtain an explicit RK method where the *number of effective stages equals the order*. This is an optimal result and compares favourably with the situation for classical (uniprocessor) explicit RK methods, where the number of stages increases faster than linearly if we want a high order.

Next we consider the case of *stiff* problems, leading us to *implicit* methods, i.e., to $D \neq \mathrm{O}$. Before specifying particular choices of D, we first want to discuss an aspect of the corrector which is relevant with respect to stiffness. In integrating stiff ODEs, a favourable property of the method is that it is "stiffly accurate". This means that the RK method satisfies $\boldsymbol{b}^{\mathrm{T}} = \boldsymbol{e}_s^{\mathrm{T}} A$, with $\boldsymbol{e}_s$ the sth unit vector. Hence, $\boldsymbol{b}^{\mathrm{T}}$ equals the last row of A, or equivalently, the last component of the stage vector $\boldsymbol{Y}$ is an approximation to the solution at the new step point t_{n+1}. Therefore, in case of a stiffly accurate corrector, (5.26) will be replaced by

$$y_{n+1} := \boldsymbol{e}_s^{\mathrm{T}} \boldsymbol{Y}^{(m)}. \tag{5.27}$$

Now, we return to the discussion of the matrix D; we distinguish two cases:

(i) D is such that after a *prescribed* number of iterations the resulting method has good stability properties.

(ii) Another option is to *solve* the corrector, i.e., to iterate (5.23) until convergence is obtained, and to choose D in such a way that convergence is as fast as possible.

In the following two subsections these cases will be briefly discussed.

Diagonal iteration with a prescribed number of iterations

The strategy to fix the number of iterations is motivated by the following theorem:

Theorem 5.2 Let p^* be the order of the underlying corrector (5.21) and (5.22). Then the order p of the resulting PDIRK method {(5.23, 5.24 and 5.25), (5.26), (5.27)} is given by

$$\begin{array}{ll} \min\{p^*, m + r\} & \text{for all matrices } B \text{ and } D, \\ \min\{p^*, m + 1 + r\} & \text{if } B\boldsymbol{e} = A\boldsymbol{e}, \\ \min\{p^*, m + 2 + r\} & \text{if, in addition, } BA\boldsymbol{e} = A^2\boldsymbol{e}, \end{array}$$

where r takes the value 1 if y_{n+1} is defined by (5.26) (i.e., the non-stiffly accurate case) and $r = 0$ if y_{n+1} is defined by (5.27) (the stiffly accurate case).

Furthermore, if the corrector is stiffly accurate, then the corresponding PDIRK method has the same property. □

Based on this theorem, the iteration is stopped as soon as the order has reached the order of the corrector, since a continuation of the iteration process would not increase the order of the PDIRK method. For the matrix B occurring in the predictor formula (5.23) we remark that $B = \mathrm{O}$ or $B = D$ are obvious choices. Although B and D may be different diagonal matrices, the choice $B = D$ has the computational advantage that the (expensive) LU decompositions of $I - d_i h \partial f/\partial y$, which are needed during the iteration of (5.23), can also be used in solving (5.25) for $\boldsymbol{Y}^{(0)}$.

The diagonal matrix D is still free and can be used to give the resulting PDIRK method optimal stability characteristics. We distinguish two approaches: matrices D with *constant* and with *varying* diagonal entries. In the first case, i.e., D is of the form $d \cdot I$, it is possible to perform a rather thorough stability analysis. It turns out that unconditionally stable PDIRK methods can be constructed. A few of these methods are listed in Table 5.1. The relevant d-values can be found in [22]. If we allow the matrix D to have *non-constant*

corrector	matrices B and D	attainable order p	# effective stages	stability
Gauß	$B = \mathrm{O}, D = d \cdot I$	$p \leq 4, p = 6$	$p - 1$	A-stable
Gauß	$B = D = d \cdot I$	$p \leq 6, p = 8$	p	L-stable
Radau II A	$B = \mathrm{O}, D = d \cdot I$	$p \leq 6, p = 8$	p	L-stable
Radau II A	$B = D = d \cdot I$	$p \leq 8, p = 10$	$p + 1$	L-stable

Table 5.1: *Unconditionally stable PDIRK methods with $D = d \cdot I$.*

entries, then it is possible to save one iteration without reducing the order, simply by setting $B = D = \mathrm{diag}(A\boldsymbol{e})$ (cf. Theorem 5.2). Some of the resulting PDIRK methods turn out to be only $A(\alpha)$-stable, however with α close to 90°. In Table 5.2, we collect a few methods with good stability properties.

corrector	attainable order p	# effective stages	stability
Gauß/Radau IIA	$p \leq 5$	$p - 1$	strongly A-stable
Gauß/Radau IIA	$p = 6, 7$	$p - 1$	$A(\alpha)$-stable, $\alpha > 83°$
Radau IIA	$p = 3, 5, 7$	p	$L(\alpha)$-stable, $\alpha > 89°$

Table 5.2: *PDIRK methods with a non-constant D-matrix.*

Diagonal iteration until convergence

PDIRK methods with a *fixed* number of iterations, as considered in the previous subsection, are in fact special DIRK methods. DIRK methods possess a so-called *stage order* equal to 1 which, in general, drastically reduces the accuracy. As a matter of fact, in many stiff problems the actually observed order equals the stage order (or, sometimes the stage order + 1). As a consequence of this so-called *order reduction* phenomenon, the relevance of methods with a high classical order but a low stage order is questionable. Therefore, apart from the "fixed-m-strategy" we also consider the approach where the corrector is iterated *until convergence*. This implies that we can rely on all the characteristics of the corrector, like stability and accuracy behaviour and, in particular, the stage order. For example, s-stage IRK methods of Gauß and Radau type both have stage order s.

Having decided to *solve* the corrector, we can now consider (5.23) as an *iteration* process, where "iteration" has the classical meaning. This leads us automatically to a criterion for choosing the matrix D: this matrix should be such that we have *fast convergence* in (5.23).

It is easy to show that the iteration error $\boldsymbol{Y} - \boldsymbol{Y}^{(j)}$, in first approximation, satisfies the recursion

$$\boldsymbol{Y} - \boldsymbol{Y}^{(j)} = Z(z)[\boldsymbol{Y} - \boldsymbol{Y}^{(j-1)}], \quad j = 1, \ldots, m, \quad z := h\lambda, \tag{5.28}$$

where the iteration matrix Z is defined by

$$Z(z) := zD[I - zD]^{-1}[D^{-1}A - I]. \tag{5.29}$$

Here, λ denotes an approximation to the derivative $\partial f/\partial y$ and should be understood to run through the spectrum of the Jacobian matrix in case of systems of ODEs. The convergence behaviour of (5.23) is determined by the iteration matrix Z and we have the matrix D at our disposal to obtain fast convergence.

The main difficulty in choosing D is that Z depends on z, i.e., on the problem. Therefore, we cannot expect to find a uniformly "best" D-matrix. In [21], several possibilities are discussed to obtain a D-matrix leading to a satisfactory convergence behaviour.

A numerical example

To obtain insight in the actual performance of these parallel Runge-Kutta methods, we have tested a parallel implementation of a PDIRK method based on the strategy described above. For the corrector, we selected the 4-stage Radau IIA method. The predictor $\boldsymbol{Y}^{(0)}$ is obtained by extrapolating approximations obtained in the preceding step. It is to be expected that this will result in fewer iterations compared with the "trivial" predictor $\boldsymbol{Y}^{(0)} = y_n\boldsymbol{e}$. We equipped this method with a provisional strategy for error control and stepsize selection

(details concerning the implementation strategy can be found in [36]). The resulting code is termed PSODE.

We have implemented PSODE on the Alliant FX/4 computer (four parallel processors and shared memory) and applied it to several test problems. The goal of these tests is twofold: (i) we want to investigate to what extent the theoretical parallelisation can be realised in practice; in other words, how close we can approach the ideal speed-up factor 4 on this four-processor machine and (ii) we want to compare the performance of the code PSODE with that of a good sequential solver. To that purpose we select the recent (sequential) code RADAU5 of Hairer & Wanner [18]. This choice is motivated by the observation that it solves a Radau IIA method (viz., the 3-point, 5th-order one); this starting point is quite similar to that of PSODE, although the approach to obtain the Radau-solution is completely different. Furthermore, we included in our tests the code LSODE of Hindmarsh [20]. This code (based on the BDF formulas which are of linear multistep type) has formulas up to order five available, from which only those of first and second order are A-stable. Hence, LSODE is less robust as a *general* stiff solver, but, on the other hand, it is generally accepted as a good sequential solver and enjoys considerable usage over a long period.

In comparing the parallel code PSODE with the two sequential codes, we do *not* take into account effects originating from a possible "parallelisation over the loops". By this we mean that a long loop is cut into s smaller parts which are then assigned to the s processors. In the Introduction on page 42 this effect is termed "parallelism across the problem" and can in fact be used by any ODE solver. Here we merely want to test *intrinsic* parallelism (called "parallelism across the method"). In order to exclude the effects of "parallelism across the problem", LSODE and RADAU5 are run on a single processor. In fact, the amount of intrinsic parallelism offered by LSODE and RADAU5 is very modest.

Of course, if one is interested in "parallelism across the problem", then the sequential codes could be implemented on an s-processor machine. However, in that case a fair comparison would require assigning $4s$ processors to PSODE, since in each of the four concurrent subtasks of PSODE, the "parallelism across the problem" can equally well be exploited (cf. the Introduction on page 42, where we have mentioned that both parallelisation techniques are independent).

Summarizing, we may say that PSODE needs four times the number of processors given to a sequential code, simply because it possesses a 4-fold amount of intrinsic parallelism. The large number of processors utilised by PSODE reflects the current tendency in parallel computing, since modern architectures − and certainly those entering the market in the coming years − have an "almost unlimited" number of processors (*massive parallelism*).

Another aspect which is of utmost importance for the performance of a stiff code, is the amount of linear algebra per step, which in turn strongly depends on the dimension of the ODE. Prior to the discussion of our test problem, we will briefly comment on the characteristics of the various codes with respect to this aspect:

A common feature of the three codes is that they need from time to time an LU decomposition of the matrix involved in their respective iteration processes to solve the non-linear relations. Since the factorisation of a general N-dimensional matrix requires approximately $2N^3/3$ arithmetic operations, this will dominate the total costs of the integration for *large-scale problems*. In this connection we remark that both `LSODE` and `PSODE` deal with matrices of dimension N. Hence, it is to be expected that their mutual comparison is only marginally influenced if N increases and all other aspects are left unchanged.

Matters are different for the code `RADAU5`, since it has to deal with matrices of dimension $3N$. By exploiting the special structures in these matrices, Hairer and Wanner are able to reduce the total work of the LU decomposition to $10N^3/3$ operations [18], thus gaining a factor 5 compared with a direct treatment, which would have required $2(3N)^3/3$ operations. However, this number $10N^3/3$ compares unfavourably with the number $2N^3/3$ (associated with `LSODE` and `PSODE`), and causes a serious drawback for `RADAU5` when applied to large-scale problems.

To get an indication of the performances of the codes, we have applied them to a test problem originating from circuit analysis. This problem, which is extensively discussed in [17, p. 112], describes a ring modulator, that mixes a low frequency and a high frequency signal, resulting in a heavily oscillating solution. The resulting stiff system consists of 15 ODEs, some of which are strongly non-linear. Not yet fixed is the value of the capacity C_s. In our test, we give it the value 10^{-9}, which seems technically meaningful. It is reported in [17] that small C_s-values cause serious difficulties. In the limit, i.e., on setting $C_s \equiv 0$, we end up with a differential-algebraic system. The integration interval in our test is $[0,\ 10^{-3}]$. For several values of `TOL` (the local error bound) the results obtained by the codes `RADAU5`, `LSODE` and `PSODE` are collected in Table 5.3. Here, T_1 and T_4 denote the CPU time (in seconds) when the program is run on one and four processors, respectively. Recall, that we restrict the timings for the sequential codes to T_1. The accuracy is measured by means of Δ, which is defined by writing the maximum norm of the global (relative) error in the endpoint in the form $10^{-\Delta}$. Furthermore, *Nsteps* denotes the number of (successful) integration steps and $\overline{m}$ stands for the average number of (effective) iterations (and thus also f-evaluations) per step.

These results give rise to the following conclusions:

(i) with respect to our first goal, we see that the speed-up factor for `PSODE` (obviously defined by T_1/T_4) is approximately 3.7, which is pretty close to the "ideal" factor 4 on this machine. This factor rapidly converges to 4 if the dimension of the problem increases.

(ii) concerning our second goal, we observe a remarkable similarity between `RADAU5` and `PSODE`: both codes need approximately 7 iterations per step; moreover, to produce the same accuracy, the required number of steps

Method	TOL	$Nsteps$	$\overline{m}$	Δ	T_1	T_4
RADAU5	10^{-2}	1275	9.0	1.1	33.1	
	10^{-3}	2277	7.6	2.6	48.6	
	10^{-4}	3922	6.7	3.8	72.4	
	10^{-5}	6761	6.1	4.9	110.9	
LSODE	10^{-3}	7054	1.5	1.4	33.6	
	10^{-4}	9772	1.4	2.8	44.1	
	10^{-5}	13266	1.4	2.9	57.7	
	10^{-6}	17887	1.3	3.8	74.7	
	10^{-7}	23310	1.3	4.5	93.1	
	10^{-8}	30253	1.2	4.9	114.3	
PSODE	10^{-2}	1185	7.3	1.4	80.0	21.4
	10^{-3}	1561	7.3	3.1	104.5	27.8
	10^{-4}	2272	7.1	4.1	146.4	39.6
	10^{-5}	3437	6.9	5.2	212.1	57.7

Table 5.3: *Performance of the codes* RADAU5, LSODE, *and* PSODE *for the circuit problem.*

is of the same order of magnitude (for the more stringent values of TOL, the difference in the number of steps increases, which is probably due to the higher order of PSODE). There is however a striking difference between the two Radau-based codes and LSODE; this code is very cheap per step, but needs much more integration steps to produce the same accuracy. For example, to obtain a relative accuracy of about 5 digits, PSODE needs $\approx$ 3400 steps, RADAU5 twice as many, whereas for LSODE this number is 9 times as large. Taking into account the computational effort per step of the various codes, the comparison with PSODE yields a double amount of time both for LSODE and RADAU5. Approximately the same ratios are observed in the low-accuracy range (say, $\Delta = 3$).

As mentioned before, this example is only a model problem describing a small (part of an) electrical circuit, and is still far away from a real-life application. However, even for this small system of ODEs, the performance of (this provisional version of) PSODE is already superior by a factor of 2 to that of the (well-established) codes LSODE and RADAU5.

Summarizing, we can say that

– the PSODE-approach is much more promising to serve as the basis for an efficient, "all-purpose" stiff solver than the LSODE-approach. This is due to the improved mathematical qualities, viz., the high order in combination

with A-stability.

– In comparison with `RADAU5`, `PSODE` has the advantage that in large-scale problems, the (dominating) LU factorisations require a factor 5 less computational effort. In this connection we remark that a few preliminary experiments with a problem of dimension 75 reveal that the overall gain of `PSODE` is already more than a factor 4. For really large-scale problems we expect that the speed-up factor will be in the range 6–8, depending on the required accuracy. This number is composed of the asymptotic factor 5 coming from the linear algebra part and the remaining factor 1.2–1.6 originating from the higher order of `PSODE`.

Conclusions

It has been shown that iterating a *fully implicit RK* method leads in a natural way to parallel integration methods. This approach can be used both for stiff and non-stiff ODEs. Although it is *conceptually* not necessary to start with a fully implicit RK method, such IRKs are an excellent choice to serve as a method, underlying the iteration process.

In the *non-stiff* case, the Gauß methods are recommended because of their highly accurate behaviour. Moreover, the optimal order of these IRKs with respect to the number of stages, minimises the number of required processors. Following this approach, it is possible to construct explicit RK methods for which the (effective) number of stages equals the order. This property holds for an arbitrarily high order and is principally impossible within the class of sequential explicit RK methods.

For *stiff* equations, a stiffly accurate IRK is a good choice; in particular, Radau IIA methods are suitable candidates. In the stiff case, the parallel, diagonally-implicit iteration leads to methods with nice features, both from a computational and a mathematical point of view. The property that only one matrix of the ODE dimension has to be factorised per step, reduces the amount of linear algebra to an acceptable level. We have seen that performing a fixed number of iterations results in L-stable methods with a high classical order, but with a (at least, formally) low stage order. Alternatively, iterating until convergence yields a high classical order as well as a high stage order. Moreover, already after a modest number of iterations, these methods are unconditionally stable.

Exercise 5.10 For the explicit PDIRK methods (i.e., with $D = \mathrm{O}$), it is said (in the sentence following Theorem 5.1) that they belong to the class of explicit Runge-Kutta methods. Verify this statement, using the general definition (5.19) and (5.20). □

Exercise 5.11 Furthermore, it is said that these RK methods have $s \cdot m + 1$ stages. At first sight this is a bit surprising since m iterations of (5.23), each containing s stages, seem to result in $s(m+1)$ stages. Show that the number $s \cdot m + 1$ is correct indeed. □

Exercise 5.12 The central part of these explicit, parallel RK methods is given by the iteration (cf. (5.23)):

$$\mathbf{Y}^{(j)} = y_n \mathbf{e} + hAf(\mathbf{Y}^{(j-1)}), \quad j = 1, \ldots, m.$$

Write a piece of Fortran-code for this central part; choose your data structures. □

Exercise 5.13 Verify that for a stiffly accurate corrector (which satisfies $\mathbf{b}^{\mathrm{T}} = \mathbf{e}_s^{\mathrm{T}} A$), the step point formula (5.27) is the correct analogue of (5.26). □

Exercise 5.14 As said, the code RADAU5 is based on a 3-point (fully implicit) Radau IIA method. Hence, it requires in each step the solution of the system (cf. (5.22))

$$\mathbf{Y} - y_n \mathbf{e} - hAf(\mathbf{Y}) = 0.$$

The application of a modified Newton process requires the solution of linear systems of the form

$$\begin{pmatrix} I - a_{11}hJ_n & -a_{12}hJ_n & -a_{13}hJ_n \\ -a_{21}hJ_n & I - a_{22}hJ_n & -a_{23}hJ_n \\ -a_{31}hJ_n & -a_{32}hJ_n & I - a_{33}hJ_n \end{pmatrix} \mathbf{x} = \mathbf{b}, \tag{5.30}$$

where J_n denotes the Jacobian matrix $\frac{\partial f}{\partial y}(y_n)$, $\mathbf{x}$ denotes the correction in this particular Newton iteration, and $\mathbf{b}$ is a known righthand side vector (depending on the previous Newton iteration). Notice that this matrix is of dimension $3N$; hence, assuming that J_n is a full matrix, an LU factorisation of the matrix in (5.30) would require $\frac{2}{3}(3N)^3$ floating-point operations.

Hairer and Wanner suggest some transformations for the system (5.30) (details can be found in [18, p. 131]) ending up with the modified system

$$\begin{pmatrix} \gamma I - J_n & 0 & 0 \\ 0 & \alpha I - J_n & -\beta I \\ 0 & \beta I & \alpha I - J_n \end{pmatrix} \tilde{\mathbf{x}} = \tilde{\mathbf{b}}, \tag{5.31}$$

where γ and $\alpha \pm i\beta$ respectively are the real and complex eigenvalues of $h^{-1}A^{-1}$.

a. Show that the total work of the LU factorisation can be reduced to $\frac{10}{3}N^3$ by transforming the real subsystem of dimension $2N$ into the N-dimensional complex system

$$[(\alpha + i\beta)I - J_n](\tilde{x}_2 + i\tilde{x}_3) = \tilde{b}_2 + i\tilde{b}_3.$$

b. Show that on a 2-processor machine, this code possesses a small amount of inherent parallelism by which the effective number of operations can be further reduced to $\frac{8}{3}N^3$. □

5.2.7 Partial differential equations

We illustrate two important vectorisation/parallelisation techniques on rather simple iterative solution methods for partial differential equations. In general one uses (preconditioned) Conjugate Gradient methods, multigrid methods or GMRES, just to name three of the more popular methods, to solve PDEs (see also Section 5.2.5). On the other hand, the relaxation methods we discuss, can be used as preconditioner for Conjugate Gradient methods or as smoother for multigrid methods.

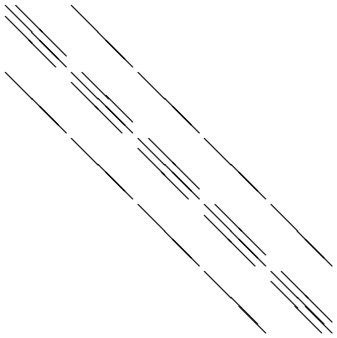
We consider the Poisson equation on an $n \times n$ grid

$$u_{xx} + u_{yy} = f.$$

Discretisation with central differences gives us

$$\frac{1}{h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) + \frac{1}{h^2}(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}) = f_{i,j}.$$

The resulting $n^2 \times n^2$ matrix of the discrete system will have the following sparse structure:



It is not very attractive to use a direct solver because of the fill-in in the original sparse matrix. The matrix will become dense between the two outer diagonals; so we will need much more storage but also the complexity $\mathcal{O}(n^4)$ will be more than the complexity for say multigrid $\mathcal{O}(n^2)$.

Two well-known iterative methods are Jacobi-relaxation and Gauß-Seidel-relaxation.

In a Jacobi step we compute a new approximation u' of the solution as follows:

for $j = 1$ to n
  for $i = 1$ to n
    $u'_{i,j} := -\frac{1}{4}(h^2 f_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1})$

It is clear that all computations are completely uncoupled: if we compute a new u-value for a grid-point we only use already existing information. The algorithm is completely vectorisable and by distributing the work over multiple processors also completely parallelisable. The (collapsed) loop-length is n^2.

From a numerical point of view we can do better: instead of using old information, we can use new information the moment it becomes available. A Gauß-Seidel step using lexicographical ordering looks as follows:

for $j = 1$ to n
    for $i = 1$ to n
        $u'_{i,j} := -\frac{1}{4}(h^2 f_{i,j} - u_{i+1,j} - u'_{i-1,j} - u_{i,j+1} - u'_{i,j-1})$

Now we obtain a better convergence, but the computations have become data dependent: for the computation of $u'_{i,j}$ we need the just computed value of $u'_{i-1,j}$.

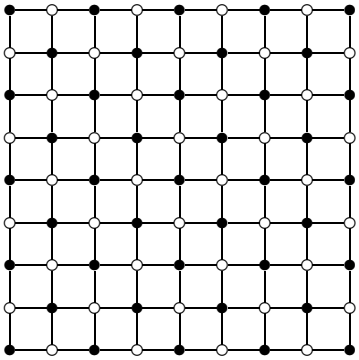
We will discuss two possibilities to improve on Jacobi relaxation with respect to both convergence and vectorisation/parallelisation: red-black relaxation and the hyperplane method. Both methods are based on re-ordering of the computations.

If we stay with Gauß-Seidel relaxation using the lexicographical ordering described above, we observe that we use already updated values from west-neighbours and south-neighbours in order to compute a new value. So if we perform our operations using a diagonal-ordering of the grid-points ($i + j = d$) we see that we only need the just updated values on the previous diagonal ($i + j = d - 1$) and old values on the current and next diagonal.

for $d = 2$ to $2n$
    for all $(i, j) \in \{(i, j) | i + j = d\}$
        $u'_{i,j} := -\frac{1}{4}(h^2 f_{i,j} - u_{i+1,j} - u'_{i-1,j} - u_{i,j+1} - u'_{i,j-1})$

The computations in the for all-loop are uncoupled so completely vectorisable and by distributing the work also parallelisable. We have $2n - 1$ vector/parallel loops with an average length of $n/2$. The outer loop is still data dependent. This technique is known as the hyperplane method.

Another possibility for improving on Jacobi relaxation is obtained by using a checkerboard (or red/black) ordering:



We notice that when we perform a Jacobi update of an u-value on a white grid-point we only use u-values on the neighbouring black grid-points and vice versa. So, the first half step we update all values on the white grid-points simultaneously using only u-values on black grid-points. The second half-step we use the — just updated — u-values on the white grid-points in order to update the u-values on black grid-points. It is clear, that every half-step is completely vectorisable/parallelisable with a loop-length of $n^2/2$. This loop-length is slightly worse than in the Jacobi-case, but much better than in the Gauß-Seidel-case.

Exercise 5.15 Construct a colouring-scheme for a 9-point discretised 2-dimensional PDE, (so $u_{i+1,j+1}$, $u_{i-1,j+1}$, $u_{i-1,j-1}$, and $u_{i+1,j-1}$ also occur in the discrete system) such that all work of a Gauß-Seidel like relaxation can be performed completely decoupled during the subsequent intermediate Jacobi steps (analogous to the red/black ordering for 5-point stencils). □

5.3 Nonnumerical algorithms

5.3.1 Sorting

Complete books, e.g. [23], are devoted to sorting and much work is done on parallel sorting methods and parallel sorting networks. In this subsection we discuss the vectorisation of quicksort [23] and we demonstrate a parallel sorting method.

At first sight, sorting does not seem to be a natural algorithm for vectorisation. Since we want to discuss vectorisation we restrict ourselves to sorting numerical data. Quicksort is an attractive sorting algorithm on scalar computers. Perhaps surprisingly this algorithm can be fully vectorised. Instead of giving a completely vectorisable algorithm in pseudo-code, we discuss the heart of the quicksort algorithm.

The basic idea is to pick a number in the array to be sorted and move this number to its final position in the sorted array. While determining this position, we move all smaller numbers to the left and all larger numbers to the right of the number we picked. Thus the array is partitioned in such a way, that we can apply the same algorithm recursively on the left part and on the right part of the array, until the whole array has been sorted.

For example, consider an array x of eight elements: 55 19 60 95 45 26 75 14. We pick the first element $x_1 = 55$ and use two pointers $i = 2$, and $j = 8$, initially. Increase i until x_i becomes greater than x_1; decrease j until x_j becomes less than x_1. At this point we interchange x_i and x_j. Repeat until j becomes less than i. Now we interchange x_1 and x_j. We end up with a partitioned array.

initial array	55	19	60	95	45	26	75	14
initial pointers		$i\uparrow$						$j\uparrow$
1^{st} exchange			14					60
2^{nd} exchange				26		95		
$j < i$					$j\uparrow$	$i\uparrow$		
partitioned array	[45	19	14	26]	55	[95	75	60]

Quicksort is sometimes called partition-exchange sorting. The searching in the exchanging part is fully vectorisable — in fact the complete partitioning can be rewritten in terms of compress operations. On the Cyber-205, it could even be performed using a special machine-instruction: the so-called arithmetic compress instruction.

Parallel sorting is a completely different story. The computational complexity for optimal sorting algorithms on a single processor is $\mathcal{O}(n \log n)$, whereas merging two sorted arrays, of length m and n, has complexity $\mathcal{O}(\lfloor \frac{m+n-1}{2} \rfloor)$. A divide-and-conquer technique can be applied. We consider a shared-memory two processor machine. Furthermore, we assume that the to-be-sorted array

has been divided in two parts x and y, of length m and n, respectively, and each processor has already (quick)sorted its own part. Trivially, sorting of x and y can be performed completely in parallel. For the following step we use a so-called "odd-even merge" algorithm.

Say we have $x_1, x_2, \ldots, x_m$ (quick)sorted by processor p_1, and $y_1, y_2, \ldots, y_n$ (quick)sorted by processor p_2. Now processor p_1 can merge the "odd sequences" $x_1, x_3, \ldots, x_{2\lceil m/2\rceil - 1}$ and $y_1, y_3, \ldots, y_{2\lceil n/2\rceil - 1}$ obtaining the sorted result $v_1, v_2, \ldots, v_{\lceil m/2\rceil + \lceil n/2\rceil}$; analogously processor p_2 can merge the "even sequences" yielding the sorted result $w_1, w_2, \ldots, w_{\lfloor m/2\rfloor + \lfloor n/2\rfloor}$. Both of these merges can be performed in parallel. Finally, both processors can work simultaneously on the comparison-interchange operations

$$v_1, w_1 : v_2, w_2 : v_3, w_3 : v_4, \ldots, w_{\lfloor m/2\rfloor + \lfloor n/2\rfloor}$$

the result will be sorted. For example, consider the following array:

14 75 95 45 26 60 19 55 96 57 06 89 79 25 44 89

divide it in two parts x and y:

x: 14 75 95 45 26 60 19 55
y: 96 57 06 89 79 25 44 89

(quick) sort both parts independently:

x: 14 19 26 45 55 60 75 95
y: 06 25 44 57 79 89 89 96

merge the odd subsequences on p_1:

v: 06 14 26 44 55 75 79 89

merge the even subsequences on p_2:

w: 19 25 45 57 60 89 95 96

compare-interchange using both processors:

06,
19:14, 25:26, 45:44, 57:55, 60:75, 89:79, 95:89,
96.

Finally, the sorted result will be:

06 14 19 25 26 44 45 55 57 60 75 79 89 89 95 96.

Exercise 5.16 Describe how we could use the (level 1 BLAS) ISAMAX routine (completely vectorisable) to sort an array of positive REALS. Analyse the complexity of this algorithm and compare the result with quicksort. □

5.3.2 Factorisation

Factorisation of integers is a classical problem in number theory. After the discovery, in 1978, by Rivest, Shamir and Adleman, of certain cryptosystems, the security of which depends on the difficulty of factoring large integers, the factorisation problem has enjoyed renewed and wide-spread interest. The *computational complexity* of factorisation is not known, but practical experience with the fastest known factoring methods indicates that the work to factorise a number N grows *exponentially* with N.

Since Euclid's time it has been known that any natural number has a unique prime power decomposition $N = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$ ($p_1 < p_2 < \dots < p_k$ being primes, and α_j being positive integers), and for many purposes one likes to know an efficient algorithm for finding this decomposition. Note that if we have an algorithm to find a non-trivial factor f of N this can be applied recursively to obtain the complete prime power decomposition of N. Of course, if N itself is a prime number, we are finished. From now on we assume that the number N we want to factorise is composite. This can be checked in practice very easily by selecting a (small) number a which has no common divisor with N, and by computing $b = a^{N-1} \bmod N$. If $b \neq 1$ then we are sure that N is composite. If $b = 1$, N will be prime in most, but not in *all* cases. The number b can be computed in $\mathcal{O}(\log(N))$ steps by looking at the binary representation of N. For details we refer to [31].

Some known factorisation algorithms are not suited, some others turn out to be very well-suited to parallelisation. Here we describe two simple factoring algorithms, "trial division" and "Pollard rho", and discuss their parallel implementation.

Trial division

Trial division is a straightforward factorisation algorithm. One just tries potential divisors $d = 2, 3, \dots$ until one of the following events occurs:

[N **is prime**] $d > N^{1/2}$, in which case N is prime; or

[N **is composite**] $d < N$ and $d|N$, in which case d is a non-trivial prime divisor of N; or

[**not yet finished**] d exceeds some preassigned bound $B < N^{1/2}$, in which case we only can say that any prime factor of N exceeds B.

Of course this can be refined, by just using the primes as trial divisors, but if B is large this can cause severe side problems of computing and storing these primes. A more simple refinement is possible if N is odd and $N \neq 0 \bmod 3$; in that case we can let d run through the integers $\equiv 1, 5 \bmod 6$, or $\equiv 1, 7, 11, 13, 17, 19, 23, 29 \bmod 30$.

The parallel implementation of trial division is straightforward. With P processors we can perform up to P trials in parallel. Thus, provided that $P \ll p$, where p is the smallest prime divisor of N, a linear speed-up is obtained.

Exercise 5.17 Write a program to factorise numbers with the trial method where the trial divisors d run through a sequence of integers which are not divisible by $2, 3, 5, 7$. Apply this to the number $N = 9904156957$. □

The Pollard "rho" algorithm

Pollard's "rho" algorithm uses an iteration of the form $x_{i+1} = f(x_i) \bmod N$, $i \geq 0$, where N is the number to be factorised, x_0 is a random starting value, and f is a polynomial with integer coefficients, for example, $f(x) = x^2 + a$ ($a \neq 0 \bmod N$).

Let p be the smallest prime factor of N, and j the smallest positive index such that $x_{2j} = x_j \bmod p$. We can expect that $j = \mathcal{O}(p^{1/2})$. The argument for this expectation is related to the well-known "birthday-paradox" (if you have a class with 30 children, the probability that at least two of them have the same birthday date is nearly 0.7). Of course, we do not know p in advance, but we can detect x_j by taking greatest common divisors as follows. We simply compute $\gcd(x_{2i} - x_i, N)$ for $i = 1, 2, \ldots$ and stop as soon as some $\gcd > 1$ is found. (The greatest common divisor gcd of two numbers a and b can be computed efficiently by means of Euclid's algorithm, which is based on the relation $\gcd(a, b) = \gcd(a \bmod b, b)$. Since $0 \leq a \bmod b < b$ we can apply this relation, after interchanging the arguments, repeatedly until we find $a \bmod b = 0$. The number of steps is $\mathcal{O}(\log(\max(a, b)))$.)

Parallel implementation of this algorithm does not give linear speed-up. One possibility is to try several different pseudo-random sequences (generated by different polynomials f). If we have P processors and use P different sequences in parallel, the probability that the first k values in each sequence are distinct mod p is approximately $\exp(-k^2 P/(2p))$, so the speed-up is $\mathcal{O}(P^{1/2})$ and the efficiency is only $\mathcal{O}(P^{-1/2})$.

Exercise 5.18 Write a program to factorise numbers with the Pollard rho method on P processors for some small values of $P \geq 2$ (choosing $f(x) = x^2 + a$ for various values of a) and apply this to the number $N = (10^{17} - 1)/9$. Compare the speed-up with the expected one. Note that you need multiprecision routines to compute $f(x) \bmod N$. □

5.4 Systolic Algorithms

In this section we briefly present one kind of parallel algorithms which are particularly suitable for implementation on distributed-memory parallel computers. This suitability emanates from the uniform granularity of the processes involved and the need for only local synchronised communication between the processors they are mapped onto. These algorithms are generally referred to as *systolic algorithms*, a concept which became very popular in the last decade, though the original motivation was driven by the potential advantages for direct VLSI implementation rather than parallel computer implementation. In the following we present an example of a systolic algorithm, show how this algorithm can be expressed in a high-level programming language with parallel constructs and finally comment on the general properties of systolic algorithms and refer the reader to further literature on this topic.

The matrix multiplication algorithm presented below is due to L. E. Cannon [3] and was conceived a decade before the term "systolic algorithm" was introduced in computer science. It was developed for the needs of a specific application, Kalman filtering , in time when parallel computing was considered to be a rather exotic research activity. Twenty years later this algorithm became the basis of matrix multiplication subroutines for several very powerful parallel computers which present the state of the art in high-performance computing and are extensively used by the scientific research community. In this context, this parallel algorithm is a good example of the usefulness of algorithmic research on the long-term.

Cannon's algorithm applies to the multiplication of square $n \times n$ matrices

$$c_{i,j} := \sum_{k=1}^{n} a_{i,k} b_{k,j} \ , \quad i = 1, \ldots, n; \ j = 1, \ldots, n$$

written more concisely as

$$C := A.B \ .$$

The algorithm maps most naturally onto a square processor array of size $n \times n$ with torus interconnections but can also easily be adapted to other parallel system models such as a linear array of n processors, a hypercube of n^2 processors, an X-net or a fat tree interconnected system, etc. As it will be shown in the following, Cannon's matrix multiplication algorithm can be encoded in a series of data-parallel elemental and communication operations with data arrays without any reference to the way in which these operations are realised on a concrete parallel computer. The only requirement for an efficient implementation is that the elements of a matrix can be distributed in such a way among the processors of a parallel computer, that circular shifts by one position along the rows and columns of a matrix can be done in time that is small as compared with the time needed for one elemental array multiplication and one addition.

Figure 5.7a shows the distribution of the elements of an $n \times n$ matrix A among the n^2 processors of a parallel computer. For the present, no assumptions are made for the interconnection pattern and the processors are shown arranged in a two-dimensional square grid which corresponds to the usual way of visualizing a matrix. Figure 5.7b shows a conform distribution of the elements of three matrices A, B and C. "Conform" means that counterpart matrix elements, i.e., matrix elements with the same subscript pair such as $a_{2,3}$, $b_{2,3}$ and $c_{2,3}$, are mapped onto the same processor. Cannon's algorithm requires a different starting distribution of the elements of the matrices A, B and C and this distribution is shown in Figure 5.7c. While the matrix C is distributed as normally, the elements of A and B are rearranged with respect to C. More precisely, the elements of the i^{th} row of A and the j^{th} column of B are shifted cyclically by $i-1$ and $j-1$ positions to the left and upwards, respectively $(i = 1, \ldots, n, \;\; j = 1, \ldots, n)$.

Here we are not concerned with the problem of how this rearrangement can be carried out on a particular parallel computer. On a 2-D torus, for instance, a circular shift by $i-1$ positions, $i < n/2$, can be carried out as a series of $i-1$ elementary shifts by one position and thus require time $\mathcal{O}(i)$. For $i \geq n/2$, the shift can be realised by $n - i + 1$ elementary shifts. The worst case is given by the middle row of A and middle column of B which have to be circularly shifted by $n/2$ positions, a process which requires time $\mathcal{O}(n)$. On a system of hypercube-connected processors, the necessary matrix elements rearrangements can be realised more efficiently as series of shifts at distances which are powers of two. On a hypercube the latter shifts can be carried out in constant time, and therefore the total time required for rearranging the matrices A and B will be $\mathcal{O}(\log n)$.

Cannon's algorithm starts with the data distribution shown in Figure 5.7c and involves a series of n steps in each of which each processor computes the product of the components of A and B it currently has and accumulates the result in the component of C which is permanently assigned to it. After that each processor forwards the components of A and B it has to the West and North neighbours replacing them by components it receives from the East and South, respectively. This processor function for one step is specified in Figure 5.8. The left-most column and the top row of processors forward data to the right-most column and the bottom row, respectively.

Figures 5.9 and 5.10 illustrate the movement of data and the computation of products for the case of 4×4 matrices ($n = 4$). Note that the efficient execution of the algorithm, in particular the realisation of the circular column and row shifts, requires that the processors which hold matrix components with neighbouring subscript values, e.g. $a_{2,3}$ and $a_{2,4}$ or $a_{2,3}$ and $a_{3,3}$, communicate directly. The same requirement holds for matrix elements which are neighbours in modulo n sense, e.g. $a_{i,n}$ and $a_{i,1}$ or $b_{n,j}$ and $b_{1,j}$ $(i = 1, \ldots, n;\; j = 1, \ldots, n)$. This effectively means that the processors have to be interconnected in torus or that the torus is a subgraph of the connectivity graph of the system as for

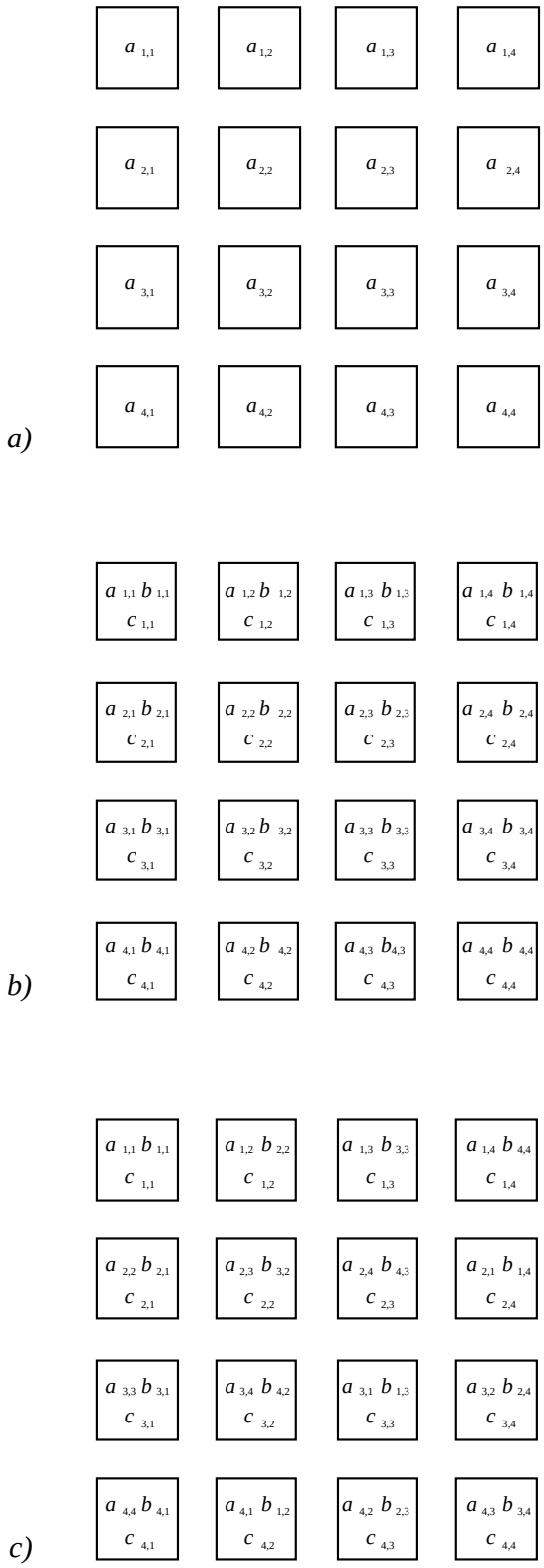


Figure 5.7: *Distribution of a matrix A (a), conform distribution of three matrices A, B and C (b), and distribution required for Cannon's matrix multiplication algorithm (c).*

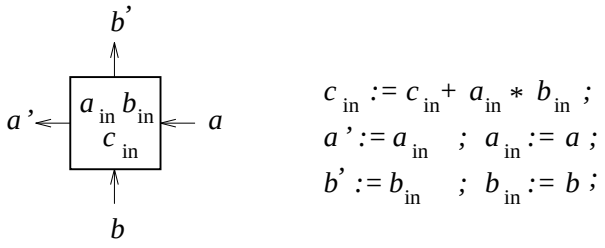


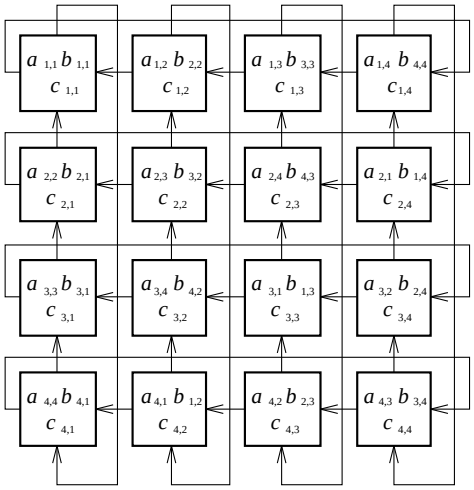
Figure 5.8: *Processor function for Cannon's matrix multiplication algorithm*

instance is the case of hypercube interconnections. Such interconnections are assumed to be given in Figures 5.9 and 5.10.

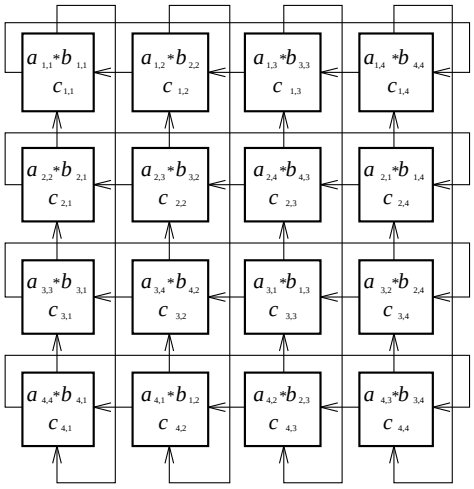
As far as the components of A and B are concerned, the effect of the cell function, the interconnection pattern and the starting distribution shown in Figures 5.7c and 5.9a is that the rows of A and the columns of B circulate in the rows and columns of the torus, respectively. The movement of data is synchronised, i.e., all rows of A are shifted synchronously by one position to the left and at the same time all columns of B are shifted by one position upwards. Then a local computation follows which is executed in all processors and then the communication procedure is repeated again, followed by computations, etc.

The algorithm is quite simple to understand, if it is analysed on a row-by-row basis for the matrix A and column-by-column basis for the matrix B. Let us, for instance, consider the first row of A and the first column of B. At the beginning of the algorithm, the components of the first row of A and the first column of B are arranged in the "normal" order in the top row and leftmost column of processors, respectively. In the first step ($t = 1$), their first components $a_{1,1}$ and $b_{1,1}$ are in the top-left processor where the product $a_{1,1}b_{1,1}$ is computed and accumulated in the variable $c_{1,1}$ which resides permanently in this processor (Figure 5.9b). Since the first row of A is shifted to the left and the first column of B is shifted upwards, $a_{1,2}$ and $b_{2,1}$ meet in the considered processor in the next, second step. There they are multiplied and their product is accumulated in $c_{1,1}$ (Figure 5.9c). In a similar way, the other components of the first row of A and the first column of B pass through the top-left processor where they are multiplied with each other and their products are accumulated (Figure 5.10d and 5.10e). In this way, in n clock periods the innerproduct of the first row of A with the first column of B is computed and assigned to $c_{1,1}$ in the top-left processor.

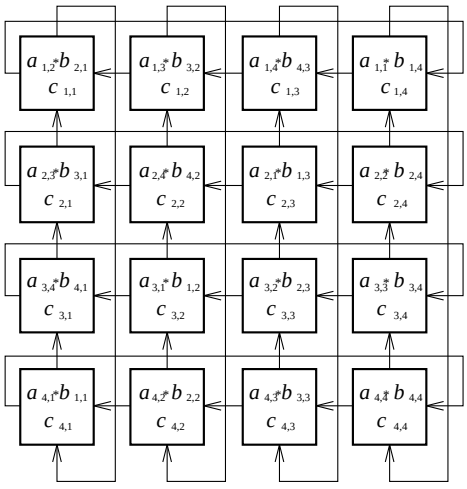
Now, note that since the first row of A is shifted circularly to the left, all components of this row of A pass through all processors of the first row of the processor array. Only the sequence in which this is done differs from processor to processor. The component $a_{1,2}$, for instance, enters the second processor at time (step) $t = 1$, followed by $a_{1,3}$ at $t = 2, \ldots, a_{1,n}$ at $t = n - 1$ and finally by $a_{1,1}$ at $t = n$. Since the components of the second column of B reside in the



a) $t = 0$

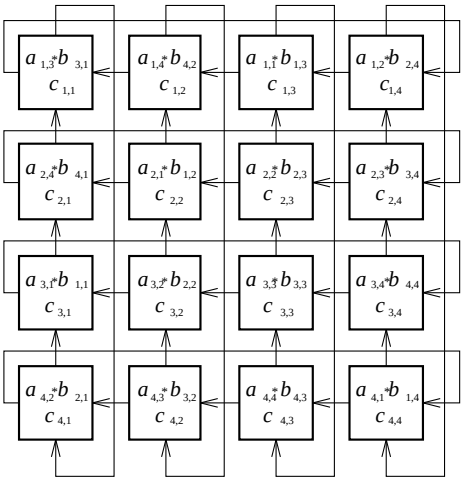


b) $t = 1$

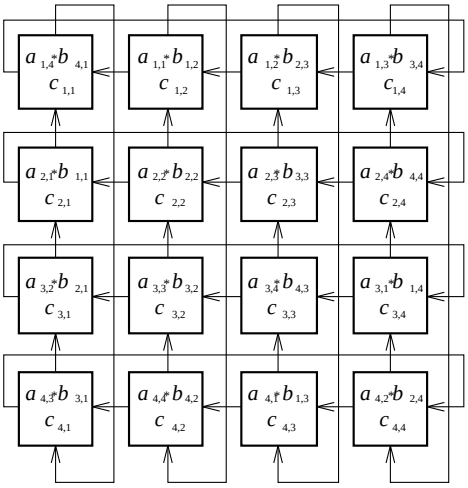


c) $t = 2$

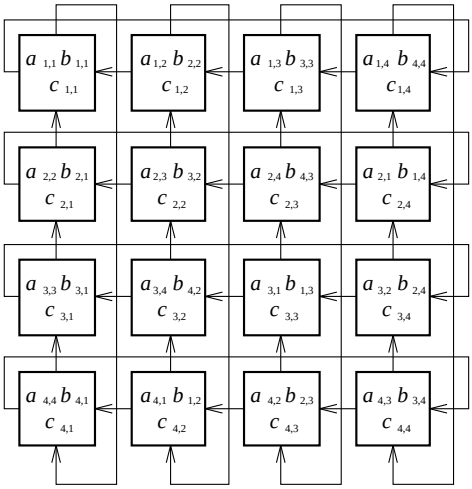
Figure 5.9: *Data flow and computational activities*



d) $t = 3$



e) $t = 4$



f) $t = 5$

Figure 5.10: *Data flow and computational activities (continued)*

second column of the array and are shifted circularly upwards, it is possible to compute the innerproduct of the first row of A with the second column of B. In order to do that, the components of the second column of B have to enter the second processor in the top row of the processor array in the correct sequence, i.e., starting with $b_{2,2}$ at $t = 1$ followed by $b_{3,2}$ at $t = 2, \ldots, b_{n,2}$ at $t = n - 1$ and finally by $b_{1,2}$ at $t = n$. Similar considerations apply to the other rows of A and columns of B. This suggests an explanation for the particular ordering (starting data configuration) of the elements of A and B.

In general, the innerproduct of the i^{th} row of A with the j^{th} column of B is computed in n steps by the processor which holds component $c_{i,j}$ of C. All such innerproducts are computed in parallel and, therefore, with n^2 processors the algorithm requires n steps. Assuming the torus interconnections are given and used, each such step takes constant time (independent of n) and, hence, the total running time is $\mathcal{O}(n)$. Of course, the matrix elements rearrangement phase, in which the starting data configuration is produced, increases the total time. Using torus interconnections this phase requires time $\mathcal{O}(n)$. On a hypercube, it can be carried out in time $\mathcal{O}(\log n)$. Although the total running time is increased by the necessary matrix rearrangements, in both cases the time complexity $\mathcal{O}(n)$ of the algorithm is not worsened.

Note that after n steps the components of A and B are in the same position as at the beginning of the algorithm (compare Figure 5.9a and Figure 5.10f). Typically, matrix multiplication will be only one in a series of computations with the involved matrices. Therefore, the "natural" distribution of the matrices A and B has in general to be restored after the computations are completed. This can be done by circular shifting of the i^{th} row of A to the right at a distance $i - 1$ $(i = 1, \ldots, n)$ and circular downwards shifting of the j^{th} column of B at a distance $j - 1$ $(j = 1, \ldots, n)$. Similar to the initial rearrangement phase, this final phase will increase the overall execution time but not worsen the time complexity of the algorithm.

Similar algorithms can be given in which one of the matrices A and B is stationary while the other one and the result matrix C move in mutually perpendicular directions. The reader is invited to design such algorithms as an exercise and referred to [30, chapter 11] for a methodology for the systematic design and restructuring of parallel systolic algorithms.

Finally we show how Cannon's algorithm can be encoded in a high-level programming language with array constructs. Here follows a Fortran 90 expression of the algorithm (program rows are labeled for ease of reference):

```
0       REAL, DIMENSION(n,n) :: a, b, c

10      a = CSHIFT( a, SHIFT = (/0:n-1/), DIM = 2 )
11      b = CSHIFT( b, SHIFT = (/0:n-1/), DIM = 1 )

20      DO it = 1, n
21         c = c + a*b
```

```
22          a = CSHIFT( a, SHIFT = 1, DIM = 2 )
23          b = CSHIFT( b, SHIFT = 1, DIM = 1 )
24        ENDDO

30        a = CSHIFT( a, SHIFT = -(/0:n-1/), DIM = 2 )
31        b = CSHIFT( b, SHIFT = -(/0:n-1/), DIM = 1 )
```

Statement 0 is a declaration of three real matrices A, B and C of size $n \times n$. Statements 10 and 11 realise the necessary rearrangement of the matrices A and B to achieve required starting data configuration. The parameter DIM in these statements specifies which of the indices is changed. In statement 10, DIM = 2 and this means that the second index of A is changed; this is equivalent to shifting the elements of A along the rows. Similarly, the elements of B are shifted along the columns (DIM = 1 in statement 11 means that the first, row index is changing). The parameter SHIFT is a vector which specifies for each row of A and each column of B the number of positions by which they are shifted. In this case, SHIFT$(i) = i - 1$, $i = 1, \ldots, n$, and this means that the i^{th} row of A and the i^{th} column of B are shifted at distance $i - 1$.

The actual algorithm, as it is illustrated by Figure 5.9b through Figure 5.9e, is described by statements 20 through 24 and consists of n steps counted by the loop counter it (see statement 20). Each step begins with an elemental array operation (statement 21) in which the component of A at position (i, j) is multiplied with the component of B at position (i, j) and the result is added to the component of C which is in the same position ($i = 1, \ldots, n$; $j = 1, \ldots, n$). Note that the components of A and B at position (i, j) are not necessarily the components $a_{i,j}$ and $b_{i,j}$ of the original matrices A and B because of the initial rearrangement phase and later on because of the further rearrangement operation with these matrices. (Thus for instance, the component of A which is in position $(2, 1)$ in the first step is $a_{2,2}$ which is replaced by $a_{2,3}$ in the second step and by $a_{2,4}$ in the third step.) After the elemental array operation specified by statement 21, the matrix A is shifted cyclically by one position to the left (statement 22) and the matrix B is shifted cyclically by one position upwards (statement 23). Note that the parameter SHIFT is a scalar (1) and this means that all rows of A and all columns of B are shifted at the same distance (of one).

Finally, the "normal" distribution of A and B is restored by statements 30 and 31 which specify operations that are inverse to those described by statements 10 and 11, respectively. ("Normal" means that the elements of A and B at position (i, j) are the elements $a_{i,j}$ and $b_{i,j}$ of the original matrices A and B.)

Note that the above program does not specify in any way the number of processors in the executing parallel system nor a particular interconnection pattern. The program is merely a specification of a set of operations to be carried out with the three matrices involved. These operations can be executed on different parallel systems and also on any sequential computer. The parallelism inherent to the algorithm is encoded in data-parallel operations, i.e., operations which

are applied to all elements of a data array. The efficient execution of the algorithms, in particular the efficient execution of the data-parallel communication operations specified by statements 22 and 23, on a distributed-memory parallel computer requires, however, that the processors of the parallel computer can be directly connected in a torus.

Finally, we should note that in spite of the expressive power of Fortran 90 and the rather compact algorithm specification, there are some shortcomings of the language with respect to the specification of parallel processes. In particular, the statements in the pairs 10 and 11, 22 and 23, and 30 and 31 will be executed in the specified sequence whereby in the algorithm they can be executed in parallel.

In Cannon's algorithm the elements of the matrices A and B are shifted from processor to processor, covering the same distance in each step. The data movement is steady in this sense but at the same it is time discontinuous, in that it is interrupted by the stops of data in the processors for computations. This movement resembles the pulsing movement of the blood under the contractions of the heart, called *systoles* in physiology, and this analogy was the reason to give such algorithms the attribute *systolic* [24, 25].

The concepts of a systolic algorithm and a systolic array became very popular in the beginning of the 1980s. These concepts were introduced in computer science and electrical engineering by means of instances of such algorithms and models of computing structures for concrete problems such as convolution, matrix multiplication, computing the inverse of a matrix, etc. The regularity of systolic arrays and the high efficiency of systolic algorithms do not only have practical implications in VLSI design and parallel computing but do also strongly appeal to one's intellect, aesthetic sense and curiosity to "to see behind the trick". These were certainly some of the reasons for the fast proliferation and wide acceptance of these concepts in the computer science and electrical engineering research communities. One of the effects of the enthusiasm for systolic algorithms in the past decade was a wave of worldwide activities in this area. As a result of these activities, a large number of highly efficient systolic algorithms have been proposed for many computationally intensive problems. For lack of space it is not possible to even only list here the wealth of results achieved in this area. Therefore, we close this subsection by referring the reader to the monograph [30] for a unified representation of a large number of such algorithms for a wide variety of problems as well as for formal definitions of the concepts.

Bibliography

[1] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Dunato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, PA, 1993.

[2] J.C. Butcher, *The numerical analysis of ordinary differential equations, Runge-Kutta and general linear methods*, Wiley, New York, 1987.

[3] L.E. Cannon, *A Cellular Computer to Implement the Kalman Filter Algorithm*, PhD Thesis, Montana State University, Bozeman, Montana, U.S.A., 1969.

[4] P. Chartier, *L-stable parallel one-block methods for ordinary differential equations*, SIAM J. Numer. Anal. **31** (2), 552–571, 1994.

[5] M.T. Chu, H. Hamilton, *Parallel solution of ODE's by multi-block methods*, SIAM J. Sci. Stat. Comput. **8**, 342–353, 1987.

[6] L. Csanky, *Fast parallel matrix inversion algorithms*, SIAM J. Computing **5**, 618–623, 1976.

[7] J.J.M. Cuppen, *A divide-and-conquer method for the symmetric tridiagonal eigenproblem*, Numer. Math. **36**, 177–195, 1981.

[8] J.W. Demmel, M.T. Heath, H.A. van der Vorst, *Parallel Numerical Linear Algebra*, Acta Numerica 1993, 111–197.

[9] D.S. Dodson, J.G. Lewis, *Proposed Sparse Extensions to the Basic Linear Algebra Subprograms*, Signum **20**, 22–25, 1985.

[10] J. Dongarra, I. Duff, D. Sorensen, H. van der Vorst, *Solving linear systems on vector and shared memory computers*, SIAM, Philadelphia, PA, 1991.

[11] I.S. Duff, G.A. Meurant, *The effect of ordering on preconditioned conjugate gradient*, BIT **29**, 635–657, 1989.

[12] R. Fletcher, *Conjugate gradient methods for indefinite systems*, volume 506 of *Lecture Notes Math.*, 73–89, Springer-Verlag, Berlin–Heidelberg–New York, 1976.

[13] R.W. Freund, G.H. Golub, N.M. Nachtigal, *Iterative solution of linear systems*, Acta Numerica 1992, 57–100.

[14] R. W. Freund and N. M. Nachtigal, *QMR: a quasi-minimal residual method for non-Hermitian linear systems*, Numer. Math. **60**, 315–339, 1991.

[15] E.D. de Goede, *Numerical methods for the three-dimensional shallow water equations*, Doctor's Thesis, CWI Amsterdam, 1992.

[16] G. Golub, C. Van Loan, *Matrix Computations*, John Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[17] E. Hairer, C. Lubich, M. Roche, *The numerical solution of differential-algebraic systems by Runge-Kutta methods*, Lecture Notes in Mathematics **1409**, Springer-Verlag, Berlin, 1989.

[18] E. Hairer, G. Wanner, *Solving ordinary differential equations, II: Stiff and differential-algebraic problems*, Springer Series in Comp. Math., vol. **14**, Springer-Verlag, Berlin, 1991.

[19] M.R. Hestenes, E. Stiefel, *Methods of conjugate gradients for solving linear systems*, J. Res. Nat. Bur. Stand. **49**, 409–436, 1954.

[20] A.C. Hindmarsh, *LSODE and LSODI, two new initial value ordinary differential equation solvers*, ACM/SIGNUM Newsletter **15** (4), 10–11, 1980.

[21] P.J. van der Houwen, B.P Sommeijer, *Iterated Runge-Kutta methods on parallel computers*, SIAM J. Sci. Stat. Comput. **12**, 1000–1028, 1991.

[22] P.J. van der Houwen, B.P. Sommeijer, W. Couzy, *Embedded diagonally implicit Runge-Kutta algorithms on parallel computers*, Math. Comp. **58**, 135–159, 1992.

[23] D.E. Knuth, *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Addison Wesley, 1973.

[24] H.T. Kung, C.E. Leiserson, *Systolic arrays (for VLSI), Sparse Matrix Proc. 1978*, Society for Industrial and Applied Mathematics, 256–282, 1979.

[25] H.T. Kung, *Why systolic architectures*, Computer **15** (1), 37–46, 1982.

[26] C. Lanczos, *Solution of systems of linear equations by minimised iterations.* J. Res. Nat. Bur. Stand. **49**, 33–53, 1952.

[27] B. N. Parlett, D. R. Taylor, Z. A. Liu, *A look-ahead Lanczos algorithm for unsymmetric matrices*, Math. Comp., **44**, 105–124, 1985.

[28] C. C. Paige, M. A. Saunders, *Solution of sparse indefinite systems of linear equations*, SIAM J. Numer. Anal. **12**, 617–629, 1975.

[29] C. C. Paige, M. A. Saunders, *LSQR: An algorithm for sparse linear equations and sparse least squares*, ACM Trans. Math. Soft. **8**, 43–71, 1982.

[30] N. Petkov, *Systolic Parallel Processing*, Elsevier Science Publ., Amsterdam, 1993.

[31] H.J.J. te Riele, *Factoriseren en Primaliteitstesten, een inleiding*, Report NM-N8804, CWI Amsterdam, October 1988 (in Dutch).

[32] U. Schendel, *Introduction to Numerical Methods for Parallel Computers* (translated from German). Ellis Horwood Ltd., Chichester, 1984.

[33] Y. Saad, M. H. Schultz, *GMRES: a generalised minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput. **7**, 856–869, 1986.

[34] G. L. G. Sleijpen, D. R. Fokkema, *Bi-CGSTAB(ℓ) for linear equations involving unsymmetric matrices with complex spectrum*, Technical Report 772, Department of Mathematics, Utrecht University, 1993.

[35] B.P. Sommeijer, *Parallelism in the numerical integration of initial value problems*, Thesis, Univ. of Amsterdam, 1992.

[36] B.P. Sommeijer, *Parallel-iterated Runge-Kutta methods for stiff ordinary differential equations*, J. Comput. Appl. Math. **45**, 151–168, 1993.

[37] P. Sonneveld, *CGS: a fast lanczos-type solver for nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput. **10**, 36–52, 1989.

[38] H.S. Stone, *Problems of parallel computation*, In: J.F. Traub, editor, "Complexity of Sequential and Parallel Numerical Algorithms", 1–16, Academic Press, 1973.

[39] A. van der Sluis, H. A. van der Vorst, *Numerical solution of large sparse linear algebraic systems arising from tomographic problems*, in: G. Nolet, editor, *Seismic Tomography*, chapter 3, 49–83, Reidel Pub. Comp., Dordrecht, 1987.

[40] H. A. van der Vorst, *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems*, SIAM J. Sci. Statist. Comput. **13**, 631–644, 1992.

[41] H.A. van der Vorst, *Parallel rekenen en supercomputers*. Academic Service, Schoonhoven, The Netherlands, 1988 (in Dutch).

[42] H.A. van der Vorst, K. Dekker, *Vectorisation of linear recurrence relations*, Siam J. Sci. Stat. Comput. **10**, 27–35, 1989.

[43] H.H. Wang, *A parallel method for tridiagonal equations*, ACM Trans. Math. Software **7**, 170–183, 1981.