Developments and Trends in the Parallel Solution of Linear Systems

Iain S. Duff* Henk A. van der Vorst†

April 13, 1999

Abstract

In this review paper, we consider some important developments and trends in algorithm design for the solution of linear systems concentrating on aspects that involve the exploitation of parallelism. We briefly discuss the solution of dense linear systems, before studying the solution of sparse equations by direct and iterative methods. We consider preconditioning techniques for iterative solvers and discuss some of the present research issues in this field.

Keywords: linear systems, dense matrices, sparse matrices, tridiagonal systems, parallelism, direct methods, iterative methods, Krylov methods, preconditioning.

AMS(MOS) subject classifications: 65F05, 65F50.

1 Introduction

Solution methods for systems of linear equations

$$Ax = b, \tag{1}$$

where A is a coefficient matrix of order n and x and b are n-vectors, are usually grouped into two distinct classes: direct methods and iterative methods. However,

*CCLRC - Rutherford Appleton Laboratory, Oxfordshire, England and CERFACS, Toulouse, France

†Mathematical Institute, Utrecht University, Budapestlaan 6, Utrecht, the Netherlands. The work of this author was supported by the Netherlands organization for scientific research NWO, through project 95MPR04.

E-mail: I.Duff@rl.ac.uk, vorst@math.uu.nl

this distinction is not really productive or even meaningful. The archetype for direct methods is Gaussian elimination and the archetype for an iterative method is Richardson's iteration method. In reality the two classes cannot be clearly distinguished. A common and cheap technique for improving the solution obtained by pure Gaussian elimination is iterative refinement, which can be reformulated as Richardson's iterative method with Gaussian elimination as a preconditioner. On the other hand, true iterative methods are usually accelerated by preconditioning which often involves the direct solution of a nearby system. Usually this nearby system is chosen in such a way that it can be solved more economically or so that the solution process can better exploit parallelism. In any case, the solution of a system used in preconditioning must be much cheaper than the solution of the original system. Thus, the main difference between solving the original system and a preconditioning by direct methods is that, for the former case, we have to rearrange computations or modify the algorithm in order to obtain better parallelism, whereas with preconditioning we may simply discard entries from the nearby system if this improves parallelism (under the condition that the convergence of the preconditioned iterative solver is not affected too much).

In the early days of parallelism, attention was focussed on fine-grained forms of parallelism. For direct methods, that led to the design of vectorizable algorithms. A major bottleneck with this approach was that the poor computation to memory access ratio caused poor performance on many architectures, so that the building blocks had to be matrix-vector operations rather than vector operations. For current parallel computers, it has been necessary to look for even more coarse-grained parallelism and this has been achieved by using building blocks of matrix-matrix multiplication. This trend is nicely characterized by the successive levels of the Basic Linear Algebra Subroutines: the Level 1 BLAS in 1979 [137], the Level 2 BLAS in 1988 [72], and the Level 3 BLAS in 1990 [73]. For dense systems, it has become evident that the Level 3 BLAS can lead to optimal performance on a variety of modern architectures, including serial computers, because of the potential to handle memory hierarchy in an efficient way. Additionally, for large sparse matrices this has given rise to widely accepted direct methods, for example the multifrontal methods and supernodal techniques. We will discuss these methods in Section 2.

For linear systems of very special form, for instance, tridiagonal matrices, standard direct methods may not offer sufficient possibilities for efficient implementation. Nevertheless, in this case also, the development has advanced from fine-grained methods (cyclic reduction; recursive doubling) to more coarse-grained approaches (Wang's method, Mehrmann's algorithm, Bondeli's divide and conquer algorithm). These techniques will be reviewed in Section 3.

For iterative methods, attention has been paid to the iteration technique itself (block methods, s-step methods) and to the preconditioner. The techniques for creating more parallelism in the iterative algorithm seem to receive less attention now, mainly because modern computers can handle the inner products sufficiently well. In the past, this was a notorious bottleneck and many techniques revolved around the effective grouping of inner products, often sacrificing some of the numerical stability of the method. We will discuss this in more detail in Section 4.

For preconditioners, the early approaches were also aimed at detecting parallelism at a rather fine-grained level (hyperplane, truncated Neumann, frontal approach). For the preconditioner, we now see very much the same implementation techniques as for direct methods, but one often concentrates on the selection of special structured preconditioners which makes a separate discussion for these techniques necessary. In many cases, the parallelism in the preconditioner is based on a domain-splitting technique, which leads to the solution of large independent blocks. Techniques for creating parallel preconditioners will be reviewed in Section 5.

2 Parallel techniques for direct solvers

The first real library of subroutines for linear algebra on *dense* matrices was developed in Algol by Wilkinson and Reinsch [208]. These were used as the basis for the LINPACK Project where a wide range of software for solving dense systems of equations was developed in Fortran and is described in the LINPACK book [71]. The LU factorization code has been used as a basis for the benchmarking of computers with the latest results being available on the Web [70]. The codes in the LINPACK package used Level 1 BLAS [137] and were portable over a wide range of machines but sometimes performed poorly on vector or cache-based machines. This was addressed in the development of the LAPACK package for linear algebra [11]. Codes in this package incorporated Level 2 and Level 3 BLAS ([72] and [73] respectively) and had a much improved performance on modern architectures. Many vendors of shared memory computers offer parallel versions of the BLAS and so at this level parallelization is trivial. However, LAPACK was not designed for parallel machines and, in particular, not for machines with distributed memory that use message passing to communicate data. This last class of machines is targeted by the ongoing ScaLAPACK project [33] that supports distributed computation using tools like the BLACS (Basic Linear Algebra Communications Routines) [207].

Sparse factorization
The main issue in the implementation of direct methods for solving *sparse* linear equations lies in preserving as much sparsity as possible in the matrix factors. For this reason, the LU decomposition is favoured and strategies are developed to retain much of the sparsity of the original matrix in the LU factors. This leads to a need to compromise the numerical pivoting strategy in order to choose pivots to limit the fill-in. A common strategy for limiting fill-in, due to Markowitz [144], chooses entries so that the product of the number of other entries in the row and column of the candidate pivot is minimized. An entry is accepted as a pivot only if it is within a threshold of the largest in its column. This Markowitz-threshold strategy and a

range of other similar possibilities are discussed in detail by [80]. Complicated data structures are designed so that only the nonzero entries of the matrix and of the factors need to be held. This, coupled with the fact that it is often non-trivial to determine what part of the matrix is updated at each pivot step, has meant that early experiments on parallelizing sparse direct methods gave poor speedups [95].

However, there are three levels at which parallelism can be exploited in the solution of sparse linear systems by direct methods. We consider each of these levels in turn.

The finest grain lies in the elimination operations themselves. Most modern sparse codes (we will discuss this more shortly) use higher level BLAS to perform most or all of the floating-point operations and so can benefit from their efficient implementation and parallel support in a similar way to software for dense linear systems.

At the coarsest level, techniques for partitioning the matrix are often designed for parallel computing and are particularly appropriate for distributed memory computers. Indeed, partitioning methods are often only competitive when parallelism is considered. The PARASPAR package [210] uses a preordering to partition the original problem. The MCSPARSE package [91, 101] similarly uses a coarse matrix decomposition to obtain an ordering to bordered block triangular form.

At an intermediate level, we can use the sparsity of the matrix to advantage. Clearly, there can be substantial independence between pivot steps in sparse elimination. For example, if the matrix were a permutation of a diagonal matrix all operations could be performed in parallel. Two matrix entries a_{ij} and a_{rs} can be used as pivots simultaneously if a_{is} and a_{rj} are zero. These pivots are termed *compatible*. This observation [38] has been the basis for several algorithms and parallel codes for general matrices. The central theme is to select a number of compatible pivots that would give a diagonal block if ordered to the top left of the matrix. The update from all these pivots is then performed in parallel. The procedure is repeated on the reduced matrix. The algorithms differ in how the pivots are selected (clearly one must compromise criteria for reducing fill-in in order to get a large compatible pivot set) and in how the update is performed. Alaghband [1] uses compatibility tables to assist in the pivot search. She uses a two-stage implementation where first pivots are chosen in parallel from the diagonal and then off-diagonal pivots are chosen sequentially for stability reasons. She sets thresholds for both sparsity and stability when choosing pivots. Davis and Yew [56] perform their pivot selection in parallel, which results in the nondeterministic nature of their algorithm because the compatible set will be determined by the order in which potential compatible pivots are found. Their algorithm, D2, was designed for shared-memory machines and was tested extensively on an Alliant FX/8. The Y12M algorithm [211] extends the notion of compatible pivots by permitting the pivot block to be upper triangular rather than diagonal, which allows them to obtain a larger number of pivots, although the update is more complicated. For distributed memory architectures, van der Stappen, Bisseling, and van de Vorst [195] distribute the matrix over the processors in a grid fashion, perform a parallel search for compatible pivots, choosing entries of low Markowitz cost that satisfy a pivot threshold, and perform a parallel rank-m update of the reduced matrix, where m is the number of compatible pivots chosen. We show some results from using their code in Table 1. The slow processor speed masks the communication costs but nevertheless these results show that good speedups can be obtained on some machines. Their code was originally written in OCCAM and run on a network of 400 transputers, but they have since developed a version using PVM [133].

In the context of reduced stability of the factorization due to the need to preserve sparsity and exploit parallelism, it is important that sparse codes offer the possibility of iterative refinement both to obtain a more accurate answer and to provide a measure of the backward error. Quite recently there has been suggestions made to compute in increased precision to avoid some of the problems from this compromise of stability pivoting [139].

Matrix	Order	Entries	Number of processors			
			1	16	100	400
			Secs.	Speedup		
SHERMAN 2	1080	23094	1592	15	49	102
LNS 3937	3937	25407	2111	13	56	89

Table 1: Factorization times in seconds on one processor and speedups on more than one for code of van der Stappen, Bisseling, and van de Vorst (1993) on a PARSYTEC SuperCluster FT-400

Elimination tree

A common structure for both visualizing and implementing parallelism is the elimination tree, or other trees derived from it. The elimination tree is defined for any sparse matrix whose sparsity pattern is symmetric. For a sparse matrix of order n, the elimination tree is a tree on n nodes such that node j is the father of node i if entry (i, j), $j > i$ is the first entry below the diagonal in column i of the triangular factors. An analogous graph for an unsymmetric patterned sparse matrix is the directed acyclic graph [55, 102]. The main property that we exploit in this tree is that computations corresponding to nodes that are not ancestors or descendants of each other are independent (see, for example, [77, 140]). The tree can thus be used to schedule parallel tasks. For shared memory machines, this can be accomplished through a shared pool of work with fairly simple synchronizations that can be controlled using locks protecting critical sections of the code [7, 78].

Since any sparse LU or QR factorization can be represented by an elimination tree, we can use this structure to exploit parallelism. One of the main issues for an efficient implementation on shared memory machines concerns the management of data, which must be organized so that book-keeping operations such as garbage collection do not cause too much interference with the parallel processing.

Sparse Cholesky factorization by columns can be represented by an elimination tree. These can either be left-looking (or fan-in) algorithms, where updates are performed on each column in turn by all the previous columns that contribute to it, then the pivot is chosen in that column and the multipliers calculated; or a right-looking (or fan-out) algorithm where, as soon as the pivot is selected and multipliers calculated, that column is immediately used to update all future columns that it modifies. The terms left-looking and right-looking are discussed in detail in the book [74]. Either way, the dependency of which columns update which columns is determined by the elimination tree. If each node of the tree is associated with a column, a column can only be modified by columns corresponding to nodes that are descendants of the corresponding node in the elimination tree.

One approach to using higher level BLAS in sparse direct solvers is a generalization of a sparse column factorization. Higher level BLAS can be used if columns with a common sparsity pattern are considered together as a single block or supernode and algorithms are termed column-supernode, supernode-column, and supernode-supernode depending on whether target, source, or both are supernodes.

Another major approach for utilizing Level 3 BLAS within a sparse direct code is a multifrontal technique [84]. In this approach, the nonzero entries of the pivot row and column are held in the first row and column of a dense array and the outer-product computation at that pivot step is computed within that dense submatrix. The dense submatrix is called a *frontal matrix*. Now, if a second pivot can be chosen from within this dense matrix (that is there are no nonzero entries in its row and column in the sparse matrix that lie outside this frontal matrix), then the operations for this pivot can again be performed within the frontal matrix. In order to facilitate this multiple elimination within a frontal matrix, an assembly tree is preferred to an elimination tree where, for example, chains of nodes are collapsed into a single node so that each node can represent several eliminations. Ordering schemes, artificial enlargement of frontal matrices, and fill-in in Gaussian elimination can be used so that the kernel of the multifrontal scheme can be represented by the computation

$$F_{22} \leftarrow F_{22} - F_{21} F_{11}^{-1} F_{12} \tag{2}$$

performed within the dense frontal matrix

$$\begin{pmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{pmatrix}.$$

These operations can clearly be performed using the Level 3 BLAS. If the factorization is represented by an elimination tree, nodes of this tree can be amalgamated to form an assembly tree where each node corresponds to a factorization of the form (2) and each edge to the communication of the resulting Schur complement (F_{22}) to the parent node.

Several authors have experimented with these different algorithms (right-looking, left-looking, and multifrontal) and different blockings. Ng and Peyton [155] favour the left-looking approach and Amestoy and Duff [8] show the benefits of Level 3 BLAS within a multifrontal code on vector processors. Rothberg and Gupta [171] find that, on cache-based machines, it is the blocking that affects the efficiency (by a factor of 2 to 3) and the algorithm that is used has a much less significant effect. Demmel, Eisenstat, Gilbert, Li, and Liu [66] have extended the supernodal concept to unsymmetric systems although, for general unstructured matrices, they cannot use regular supernodes for the target columns and so they resort to Level 2.5 BLAS, which is defined as the multiplication of a set of vectors by a matrix where the vectors cannot be stored in a two-dimensional array. By doing this, the source supernode can be held in cache and applied to the target columns or blocks of columns of the "irregular" supernode, thus getting a high degree of reuse of data and a performance similar to the Level 3 BLAS. We now examine some of the history of attempts to parallelize sparse direct codes.

Historic survey of techniques to exploit parallelism

Early experiments on the massively parallel SIMD Connection machine [103] were a little disappointing although they did indicate the possibility of using massive parallelism in sparse factorization. The experiments did show that a grid-distributed multifrontal implementation substantially outperformed an algorithm based on a left-looking approach that exploited the parallelism in the elimination tree. Conroy, Kratzer, and Lucas [52] used a mapping strategy that can trade work against data movement to design a multifrontal algorithm for machines supporting a data-parallel programming model. They tested their code on a TMC CM-5 and a MasPar MP-2 and showed it to be competitive with codes on vector supercomputers. Manne and Hafsteinsson [143] have implemented a supernodal fan-out algorithm on the MasPar MP-2 and use a graph colouring algorithm to map the matrix to processors.

The earliest work on parallelizing sparse codes for distributed memory machines was based on column oriented Cholesky factorizations, either the fan-in or the fan-out algorithm. The original codes just used a column-column formulation of the algorithm as in the fan-in algorithm of [94] but it was soon apparent that better efficiency could be obtained using a supernode-column fan-in approach as in [156]. Some of this early work on parallel algorithms for distributed memory computers is reviewed by Heath, Ng, and Peyton [117]. For distributed memory machines, processors can be assigned work corresponding to subtrees, but this requires quite balanced trees. A breadth-first search strategy can be used to assign work to processors using a heuristic bin-packing algorithm to achieve reasonable load balancing [92]. However, the results of runs on L-shape domains on an INTEL iPSC/2 comparing their strategy with wrap mapping and with a nested-dissection ordering and subtree-to-subcube mapping are rather flat, showing only a slight advantage for their heuristic. Pothen and Sun [164] have adapted the heuristic of [92] to a multifrontal scheme and have compared this with a generalized version of a subtree-to-subcube mapping and have found their algorithm to be twice as fast on an INTEL iPSC/2. All approaches to sparse Cholesky factorization have been used to develop parallel factorization routines on hypercubes: fan-out [96], fan-in [15], and multifrontal [189, 190]. In comparative testing of all three approaches none of the methods shows very high performance [16]. Even in a distributed environment, it is very beneficial if some memory is available as shared memory to hold information such as mapping vectors [99]. The first work to exploit parallelism at all phases of the sparse solution process was by [212], later developed by [104] and [213]. More recently, the work of [118] also exploits parallelism in all phases, although the matrix must be held in Cartesian form; that is, in a two or three dimensional coordinate system. While this is quite natural in the context of discretized PDEs, it is not a convenient interface in general.

Schreiber [180] presents a clear discussion showing that a one-dimensional mapping of columns or block columns to processors is inherently unscalable and that a two-dimensional mapping is needed to obtain a scalable algorithm. Bisseling, Doup, and Loyens [31] have developed an interior point solver for linear programming containing sparse algorithms for Cholesky factorisation, triangular system solution, and matrix multiplication, all based on a two-dimensional mapping of data to processors. Dumitrescu *et al.* [85] find that a two-dimensional block fan-in algorithm (with proportional mapping of the elimination tree) is usually superior to a two-dimensional block fan-out algorithm. They record performances of over 360 Mflop/s on 32 nodes of an IBM SP1. Rothberg [170] compares a block fan-out algorithm using two-dimensional blocking with a panel multifrontal method using one-dimensional blocking and favours the former, obtaining a performance of over 1.7 Gflop/s on 128 nodes of an Intel Paragon, which is about 40% of the performance of the GEMM kernel on that machine. He points out that the benefit of using higher level BLAS kernels, coupled with the then recent increases in local memory and communication speed of parallel processors, had at last made the solution of large sparse systems feasible on such architectures. The 2-D block fan-out algorithm is further investigated by [172], and some block mapping heuristics are used to improve the performance to over 3 Gflop/s for a 3-D grid problem on a 196-node Intel Paragon [173]. A similar type of 2-dimensional mapping is used [111] in an implementation of a multifrontal method, where much of the high performance is obtained through balancing the tree near its root and using a careful and regular mapping of the dense matrices near the root to enable a high level of parallelism to be maintained when the trivial parallelism from subtree assignment is exhausted. Although the headline figure of nearly 20 Gflop/s on the CRAY T3D was obtained on a fairly artificial and essentially dense problem, large sparse problems from structural analysis were factorized at between 8 and 15 Gflop/s on the same machine for which a tuned GEMM code will execute at around 50 Gflop/s. This code is available in compiled form on an IBM SP2 [110] and source code versions of a portable implementation are available from the authors. More recently, Li and Demmel [139] have been developing a version of the SuperLU code [67] for distributed memory machines and the MUMPS multifrontal code [9], developed within the EU PARASOL Project, is also targeting message passing architectures.

Parallelism of other phases than factorization
Partly because of the success of fast and parallel methods for performing the numerical factorization, other phases of the solution are now becoming more critical on parallel computers. The package [118] executes all phases in parallel, and there has been much recent work in finding parallel methods for performing the reordering. This has been another reason for the growth in dissection approaches (for example, see [130, 168]). Parallelism in the triangular solve can be obtained either using the identical tree to the numerical factorization [10] or by generating a tree from the sparsity pattern of the triangular factor [12]. However, in order to avoid the intrinsically sequential nature of a sparse triangular solve, it is possible to hold the denser but still sparse L^{-1}, or better a partitioned form of this to avoid some of the fill-in that would be associated with forming L^{-1} explicitly [6]. Various schemes for this partitioning have been proposed to balance the parallelism (limited by number of partitions) with the fill-in (for example, [4, 5, 161]) and, more recently, the selective inversion of submatrices produced by a multifrontal factorization algorithm has been proposed [167].

Effect of ordering schemes
We now turn to the crucial issue of ordering the rows and columns (choosing the pivots) to preserve sparsity and to exploit parallelism. The classical technique for symmetric systems is to choose at each stage the entry from the diagonal with the least number of entries in its row and column. This minimum degree criterion was first proposed in 1967 [194] and has stood the test of time well. George [93] proposed a different class of orderings based on a non-local strategy of dissection. In his *nested dissection* approach, a set of nodes is selected to partition the graph, and this set is placed at the end of the pivotal sequence. The subgraphs corresponding to the partitions are themselves similarly partitioned and this process is nested with pivots being identified in reverse order. Minimum degree, nested dissection and several other symmetric orderings were included in the SPARSPAK package [97, 98]. Many experiments were performed using the orderings in SPARSPAK and elsewhere, and the empirical experience at the beginning of the 90s indicated that minimum degree was the best ordering method for general or unstructured problems.

However, when one considers parallel implementation, a problem with the minimum degree ordering is that it tends to give elimination trees that are not well balanced and so not ideal for use as a computational graph for driving a parallel algorithm. The elimination tree can be massaged [141] so that it is more suitable for parallel computation but the effect of this is fairly limited for general matrices. The use of dissection techniques would appear to offer the promise of much better balanced trees, although the inferior performance of the early dissection codes needs to be addressed for them to be viable. We now discuss recent advances in dissection techniques.

The beauty of dissection orderings is that they take a global view of the problem; their difficulty until recently has been the problem of extending them to unstructured problems. Recently, there have been several tools and approaches that make this extension more realistic. The essence of a dissection technique is a bisection algorithm that divides the graph of the matrix into two partitions. If node separators are used, a third set will correspond to the node separators. Recently, there has been much work on obtaining better bisections even for irregular graphs. Perhaps the bisection technique that has achieved the most fame has been spectral bisection [23]. In this approach, use is made of the Laplacian matrix that is defined as a symmetric matrix whose diagonal entries are the degrees of the nodes and whose off-diagonals are -1 if and only if the corresponding entry in the matrix is nonzero. This matrix is singular because its row sums are all zero but, if the matrix is irreducible, it is positive semidefinite with only one zero eigenvalue.

A currently favoured approach is for the dissection technique only to be used for the top levels and the resulting subgraphs to be ordered by a minimum degree scheme. This hybrid technique was used many years ago [100] and is included in many current implementations (for example, [17, 122]). Current empirical evidence would suggest that these schemes are at least competitive with minimum degree on some large problems from structural analysis [17, 169] and from financial modelling [29]. In these studies, dissection techniques outperform minimum degree by on average about 15% in terms of floating-point operations for Cholesky factorization using the resulting ordering, although the cost of these orderings can be several times that of minimum degree and may be a significant proportion of the total solution time [17].

Of course, dissection techniques are important for purposes other than generating an ordering for a Cholesky factorization. They can be used to partition an underlying grid for domain decomposition and are equally useful for the parallel implementation of many iterative methods. Two of the major software efforts for developing graph partitioning based on some of the above techniques are CHACO [121] and METIS [129].

3 Solution of tridiagonal systems

In the early days of parallel computing, much attention was paid to the parallel solution of banded linear systems, in particular, tridiagonal systems. In this case, as in the more general case mentioned in Section 1, there has been a trend from fine-grained parallelism towards coarse-grained parallelism. We will briefly discuss this development, solely for the sake of illustration, because in our opinion most of the algorithms are only practical for very large narrow-banded systems and such systems occur infrequently. Moreover, because of numerical stability, the techniques can only be safely used for diagonally dominant systems. Generalizations, in particular of the cyclic reduction technique, to block tridiagonal systems can be practical for specific classes of matrices. As a prime example, we think of the Fast Poisson Solvers.

Discussion of approaches to parallel factorization
For the factorization of tridiagonal matrices there is not much that can be done in parallel if we do not want to do additional work, by rearranging the order of computation. For large enough bandwidths, we can use block algorithms and Level 3 BLAS as for dense matrices (see Section 2). The different parallel techniques for the factorization of tridiagonal matrices fall into four classes:

1. *Twisted factorization* [19]
2. *Recursive doubling* [187]
3. *Cyclic reduction* [120, 135]
4. *Divide and Conquer* [34, 42, 146, 147, 205]

We will briefly discuss these techniques in the following paragraphs.

Twisted factorization
Twisted factorization amounts to starting Gaussian elimination simultaneously from top-down and from bottom-up. In matrix notation, if T is the matrix being factorized, this leads to a factorization $T = RS$, in which the first half of R is a lower bidiagonal matrix and the second half is an upper bidiagonal matrix. The matrix S has the transpose form of this structure. The process is also known, for obvious reasons, as the BABE (*Burn At Both Ends*) algorithm. It was known to Wilkinson and was used in LINPACK [71]. This factorization was analysed by Babuska [19], who studied it for the accurate computation of the k-th component of the solution vector. The degree of parallelism is clearly only two. The approach is sometimes used for the construction of parallel preconditioners, where it is generalized to block tridiagonal systems. This may lead to additional parallelism on top of other techniques for the construction of parallel preconditioners, see Section 5 for more details.

Recursive doubling
The *recursive doubling* approach works as follows. We assume that the matrix T has been scaled so that $\mathrm{diag}(T) = I$, and we write T as $T = B + I + C$, where B

is the strictly lower diagonal part and C is the strictly upper diagonal part. After multiplying the system by $-B+I-C$ we obtain a matrix with again three nonzero diagonals, but now the off-diagonal nonzero entries are in positions (i,j) with $|i-j|=2$, that is their distance to the main diagonal is doubled. We can repeat this and, after $\lceil\log_2(n)\rceil$ steps, there is only a diagonal matrix left. The only operation is to multiply vectors and matrices by matrices with three nonzero diagonals so that all steps can be done in parallel, and the system $Tx=b$ can be solved in $\lceil\log_2(n)\rceil$ steps. This approach is elegant but has not led to practical implementations, because it requires n processors to complete in $\lceil\log_2(n)\rceil$ steps. The parallelism is too fine-grained, after each step there is much communication for redistribution of the data, and progressively more processors become idle in the process, which leads eventually to poor efficiency. We are not aware of generalizations of recursive doubling for banded systems.

Cyclic reduction
With *Cyclic reduction*, the entry in position $(2i, 2i-1)$ is eliminated using the $(2i-1)$-th row, and entry $(2i, 2i+1)$ is eliminated using row $2i+1$. This leads to fill-in in positions $(2i, 2i-2)$ and $(2i, 2i+2)$. This eliminates all odd numbered unknowns, and clearly the even numbered unknowns are now coupled only to each other and we have a tridiagonal system for the even numbered unknowns. This reduction to a tridiagonal system of order $n/2$ can be done in parallel. If this reduced system is solved, the odd numbered unknowns can be recovered in parallel as well. The reduction can be nested until the remaining system is small enough to be solved by a few serial computations. From a parallel point of view, this approach is not very practical because the parallelism is too fine-grained, but since the computational complexity is only increased by a factor of about two, the technique has been exploited for constructing efficient code for vector computers for which the vector speed is significantly faster than the scalar speed [59, 159, 178, 201]. The technique was generalized to banded systems by Dubois and Rodrigue [75]. For block tridiagonal systems, recursive reduction can lead to very efficient (serial as well as parallel) algorithms (for example, in Fast Poisson Solvers [37, 191, 80]). The performance of a block tridiagonal solver, based on a block cyclic reduction technique, on the 28-processor Fujitsu VPP500, is reported in [188].

Divide and conquer
For *Divide and Conquer*, we partition T, perhaps after augmenting it, into p blocks of size k. We eliminate in parallel the lower diagonal entries within the diagonal blocks of T. Of course, this leads to fill-in outside the diagonal blocks. In what is known as Wang's method ([205], but proposed earlier in a more general form by [42]) the method proceeds by eliminating the upper diagonal entries in each diagonal block, and the entry in the off-diagonal block above the diagonal block, starting with those in row $k-2$ and upwards. This produces fill-ins in the last column of each

diagonal block, in the same column of T where the fill-ins in the downward sweep occurred.

With row k of the first block, the fill-ins below the diagonal entry can be eliminated in parallel, without producing new fill-ins. This can be repeated for the other diagonal blocks in succession. After the downward sweep, a similar upward sweep can be done for the elimination of the fill-ins in the upper triangular part of T. These steps require redistribution of the fill-ins for each off-diagonal block over the processors, because all the fill-in columns cannot be processed simultaneously. This part of the process yields fine-grained parallelism. It was shown in [153], that this is not efficient because of all the data movement. Therefore, the original variant has been superseded by a more coarse-grained one [80, 153, 154], which proceeds as follows. The downward and upward sweep for elimination of off-diagonal elements is exactly the same as for the original algorithm, but the removal of the spikes is different. If we take row k from each block of the partially reduced matrix, then these rows together form a tridiagonal system of dimension p. We solve this small system in serial mode, and then the remaining part of the system can be solved again in parallel, without further data redistribution.

In [153] it is shown that, if the size of the blocks is much larger than p, then the wall-clock time on a p-processor system is roughly $p/2$ times less than standard Gaussian elimination on a single processor: the parallel process is almost perfectly parallel, which gives a factor p, but is about twice as expensive as serial Gaussian elimination. For the original fine-grained approach the speedup is bounded by the minimum of $p/2$ and α/β, where α denotes the cpu-time for one floating-point operation and β is the time for transport of one datum to another processor. For modern computers, this ratio is usually less than 1.0, which makes the original method impractical. A generalization of Wang's divide and conquer technique for banded systems is discussed in [153].

In a pure Divide and Conquer approach, the tridiagonal matrix would be split into smaller tridiagonal matrices and one would concentrate on the elimination of the few off-diagonal entries that are left over outside the blocks. The algorithm by Bondeli [34] is based on the observation that if we split off from T a block diagonal part (which can be processed in parallel), then the remaining off-diagonal blocks have only at most a single nonzero entry. They can be interpreted as rank one matrices, for example $t_{k,k+1}e_k e_{k+1}^T$. These matrices link the unknowns associated with the different blocks and, by using expressions for rank-one updates; the corrections for all the blocks can be computed from the solution of a small p by p subsystem. If one simply splits off blocks along the diagonal, without attempting to correct these blocks for the off-block elements, then this can be interpreted as a Dirichlet-Dirichlet coupling in a domain decomposition setting. Mehrmann [146] has generalized Bondeli's approach by changing some entries within the diagonal blocks or, in domain decomposition terms, by changing the interface conditions. This leads to higher rank update corrections that can be removed iteratively. He also discusses the parallel solution of block tridiagonal matrices related to PDE's in two dimensions. In [192], the Sherman-Morrison-Woodbury formula (see [105]:page 50) is used to eliminate the rank-one corrections. All these approaches lead to coarse-grained parallelism, and a reduction of the serial step. Bondeli's approach has even a slightly lower computational complexity than that of Wang. He reports, on experiments with systems of order 8190, a parallel efficiency of the order of 90%, which reduces to an effective efficiency of about 45%, relative to standard Gauss elimination on a single processor.

Discussion
In conclusion, we see that there has been a shift from almost completely parallel, but fine-grained and communication-intensive algorithms to algorithms that have coarse-grained parallelism with very low communication, but at the price of a serial step. This serial step becomes relatively less important when the block sizes can be chosen large enough. The fine-grained cyclic reduction approach may still be useful for vector processing, and for special block tridiagonal systems. The modern coarse-grained algorithms are only attractive for parallel processing if the size of the matrix is large, say in the order of a few hundred times the number of processors. Moreover, the computational complexity of the divide and conquer approach is about twice as great as the computational complexity of the serial implementation of Gaussian elimination. This means that this approach is seldom useful for the tridiagonal systems associated with single grid-lines, that arise when discretizing PDEs over regular grids with finite differences in two or three dimensions. The tridiagonal subblocks in the matrices will not often be of order greater than, say, 1000. For such block tridiagonal matrices, one usually considers solution techniques in which the tridiagonal subsystems are solved in parallel (for instance as in Block Jacobi Preconditioning) rather than solving each separate tridiagonal subsystem in parallel. We will discuss these types of solution techniques in Section 5.

4 Iterative methods

In our discussion of the past and current trends in the parallelisation of iterative methods, we distinguish between two different aspects of the iterative solution of a linear system $Ax = b$. The first one that we address in this section is the particular acceleration technique for a sequence of iteration vectors, that is the technique used to construct a new approximation for the solution x, with information from previous approximations. This leads to specific iteration methods, such as SOR, Conjugate Gradients, etc. The second aspect is the transformation of a given system to one that can be more efficiently solved by a particular iteration method. This is called preconditioning, and it will be addressed in the next section.

For the early parallel computers, in existence in the eighties, it was observed that the single iteration steps of most iteration methods offered too little opportunity for effective parallelism, in comparison with, for instance, direct methods for dense matrices. In particular, the few inner products that are required per iteration step for many iteration methods, were identified as obstacles because of communication. This has led to attempts to combine iteration steps, or to combine the message passing for different inner products. For more modern parallel computers, the inner products are relatively less important and we see that attention has shifted more to the preconditioning aspect. Nevertheless, we believe that there is still value in many of the approaches for parallelism in the iteration methods, in particular, because they may help to reduce performance degrading effects of memory hierarchy. This is still a problem for iterative sparse matrix solvers. As we have seen in the section on direct methods, one tries to exploit blocking techniques as much as possible in order to exploit local memories and to hide delaying effects of data transport from slower memory. For some relevant large sparse systems, this leads to performances of about 50% of those for large dense matrices. For standard iterative solvers this is more in the order of 10%, which makes it still opportune to seek for techniques that help to bridge that gap.

Comments on methods and kernels

The currently most popular iterative methods belong to the class of Krylov subspace methods. We will first give a brief overview of some characteristic and well known Krylov subspace methods, where we denote the Krylov subspace by $K^i(A; r_0)$ which is defined as the space spanned by the vectors $\{r_0,\ Ar_0,\ A^2r_0,\ \ldots,\ A^{i-1}r_0\}$.

(a) If A is symmetric positive definite, then the *Conjugate Gradient* method [123] generates, using two two-term recurrences, the x_i for which $(x - x_i, A(x - x_i))$ (the so-called $A-$norm or energy norm) is minimized over all vectors in the current Krylov subspace $K^i(A; r_0)$, where r_0 is the initial residual $b - Ax_0$.

(b) If A is only symmetric but not positive definite, then the *Lanczos* [136] and the *MINRES* methods [160] may be considered. In MINRES, the $x_i \in K^i(A; r_0)$ is determined for which the two-norm of the residual ($\|b - Ax_i\|_2$) is minimized, while the Lanczos method leads to an x_i for which $b - Ax_i$ is perpendicular to the Krylov subspace. From a parallel implementation point of view, these methods have very similar performance to Conjugate Gradients.

(c) If A is unsymmetric, then we can compute the $x_i \in K^i(A, r_0)$, for which the residual is minimized in the Euclidean norm. This is done by the GMRES method [177]. Unfortunately, this requires i inner products at the i-th iteration step, as well as i vector updates. In a parallel environment, the communication required for the inner products may seriously degrade the performance of GMRES.

(d) If A is unsymmetric, it is in general not possible to determine an optimal $x_i \in K^i(A, r_0)$ with short recurrences. This was proved in [87]. However, with short recurrencies as in Conjugate Gradients, we can compute the $x_i \in K^i(A; r_0)$, for which $b - Ax_i \perp K^i(A^T; s_0)$ (usually, one selects $s_0 = r_0$). This leads to the *Bi-Conjugate Gradient method* [88]. Apart from the fact that this requires operations with A^T, this method has very much the same parallel properties as Conjugate Gradients. A clever variant is QMR [89] that has smoother convergence behaviour and is more robust than Bi-Conjugate Gradients.

(e) An interesting observation is that the operations with A^T in the Bi-Conjugate Gradient method can be replaced by operations with A itself, by using the observation that $< x, A^T y >$ equals $< Ax, y >$, where $< \dots >$ represents the inner-product computation. Since the function of the multiplications by A^T in Bi-CG serves only to maintain the dual space to which residuals are orthogonalized, the replacement of this operator by A allows us to expand the Krylov subspace and to find better approximations, for virtually the same costs per iteration as for Bi-Conjugate Gradients. Indeed, a rule of thumb is that the CGS method converges twice as fast as Bi-CG, although the downside is that it can diverge twice as fast also !! This leads to so-called hybrid methods such as CGS [186], Bi-CGSTAB [200], Bi-CGSTAB(ℓ) [183], TFQMR [90], and hybrids of QMR [39]. In particular the Bi-CGSTAB(ℓ) methods offer some possibilities for further improving the parallel performance, as we will see later.

The dominant computational kernels for these iterative methods are:

1. Sparse matrix-vector product computations, that is the computation of Ap and/or $A^T p$, for vectors p that are generated by the algorithm.

2. A preconditioning step, that is either the computation of Mq, where M is an approximation for A^{-1}, or the solution of a system $Kz = q$, where K is an approximation for A. See Section 5 for more information on this.

3. Vector updates (so-called (s)axpy's).

4. Inner products.

There is a current activity, called the BLAS Technical Forum, which includes an attempt to standardize and provide model implementations for the sparse BLAS, that includes both triangular solves with sparse triangular matrices and sparse matrix-vector and sparse matrix-(dense) matrix multiplication. This activity is described in the Web page `http://www.netlib.org/cgi-bin/checkout/blast/blast.pl` from where current versions of the standard can be obtained. A proposal for a User Level interface to a set of sparse BLAS kernels specifically designed for use with iterative methods is presented in [82].

For a distributed memory machine, at least some of the steps require communication between processors: the accumulation of inner products and the computation of the sparse matrix-vector products (the amount of communication depending on the nonzero structure of the matrix and the distribution of the nonzero entries over the processors).

Sparse matrix-vector products
We will first discuss parallel aspects of the sparse matrix-vector product computation. In the early days of parallel computing when the architecture and software were more primitive than today, the interconnection of the processors was more visible to the programmer and it was very important to tune one's code for the particular interconnect, particularly to avoid sending data over long paths in the interconnection networks between processors. This caused a substantial preoccupation with the distribution of the nonzero entries over the processors. For instance, McBryan and van de Velde discuss the sparse matrix-vector product computations on hypercube architectures [145]. Ortega ([159]:Chapter 3) discusses a simple scheme for grid-oriented problems, based on domain decomposition. Radicati and Vitaletti [166] investigated, amongst others, a generalization of the diagonal representation of a matrix, that was suggested for regularly structured matrices by Madsen, Rodrigue and Karush [142]. This *compressed-diagonal representation* led to a good performance on parallel shared memory vector computers, in particular the IBM 3090 VF computers. Pommerell [162] discusses the implementation of sparse matrix-vector products for various data storage schemes and for various multiprocessor architectures. He reports performances for irregular sparsity patterns, using distribution heuristics suggested in [163], associated with matrices from semiconductor device simulations. For finite-element applications related to CFD, Johan *et al.* [125, 126] report on a parallel implementation with minimal communication overhead on the CM-5 computer. In this study, the emphasis was on data parallel techniques for the sparse matrix-vector product. Bisseling and McColl [32] have investigated the performance of two-dimensional matrix distributions in the general-purpose framework of the bulk synchronous parallel model. They found that good performance can only be achieved if the underlying physical structure of the matrix is exploited. See also [181] for the partitioning of unstructured matrices for parallel processing. Research on minimizing the number of messages, for the matrix-vector product, and the lengths of the paths over which the messages are sent have been reported in [45, 60]. In [61], it is shown that, if the matrices come from discretized PDEs, the time for communication can almost completely be overlapped with computational work. The distribution of the matrix over the processors is such that there is only neighbour-neighbour communication, if carried out on a mesh-connected parallel machine.

Access to main memory is a very serious issue, even for serial performance. In fact, parallelism does sometimes alleviate the degradation in performance that is observed for a single processor. If the number of processors is increased, then we sometimes see a superlinear speedup. This is because more of the data can be kept in the local memories (caches) of the processors. For much larger problems, the required vectors usually cannot be kept entirely in fastest memory (cache or registers). In that case, the vectors for each operation have to be transferred from slower memory to the caches and because there is not enough computational work to do, the speed of data transport from the slowest part of memory becomes the dominating factor. Suggestions have been made to rearrange operations in iterative algorithms to make Level 3 BLAS operations possible (be it to a very modest degree). For Conjugate Gradients, the most well known approach is to combine a number of s successive steps [46]. In this approach, the vectors $\{Ar_0,\ A^2r_0,\ ....\ A^{s-1}r_0\}$ are first all computed before any orthogonalization of the basis is performed. The main problem is that the success of this approach depends critically on the eigenvalue distribution of the matrix A, because it can lead to unstable computations, even for modest values of s. This may explain why this approach has not become very popular because the gain in performance does not necessarily compensate for the increased instability (see also [175]). A similar approach, suggested by Chronopoulos and Kim [47] for restarted processes such as GMRES(m), seems to be more effective, mainly because the restarts prevent accumulated instability. The algorithm can be made more robust by orthogonalization, in the A^TA norm, of the direction vectors within each s-block [49]. This permits values of s up to 16. They show, by experiments on a CRAY C90, small performance improvements with respect to a parallel implementation of regular GMRES. Li [138] proposes a variation on the s-step GMRES method that performs better on processors with cache memory (even in uniprocessor mode). His experiments also suggest a better behaviour with respect to numerical stability.

A partial solution for the reduction of memory references is to combine the updates for the approximate solution into one single block-vector update for a number of successive iteration steps [197]. This does not affect the numerical stability of the scheme. It can even be used to improve the accuracy of the iteration method, especially if it is done for irregularly converging methods such as CGS, in a way known as *reliable updating* [115, 184]. The main problem in this case is that if large errors are introduced into the iterates x_i (that are not usually computed explicitly), they can swamp any meaningful information previously present in the iterates, from which it is impossible to recover later (see, for instance, [107, Chapter 7.3]). In the reliable updating technique, corrections to the approximation are not applied at each step but are grouped so that the resulting update is of more comparable magnitude to the solution vector. Thus the disastrous consequences of blips in the convergence process can be avoided. The decision on how to group updates is based on monitoring the residual (which is normally computed at each step) and the update is performed only when the residual vector has decreased in norm since the previous group update. With certain precautions to preventing groups being too large, this leads to very accurate approximate solutions.

A particularly favourable situation is when there are several right-hand sides for which the system has to be solved. The Krylov subspace methods generate different subspaces for different right-hand sides, but one can try to construct a slightly bigger subspace that approximates the union of all the different subspaces. This approach is known as a *Block method*, see for instance [157, 158, 182]. From the point of view of computational complexity, this is not very effective since the subspace is in general not optimal for any of the systems separately but, for parallel processing and for systems with limited fast local memory, many of the computations can be logically combined. This may lead to better use of local data and less communication overhead. If there are many more right-hand sides than processors, then it usually is more efficient to give each processor its own set of linear systems (if memory permits the storage of all the required information).

Inner products

The emerging feeling is that, with adequate coding, the matrix-vector products do not necessarily lead to serious performance-degrading communication problems on modern parallel computers, not even for relatively small-sized problems. This is not the case for the inner products. The common approach in Krylov subspace iteration methods is to generate an orthogonal basis for the Krylov subspace and then to consider the restriction of the given system $Ax = b$ with respect to this basis. Schematically, the computationally intensive part of these methods can be represented as follows:

Let v_1, ..., v_{i-1} be the orthogonal basis at step $i-1$, then v_i is constructed, with Modified Gram-Schmidt, as:

$$\begin{array}{l} t = Av_{i-1} \\ \text{for } j = 1, \ldots, i-1 \text{ do} \\ \quad t = t - (t^T v_j) v_j \\ v_i = t / \|t\|_2 \end{array} \tag{3}$$

Inner products always act in a parallel environment as synchronization points and require global communication. On distributed memory machines they form, apart from the preconditioning, a major bottleneck. It is thus no surprise that this aspect has received much attention in the literature. For the inner product, we need global communication for both the reduction operation and for the broadcast of the assembled inner product, because all processors need to know the result. For a $p \times p$ processor grid ($P = p \times p$), these communication costs are proportional to p. This means that, for a constant length of vector segment per processor, these communication costs will dominate when p is large enough. This is unlike the situation for the matrix-vector product and may be a severely limiting factor in achieving high speedups in a massively parallel environment, when n/P is only modest, say a few hundred.

Of course, implementing an inner-product in parallel can lead to a different sequencing of the arithmetic operations in the inner product and thus, because of finite precision, could result in a different value for the computed product. Since the value could have an important bearing on the convergence of the iterative method, concern has been expressed about this. We will not comment further other than to remark that such a high sensitivity could well be a sign of poor conditioning of the underlying problem.

We will first discuss this situation in more detail for the Conjugate Gradient method. In the Conjugate Gradient method, the new basis vector has to be made orthogonal to only the previous two basis vectors and this might lead one to expect that the inner products are less of a problem for this method. In [54], it is shown that, for matrices of order $90000P$ with only five nonzero entries per row, the communication will dominate when the number of processors P is greater than 400 on a Parsytec GCel computer. Note that the order of the matrix scales with the number of processors, but nevertheless the communication for the two inner products will eventually dominate. The main problem is that the communication for the two inner products cannot be combined, because the two inner products cannot be executed immediately after each other.

Several authors ([46, 58, 86, 150, 151]) have attempted to reduce the number of synchronization points (and to improve the computation to memory reference ratio). The scheme of Meurant [150] is the prototype for these attempts: the two separated inner products can be replaced by three consecutive inner products. They can be computed in parallel and the communications can be combined. The iteration parameters can be computed from these three inner products, although at the cost of reduced numerical stability. In this scheme, the ratio between computations and memory references is also better than for the standard scheme. A similar scheme has been suggested by Bücker and Sauren [36] for the BiCG and the QMR method although they do not give any actual parallel performance results. The numerical stability of these and other schemes is a point of serious concern, since the restructured computations may lead to cancellation in the iteration parameters of CG. See [57] for a discussion on the stability aspects of these GG schemes with reduced synchronization.

In [65] another variant of CG was suggested, in which there is more possibility for overlapping all of the communication time with useful computations. This variant is nothing but a rescheduled version of the original CG scheme and is therefore equally stable. The key trick in this approach is to delay the updating of the solution vector by one iteration step. This creates a possibility for overlap, since the update does not have to wait for the completion of the inner products. De Sturler [63] reports some modest improvements ($\leq 10\%$), using this approach for rather small systems (order 10000) on a mesh-based Parsytec Supercluster (400 processors).

For stability reasons, the GMRES algorithm [177] is based upon the modified Gram-Schmidt orthogonalization scheme, described in (3), where clearly the number of inner products per iteration step increases linearly, since the new basis vector for the Krylov subspace has to be made orthogonal to all previous basis vectors. In practical implementations, one often prefers to restart after each cycle of m iterations (instead of increasing the value of m), and this is referred to as GMRES(m). This then limits the maximum number of synchronization points and the communication overhead (start-up time and the time for message passing) to m on any one iteration. An alternative would be to compute first a suitable, but not necessarily orthonormal, basis [48]:

$$\{v_1, f_1(A)v_1, \ldots, f_{m-1}(A)v_1\}, \tag{4}$$

(the f_i denote non-degenerate polynomials of degree i) and to orthogonalize the $\{f_i(A)v_1\}$ simultaneously. This offers more possibilities for parallelism. The notation f_i is used to indicate that the successive vectors are elements of Krylov subspaces of increasing dimension. A suitable mechanism for creating such a basis is to use the h_{ij} of a previous GMRES(m) cycle ([185, Section 6], see also [20]). Although the resulting basis will not be orthogonal, it may still be a good basis since the construction rules for the subspace may not have changed too much from the previous sequence. The Gram-Schmidt orthogonalization can then be performed on these basis vectors using Level 2 BLAS and the required communication can be largely overlapped [61, 64]. The resulting entries for the new H matrix can then be used for the next sequence of steps, as before. See [64] for an analysis of the parallel performance of this approach and for illustrative experiments. The remarkable result is that, in spite of the large number of inner products in GMRES, we can obtain a more parallelizable code than for Conjugate Gradients although CG has only two inner products per step. Vuik, van Nooijen, and Wesseling [204] show that almost optimal scalability can be obtained for GMRES on a 128-processor CRAY T3D, even with modified Gram-Schmidt for the orthogonalization step, if one uses a domain decomposition inspired partitioning for the evaluation of the inner products, the matrix-vector product, and the preconditioning. The linear systems came from CFD applications, and the number of unknowns was up to $65,536$ per processor.

In recent years, so-called hybrid variants of Bi-CG have been proposed. The most well-known variants are CGS [186], Bi-CGSTAB [200], and Bi-CGSTAB(ℓ) [114, 183, 185]. In these variants, the operations with A^T have been replaced by operations with A, and these operations are exploited in an attempt to construct better approximations to the solution, as we discussed in bullet (e) near the beginning of this section. The CGS method, which just replaces the A^T product by the same product with A, corresponds to applying the product of two Bi-CG polynomials, but one has the freedom to use other polynomials for the replaced A^T operations. For example, if GMRES(1) iterations are used, we obtain the Bi-CSTAB algorithm. The Bi-CGSTAB(ℓ) methods can be seen as products of Bi-CG and GMRES(ℓ), that is the effect in the $i \times \ell$-th iteration can be interpreted as the combined effects of $i \times \ell$ steps of Bi-CG and i times a GMRES process with ℓ iterations. These hybrid methods offer better possibilities for parallelism, especially on distributed memory computers. In these hybrid methods, the GMRES part involves ℓ steps on top of every successive ℓ Bi-CG steps (in practice, $\ell \leq 4$), and the required basis can be generated in some convenient way (as in (4)), so that the inner products for the orthogonalization can be computed simultaneously. Also some of the vector updates can be combined which leads to fewer memory references. For Bi-CGSTAB(ℓ), this is discussed in more detail in [74, Section 8.2.11].

Discussion
We have the feeling that the efforts for constructing more effective iterative methods have come to a (temporary?) stop. The very large systems that one wants to solve with parallel computers, usually have a special structure that can be exploited to obtain an efficient direct solution or, if that leads to too much fill-in, have a rather regular structure in the sense that for an iterative solver, the load balancing per iteration is not any longer a serious point of concern. The main objective of parallelization should always be to reduce wall-clock time and the approaches that have been sketched in this section lead to marginal improvements when compared with what an effective preconditioning can do. A good preconditioner may reduce the CPU time by a large factor, whereas the approaches in this section lead to modest percentages, especially for large problems (say $n/P \gg 10000$). Heise and Jung [119] report on experiments on the Power Xplorer (16 processors), CG-Power Plus (128 processors), Multicluster 2 (16 Transputers), and a Workstation cluster (8 SPARCs), that show very good scalability for preconditioned Conjugate Gradients applied to a mixed FEM-BEM problem with $1,514,008$ unknowns. Parallelism was exploited in the matrix-vector product and a domain decomposition preconditioner; the Conjugate Gradient method itself had not been modified. More general software packages for iterative methods also try to identify the parallelism in the separate computational elements (matrix-vector products, inner products, axpy's, and preconditioning steps), instead of reformulating the iterative schemes in the ways that we have sketched above. For information on some of these packages we refer to:

- AZTEC. For more information see:
  `http://www.cs.sandia.gov/HPCCIT/aztec.html`
- ITPACK. For the parallel implementation of this package see: [131].
- PARASOL. For more information see Section 6.
- PETSc. For more information see:
  `http://www.mcs.anl.gov/petsc/`

- PIM. Information for this parallel software package can be obtained from: `ftp://unix.hensa.ac.uk/pub/misc/netlib/pim/`
- PINEAPL. For more information see: `http://www.nag.co.uk/projects/PINEAPL/`
- SPARSKIT and SPARSELIB. For more information see: `http://www/cs.umn.edu/~saad/`

5 Parallel Preconditioning

There are many occasions and applications where iterative methods fail to converge or converge very slowly. The usual remedy is to apply a preconditioner, that is instead of $Ax = b$, one solves $KAx = Kb$ or a spectrally equivalent system, for example, $AKy = b$. The general problem of finding an efficient preconditioner, is to identify a linear operator K (the *preconditioner*) with the properties that:

1. K is a good approximation to A in some sense.
2. The cost of the construction of K is not prohibitive.
3. The system $Ky = z$ is much easier to solve than the original system.

By efficient, we mean that the iteration method converges much faster, in terms of CPU time, for the preconditioned system.

The choice of K varies from purely "black box" algebraic techniques which can be applied to general matrices to "problem dependent" preconditioners which exploit special features of a particular problem class. Although problem dependent preconditioners can be very powerful, there is still a practical need for efficient preconditioning techniques for large classes of problems. We refer the reader to [18, 41, 176] for further discussions on this. In this section, we will not go in details about preconditioning but rather give a sketch of important ideas for the construction of parallel preconditioners. For more details on implementation for high performance computers, see [74].

Originally, preconditioners were based on direct solution methods in which part of the computation is skipped. This leads to the notion of *Incomplete* LU (or ILU) *factorization* [18, 149, 176], The incomplete factors $\widetilde{L}$ and $\widetilde{U}$ define the preconditioner $K = (\widetilde{L}\widetilde{U})^{-1}$. In the context of an iterative solver, this means that we have to evaluate expressions like $z = (\widetilde{L}\widetilde{U})^{-1}y$ for any given vector y. This is done in two steps: first obtain w from the solution of $\widetilde{L}w = y$ and then compute z from $\widetilde{U}z = w$. Straightforward implementation of these processes leads to recursions, for which vector and parallel computers are not ideally suited. This sort of observation has led to reformulations of the preconditioner, for example, with reordering techniques or with blocking techniques. It has also led to different types of preconditioners, including diagonal scaling, polynomial preconditioning, and truncated Neumann series. These approaches may be useful in certain circumstances, but they tend to increase the computational complexity, because they often require more iteration steps or make each iteration step more expensive. Diagonal scaling can be done explicitly, without any further complications for parallel processing. This can be done before and in addition to the construction of another preconditioning.

Changing the Order of Computation

In some situations, it is possible to change the order of the computations without changing the results. A prime example is the ILU preconditioner for the 5-point finite-difference operator over a rectangular m by m grid. Suppose that we have indexed the unknowns according to their positions in the grid, lexicographically as $x_{1,1}$, $x_{1,2}$, $\ldots$, $x_{1,m}$, $x_{2,1}$, $\ldots$, $x_{m,m}$. Then, for the standard ILU(0) preconditioner, in which all fill-ins are discarded, it is easily verified that the computations for the unknowns $x_{i,j}$ can be done independently of each other along diagonals of the grid (grid points for which the sum of the indices is constant). This leads to vector code but, because there is only independence along each diagonal, the parallelism is too fine-grained. In three-dimensional problems, there are more possibilities for obtaining vectorizable or parallel code. For the standard 7-point finite-difference approximation of elliptic PDEs over a regular rectangular grid, the equivalent of the diagonal in two dimensions is known as the hyperplane: a set of grid points for which the sum of the three indices is constant. It was reported in [179, 199] that this approach can lead to satisfactory performance on vector computers. For the CM-5 computer, a similar approach was developed in [30]. The obvious extension of hyperplanes (or diagonals) to irregular sparse matrices, defines the *wavefront ordering*, discussed in [166]. The success of a wavefront ordering depends very much on how well a given computer can handle indirect addressing. In general, the straightforward wavefront ordering approach gives too little opportunity for efficient parallelization.

Vuik, van Nooyen and Wesseling [204], generalize the wavefront approach to a block wavefront approach, using ideas that were originally proposed for parallel multigrid in [24]. They present results of experiments on a 128-processor CRAY 3TD. Van Duin [202, Chapter 3], uses graph concepts for the detection of parallelism. He attempts to identify strongly connected components for which independent ILU factorizations can be made. A drop tolerance strategy is used to create a large enough number of such components. This leads to the concept of MuliILU.

Reordering the Unknowns

A standard trick for exploiting parallelism is to select all unknowns that have no direct relationship with each other and to number them first. We discussed an algorithm of this kind when considering cyclic reduction of tridiagonal systems in Section 3. For the 5-point finite-difference discretization over rectangular grids, this approach is known as a *red-black ordering*. For elliptic PDEs, this leads to very parallel preconditioners. The performance of the preconditioning step is as high as the performance of the matrix-vector product. However, changing the order of the unknowns leads in general to a different preconditioner. Duff and Meurant [83] report on experiments that show that most reordering schemes (for example, the red-black ordering) lead to a considerable increase in iteration steps (and hence in computing time) compared with the standard lexicographical ordering. For the red-black ordering associated with the discretized Poisson equation, it can be shown that the condition number of the preconditioned system is only about one quarter that of the unpreconditioned system for ILU, MILU and SSOR, with no asymptotic improvement as the grid size h tends to zero [134].

One way to obtain a better balance between parallelism and fast convergence, is to use more colours [68]. In principle, since there is not necessarily any independence between different colours, using more colours decreases the parallelism but increases the global dependence and hence the convergence. In [69], up to 75 colours are used for a 76^2 grid on the NEC SX-3/14 resulting in a 2 Gflop/s performance, which is much better than for the wavefront ordering. With this large number of colours the speed of convergence for the preconditioned process is virtually the same as with a lexicographical ordering [68].

The concept of *multi-colouring* has been generalized to unstructured problems by Jones and Plassmann [128]. They propose effective heuristics for the identification of large independent subblocks of a given matrix. For problems large enough to get sufficient parallelism in these subblocks, their approach leads to impressive speedups compared to the natural ordering on a single processor.

Meier and Sameh [148] report on the parallelization of the preconditioned CG algorithm for a multivector processor with a hierarchical memory (for example the Alliant FX series). Their approach is based on a red-black ordering in combination with forming a reduced system (Schur complement).

Another approach, suggested by Meurant [151], exploits the idea of the two-sided (or twisted) Gaussian elimination procedure for tridiagonal matrices that we discussed in Section 4. This is generalized for the incomplete factorization. Van der Vorst [198] has shown how this procedure can be done in a nested way. For 3D finite-difference problems, twisting can be used for each dimension, which gives an increase in parallelism by a factor of two per dimension. This leads, without further computational overhead, to incomplete decompositions, as well as triangular solves, that can be done in eight parallel parts (2 in each dimension).

Meurant [152] reports on timing results obtained on a CRAY Y-MP/832, using an incomplete repeated twisted block factorization for two-dimensional problems. For this approach for preconditioned CG, Meurant reports a speedup of nearly 6 on an 8-processor CRAY Y-MP. This speedup has been measured relative to the same repeated twisted factorization process executed on a single processor. Meurant also reports an increase in the number of iteration steps as a result of this repeated twisting. This increase implies that the effective speedup with respect to the best non-parallel code is only about 4.

Element by Element Preconditioners
In finite-element problems, it is not always possible or sensible to assemble the entire matrix, and it is as easy to form products of the matrix with vectors as when it is held in assembled form. Furthermore it is easy to distribute such matrix multiplications to exploit parallelism. Hence preconditioners are required that can be constructed at the element level. Hughes *et al.* [124] were the first to propose such *element by element* preconditioners.

A parallel variant is suggested in [112]. For symmetric positive-definite A, they decompose each element matrix A_e as $A_e = L_e L_e^T$, and construct the preconditioner as $K = LL^T$, with

$$L = \sum_{e=1}^{n_e} L_e.$$

In this approach, non-adjacent elements can be treated in parallel. An overview and discussion of parallel element by element preconditioners is given in [203]. To our knowledge, the effectiveness of element by element preconditioners is limited, in the sense that it does not often give a substantial improvement of the CPU time.

Polynomial Preconditioning
The main motivation for considering polynomial preconditioning is to improve the parallel performance of the solver, since the matrix-vector product is often more parallelizable than other parts of the solver (for instance the inner products). By doing so, all implementation tricks for the matrix-vector product can easily be exploited. The main problem is to find effective low degree polynomials $p_k(A)$, so that the iterative solver can be applied to $p_k(A)Ax = p_k(A)b$. With m steps of a Krylov solver, this leads to a Krylov subspace

$$K^m(p_k(A)A; r_0) = \mathrm{span}(r_0, p_k(A)Ar_0, \ldots, (p_k(A)A)^{m-1}r_0),$$

and this is a subspace of the Krylov subspace $K^{(k+1)(m-1)+1}(A; r_0)$. The point is, that we have arrived in a high dimensional subspace (with "holes"), for the overhead costs of only m iteration steps. The hope is that for clever choices of p_k, this high dimensional subspace, with holes, will contain almost the same good approximation to the solution as the full Krylov subspace. If so, then we have saved ourselves all the overhead associated with the $(k+1)(m-1)$ iteration steps, that are needed to create the full subspace.

A big problem with polynomial preconditioning is that the aforementioned "holes" can cause one to miss important directions and so often many more iterations are required. Thus this form of preconditioning is usually only beneficial on a platform where inner products are expensive and for methods rich in inner products, like GMRES.

One approach for obtaining a polynomial preconditioner, reported in [76], is to use the low order terms of a Neumann expansion of $(I - B)^{-1}$, if A can be written as $A = I - B$ and the spectral radius of B is less than 1. It was suggested in [76] to use a matrix splitting $A = K - N$ and a truncated power series for $K^{-1}N$ when the condition on B is not satisfied. More general polynomial preconditioners have also been proposed (see, for example, [14, 127, 174]). These polynomials are usually shifted Chebyshev polynomials over intervals that are estimated from the iteration parameters of a few steps of the unpreconditioned solver, or from other spectral information.

Sparse Approximate Inverse (SPAI)
The main reason why explicit inverses are not used is that, for irreducible matrices, the inverse will always be structurally dense. That is to say, sparse factorization techniques will produce a dense matrix even if some of its entries are actually zero [79]. However, this need not be a problem if we follow the flavour of ILU factorizations and compute and use a sparse approximation to the inverse. Perhaps the most obvious technique for this is to solve the problem[1]

$$\min_K ||I - AK||_F, \tag{5}$$

where K has some fully or partially prescribed sparsity structure. This problem can be expressed as n independent least-squares problems for each of the n columns of K. Each of these least-squares problems only involves a few variables and, because they are independent, they can be solved in parallel. With these techniques it is possible to successively increase the density of the approximation to reduce the value of (5) and so, in principle, ensure convergence of the preconditioned iterative method [53]. The small least-squares subproblems can be solved by standard (dense) QR factorizations [53, 106, 109]. In a further attempt to increase sparsity and reduce the computational cost of the solution of the subproblems, it has been suggested to use a few steps of GMRES to solve the subsystems [44]. A recent study indicates that the computed approximate inverse may be a good alternative to ILU [106], but it is much more expensive to compute both in terms of time and storage, at least if computed sequentially. This means that it is normally only attractive to use this technique if the computational costs for the construction can be amortized by using the preconditioner for several right-hand sides. One other problem with these approaches is that, although the residual for the approximation of a column of K can be controlled (albeit perhaps at the cost of a rather dense column in K), the nonsingularity of the matrix K is not guaranteed. Partly to avoid this, an approach that approximates the triangular factors of the inverse has been proposed [132]. The nonsingularity of the factors can be easily controlled and, if necessary, the sparsity pattern of the factors may also be controlled. Following this approach,

[1] We recall that $|| \, ||_F$ denotes the Frobenius norm of a matrix, viz. $||A||_F \equiv \sqrt{\sum_{i,j} a_{i,j}^2}$.

sparse approximations to an A-biconjugate set of vectors using drop tolerances can be generated [25, 28]. In a scalar or vector environment, it is also much cheaper to generate the factors in this way than to solve the least-squares problems for the columns of the approximate inverse [27]. Van Duin [202, Chapter 5] shows how to compute (*sparsified*) inverses for incomplete Cholesky factors and Zhang [209] has developed a parallel preconditioning using incomplete triangular factors of the inverse.

Although almost every paper on approximate inverse preconditioners states that the authors are working on a parallel implementation, it is only quite recently that papers on this have appeared [21, 22, 26]. For highly structured matrices, some experiences have been reported in [108]. Gustafsson and Lindskog [113] have implemented a fully parallel preconditioner based on truncated Neumann expansions [196] to approximate the inverse SSOR factors of the matrix. Their experiments (on a CM-200) show a worthwhile improvement over a simple diagonal scaling.

Note that, because the inverse of the inverse of a sparse matrix is sparse, there are classes of dense matrices for which a sparse approximate inverse might be a very appropriate preconditioner. This may be the case for matrices that arise from problems in electromagnetism [2]. For some classes of problems, it may be attractive to construct the explicit inverses of the LU factors, even if these are considerably less sparse than the factors L and U, because such a factorization can be more efficient in parallel [5]. An incomplete form of this factorization for use as a preconditioner was proposed in [3].

Preconditioning by Blocks or Domains

Other preconditioners that use direct methods, are those where the direct method, or an incomplete version of it, is used to solve a subproblem of the original problem. This can be done in *domain decomposition*, where problems on subdomains can be solved by a direct method but the interaction between the subproblems is handled by an iterative technique.

Domain decomposition methods were motivated by parallel computing, but it now appears that the approach can also be used with success for the construction of global preconditioners. This is usually done for linear systems that arise from the discretization of a PDE. The idea is to split the given domain into subdomains, and to compute an approximation for the solution on each subdomain. If all connections between subdomains are ignored, this then leads to a *Block Jacobi* preconditioner. Chan and Goovaerts [40] showed that the domain decomposition approach can actually lead to *improved* convergence rates, at least when the number of subdomains is not too large. This is because of the well-known divide and conquer effect when applied to methods with superlinear complexity such as ILU: it is more efficient to apply such methods to smaller problems and piece the global solution together.

In many cases the preconditioner can be made more successful by coupling the domains, that by finding proper boundary conditions along the interior boundaries of the subdomains. From a linear algebra point of view, this amounts to adapting the diagonal blocks in order to compensate for the neglected off-diagonal blocks. This is only successful if the matrix comes from a PDE problem and if certain smoothness conditions on the solution are assumed. If, for instance, the solution were constant, then one could remove the off-diagonal block entries adding them to the diagonal block entries without changing the solution (similar to the case we discussed in the Section 3). Likewise, if the solution is assumed to be fairly smooth along a domain interface, one might expect this technique of diagonal block correction to be effective. Domain decomposition is used in an iterative fashion and usually the interior boundary conditions (in matrix language: the corrections to diagonal blocks) are based upon information from the approximate solutions on the neighbouring subdomains that are available from a previous iteration step.

Tan [193] studied the interface conditions along boundaries of subdomains and forced continuity for the solution and some low order derivatives at the interface. He also proposed including mixed derivatives in these relations, in addition to the conventional tangential and normal derivatives. The parameters involved are determined locally by means of normal mode analysis, and they are adapted to the discretized problem. It is shown that the resulting domain decomposition method defines a standard iterative method for some splitting $A = K - N$, and the local coupling aims to minimize the largest eigenvalues of $I - AK^{-1}$. Of course this method can be accelerated and impressive results for GMRES acceleration are shown in [193]. Some attention is paid to the case where the solutions for the subdomains are obtained with only modest accuracy per iteration step.

Recently, Washio and Hayami [206] employed a domain decomposition approach for a rectangular grid in which one step of SSOR is performed for the interior part of each subdomain. In order to make this domain-decoupled SSOR more like global SSOR, the SSOR iteration matrix for each subdomain is modified. In order to further improve the parallel performance, the inverses in these expressions are approximated by low-order truncated Neumann series. A similar approach is suggested in [206] for a block modified ILU preconditioner. Experimental results have been reported for a 32-processor NEC Cenju distributed memory computer.

Radicati and Robert [165] used an algebraic version of this approach by computing ILU factors within overlapping block diagonals of a given matrix A. When applying the preconditioner to a vector v, the values on the overlapped region are taken as the average of the two values computed by the overlapping ILU factors. The approach of Radicati and Robert has been further refined by de Sturler [62], who studies the effects of overlap from the point of view of geometric domain decomposition. He introduces artificial mixed boundary conditions on the internal boundaries of the subdomains. In [62] (Table 5.8), experimental results are shown for a decomposition into 20×20 slightly overlapping subdomains of a 200×400 mesh for a discretized convection-diffusion equation (5-point stencil). Using a twisted ILU preconditioning on each subdomain, it is shown that the complete linear system can be solved by GMRES on a 400-processor distributed memory Parsytec system with an efficiency of about 80% (this means that, with this domain adapted preconditioner, the process is about 320 times faster than ILU preconditioned GMRES for the unpartitioned linear system on a single processor). Since twisting leads to more parallelism, one can use bigger blocks (which usually means a better approximation). This helps to explain the good results.

Haase [116] suggests constructing an incomplete Cholesky decomposition on each subdomain and modifying the decomposition using information from neighbouring subdomains. His results, for the discretized Poisson equation in 3D, show that an increase in the number of domains scarcely affects the effectiveness of the preconditioner. Experimental results for a realistic finite-element model, on a 16-processor Parsytec Xplorer, show very good scalability of the Conjugate Gradient method with this preconditioner.

Heisse and Jung [119] attempt to improve the effectiveness of a domain decomposition preconditioner by using a multigrid V-cycle with only one pre- and one post-smoothing step of a parallel variant of Gauss-Seidel type to solve a coarse grid approximation to the problem. With the usual domain decomposition technique, effects of local changes in a domain that lead to global changes in the solution travel only to neighbouring domains at each iteration. The coarse grid corrections are used to get this globally relevant information more quickly to all domains. The combination with Conjugate Gradients, which is the underlying method used for the local subproblems, leads to good results on a variety of platforms, including a 64-processor GC Power Plus machine.

For general systems, one could apply a block Jacobi preconditioning to the normal equations which would result in the block Cimmino algorithm [13]. A similar relationship exists between a block SOR preconditioning and the block Kaczmarz algorithm [35]. Block preconditioning for symmetric systems is discussed in [50]; in [51] incomplete factorizations are used within the diagonal blocks. Attempts have been made to preorder matrices to put large entries into the diagonal blocks so that the inverse of the matrix is well approximated by the block diagonal matrix whose block entries are the inverses of the diagonal blocks [43]. In fact, it is possible to have a significant effect on the convergence of these methods just by permuting the matrix to put large entries on the diagonal and then scaling it to reduce the magnitude of off-diagonal entries [81].

Discussion

The history of parallel preconditioners is one with ups and downs. It is very difficult to construct efficient preconditioners for a given class of problems and even more so if they also have to be parallel. Initially, the attempts for parallelism were focussed on minimal changes to existing effective preconditioners (like ILU), or to exploiting the matrix A itself (polynomial preconditioners). These attempts have sometimes led to preconditioners that could be vectorized (hyperplane, etc), but for parallelism they are too fine grained. Polynomial preconditioning reduces the proportion of inner product and vector operations but at the penalty of increasing the number of iterations. It is thus usually only useful for overhead rich methods like GMRES on platforms where operations like inner products are relatively expensive with respect to the matrix-vector product. More recently there has been a shift of focus towards methods that offer more coarse-grained parallelism: domain decomposition methods and sparse approximate inverses. Despite some partial success, the sparse approximate inverses are still in their infancy. The domain decomposition techniques have proven to be successful for large classes of PDE related problems.

6 Current and future trends

In the previous sections of this paper, we have included a final subsection with some general comments and discussion on overall trends and pointers for what we believe are likely to be future directions. In this short final section, we attempt to summarize these and give our view on the main past trends and future directions for the parallel solution of sparse linear equations.

A clear trend that we have observed is from fine-grained to coarse-grained parallelism. On top of the line parallel vector supercomputers, for example on machines from CRAY, NEC, and Fujitsu, one strives for a combination, because of the vector-processing capabilities of the processors. The rule of thumb is that the amount of computation per parallel task should be large relative to the amount of data transport necessary for that task, so that data distribution costs are amortized. One main tool for this is the use of Level 3 BLAS. This use is well established for the solution of dense systems although the best way to incorporate these in sparse solution is still a matter for discussion.

The routing between processors has become less of problem for the numerical analyst (in the sense that less attention has to be paid to this aspect in the design of algorithms). The use of caches and hierarchical memory coupled with improved compiler software has meant that communication costs can be reduced so that operations like the inner product are no longer necessarily a severe bottleneck on modern architectures. Distribution of data is still a problem, especially since now there is a trend towards larger units of computation, and it is a problem to achieve a proper load balance for sparse matrix computations. Recently, there has been a lot of effort in the dynamic identification of parallel tasks that can be executed without too much data redistribution. This is in particular the case for direct techniques and for methods mixing iterative and direct techniques. This mixing of these techniques is a trend that has become more important in recent years. Of course, incomplete decompositions may be viewed as a bridge between direct and iterative methods, but the connection is now getting much deeper. For instance, direct methods are used unless the Schur complements require too much computation, or have too little parallelism, whence iterative schemes are considered for completing the solution.

Another trend is to nest iterative methods, as can be seen in methods like Bi-CGSTAB(ℓ), or even more in methods such as FGMRES and GMRESR that allow GMRES to be combined with virtually any other method. This can be exploited to get more parallelism.

A major concern is that the parallel performance of iterative codes lags behind that for direct-sparse and direct-dense codes, more or less in the ratio 1:5:10. This leads to more interest in situations where block techniques are obviously applicable, for instance when one has multiple right-hand sides. The changes to be made to the algorithms are not always obvious although the goal is usually to identify large enough Level 3 BLAS kernels.

Improvement to the preconditioners seems to be potentially the most rewarding and active research area, far more than trying to improve the performance of the iterative method. If blocking for multiple right-hand sides is not possible, then performance improvements for iterative methods seem to be limited to modest percentages, whereas a good preconditioner can reduce the CPU-time by a significant factor.

An example of a current project which combines many aspects of these trends is the PARASOL Project where new techniques in direct and iterative techniques are being combined with novel combinations and preconditionings within the framework of a single Library. PARASOL is an ESPRIT IV Long Term Research Project (No 20160) for "An Integrated Environment for Parallel Sparse Matrix Solvers". The main goal of this Project, which started on January 1 1996, is to build and test a portable library for solving large sparse systems of equations on distributed memory systems. There are twelve partners in five countries, five of whom are code developers, five end users, and two software houses. The software is written in Fortran 90 and uses MPI for message passing. The solvers being developed in this consortium are: two domain decomposition codes by Bergen and ONERA, a multigrid code by GMD, and a parallel multifrontal method (called MUMPS) by CERFACS and RAL. The final library will be in the public domain. For more information on the PARASOL project, see the web site at `http://www.genias.de/parasol`.

Acknowledgments

We are grateful to Rob Bisseling, Wim Bomhof, Valérie Frayssé, Luc Giraud, Karl Meerbergen, and Denis Trystram for comments on an earlier version of the text.

References

[1] G. Alaghband. Parallel sparse matrix solution and performance. *Parallel Computing*, 21(9):1407–1430, 1995.

[2] G. Alléon, M. Benzi, and L. Giraud. Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics. *Numerical Algorithms*, 16(1):1–15, 1997.

[3] F. L. Alvarado and H. Dağ. Incomplete partitioned inverse preconditioners. Technical report, Department of Electrical and Computer Engineering, University of Wisconsin, Madison, 1994.

[4] F. L. Alvarado, A. Pothen, and R. Schreiber. Highly parallel sparse triangular solution. In Alan George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*. Springer-Verlag, 1993.

[5] F. L. Alvarado and R. Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM J. Scientific Computing*, 14:446–460, 1993.

[6] F. L. Alvarado, D. C. Yu, and R. Betancourt. Partitioned sparse A^{-1} methods. *IEEE Trans. Power Systems*, 3:452–459, 1990.

[7] P. R. Amestoy. Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment. INPT PhD Thesis TH/PA/91/2, CERFACS, Toulouse, France, 1991.

[8] P. R. Amestoy and I. S. Duff. Vectorization of a multiprocessor multifrontal code. *Int. J. of Supercomputer Applics.*, 3:41–59, 1989.

[9] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal solvers within the PARASOL environment. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, No. 1541, pages 7–11, Berlin, 1998. Springer-Verlag.

[10] P. R. Amestoy, I. S. Duff, and C. Puglisi. Multifrontal QR factorization in a multiprocessor environment. *Numerical Linear Algebra with Applications*, 3(4):275–300, 1996.

[11] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, second edition*. SIAM Press, 1995.

[12] E. C. Anderson and Y. Saad. Solving sparse triangular systems on parallel computers. *Int J. High Speed Computing*, 1:73–95, 1989.

[13] M. Arioli, I. S. Duff, J. Noailles, and D. Ruiz. A block projection method for sparse matrices. *SIAM J. Scientific and Statistical Computing*, 13:47–70, 1992.

[14] S. F. Ashby. Minimax polynomial preconditioning for Hermitian linear systems. *SIAM J. Matrix Analysis and Applications*, 12:766–789, 1991.

[15] C. Ashcraft, S. C. Eisenstat, and J. W. H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM J. Scientific and Statistical Computing*, 11:593–599, 1990.

[16] C. Ashcraft, S. C. Eisenstat, J. W. H. Liu, and A. H. Sherman. A comparison on three column-based distributed sparse factorization schemes. Technical Report CS-90-09, Department of Computer Science, York University, York, Ontario, Canada, 1990.

[17] C. Ashcraft and J. W. H. Liu. Robust ordering of sparse matrices using multisection. Technical Report ISSTECH-96-002, Boeing Information and Support Services, Seattle, 1996. Also Report CS-96-01, Department of Computer Science, York University, Ontario, Canada.

[18] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, Cambridge, 1994.

[19] I. Babuska. Numerical stability in problems of linear algebra. *SIAM J. Numerical Analysis*, 9:53–77, 1972.

[20] Z. Bai, D. Hu, and L. Reichel. A Newton basis GMRES implementation. *IMA J. Numer. Anal.*, 14:563–581, 1991.

[21] S. T. Barnard, L. M. Bernardo, and H. D. Simon. An MPI implementation of the SPAI preconditioner on the T3E. Technical Report LBNL-40794 UC405, Lawrence Berkeley National Laboratory, 1997.

[22] S. T. Barnard and R. L. Clay. A portable MPI implementation of the SPAI preconditioner in ISIS++. In Michael Heath, Virginia Torczon, Greg Astfalk, Petter E. Björstad, Alan H. Karp, Charles H. Koebel, V. Kumar, R. F. Lucas, Layne T. Watson, and David E. Womble, editors, *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, pages xxx–yyy. SIAM Press, 1997.

[23] S. T. Barnard, A. Pothen, and H. Simon. A spectral algorithm for envelope reduction of sparse matrices. *Numerical Linear Algebra with Applications*, 2(4):317–334, 1995.

[24] P. Bastian and G. Horton. Parallelization of robust multigrid methods: ILU factorization and frequency decomposition method. *SIAM J. Scientific and Statistical Computing*, 6:1457–1470, 1991.

[25] M. Benzi, C. D. Meyer, and M. Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Scientific Computing*, 17:1135–1149, 1996.

[26] M. Benzi, C. D. Meyer, and M. Tůma. A two-level parallel preconditioner based on sparse approximate inverses. In D. R. Kincaid et al, editor, *Iterative Methods in Scientific Computing*, pages 1–11. IMACS, 1999.

[27] M. Benzi and M. Tůma. Numerical experiments with two sparse approximate inverse preconditioners. *BIT*, 38:234–241, 1998.

[28] M. Benzi and M. Tůma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM J. Scientific Computing*, 19(3):968–994, 1998.

[29] A. Berger, J. Mulvey, E. Rothberg, and R. Vanderbei. Solving multistage stochastic programs using tree dissection. Technical Report SOR-97-07, Programs in Statistics and Operations Research, Princeton University, Princeton, New Jersey, 1995.

[30] H. Berryman, J. Saltz, W. Gropp, and R. Mirchandaney. Krylov methods preconditioned with incompletely factored matrices on the CM-2. *J. Par. Dist. Comp.*, 8:186–190, 1990.

[31] R. H. Bisseling, T. M. Doup, and L. D. J. C. Loyens. A parallel Interior Point algorithm for linear programming on a network of transputers. *Annals of Operations Research*, 43:51–86, 1993.

[32] R. H. Bisseling and W. F. McColl. Scientific computing on Bulk Synchronous Parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing '94, Vol. I*, IFIP Transactions A, Volume 51, pages 509–514, Amsterdam, 1994. Elsevier Science Publishers.

[33] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM Press, 1997.

[34] S. Bondeli. *Divide and Conquer: parallele Algorithmen zur Lösing tridiagonaler Gleichungssysteme*. PhD thesis, ETH Zürich, Zürich, 1991.

[35] R. Bramley and A. Sameh. Row projection methods for large nonsymmetric linear systems. *SIAM J. Scientific and Statistical Computing*, 13:168–193, 1992.

[36] H. M. Bücker and M. Sauren. A parallel version of the unsymmetric Lanczos algorithm and its application to QMR. Technical Report KFA-ZAM-IB-9605, Forschungszentrum Jülich Gmbh, Jülich, Germany, 1996.

[37] B. L. Buzbee, G. H. Golub, and C W. Nielson. On direct methods for solving Poisson's equations. *SIAM J. Numerical Analysis*, 7:627–656, 1970.

[38] D. A. Calahan. Parallel solution of sparse simultaneous linear equations. In *Proceedings 11th Annual Allerton Conference on Circuits and System Theory, University of Illinois*, pages 729–735, 1973.

[39] T. F. Chan, E. Gallopoulos, V. Simoncini, T. Szeto, and C. H. Tong. A quasi-minimal residual variant of the Bi-CGSTAB algorithm for nonsymmetric systems. *SIAM J. Scientific Computing*, 15:338–347, 1994.

[40] T. F. Chan and D. Goovaerts. A note on the efficiency of domain decomposed incomplete factorizations. *SIAM J. Scientific and Statistical Computing*, 11:794–803, 1990.

[41] T. F. Chan and H. A. van der Vorst. Approximate and incomplete factorizations. In D. E. Keyes, A. Sameh, and V. Venkatakrishnan, editors, *Parallel Numerical Algorithms*, ICASE/LaRC Interdisciplinary Series in Science and Engineering, pages 167–202. Kluwer, Dordrecht, 1997.

[42] S. C. Chen, D. J. Kuck, and A. H. Sameh. Practical parallel band triangular system solvers. *ACM Trans. Math. Softw.*, 4:270–277, 1978.

[43] H. Choi and D. B. Szyld. Threshold ordering for preconditioning nonsymmetric problems with highly varying coefficients. Technical Report 96-51, Department of Mathematics, Temple University, Philadelphia, 1996.

[44] E. Chow and Y. Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM J. Scientific Computing*, 19:995–1023, 1998.

[45] A. T. Chronopoulos. Towards efficient parallel implementation of the CG method applied to a class of block tridiagonal linear systems. In *Supercomputing '91*, pages 578–587, Los Alamitos, CA, 1991. IEEE Computer Society Press.

[46] A. T. Chronopoulos and C. W. Gear. s-Step iterative methods for symmetric linear systems. *J. Comp. and Appl. Math.*, 25:153–168, 1989.

[47] A. T. Chronopoulos and S. K. Kim. s-Step Orthomin and GMRES implemented on parallel computers. Technical Report 90/43R, UMSI, Minneapolis, 1990.

[48] A. T. Chronopoulos and S. K. Kim. Towards efficient parallel implementation of Krylov subspace iterative methods. *Supercomputer*, 47:4–17, 1992.

[49] A. T. Chronopoulos and C. D. Swanson. Parallel iterative S-step methods for unsymmetric linear systems. *Parallel Computing*, 22:623–641, 1996.

[50] P. Concus, G. H. Golub, and G. Meurant. Block preconditioning for the conjugate gradient method. *SIAM J. Scientific and Statistical Computing*, 6:220–252, 1985.

[51] P. Concus and G. Meurant. On computing INV block preconditionings for the conjugate gradient method. *BIT*, pages 493–504, 1986.

[52] J. M. Conroy, S. G. Kratzer, and R. F. Lucas. Data-parallel sparse matrix factorization. In J. G. Lewis, editor, *Proceedings 5th SIAM Conference on Linear Algebra*, pages 377–381, Philadelphia, 1994. SIAM Press.

[53] J. D. F. Cosgrove, J. C. Diaz, and A. Griewank. Approximate inverse preconditionings for sparse linear systems. *Int. J. Computer Math.*, 44:91–110, 1992.

[54] L. Crone and H. van der Vorst. Communication aspects of the conjugate gradient method on distributed-memory machines. *Supercomputer*, X(6):4–9, 1993.

[55] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Analysis and Applications*, 18(1):140–158, 1997.

[56] T. A. Davis and P. C. Yew. A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization. *SIAM J. Matrix Analysis and Applications*, 11:383–402, 1990.

[57] E. F. D'Azevedo, V. Eijkhout, and C. Romine. LAPACK working note 56: Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessor. Technical report, Computer Science Department, University of Knoxville, Knoxville, TN, 1993.

[58] E. F. D'Azevedo and C. Romine. Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessors. Technical Report ORNL/TM-12192, Oak Ridge National Lab, Oak Ridge, TN, 1992.

[59] P. P. N. de Groen. Base p-cyclic reduction for tridiagonal systems of equations. *Applied Numerical Mathematics*, 8:117–126, 1991.

[60] J. De Keyser and D. Roose. Distributed mapping of SPMD programs with a generalized Kernighan-Lin heuristic. In W. Gentzsch and U. Harms, editors, *High-Performance Computing and Networking*, Lecture Notes in Computer Science 797, pages 227–232, Berlin, 1994. Springer-Verlag.

[61] E. de Sturler. A parallel restructured version of GMRES(m). Technical Report 91-85, Delft University of Technology, Delft, 1991.

[62] E. de Sturler. *Iterative methods on distributed memory computers*. PhD thesis, Delft University of Technology, Delft, the Netherlands, 1994.

[63] E. de Sturler. A performance model for Krylov subspace methods on mesh-based parallel computers. *Parallel Computing*, 22:57–74, 1996.

[64] E. de Sturler and H. A. van der Vorst. Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers. *Applied Numerical Mathematics*, 18:441–459, 1995.

[65] J. W. Demmel, M. T. Heath, and H. A. van der Vorst. Parallel numerical linear algebra. In *Acta Numerica 1993*. Cambridge University Press, Cambridge, 1993.

[66] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. Technical Report UCB//CSD-95-883, Computer Science Division, U. C. Berkeley, Berkeley, California, July 1995.

[67] J. W. Demmel, J. R. Gilbert, and X. S. Li. SuperLU users' guide. Technical report, Computer Science Division, U. C. Berkeley, Berkeley, California, February 1995. (available from netlib).

[68] S. Doi. On parallelism and convergence of incomplete LU factorizations. *Applied Numerical Mathematics*, 7:417–436, 1991.

[69] S. Doi and A. Hoshi. Large numbered multicolor MILU preconditioning on SX-3/14. *Int. J. Computer Math.*, 44:143–152, 1992.

[70] J. J. Dongarra. Performance of various computers using standard linear algebra software. Technical Report CS-89-85, University of Tennessee, Knoxville, Tennessee, 1999. Updated version at Web address `http://www.netlib.org/benchmark/performance.ps`.

[71] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM Press, Philadelphia, 1979.

[72] J. J. Dongarra, J. J. Du Croz, S. Hammarling, and R. J. Hanson. An extented set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 14:1–17, 1988.

[73] J. J. Dongarra, J. J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16:1–17, 1990.

[74] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers.* SIAM Press, Philadelphia, 1998.

[75] P. Dubois and G. Rodrigue. An analysis of the recursive doubling algorithm. In D. J. Kuck and A. H. Sameh, editors, *High speed computer and algorithm organization.* Academic Press, New York, 1977.

[76] P. F. Dubois, A. Greenbaum, and G. H. Rodrigue. Approximating the inverse of a matrix for use in iterative algorithms on vector processors. *Computing*, 22:257–268, 1979.

[77] I. S. Duff. The use of vector and parallel computers in the solution of large sparse linear equations. In P. Deuflhard and B. Engquist, editors, *Large scale scientific computing. Progress in Scientific Computing Volume 7*, pages 331–348, Boston, 1986. Birkhäuser.

[78] I. S. Duff. The influence of vector and parallel computers in the solution of large sparse linear equations. In M J D Powell and A Iserles, editors, *The State of the Art in Numerical Analysis*, pages 359–407, Oxford, 1987. Oxford University Press.

[79] I. S. Duff, A. M. Erisman, C. W. Gear, and J. K. Reid. Sparsity structure and Gaussian elimination. *SIGNUM Newsletter*, 23(2):2–8, April 1988.

[80] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, England, 1986.

[81] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. Technical Report RAL-TR-97-059, Rutherford Appleton Laboratory, 1997. To appear in *SIAM J. Matrix Analysis and Applications.*

[82] I. S. Duff, M. Marrone, G. Radicati, and C. Vittoli. Level 3 Basic Linear Algebra Subprograms for sparse matrices: a user level interface. *ACM Trans. Math. Softw.*, 23(3):379–401, 1997.

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

[84] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 9:302–325, 1983.

[85] B. Dumitrescu, M. Doreille, J.-L. Roch, and D. Trystram. Two-dimensional block partitioning for the parallel sparse Cholesky factorization. *Numerical Algorithms*, 16:17–38, 1997.

[86] V. Eijkhout. Beware of unperturbed modified incomplete point factorizations. In R. Beauwens and P. de Groen, editors, *Iterative Methods in Linear Algebra*, pages 583–591, Amsterdam, 1992. IMACS Int. Symp., Brussels, Belgium, 2-4 April, 1991, North-Holland.

[87] V. Faber and T. A. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J. Numerical Analysis*, 21(2):352–362, 1984.

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

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

[90] R. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Scientific Computing*, 14:470–482, 1993.

[91] K. A. Gallivan, B. A. Marsolf, and H. A. G. Wijshoff. Solving large nonsymmetric sparse linear systems using MCSPARSE. *Parallel Computing*, 22:1291–1333, 1996.

[92] A. Geist and E. Ng. Task scheduling for parallel sparse Cholesky factorization. *Int J. Parallel Programming*, 18:291–314, 1989.

[93] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numerical Analysis*, 10:345–363, 1973.

[94] A. George, M. T. Heath, J. W. H. Liu, and E. Ng. Solution of sparse positive-definite systems on a shared memory multiprocessor. *Int J. Parallel Programming*, 15:309–325, 1986.

[95] A. George, M. T. Heath, J. W. H. Liu, and E. Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM J. Scientific and Statistical Computing*, 9:327–340, 1988.

[96] A. George, M. T. Heath, J. W. H. Liu, and E. Ng. Solution of sparse positive definite systems on a hypercube. *J. Comput. Appl. Math.*, 27:129–156, 1989.

[97] A. George and J. W. H. Liu. The design of a user interface for a sparse matrix package. *ACM Trans. Math. Softw.*, 5(2):139–162, 1979.

[98] A. George, J. W. H. Liu, and E. G. Ng. User's guide for SPARSPAK: Waterloo sparse linear equations package. Technical Report CS-78-30 (Revised), University of Waterloo, Canada, 1980.

[99] A. George and E. Ng. Shared versus local memory in parallel sparse matrix computations. *SIGNUM Newsletter*, 23(2):9–13, 1988.

[100] A. George, J. W. Poole, and R. Voigt. Incomplete nested dissection for solving n by n grid problems. *SIAM J. Numerical Analysis*, 15:663–673, 1978.

[101] J. P. Geschiere and H. A. G. Wijshoff. Exploiting large grain parallelism in a sparse direct linear system solver. *Parallel Computing*, 21(8):1339–1364, 1995.

[102] J. R. Gilbert and J. W. H. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM J. Matrix Analysis and Applications*, 14:334–354, 1993.

[103] J. R. Gilbert and R. Schreiber. Highly parallel sparse Cholesky factorization. *SIAM J. Scientific and Statistical Computing*, 13:1151–1172, 1992.

[104] J. R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *Int J. Parallel Programming*, 16:427–449, 1987.

[105] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 1996.

[106] N. I. M. Gould and J. A. Scott. Sparse approximate-inverse preconditioners using norm-minimization techniques. *SIAM J. Scientific Computing*, 19(2):605–625, 1998.

[107] A. Greenbaum. *Iterative Methods for Solving Linear Systems*. SIAM, Philadelphia, 1997.

[108] M. Grote and H. Simon. Parallel preconditioning and approximate inverses on the connection machine. In R. F. Sincovec, D. E. Keyes, M. R. Leuze, L. R. Petzold, and D. A. Reed, editors, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 519–523, Philadelphia, 1993. SIAM.

[109] M. J. Grote and T. Huckle. Parallel preconditionings with sparse approximate inverses. *SIAM J. Scientific Computing*, 18:838–853, 1997.

[110] A. Gupta, M. Joshi, and V. Kumar. WSSMP: Watson Symmetric Sparse Matrix Package. Users Manual: Version 2.0β. Technical Report RC 20923 (92669), IBM T. J. Watson Research Centre, P. O. Box 218, Yorktown Heights, NY 10598, July 1997.

[111] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. Technical Report TR-94-63, Department of Computer Science, University of Minnesota, 1994.

[112] I. Gustafsson and G. Lindskog. A preconditioning technique based on element matrix factorizations. *Comput. Methods Appl. Mech. Eng.*, 55:201–220, 1986.

[113] I. Gustafsson and G. Lindskog. Completely parallelizable preconditioning methods. *Num. Lin. Alg. Appl.*, 2:447–465, 1995.

[114] M. H. Gutknecht. Variants of BICGSTAB for matrices with complex spectrum. *SIAM J. Scientific Computing*, 14:1020–1033, 1993.

[115] M. H. Gutknecht. Lanczos-type solvers for nonsymmetric linear systems of equations. In *Acta Numerica 1997*, pages 271–397. Cambridge University Press, Cambridge, 1997.

[116] G. Haase. Parallel incomplete Cholesky preconditioners based on the nonoverlapping data distribution. *Parallel Computing*, 24:1685–1703, 1998.

[117] M. T. Heath, E. G. Y. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.

[118] M. T. Heath and P. Raghavan. Performance of a fully parallel sparse solver. In IEEE, editor, *Proceedings of SHPCC '94, Scalable High-Performance Computing Conference. May 23-25, 1994, Knoxville, Tennessee*, pages 334–341, Los Alamitos, California, 1994. IEEE Computer Society Press.

[119] B. Heisse and M. Jung. Parallel solvers for nonlinear elliptic problems based on domain decomposition ideas. *Parallel Computing*, 22:1527–1544, 1997.

[120] D. Heller. Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems. *SIAM J. Numerical Analysis*, 13:484–496, 1978.

[121] B. Hendrickson and R. Leland. The CHACO User's Guide. Version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, Albuquerque, October 1994.

[122] B. Hendrickson and E. Rothberg. Improving the runtime and quality of nested dissection ordering. Technical Report SAND96-0868J, Sandia National Laboratories, Albuquerque, 1996. To appear in *SIAM J. Scientific Computing.*

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

[124] T. J. R. Hughes, I. Levit, and J. Winget. An element-by-element solution algorithm for problems of structural and solid mechanics. *J. Comp. Methods in Appl. Mech. Eng.*, 36:241–254, 1983.

[125] Z. Johan. *Data parallel finite element techniques for large-scale computational fluid dynamics.* PhD thesis, Stanford University, Stanford, CA, 1992.

[126] Z. Johan, K. K. Mathur, S. L. Johnsson, and T. J. R. Hughes. Mesh decomposition and communication procedures for finite element applications on the connection machine CM-5 system. In W. Gentzsch and U. Harms, editors, *High-Performance Computing and Networking*, Lecture Notes in Computer Science 797, pages 233–240, Berlin, 1994. Springer-Verlag.

[127] O. G. Johnson, C. A. Micheli, and G. Paul. Polynomial preconditioning for conjugate gradient calculations. *SIAM J. Numerical Analysis*, 20:363–376, 1983.

[128] M. T. Jones and P. E. Plassmann. The efficient parallel iterative solution of large sparse linear systems. In A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computations*, IMA Vol 56. Springer Verlag, Berlin, 1994.

[129] G. Karypis and V. Kumar. METIS: unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, 1995.

[130] G. Karypis and V. Kumar. Parallel multilevel graph partitioning. Technical Report TR-95-036, Department of Computer Science, University of Minnesota, May 1995.

[131] D. R. Kincaid and T. C. Oppe. Recent vectorization and parallelization of ITPACKV. In O. Axelsson and L. Yu. Kolotilina, editors, *Preconditioned Conjugate Gradient Methods*, pages 58–74, Berlin, 1990. Nijmegen 1989, Springer Verlag. Lecture Notes in Mathematics 1457.

[132] L. Yu. Kolotilina and A. Yu. Yeremin. Factorized sparse approximate inverse preconditionings. *SIAM J. Matrix Analysis and Applications*, 14:45–58, 1993.

[133] J. Koster and R. H. Bisseling. Parallel sparse LU decomposition on a distributed-memory multiprocessor, 1994. Submitted to *SIAM J. Scientific Computing*.

[134] J. C. C. Kuo and T. F. Chan. Two-color Fourier analysis of iterative algorithms for elliptic problems with red/black ordering. *SIAM J. Scientific and Statistical Computing*, 11:767–793, 1990.

[135] J. J. Lambiotte and R. G. Voigt. The solution of tridiagonal linear systems on the CDC-STAR-100 computer. Technical report, ICASE-NASA Langley Research Center, Hampton, VA, 1974.

[136] C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Natl. Bur. Stand*, 49:33–53, 1952.

[137] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

[138] G. Li. A block variant of the GMRES method on massively parallel processors. *Parallel Computing*, 23:1005–1019, 1997.

[139] X. S. Li and J. W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of Supercomputing*, Orlando, Florida, November 1998.

[140] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Softw.*, 12:249–264, 1987.

[141] J. W. H. Liu. Reordering sparse matrices for parallel elimination. *Parallel Computing*, 11:73–91, 1989.

[142] N. K. Madsen, G. H. Rodrigue, and J. I. Karush. Matrix multiplication by diagonals on a vector/parallel processor. *Inform. Process. Lett.*, 5:41–45, 1976.

[143] Fredrik Manne and Hjálmtýr Hafsteinsson. Efficient sparse Cholesky factorization on a massively parallel SIMD computer. *SIAM J. Scientific Computing*, 16(4):934–950, July 1995.

[144] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3:255–269, Apr. 1957.

[145] O. A. McBryan and E. F. van de Velde. Hypercube algorithms and implementations. *SIAM J. Scientific and Statistical Computing*, 8:227–287, 1987.

[146] V. Mehrmann. Divide and conquer methods for block tridiagonal systems. Technical Report Bericht Nr. 68, Inst. für Geometrie und Prakt. Math., RWTH, Aachen, 1991.

[147] U. Meier. A parallel partition method for solving banded systems of linear equations. *Parallel Computing*, 2:33–43, 1985.

[148] U. Meier and A. Sameh. The behavior of conjugate gradient algorithms on a multivector processor with a hierarchical memory. Technical Report CSRD 758, University of Illinois, Urbana, IL, 1988.

[149] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.*, 31:148–162, 1977.

[150] G. Meurant. The block preconditioned conjugate gradient method on vector computers. *BIT*, 24:623–633, 1984.

[151] G. Meurant. Numerical experiments for the preconditioned conjugate gradient method on the CRAY X-MP/2. Technical Report LBL-18023, University of California, Berkeley, CA, 1984.

[152] G. Meurant. The conjugate gradient method on vector and parallel supercomputers. Technical Report CTAC-89, University of Brisbane, July 1989.

[153] P. H. Michielse. *Parallelism in Adaptive Multigrid Solvers*. PhD thesis, Delft University of Technology, Delft, 1990.

[154] P. H. Michielse and H. A. van der Vorst. Data transport in Wang's partition method. *Parallel Computing*, 7:87–95, 1988.

[155] E. G. Ng and B. W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Scientific Computing*, 14:1034–1056, 1993.

[156] E. G. Ng and B. W. Peyton. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *SIAM J. Scientific Computing*, 14:761–769, 1993.

[157] D. P. O'Leary. The block conjugate gradient algorithm and related methods. *Linear Alg. Appl.*, 29:293–322, 1980.

[158] D. P. O'Leary. Parallel implementation of the Block Conjugate Gradient algorithm. *Parallel Computing*, 5:127–140, 1987.

[159] J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Plenum Press, New York and London, 1988.

[160] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numerical Analysis*, 12:617–629, 1975.

[161] B. W. Peyton, A. Pothen, and X. Yuan. Partitioning a chordal graph into transitive subgraphs for parallel sparse triangular solution. Technical Report ORNL/TM-12270, Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, Tennessee, December 1992.

[162] C. Pommerell. *Solution of large unsymmetric systems of linear equations*. PhD thesis, Swiss Federal Institute of Technology, Zürich, 1992.

[163] C. Pommerell, M. Annaratone, and W. Fichtner. A set of new mapping and coloring heuristics for distributed-memory parallel computers. *SIAM J. Scientific and Statistical Computing*, 13:194–226, 1992.

[164] A. Pothen and C. Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM J. Scientific Computing*, 14(5):1253–1257, 1993. Timely Communication.

[165] G. Radicati di Brozolo and Y. Robert. Parallel conjugate gradient-like algorithms for solving sparse non-symmetric systems on a vector multiprocessor. *Parallel Computing*, 11:223–239, 1989.

[166] G. Radicati di Brozolo and M. Vitaletti. Sparse matrix-vector product and storage representations on the IBM 3090 with Vector Facility. Technical Report 513-4098, IBM-ECSEC, Rome, July 1986.

[167] P. Raghavan. Efficient parallel sparse triangular solution with selective inversion. Technical Report CS-95-314, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, 1995.

[168] P. Raghavan. Parallel ordering using edge contraction. Technical Report CS-95-293, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, 1995. Submitted to *Parallel Computing*.

[169] E. Rothberg. Exploring the tradeoff between imbalance and separator size in nested dissection ordering. Technical Report Unnumbered, Silicon Graphics Inc, 1996.

[170] E. Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. *SIAM J. Scientific Computing*, 17(3):699–713, 1996.

[171] E. Rothberg and A. Gupta. An evaluation of left-looking, right-looking and multifrontal approaches to sparse Cholesky factorization on hierarchical-memory machines. Technical Report STAN-CS-91-1377, Department of Computer Science, Stanford University, 1991.

[172] E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. *SIAM J. Scientific Computing*, 15(6):1413–1439, December 1994.

[173] E. Rothberg and R. Schreiber. Improved load distribution in parallel sparse Cholesky factorization. Technical Report 94-13, Research Institute for Advanced Computer Science, 1994.

[174] Y. Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM J. Scientific and Statistical Computing*, 6:865–881, 1985.

[175] Y. Saad. Krylov subspace methods on supercomputers. Technical report, RIACS, Moffett Field, CA, September 1988.

[176] Y. Saad. *Iterative methods for sparse linear systems.* PWS Publishing Company, Boston, 1996.

[177] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Scientific and Statistical Computing*, 7:856–869, 1986.

[178] J. J. F. M. Schlichting and H. A. van der Vorst. Solving bidiagonal systems of linear equations on the CDC CYBER 205. Technical Report NM-R8725, CWI, Amsterdam, the Netherlands, 1987.

[179] J. J. F. M. Schlichting and H. A. van der Vorst. Solving 3D block bidiagonal linear systems on vector computers. *J. Comp. and Appl. Math.*, 27:323–330, 1989.

[180] R. Schreiber. Scalability of sparse direct solvers. In A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, The IMA Volumes in Mathematics and its Applications, Volume 56, pages 191–209, New York, 1993. Springer-Verlag.

[181] H. D. Simon. Partitioning of unstructured problems for parallel processing. Technical Report RNR-91-008, NASA Ames Research Center, Moffett Field, CA, 1991.

[182] V. Simoncini and E. Gallopoulos. An iterative method for nonsymmetric systems with multiple right-hand sides. *SIAM J. Scientific Computing*, 16:917–933, 1995.

[183] G. L. G. Sleijpen and D. R. Fokkema. BICGSTAB(ℓ) for linear equations involving unsymmetric matrices with complex spectrum. *ETNA*, 1:11–32, 1993.

[184] G. L. G. Sleijpen and H. A. van der Vorst. Reliable updated residuals in hybrid Bi-CG methods. *Computing*, 56:141–163, 1996.

[185] G. L. G. Sleijpen, H. A. van der Vorst, and D. R. Fokkema. Bi-CGSTAB(ℓ) and other hybrid Bi-CG methods. *Numerical Algorithms*, 7:75–109, 1994.

[186] P. Sonneveld. CGS: a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Scientific and Statistical Computing*, 10:36–52, 1989.

[187] H. S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. Assoc. Comput. Mach.*, 20:27–38, 1973.

[188] K. Sumiyoshin and T. Ebisuzaki. Performance of parallel solution of a block-tridiagonal linear system on a Fujitsu VPP500. *Parallel Computing*, 24:287–304, 1998.

[189] C. Sun. Efficient parallel solutions of large sparse SPD systems on distributed-memory multiprocessors. Technical Report CTC92TR102, Advanced Computing Research Institute, Cornell University, Ithaca, NY, 1992.

[190] C. Sun. A package for solving sparse symmetric positive definite systems on distributed-memory multiprocessors. Technical Report CTC92TR114, Advanced Computing Research Institute, Cornell University, Ithaca, NY, November 1992.

[191] R. Sweet. A parallel and vector variant of the cyclic reduction algorithm. *Supercomputer*, 22:18–25, 1987.

[192] X.-H. Sun, H.-Z. Sun, and L. Ni. Parallel algorithms for solution of tridiagonal systems on multicomputers. Technical report, Dept. of Computer Science, Michigan State University, 1989.

[193] K. H. Tan. *Local coupling in domain decomposition*. PhD thesis, Utrecht University, Utrecht, the Netherlands, 1995.

[194] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. of the IEEE*, 55:1801–1809, 1967.

[195] A. F. van der Stappen, R. H. Bisseling, and J. G. G. van de Vorst. Parallel sparse LU decomposition on a mesh network of transputers. *SIAM J. Matrix Analysis and Applications*, 14:853–879, 1993.

[196] H. A. van der Vorst. A vectorizable variant of some ICCG methods. *SIAM J. Scientific and Statistical Computing*, 3:86–92, 1982.

[197] H. A. van der Vorst. The performance of Fortran implementations for preconditioned conjugate gradients on vector computers. *Parallel Computing*, 3:49–58, 1986.

[198] H. A. van der Vorst. Large tridiagonal and block tridiagonal linear systems on vector and parallel computers. *Parallel Computing*, 5:45–54, 1987.

[199] H. A. van der Vorst. High performance preconditioning. *SIAM J. Scientific and Statistical Computing*, 10:1174–1185, 1989.

[200] 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. Scientific and Statistical Computing*, 13:631–644, 1992.

[201] H. A. van der Vorst and J. M. van Kats. The performance of some linear algebra algorithms in FORTRAN on CRAY-1 and Cyber-205 supercomputers. Technical report, Academisch Computer Centrum, Utrecht, 1984.

[202] A. C. N. van Duin. *Parallel Sparse Matrix Computations*. PhD thesis, Leiden University, Leiden, The Netherlands, 1998.

[203] M. B. van Gijzen. *Iterative solution methods for linear equations in finite element computations*. PhD thesis, Delft University of Technology, Delft, the Netherlands, 1994.

[204] C. Vuik, R. R. P. van Nooyen, and P. Wesseling. Parallelism in ILU-preconditioned GMRES. *Parallel Computing*, 24:1927–1946, 1998.

[205] H. H. Wang. A parallel method for tridiagonal equations. *ACM Trans. Math. Softw.*, 7:170–183, 1989.

[206] T. Washio and K. Hayami. Parallel block preconditioning based on SSOR and MILU. *Num. Lin. Alg. Appl.*, 1:533–553, 1994.

[207] R. C. Whaley. Lapack working note 73 : Basic Linear Algebra Communication Subprograms: analysis and implementation across multiple parallel architectures. Technical Report CS-94-234, Computer Science Department, University of Tennessee, Knoxville, Tennessee, May 1994.

[208] J. H. Wilkinson and C. Reinsch. *Handbook for Automatic Computation. Volume II Linear Algebra*. Springer-Verlag, Berlin, 1971.

[209] J. Zhang. A sparse approximate inverse technique for parallel preconditioning of general sparse matrices. Tech. Rep. 281-98, Department of Computer Science, University of Kentucky, KY, 1998.

[210] Z. Zlatev, J. Waśniewski, P. C. Hansen, and Tz. Ostromsky. PARASPAR: a package for the solution of large linear algebraic equations on parallel computers with shared memory. Technical Report 95-10, Tech Univ Denmark, Lyngby, 1995.

[211] Z. Zlatev, J. Waśniewski, and K. Schaumburg. Introduction to PARASPAR. solution of large and sparse systems of linear algebraic equations, specialised for parallel computers with shared memory. Technical Report 93-02, Tech Univ Denmark, Lyngby, 1993.

[212] E. Zmijewski. *Sparse Cholesky Factorization on a Multiprocessor*. PhD thesis, Cornell University, 1987.

[213] E. Zmijewski and J. R. Gilbert. A parallel algorithm for sparse symbolic Cholesky factorization on a multiprocessor. *Parallel Computing*, 7:199–210, 1988.