SPARSE LINEAR ALGEBRA
in and around the
APO-ENSEEIHT-IRIT group

"Algorithmique Parallèle et Optimization"
ENSEEIHT-IRIT, 2 rue Camichel, 31071 Toulouse CEDEX, France

Collaborative work with
CERFACS, 42 av. G. Coriolis, 31057 Toulouse Cedex, France
Central Computing Department, RAL, OXON OX11 0QX, England
Istituto Analisi Numerica, CNR, Pavia, Italy
University of Florida, Sciences Dept., Gainesville, USA

Abstract

We describe the work done in sparse linear algebra, in the "Algorithmique Parallèle et Optimization" group of the ENSEEIHT-IRIT laboratory. The research activities, described in this paper, result from collaborations with CERFACS, RAL and University of FLorida. These include work on computational kernels for linear algebra, the solution of sparse systems by both direct and iterative methods, the study of element-by-element preconditionners. The objective of this paper is to describe the principal research themes explored in these area. We also comment on likely future developments.

1 Introduction

We consider the solution of

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

where A is a large sparse matrix. If the matrix A is structured then it may be written as

$$A = \sum_{i=1}^{p} A_i. \tag{2}$$

Sparse structured linear systems arise in many applications. The elementary matrices A_i are usually full or nearly full matrices. Both classes of matrices (structured and unstructured) are being considered in our research studies.

For seek of clearness, and although most these topics of research are very often tighly coupled, we have divided the article into sections, each descibing a single or closely grouped research topic. The principal researchers involved in each topic are shown in parenthesis in each heading.

Kernels such as the BLAS (Dongarra, Du Croz, Duff, and Hammarling (1990)) are common building blocks used for much of our work. In Section 2, we describe the work done on the design of efficient BLAS kernels for high performance processors and multiprocessors. In Section 3, we descibe the use of a multifrontal approach for the solution of the sparse sets of equations. We consider both the **LU** and the **QR** parallel sparse factorizations. Block Cimmino and block conjugate gradient methods will be used to discuss the work done on parallel iterative methods. Then, in Section 5, we describe the iterative solution of the linear system using element-by-element preconditioners that allow one to take advantage of the structure of the linear system and that are easily parallelizable.

2 Computational kernels (Daydé and Duff)

The building blocks for much of our work, both in the solution of sparse as well as full systems and in more complicated areas of scientific computation, are the Basis Linear Algebra Subprograms known as the BLAS. For reasons of efficiency, we are interested in the higher level BLAS, in particular the Level 3 BLAS (Dongarra, Du Croz, Duff, and Hammarling (1990)) which include kernels like the matrix-matrix multiply routine _GEMM. Indeed, in Daydé, Duff, and Petitet (1994) and Kågström, Ling, and Loan (1993), it is shown how all the Level 3 BLAS routines can be designed for high performance using the _GEMM kernel so it is reasonable to only consider the performance of that routine here.

In particular, we have developped a set of the Level 3 BLAS single and double precision computational kernels suitable as a platform for efficient implementation of BLAS on RISC processors (Daydé and Duff (1995)). In Table 1, we show the performance of the double precision matrix-matrix product DGEMM on a set of RISC workstations, obtained using this block version of the BLAS. We also report the results from the standard Fortran implementation of DGEMM and the performance of the manufacturer-supplied version when available.

Processor	Computational kernel	Order of matrices				Peak
		32	64	96	128	Perf.
DEC3000/300-AXP	DGEMM standard	31.5	23.9	25.1	23. 1	150
	DGEMM blocked	46.2	48.7	47.4	48. 4	
IBM 750	DGEMM standard	16.9	29.6	23.5	25. 2	125
	DGEMM blocked	33.8	66.6	81.3	84. 5	
	DGEMM library	33.8	88.7	89.4	96. 1	
HP 715/64	DGEMM standard	11.3	15.7	16.5	16. 9	128
	DGEMM blocked	16.9	29.6	35.7	36. 4	
	DGEMM library	33.8	44.4	47.0	38. 4	
SUN SPARC 10/41	DGEMM standard	11.3	8.9	9.0	9. 0	40
	DGEMM blocked	16.9	19.0	19.9	18. 7	

Table 1: Performance in Mflops of the block implementation of DGEMM on RISC workstations (case 'No transpose', 'No transpose').

This blocked implementation of the BLAS, in this case DGEMM, gives a gain in performance of greater than a factor of two compared with the standard Fortran coded version. Furthermore, we have observed that the performance is even better if the matrices are already held in the cache (which was not the case in these experiments). The performance for parallel versions of _GEMM on the BBN TC2000 and the KSR1 are given in Table 2 on matrices of order 512. We show results for both single (32-bit) and double (64-bit) precision on the BBN and for normal (64-bit) precision on the KSR1. We see that the matrices are too small to get maximum performance. Indeed, on matrices of order 1536, SGEMM can reach 150 Mflop/s on 24 processors on the BBN TC2000, and it is also possible to get 1320 Mflop/s with 72 processors on the KSR1 for matrices of order 768 (Amestoy, Daydé, Duff, and Morère (1995)).

This parallel version of the BLAS has been used to exploit, in a transparent manner, parallelism in codes from the LAPACK Library. We show in Table 3 the performance of LAPACK codes corresponding to blocked **LU** (GETRF) and Cholesky (POTRF) factorizations on the BBN TC2000. Only the parallelism internal to the BLAS has been exploited. The matrices are declared "shared interleaved" on the BBN TC2000 (that is, they are logically shared but physically distributed over the memory of the processors).

These results show that most of the possible parallelism in these LAPACK routines has been exploited.

Computer	Precision	Uniproc.	Number of processors					
			1	2	4	8	16	24
BBN TC2000	32 bits	7.8	6.6	13.4	26.2	52.1	98.8	124.4
	64 bits	2.7	2.5	4.9	9.7	19.2	37.2	47.0
KSR1	64 bits	27.5	25.4	42.9	81.9	165.4	305.4	418.3

Table 2: Performance in Mflop/s of GEMM with matrices of order 512 on a BBN TC2000 and a KSR1.

	Precision	Number of processors					
		1	2	4	8	16	24
LU	single	5.4	10.1	18.1	31.1	49.1	60.6
	double	2.0	3.7	6.7	11.3	18.8	24.1
Cholesky	single	6.1	11.6	21.4	38.6	67.6	91.3
	double	2.3	4.4	8.3	14.8	25.9	35.9

Table 3: Performance in Mflop/s on the BBN TC2000 of LU and Cholesky factorizations on matrices of order 2000.

We observe that there are better speedups for the Cholesky factorization because it has a greater percentage of Level 3 BLAS operations than the **LU** factorization.

3 Direct solution of sparse systems (Amestoy, Duff, Puglisi)

We consider the sparse **LU** and **QR** factorizations of square and overdetermined matrices on shared memory multiprocessors. We use a multifrontal approach to design our factorization algorithms. Further background on this approach can be obtained from the original papers by Duff and Reid (Duff and Reid (1983), Duff and Reid (1984)). As is common in sparse elimination, the factorization is split into a symbolic phase, which performs an analysis using only the sparsity pattern of the matrix, and a numerical factorization phase.

In a multifrontal method, the sparse factorization proceeds by a sequence of factorizations on small dense matrices, called frontal matrices. The ordering for the sequence of computations and the frontal matrices are determined by a computational tree, called elimination tree, where each node represents a full matrix factorization and each edge the transfer of data from child to parent node. This elimination tree is determined from the sparsity pattern of the matrix and from a reordering that aims at minimizing the fill-in during the numerical factorization (such as the minimum degree we use here). During the numerical factorization, eliminations at any node can proceed as soon as those at the child nodes have completed. This will be referred to as tree parallelism. Note that the factorization at each node is done using full linear algebra and direct addressing so that we can use the BLAS. All the indirect addressing is confined to the assembly process.

Multifrontal LU factorization

We have developed a parallel multifrontal code for the solution of unsymmetric equations. The results in this section are from runs with this code, called MUPS, and experimental versions of this code (Amestoy and Duff 1989, 1993).

During the **LU** factorization, if we only exploit the tree parallelism, the speed-up is very disappointing. The actual speed-up depends on the problem but is typically only 2 to 3 irrespective of the number of processors. This poor performance is caused by the fact that the tree parallelism decreases while going towards the root of the tree. Moreover, Amestoy and Duff (1993) have observed that typically 75% of the work is performed in the top three levels of the elimination tree. It is thus necessary to obtain further parallelism within the large nodes near the root of the tree (so-called node parallelism) by using parallel versions of the BLAS in the factorizations within the nodes. When combining both tree and node parallelism the situation becomes much more encouraging and we show typical speed-ups for a range of computers in Table 4. A medium size sparse matrix, *BCSSTK15* from the Harwell-Boeing set (Duff, Grimes, and Lewis (1992)), is used to illustrate our discussion. This is a structural analysis matrix of order 3948 with 117816 nonzeros. A minimum degree ordering is used in the analysis and the number of floating-point operations for the LU factorization is 443 million.

		(1)		(2)	
Computer	nprocs	Mflops	(speed-up)	Mflops	(speed-up)
Alliant FX/80	8	15	(1.9)	34	(4.3)
IBM 3090E/3VF	3	83	(1.9)	105	(2.4)
IBM 3090J/6VF	6	126	(2.1)	227	(3.8)
CRAY-2	4	316	(1.8)	404	(2.3)
CRAY Y-MP	6	529	(2.3)	1119	(4.8)
CRAY C90	4	1124	(2.2)	1486	(2.9)

Table 4: Performance summary of the multifrontal **LU** factorization on matrix BCSSTK15. In column (1) we exploit only parallelism from the tree; in column (2) we combine the two levels of parallelism.

Note that, based on recent work done by Amestoy, Davis and Duff on the minimum degree (AMD: Approximate Minimum Degree, Amestoy, Davis, and Duff (1994)), the efficiency of the Minimum Degree has been significantly improved mainly in terms of of computionnal time but also in terms of fill-in in the LU factors and floating point operations to perform the numerical phase. The AMD algorithm is based on a new upper bound for the degree of nodes in the elimination graph that can be easily computed in the context of a minimum degree algorithm. We have demonstrated that this upper-bound for the degree is more accurate than all previously used degree approximations. We have experimentally shown that we can replace an exact degree update by our approximate degree update and obtain almost identical fill-in in the factors. The AMD algorithm is highly competitive with other ordering algorithms. It is typically faster than other minimum degree algorithms, and produces comparable results to Multiple Minimum Degree (from Liu, which is also based on external degree) in terms of fill-in and the number of floating-point operations needed to compute the factors. AMD typically produces better results, in terms of fill-in in the factors and computing time, than the MA27 minimum degree algorithm (based on true degrees). Compared with general nested dissection, AMD is often faster and usually produces better orderings. For example, on matrix BCSSTK15, the time to compute the ordering is reduced by a factor of two while the number of floating-point operations for the LU factorization drops from 443 million to 318 million. On a large range of real matrices, the time to compute the ordering with classical implementations of the Minimum degree algorithm has been decreased by a significant factor (between 2 and 60) when using the new AMD ordering.

The parallel multifrontal solver, MUPS, is designed for shared memory multiprocessors but it was not too difficult to develop a version that could run on the virtual shared memory environment of the BBN TC2000 (Amestoy, Daydé, Duff, and Morère (1995)). However, because access to memory on this machine is not uniform (remote access is not cached by default and takes about 3.5 times the time of local access for a read, and 3.0 for a write), the performance of the shared memory code was not good. It is necessary to pay more attention to data locality in order to get acceptable performance and we show, in Table 3, two versions of the BBN multifrontal code. Version 1 is the relatively straightforward adaptation of the code while version

	Number of processors					
	1	2	4	8	16	26
Version 1	1.35	1.95	2.61	2.93	4.30	4.66
Version 4	2.05	2.86	5.15	8.05	12.32	14.29

Table 5: Performance in Mflops for multifrontal factorisation phase on the BBN TC2000 (matrix BCSSTK15).

4 represents many refinements to this, including explicit copying of data to local memory so that it can be cached and effectively reused during the computation so reducing the amount of remote access. The results in Table 3 show quite clearly that it is still vital to respect data locality when using virtual shared memory. If one does so, then good performance can be obtained.

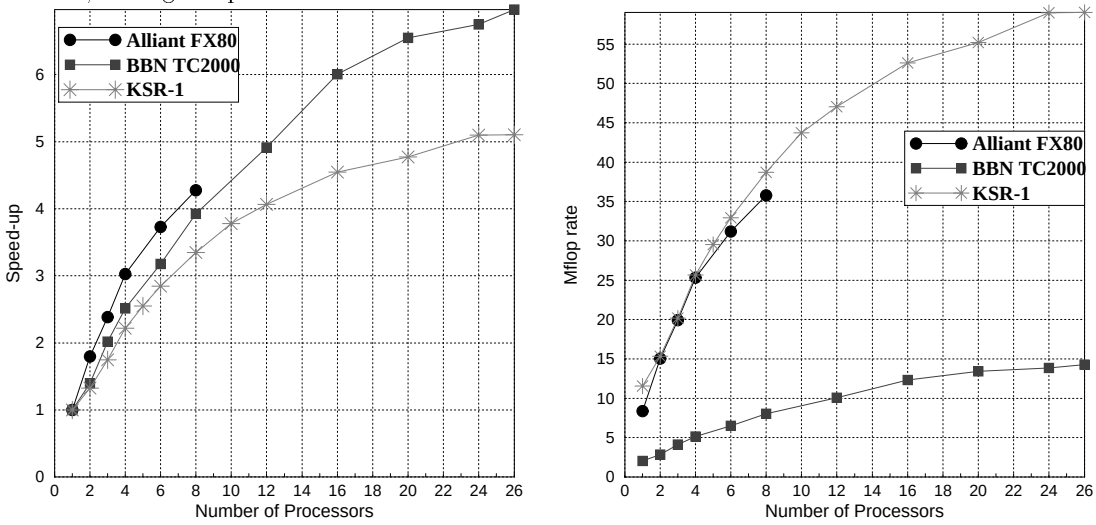

Figure 1. Performance obtained during multifrontal LU factorization on shared and virtual shared memory computers.

A similar approach has been used on another virtually shared memory computer the KSR-1 from Kendall-Square. On this all-cached memory machine, the shared memory code based on tuned level-3 BLAS kernels performs relatively better that on the BBN TC2000. However tuning could not be done in the same way as on the BBN since it is relatively much more complex to control manually the data locality on the KSR because of data caching mecanism which automatically modifies data locality. We compare, in Table 1, the performance obtained during multifrontal LU factorisation on the shared memory Alliant computer with that obtained on the virtually shared memory computers BBN TC2000 and KSR-1. The best version of the code for each computer was used to run the experiments. Although speedups on the BBN are better than on the KSR-1, it should be stressed that the KSR-1 tuned code is a much straightforward adaptation of the shared memory code.

Multifrontal QR factorization

The elimination tree for the **QR** multifrontal factorization is built from the symbolic factorization of the matrix A^TA. A detailed description of the multifrontal approach for the **QR** factorization can be found in Puglisi (1993), Amestoy, Duff, and Puglisi (1994).

In Table 6 we describe the test problems used to illustrate our discussion.

Matrix	m	n	Nzeros	Description
Nfac80	24964	6400	99856	80x80 grid with 4 eqns/elements
Nfac90	31684	8100	126736	90x90 grid with 4 eqns/elements
Nfac100	39204	10000	156816	100x100 grid with 4 eqns/elements
large	28254	17264	75018	animal science
medium2	18794	12238	75039	animal science
large2	56508	34528	225054	animal science

Table 6: Test matrices.

The matrices called Nfacxxx in Table 6 were used by Lewis, Pierce, and Wah (1989) and arise in the natural formulation of the finite-element method (Argyris and Bronlund (1975)). The other test problems in Table 6 are rectangular matrices arising in animal breeding science for the estimation of breeding values. The data was used for pig breeding in Switzerland and was supplied by Andreas Hofer (1990).

A very important issue in designing an algorithm for the **QR** factorization of sparse matrices is the transient fill-in. In fact, the orthogonal factorization of sparse matrices is characterized by fill-in that can appear and disappear throughout the computation and that can substantially increase the total number of arithmetic operations to be performed, the size of the temporary storage and the space required for storing the orthogonal transformations. This kind of fill-in depends on the way in which the rows are zeroed out. Therefore, we have experimented with 3 node factorization strategies in order to minimize the transient fill-in. With the original strategy (*strategy 1*), the frontal matrices are treated as full rectangular matrices and, at each node of the elimination tree, we eliminate the number of fully summed variables. The second one (*strategy 2*) eliminates the dependency between rows of the matrix as soon as possible (triangularization of rectangular overdetermined frontal matrices) and the third one (*strategy 3*) annihilates as many entries as possible under the diagonal in each frontal matrix (triangularization of each frontal matrix). We show, in Table 7, some results concerning the three node factorization strategies.

We found that strategy 3 performs better in terms of memory requirement, number of operations and time (up to 10 time faster than strategy 1 in a uniprocessor environment). This strategy can be viewed as a generalization of Reid's algorithm (Reid (1967)) for banded matrices.

Our **QR** factorization is based on the multifrontal approach and a well-known technique for improving the performance of multifrontal methods is the use of node amalgamation. We have studied the influence of this technique on the performance of the factorization and solve steps. Node amalgamation significantly improves the uniprocessor computing time, while increasing the size of the working space. Furthermore, we see in Table 8 (compare rows 3 and 4) that also in a multiprocessor environment, using node amalgamation, the factorization time decreases quite significantly, even if we have lost a small part of the parallelism of the method. This improvement is essentially due to the reduction of indirect operations during the assembly of the frontal matrices. It should be pointed out that node amalgamation increases the memory space for the **R** factors increases but not necessarily that for the **Q** factors.

We have then studied the parallel implementation of the factorization and solve step on shared memory multiprocessor machines. The multifrontal methods have an intrinsic parallelism represented by the elimination tree: computations corresponding to disjoint subtrees can be performed in parallel. Another possible kind of parallelism, that can be exploited, is the node parallelism. Node parallelism can coexist with the tree parallelism or/and can be exploited at the top of tree when the tree structure does not supply any more parallelism. It consists in subdividing the full factorization into smaller independent tasks that can be run in parallel. We have considered both kind of parallelism, using as much as possible the tree parallelism and using the node parallelism as soon as we detect that only one processor will remain active until the end of the factorization. It can be seen in Table 8 that node parallelism significantly improves the speedup of the

Matrix	Strat.	flops $\times 10^6$	Space for **Q** Reals $\times 10^6$	Space for **Q** Integers $\times 10^6$	Number Hous. transf.	Temporary storage$\times 10^3$	1 proc. time (sec.) Fact.	1 proc. time (sec.) $\mathbf{Q}^T\mathbf{b}$
large	1	2356.0	3.9	0.3	17264	3055	1241.12	7.71
	2	114.2	0.7	0.2	44788	83	475.49	3.69
	3	95.9	0.6	0.2	60411	80	477.44	4.54
medium2	1	3197.6	4.1	0.1	12238	2367	1022.68	5.70
	2	516.0	1.4	0.1	37248	233	294.11	3.54
	3	281.4	0.9	0.2	63737	169	254.34	4.56
large2	1	-	-	-	-	-	-	
	2	5651.7	7.2	0.5	125328	712	2684.72	14.56
	3	3339.9	4.7	0.7	208893	678	2139.57	16.94
Nfac80	1	2605.1	6.6	0.3	6400	2892	871.07	9.95
	2	41.1	0.5	0.2	52846	27	93.20	3.24
	3	41.1	0.5	0.2	52854	27	92.44	3.24
Nfac100	1	-	-	-	-	-	-	-
	2	92.1	0.9	0.2	82,864	63	178.56	5.14
	3	92.1	0.9	0.2	82,872	63	177.84	5.14

Table 7: Comparison between the three strategies of node factorization. The run are made on one processor of the Alliant FX/80. In the case of large2 and Nfac100, the memory requirement with strategy 1 was too large.

method.

Techniques	**large** Time	**large** Spd.	**medium2** Time	**medium2** Spd.	**large2** Time	**large2** Spd.	**Nfac80** Time	**Nfac80** Spd.	**Nfac100** Time	**Nfac100** Spd.
strategy 3	81.11	5.89	70.85	3.59	815.92	2.62	14.98	6.17	32.63	5.45
node amalg.	40.69	5.03	50.53	3.00	630.02	2.18	10.21	5.65	22.90	4.89
node paral.	33.33	6.14	34.59	4.38	406.06	3.38	9.88	5.84	19.67	5.69
relax. BLAS 3	32.73	6.19	28.85	4.96	218.74	4.45	9.88	5.84	19.67	5.69

Table 8: Time (in seconds) and speedups (column Spd) of the factorization step using different techniques on 8 processor of an Alliant FX/80.

To increase the benefit of node parallelism, we have studied a block Level 3 BLAS based full **QR** factorization algorithm. Using Level 3 BLAS, we hoped also to increase the efficiency of the algorithm in an uniprocessor environment. But, because of the block upper triangular structure of the frontal matrices, little efficient use of Level 3 BLAS was possible. In order to enable more Level 3 operations, we have relaxed the sparsity structure of the frontal matrix. In this way we could achieve our goals, that is increasing the uniprocessor Megaflop rate and the parallelism when we switch to the node level parallelism.

We summarize in Table 8 the improvement obtained during factorization using the different techniques in a multiprocessor environment. We see that a significant part of the performance improvement comes from the increase in the parallelism of the method.

Finally, we have studied the performance of the tree parallelism for computing the matrix-vector multiplication $\mathbf{Q}^T\mathbf{b}$, and solving the two triangular systems $\mathbf{Rx} = \mathbf{b}$ and $\mathbf{R}^T\mathbf{x} = \mathbf{b}$.

We see in Table 9 that the tree provides a good parallelism for the $\mathbf{Rx} = \mathbf{b}$ step and for the $\mathbf{Q}^T\mathbf{b}$ operation.

Matrices	Strat.	Nb of Proc.	$\mathbf{Q}^{\mathrm{T}}\mathbf{b}$ time	$\mathbf{Q}^{\mathrm{T}}\mathbf{b}$ Speedup	$\mathbf{Rx} = \mathbf{b}$ time	$\mathbf{Rx} = \mathbf{b}$ Speedup	$\mathbf{R}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$ time	$\mathbf{R}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$ Speedup
large	3	1	2.59		2.69		2.42	
		8	0.62	4.18	0.42	6.40	2.09	1.18
medium2	3	1	2.84		1.79		1.92	
		8	0.77	3.69	0.32	5.59	2.78	0.69
large2	3	1	10.98		5.49		3.07	
		8	3.53	3.11	1.11	4.95	10.87	0.28
Nfac80	3	1	1.80		1.02		1.32	
		8	0.34	5.29	0.18	5.67	1.11	1.19
Nfac100	3	1	2.98		1.65		2.07	
		8	0.70	4.26	0.27	6.11	1.79	1.16

Table 9: Time (in seconds) for computing $\mathbf{Q}^{\mathrm{T}}\mathbf{b}$, $\mathbf{Rx} = \mathbf{b}$ and $\mathbf{R}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$.

On the contrary for the $\mathbf{R}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$ step, the overhead introduced by the parallelism is such that it is preferable to compute the forward-substitution in sequential manner.

4 Block iterative methods (M. Arioli, L. A. Drummond, I. S. Duff, D. Ruiz)

We study the implementation in multiprocessor environments of block iterative methods from the class of "row projection methods". This class of iterative methods is widely used in the frame of image reconstruction, for example, and is recognized to be quite robust (cf. Kamath and Sameh (1988), Bramley and Sameh (1992)).

We have developed a block version of the method of Cimmino (see Cimmino (1939)) for the solution of consistant sparse linear systems

$$\mathbf{Ax} = \mathbf{b} \tag{3}$$

where $\mathbf{A}$ is an $m \times n$ sparse matrix ($m \leq n$), and $\mathbf{b}$ is an m-vector. In this block iterative method, $\mathbf{A}$ is partitioned in a row-oriented way

$$\begin{pmatrix} \mathbf{A}^1 \\ \vdots \\ \mathbf{A}^p \end{pmatrix} \mathbf{x} = \begin{pmatrix} \mathbf{b}^1 \\ \vdots \\ \mathbf{b}^p \end{pmatrix} \tag{4}$$

where $1 \leq p \leq n$. The current iterate is projected simultaneously onto the manifolds $\mathbf{A}^i\mathbf{x} = \mathbf{b}^i$, $i = 1, \ldots, p$, where $\mathbf{b}^i$ is the block of $\mathbf{b}$ corresponding to $\mathbf{A}^i$, and the new iterate is obtained from a convex combination of all the resulting vectors. This gives rise to a general purpose iterative solver with obvious parallelism, suitable both for shared memory and distributed memory parallel computers. If we take $p = n$, which means that each manifold from (4) is a hyperplane defined by one equation in (3), we then recover the algorithm of Cimmino (see Sloboda (1991)).

A general study of the convergence of this method, and of other related block-row and block-columns methods, like the block SOR method for instance, can be found in Elfving (1980). Elfving calls this method the "block-row Jacobi" method, because the iteration matrix of the block Jacobi method applied to $\mathbf{AA}^T$ and the iteration matrix of block Cimmino are similar. If the block rows $\mathbf{A}^i$ are nearly mutually orthogonal, i.e. $\mathbf{AA}^T$ is strongly block-diagonally dominant, we can expect that the method will converge very quickly. The basic method can be accelerated by using the conjugate gradient algorithm. This acceleration is very

dependent on a partitioning of the original system and we discuss several possible partitionings. The case of block tridiagonal matrices, which is very common in PDE discretization problems for instance, is very important because, for such structures, we can introduce a partitioning with a high degree of parallelism and a fast convergence. To this end, we exploit algorithms which maximize the minimum element on the diagonal, and which reorder $\mathbf{A}$ to block tridiagonal form. In this way, we can identify a partitioning (4) in which $\mathbf{A}^i$ will be structurally orthogonal to $\mathbf{A}^{i+2}$ $(i = 1, 2, ..., p-2)$, and which is therefore equivalent to a partitioning in two blocks. For such a partitioning, Elfving (1980) shows that the eigenvalues of the iteration matrix of the Block Cimmino method are the cosines of the principal angles between the two manifolds defined by these two blocks. In this case, the spectrum of the iteration matrix lies strictly between -1 and 1, is symmetric around 0, and has many eigenvalues clustered around 0. For iteration matrices which are not too ill-conditioned, the Conjugate Gradient algorithm is a good method for accelerating the convergence of the iterative scheme.

Underdetermined systems corresponding to the subproblems of the partitioned system are solved using the Harwell sparse symmetric indefinite solver MA27 on an augmented system. These systems are independent and can be solved in parallel. Additionnally, each of these augmented systems constitutes a sparse linear system which can also be solved using multifrontal sparse direct solvers (see Amestoy (1991)). This enables then to exploit two levels of parallelism in the method, one directly determined by the partitioning (4) and resulting in loosely coupled tasks synchronized in the iterative part of the algorithm, and a second one depending on the sparse structure of the subsystems. We must not forget also the third level of parallelism inherent to the use of dense kernels (the BLAS, for instance; see Anderson, Bai, Bischof et al. (1992)) at all stages of the computations. The potential of this third level of parallelism can indeed be quite high, as it has already been highlighted in the case of multifrontal sparse direct solvers (see Amestoy (1991)). We finally mention that, in the context of this study, the usual three stages of multifrontal methods (analysis, factorization, and forward and backward substitution) can be executed in parallel on different subsystems.

Our first concern was the study of the first level of parallelism, e.g. the one coming from the partitioning of the original linear system. We have therefore disabled the second and third levels of parallelism mentioned above, in order to isolate and better understand the influence of the partitioning strategy on the resulting degree of parallelism in the method. The various aspects of our algorithm have been tested on an eight processor Alliant FX/80. The effect of partitioning on the number of iterations and overall elapsed time for solution has been studied (see Arioli, Duff, Noailles, and Ruiz (1992)). Our numerical experiments reinforce what we expect from the theory, showing that we can vary the number of iterations and tune the speedup, using simple heuristics like the size of the subblocks in the two-block partitioning (see Figure 2 as an example). The results also indicate that the row projection methods are well suited for parallel processing. Bramley (1989) has also experienced the efficiency of these methods in multiprocessor environments. His approach to the implementation of row projection methods mainly differs from ours through the choice of the normal equations approach instead of the augmented systems for the solution of the subproblems. We also verified experimentally that the two approaches are complementary, in the sense that they are suitable for different partitioning strategies.

One important step in our study has been to determine the limits of the method. The experiments enabled to see that, for iteration matrices which are not too ill-conditioned, the block Cimmino method is a good preconditioner for the conjugate gradient scheme (with respect to robustness). An analysis of the iteration matrix for the conjugate gradient acceleration leads us to consider rather unusual and novel scalings of the matrix that alter the spectrum of the iteration matrix to reduce the number of CG iterations. These appeared to be good complementary preconditioners when the convergence would be otherwise slow. On the other hand, for ill-conditioned problems, the convergence of the conjugate gradient acceleration is slow. The iteration profile of the residuals shows long plateaux eventually followed by a sharp drop. More sophisticated preconditioning techniques than the previous diagonal ones help by shortening the plateaux, but do not remove them (see, for instance, the convergence profile of the classical conjugate gradient in Figure 3). In some cases, the reduction of the plateaux is not sufficient for balancing the extra amount of computation induced by these preconditioning techniques.

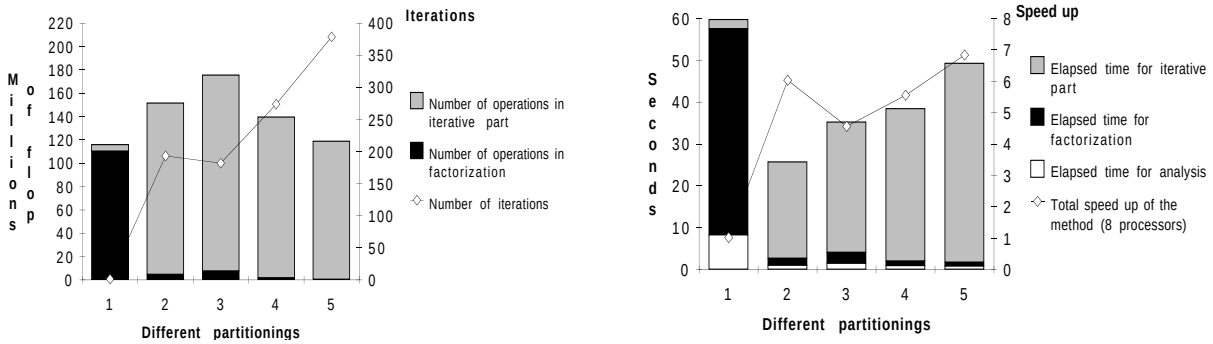

Figure 2: 2 dimentional PDE test problem on a 64×64 grid. $\omega_k \leq 10^{-14}$.

We then analysed the reasons for this poor convergence of the conjugate gradient acceleration. The spectrum of the iteration matrix exhibits, in these cases, a strong clustering of eigenvalues, which emphasizes the fact that the block Cimmino method is in general a good preconditioner for the conjugate gradient method. However, some clusters of eigenvalues also occur at the ends of the spectrum, and in the case of ill-conditioned problems, are the cause for the poor convergence of the conjugate gradient method. We therefore tried variants of the block Lanczos method for solving linear systems, including the block conjugate gradient method of O'Leary (1980), for accelerating our basic block iterative method. In contrast to the classical conjugate gradient algorithm, these block techniques allow the computation of multiple eigenvalues, and thus present a nice ability for detecting clusters of eigenvalues in finite arithmetic. On the basis of the comparisons between the Lanczos and the conjugate gradient methods, we identified a numerically stable variant of the block conjugate gradient method (see Arioli, Duff, Ruiz, and Sadkane (1992)). We also gave experimental evidence for the fact that the block conjugate gradient method behaves more like a direct method as opposed to the classical conjugate gradient method, which in case of well conditioned problems behaves like a stationary iterative method with a linear rate of convergence. Furthermore, in some test examples, these techniques are convergent whereas the Conjugate Gradient method is not. Finally, it must be emphasized that even if these block acceleration techniques may increase the amount of arithmetic, they can still be more efficient in a parallel environment since they allow the use of efficient level 3 BLAS kernels. This last point is illustrated in Figures 3 and 4, where on an eight processor Alliant FX/80, the effect of BLAS 3 kernels enhances the efficiency with increasing block sizes, and can even overcome the potential increase in the amount of work.

The first axe of research under current investigation concerns the study of this class of block iterative methods in distributed memory parallel computers and/or heterogeneous network of computers. A new version of the block Cimmino code, well suited for this kind of multiprocessor environments, has been developed. This work takes place in the frame of the PHD thesis of L. A. Drummond at CERFACS, which has started in January 1992.

In this framework, the different heterogeneous processors are classified into single computing elements, shared memory clusters, or distributed memory clusters. The purpose of this classification is to reuse whenever possible the shared memory implementation inside the shared memory clusters and to exploit the internal levels of parallelism (see the above paragraphs for a discussion about the different levels of parallelism in the method). Preliminary results have been obtained from experiments performed on clusters of workstations, using PVM 3.2 to express the first level of parallelism coming from the partitioning (4).

As mentioned previouly, we accelerate the block Cimmino method with the block conjugate gradient algorithm. This requires to set up an efficient distributed version of the block conjugate gradient algorithm. Three parallel implementations of the Block-CG algorithm have therefore been experimented. In the first implementation, we choose an interprocess communication scheme in which every process has knowledge about

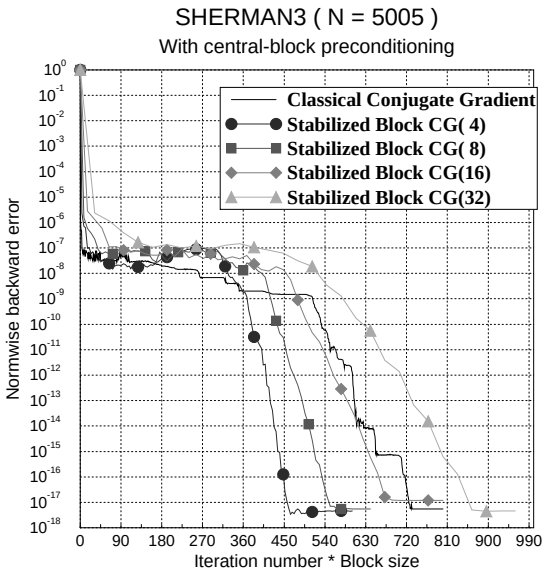

Figure 3: Normalized iteration counts performed by the stabilized block conjugate gradient acceleration on problem SHERMAN3.

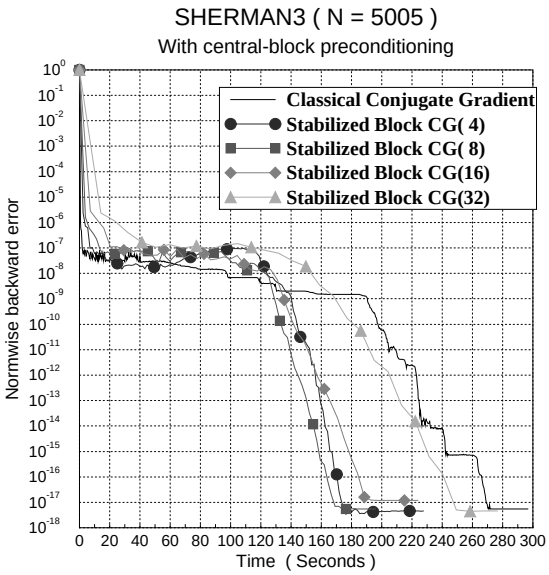

Figure 4: Time in seconds of the stabilized block conjugate gradient acceleration on problem SHERMAN3. Computations performed on an Alliant FX/80 (8 processors).

the information it will need to share with its neighbours. After completion of the data exchanges, there may be redundant computations (All-to-All). To reduce the number of messages and redundant computations, we develop a second implementation in which a process is assigned the task of merging local results from neighbouring processes, sequentially performs some steps of the Block-CG implementation, and broadcasts results back to neighbouring processes to resume the parallel execution of the Block-CG (Master-Slave : distributed). Finally, we developped an implementation where a single process performs the steps of the Block-CG, and only the $matrix - matrix$ products that involve the iteration matrix are performed in parallel (Master-Slave : centralized). In Arioli, Drummond, Duff, and Ruiz (1994a), we present results obtained for the three implementations using PVM 3 on a BBN TC2000 computer (see Table 10) and a heterogeneous network of IBM RS6000 and SUN Sparc 10 workstations.

Laplace Matrix 4096 x 4096 (Block size = 4, 171 iterations) Elapsed Time of sequential version = 279142						
Number of	All-to-All		Mstr-Slv: distributed		Mstr-Slv: centralized	
PE's	Elps. Time	Speed-up	Elps. Time	Speed-up	Elps. Time	Speed-up
1	278827	1.001	279436	0.999	-*-	-*-
2	143083	1.951	143419	1.946	301884	0.925
4	71393	3.910	71244	3.918	278184	1.003
8	40755	6.849	38798	7.195	273320	1.021
12	40668	6.864	29747	9.384	279414	0.999
16	57759	4.833	25452	10.967	283649	0.984

Table 10: Test matrix generated from a discretization on a 64×64 grid: Laplace's equation. Times shown in table are in microseconds. The experiments are performed on the BBN TC2000.

In the Parallel distributed Cimmino solver the number of generated subsystems, for numerical reasons, is related with the structure of the problem and not with the number of available computing elements. Therefore, we implemented a scheduler that statically distributes tasks to the computing elements trying to keep the work load balanced among the processing elements and to take advantage of available interconnection networks. Part of our current research objectives is to test the scheduler in a heterogeneous environment using

PVM 3.2. In this respect, we did some experiments on a heterogeneous network of SUN, IBM RS6000, and Hewlett Packard workstations (see Arioli, Drummond, Duff, and Ruiz (1994b)), which helped us to tune the parallel scheduler to better distribute the workload among the heterogeneous processing elements. For example, it has been verified that, in heterogenous networks of workstations, it is really important to avoid as much as possible communication.

In addition to the study of the parallelism of the method and to the development of tools well suited for distributed memory environments, we study some preconditioning techniques as well as techniques for preprocessing the linear system of equations. The Block Cimmino method presents indeed some similarities with domain decomposition methods. From this similarities, we plan to investigate the possible application of different techniques commonly used with the domain decomposition methods to the Block Cimmino method. On the basis of the analysis of the pattern and/or the values of the matrices $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$, we also try to define some techniques for automatic partitioning. These different aspects still require a lot of investigations and are planned for longer term research activities.

An effort has been spent also on the comparison of the convergence of the block conjugate gradient algorithm with that of the classical conjugate gradient algorithm. Out of this study, we propose an approach which would enable to compute consecutive solutions with the same matrix and different right-hand sides with a much cheaper cost than the corresponding successive runs of the block conjugate gradient algorithm. The approach is based on the combination of Subspace Inverse Iteration for eigenvalues and eigenvectors computation with Block-CG and Classical-CG for the solution of linear systems. The theoretical study of the properties of the algorithm resulting from this combination is under current study.

Finally, we intend to investigate to what extent the different tools and/or techniques developed in this study can be generalized and reused in a more general framework than that of the row-projections iterative methods.

5 Use of element-by-element preconditioners (Daydé, l'Excellent, Gould)

We now report on the iterative solution of the sparse structured linear systems. We are especially interested by the linear systems that come from the solution of large scale nonlinear optimization problems using the all-pervasive property of partial separability, first studied by Griewank and Toint (1982) The function f is said to be *partially separable* if $f(x) = \sum_{i=1}^{p} f_i(x)$, where each *element* function f_i has a large invariant subspace. Crucially, the Hessian matrix of such a function is of the form of (2).

In unconstrained optimization, one is normally concerned with obtaining an (approximate) solution, d, to the Newton equations

$$\nabla_{xx} f(x) d = -\nabla_x f(x)$$

If f is partially separable , these equations become :

$$\left(\sum_{i=1}^{m} A_i \right) d = - \left(\sum_{i=1}^{m} g_i \right),$$

where $g_i = \nabla_x f_i(x)$ and $A_i = \nabla_{xx} f_i(x)$, is sought.

We assume that A is a large and normally positive definite symmetric n by n matrix. The preconditioned conjugate gradient method is a very attractive iterative solution technique for such problems. It requires, at each iteration, a matrix-vector product of the form :

$$w = \left(\sum_{i=1}^{m} A_i \right) v = \sum_{i=1}^{m} (A_i v)$$

for given input v, and the solution of a linear system :

$$Pz = r$$

where P is the preconditioner.

As A has a special structure, we would hope to construct appropriate preconditioners which mimic the structure of A or which use the structure in some other way. In Daydé, L'Excellent, and Gould (1994), we consider a number of existing and new Element-by-Element preconditioners, originally designed for the finite element solution of partial differential equations and report that these preconditioners appear to be most effective in the optimization context. We study here the use of Element-By-Element (EBE) preconditioners that were introduced by Hughes, Levit, and Windget (1983) and Ortiz, Pinsky, and Taylor (1983) and have been successfully applied in a number of applications in engineering and physics (see, for example, Hughes, Ferencz, and Hallquits (1987), and Erhel, Traynard, and Vidrascu (1991)). A detailed analysis of this technique is given by Wathen (1989). The following element-by-element preconditioners are considered : the Element matrix Factorization (EMF) of Gustafsson and Linskog (1986) based on a Cholesky factorization of each element; the Finite Element preconditioner (FEP) of Kaasschieter (1989); the one-pass (EBE) and two-pass (EBE2) element-by-element preconditioners of Hughes, Levit, and Windget (1983) and Ortiz, Pinsky, and Taylor (1983), initially described and used in the context of finite element techniques for partial differential equations; and the "Gauss Seidel" EBE preconditioner (GSEBE).

These preconditioners have some nice features. They can be computed element-wise and most of them do not require assembly. Furthermore, they allow efficient parallelization of the matrix-vector product and of the solution step involved in the preconditioned gradient iteration.

We only describe here the EBE preconditioner since, in our experience, this is one of the most promising.

Assuming that A is positive definite, we may rewrite A as

$$A = \sum_{i=1}^{p} M_i + \sum_{i=1}^{p} (A_i - M_i) = M + \sum_{i=1}^{p} (A_i - M_i),$$

where $M_i = diag(A_i)$ and $M = \sum_{i=1}^{p} M_i = diag(A)$. Now, let $M = L_M L_M^T$ be the Cholesky factorization of M; of course L_M is simply a diagonal matrix. Then,

$$A = L_M \left(I + \sum_{i=1}^{p} L_M^{-1} (A_i - M_i) L_M^{-T} \right) L_M^T = L_M \left(I + \sum_{i=1}^{p} E_i \right) L_M^T,$$

where $E_i = L_M^{-1} (A_i - M_i) L_M^{-T}$. The EBE preconditioner is given by :

$$P_{EBE} = L_M \left(\prod_{i=1}^{p} L_i \right) \left(\prod_{i=1}^{p} D_i \right) \left(\prod_{i=p}^{1} L_i^T \right) L_M^T \tag{5}$$

where the L_i and D_i factors come from the LDL^T factorization of the matrices $I + E_i$.

The computation of this preconditioner is completely parallel. We are free to order the elements in any way we choose and may thus encourage parallelism by consecutively ordering non-overlapping elements so that backsolves can be performed in parallel. We apply therefore a graph coloring algorithm on the elements.

In Table 13, we report on the application of EBE, EBE2 and GSEBE on two test problems : CEGB2802 and MAN5976, described in Table 12.

Problem name	Order	Number of elements	Min element size	Max element size	Mean element size	Degree of overlap	Condition number
CEGB2802	2694	108	42	60	58.7	2.4	5.7×10^4
MAN5976	5882	785	20	20	20.0	2.7	5.0×10^1
CBRATU3D	4934	4934	5	8	7.5	7.5	3.4×10^1
NET3	512	531	1	6	2.6	2.7	2.4×10^9

Table 11: Summary of the characteristics of each test problem

Problem name	Precondi-tioner	Calculating the preconditioner	Number of iterations	Time for convergence	Time per iteration	Resi dual norm (log10)
CEGB2802	None	0	2040	481.5	0.24	-9.0
	Diagonal	0.035	661	158.4	0.24	-9.1
$k = 5.7 \times 10^4$	EBE	6.4	129	75.7	0.58	-9.0
	EBE2	6.3	145	131.0	0.90	-9.0
	GS EBE	0.68	362	211.2	0.58	-9.0
MAN5976	None	0	68	22.7	0.33	-9.1
	Diagonal	0.081	28	9.8	0.33	-9.1
$k = 5.0 \times 10^1$	EBE	5.2	10	10.1	0.91	-9.4
	EBE2	5.0	11	17.5	1.45	-9.6
	GS EBE	0.85	12	11.9	0.91	-9.3

Table 12: Results for CEGB2802 and MAN5976. Times are in seconds

We observe that element-by-element preconditioners do not appear to be significantly more effective than diagonal preconditioning on some examples (cf MAN5976). However we observe that, as we might hope, these preconditioners do appear to be more effective than their diagonal counterparts for the ill-conditioned problems.

Clearly, the efficiency of the EBE preconditioner depends on the partitioning of the initial matrix and on the magnitude of the off-diagonal elements of the elementary matrices. As the decomposition of A is, in general, not unique, different decompositions may significantly affect the performance of the preconditioner. Indeed it may happen that the local variable set for one element is completely contained within another and it would pay to merge the two elements into one single.

In order to improve both the computational performance and the numerical convergence of the method, we roughly decide to amalgamate two elements into a group if their overlap is large. The advantages of such techniques may be the following :

- the number of operations in the matrix-vector product and in the preconditioning step is decreased ;
- the elements become larger and more effective use of BLAS and LAPACK routines can be made ; and
- the behaviour of element-by-element preconditioners is improved since each element contains more global information.

The main difficulty is to define a suitable heuristic to control this amalgamation process. Two strategies were considered : **amal1** is an amalgamation heuristic that aims at minimizing the time spent in matrix-vector products; **amal2** is another amalgamation heuristic well suited for the EBE preconditioner since it tries to minimize the time spent both in the matrix-vector products and in the forward-backward solution. In Table 14 we report the number of iterations, the time and the speed-up per iteration, and the solution time (in seconds), of a diagonal preconditioner and the EBE preconditioner on the Alliant FX/80 using our two amalgamation heuristics. The matrix **NET3** is of order 512 and has 538 elements, while the second one, **CBRATU3D**, is of order 4394 and has 4394 elements. Both problems are described in Table 12.

Problem	Preconditioner		Iterations		Time and speed-up per iteration		Time for solution	
			seq	par	seq	8 procs	seq	8 procs
NET3	Diagonal	initial	1561		0.04	3.72	57.3	15.4
		amal1	1595		0.01	2.77	23.8	2.9
	EBE	initial	628	696	0.14	4.90	88.1	19.9
		amal2	140	137	0.04	3.35	5.18	1.5
CBRATU3D	Diagonal	initial	53		0.59	4.50	31.9	7.1
		amal1	53		0.50	4.57	27.3	6.0
	EBE	initial	24	25	1.95	5.74	48.8	8.8
		amal2	19	19	1.28	4.94	25.7	5.2

Table 13: Performance summary of the diagonal and the EBE preconditioners on the Alliant FX/80.

It appears that amalgamation can be very efficient and that the ratio EBE to diagonal preconditioner solution times decreases as the amalgamation is applied, both because typically because fewer iterations are required following the amalgamation and because an increase in element sizes encourages efficient vectorization for EBE and matrix-vector products — diagonal preconditioning is already completely vectorizable.

It is clear that preprocessing should be applied to any large-scale optimization problem, and that in our case it is important to amalgamate elements and find a suitable colouring for later calculations. This is a difficult task as many criteria need to be taken into account. This preprocessing may well be quite costly but we expect there to be longer-term payoffs.

We currently study the implementation of an Incomplete Cholesky (element-wise) preconditioner. For the numerical quality of the preconditioner, larger sparse elements could be considered and a compromise found between :

- using large elements with low overlap ; and
- keeping enough elements for an efficient parallelization and avoid increasing too much the work at each iteration.

A public domain package implementing these element-by-element techniques is under development.

6 Concluding remarks and perspective

We clearly want to continue our work on designing efficient linear algebra kernels for RISC processors, vector processors and virtual shared memory computers.

In collaboration with CERFACS and RAL, we plan to develop a distributed memory version of the sparse LU multifrontal code (MUPS) based on a portable environnement, such as PVM or MPI. In parallel, we will also investigate the extension of the multifrontal QR algorithm in the same environement.

In collaboration with CERFACS and Pavia, we will continue to investigate the numerical aspects of iterative methods, and the distributed memory implementation of block iterative methods.

In collaboration with RAL, we plan to develop a sparse cholesky multifrontal code (complete and incomplete factorization) for symmetric structured, possibly unassembled, matrices. This will be extremely useful for matrices arising in large scale optimization problems, but can obviously be used on a wide range of problems. We will also continue our studies on the use of element-by-element preconditioners and will consider their use within an optimization software.

References

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

P. R. Amestoy and I. S. Duff, (1993), *Memory allocation issues in sparse multiprocessor multifrontal methodsl methods*, Int. J. of Supercomputer Applics., **7**, 64–82.

P. Amestoy, T. A. Davis, and I. S. Duff, (1994), *An approximate minimum degree ordering algorithm*, Technical Report TR-94-039, CIS Dept., Univ. of Florida. (submitted to SIAM J. Matrix Analysis and Applications).

P. R. Amestoy, M. J. Daydé, I. S. Duff, and P. Morère, (1995), *Linear algebra calculations on a virtual shared memory computer*, Int Journal of High Speed Computing, ??–?? To appear.

P. R. Amestoy, I. S. Duff, and C. Puglisi, (1994), *Multifrontal QR factorization in a multiprocessor environment*, Tech. Rep. TR/PA/94/09, CERFACS, Toulouse, France. submitted to Int. Journal of Num. Linear Alg. and Appl.

P. R. Amestoy, (1991), *Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment*, PhD Thesis TH/PA/91/2, CERFACS, Toulouse.

E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, (1992), *LAPACK : A portable linear algebra library for high-performance computers*, SIAM, Philadelphia.

J. H. Argyris and O. E. Bronlund, (1975), *The natural factor formulation of the stiffness for the matrix displacement method*, Comput. Meth. Appl. Mech. Engrg., **5**, 97–119.

M. Arioli, A. Drummond, I. Duff, and D. Ruiz, (1994a), *Parallel Block Iterative Solvers for Heterogeneous Computing Environments*, in Algorithms and Parallel VLSI Architectures III, Proceedings of The International Workshop, Elsevier.

M. Arioli, A. Drummond, I. S. Duff, and D. Ruiz, (1994b), *A Parallel Scheduler for Block Iterative Solvers in Heterogeneous Computing Environments*, Technical Report, CERFACS, Toulouse, France. to appear in the Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing.

M. Arioli, I. S. Duff, J. Noailles, and D. Ruiz, (1992), *A Block Projection Method for Sparse Matrices*, SIAM J. Scient. Statist. Comput., **13**, 47–70.

M. Arioli, I. S. Duff, D. Ruiz, and M. Sadkane, (1992), *Block Lanczos techniques for accelerating the Block Cimmino method*, Technical Report TR/PA/92/70, CERFACS, Toulouse, France.

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

R. Bramley, (1989), *Row projection methods for linear systems*, PhD Thesis 881, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL.

G. Cimmino, (1939), *Calcolo approssimato per le soluzioni dei sistemi di equazioni lineari*, Ric. Sci. Progr. tecn. econom. naz., **9**, 326–333.

M. J. Daydé and I. S. Duff, (1995), *A Block Implementation of Level 3 BLAS for RISC Processors*, Technical Report (in preparation), ENSEEIHT-IRIT.

M. J. Daydé, I. S. Duff, and A. Petitet, (1994), *A Parallel Block Implementation of Level 3 BLAS Kernels for MIMD Vector Processors*, ACM Transactions on Mathematical Software, **20**, 178–193.

M. J. Daydé, J. Y. L'Excellent, and N. I. M. Gould, (1994), *On the Use of Element-by-Element Preconditioners to Solve Large Scale Partially Separable Optimization Problems*, Technical Report, ENSEEIHT-IRIT, Toulouse, France. RT/APO/94/4.

J. J. Dongarra, J. J. Du Croz, I. S. Duff, and S. Hammarling, (1990), *Algorithm 679. A set of Level 3 Basic Linear Algebra Subprograms.*, ACM Transactions on Mathematical Software, **16**, 1–17.

I. S. Duff and J. K. Reid, (1983), *The multifrontal solution of indefinite sparse symmetric linear systems*, ACM Transactions on Mathematical Software, **9**, 302–325.

I. S. Duff and J. K. Reid, (1984), *The multifrontal solution of unsymmetric sets of linear systems*, SIAM Journal on Scientific and Statistical Computing, **5**, 633–641.

I. S. Duff, R. G. Grimes, and J. G. Lewis, (1992), *Users' Guide for the Harwell-Boeing Sparse Matrix Collection (Release I)*, Technical Report RAL 92-086, Rutherford Appleton Laboratory.

T. Elfving, (1980), *Block-iterative methods for consistent and inconsistent linear equations*, Numer. Math., **35**, 1–12.

J. Erhel, A. Traynard, and M. Vidrascu, (1991), *An element-by-element preconditioned conjugate gradient method implemented on a vector computer*, Parallel Computing, **17**, 1051–1065.

A. Griewank and P. L. Toint, (1982), *On the unconstrained optimization of partially separable functions*, in Nonlinear Optimization, M. J. D. Powell, ed., Academic Press, London and New York.

I. Gustafsson and G. Linskog, (1986), *A preconditioning technique based on element matrix factorization*, Computational Methods in Applied Mechanics, **55**, 201–220.

A. Hofer, (1990), *Schätzung von Zuchtwerten feldgeprüfter Schweine mit einem Mehrmerkmals-Tiermodell*, PhD thesis, EHT-Zurich.

T. J. R. Hughes, R. M. Ferencz, and J. O. Hallquits, (1987), *Large-scale vectorized implicit calculations in solid mechanics on a CRAY X-MP/48 utilizing EBE preconditioned conjugate gradients*, Computational Methods in Applied Mechanics and Engineering, **61**, 215–248.

T. J. R. Hughes, I. Levit, and J. Windget, (1983), *An element-by-element solution algorithm for problems of structural and solid mechanics*, Compututational Methods in Applied Mechanics and Engineering, **36**, 241–254.

E. F. Kaasschieter, (1989), *A general finite element preconditioning for the conjugate gradient method*, BIT, **29**, 824–849.

B. Kågström, P. Ling, and C. V. Loan, (1993), *Portable High performance GEMM-Based Level-3 BLAS*, in To appear Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, SIAM.

C. Kamath and A. Sameh, (1988), *A projection method for solving nonsymmetric linear systems on multiprocessors*, Parallel Computing, **9**, 291–312.

J. G. Lewis, D. J. Pierce, and D. C. Wah, (1989), *A Multifrontal Householder QR Factorization*, Technical Report ECA-TR-127, Engineering and Scientific Services Division, Boeing Computer Services.

D. P. O'Leary, (1980), *The block conjugate gradient algorithm and related methods*, Linear Algebra and its Applications, **29**, 293–322.

M. Ortiz, P. M. Pinsky, and R. L. Taylor, (1983), *Unconditionally stable element-by-element algorithms for dynamic problems*, Compututational Methods in Applied Mechanics and Engineering, **36**, 223–239.

C. Puglisi, (1993), *QR Factorization of large sparse overdetermined and square matrices using a multifrontal method in a multiprocessor environment*, PhD Thesis TH/PA/93/33, CERFACS, Toulouse.

J. K. Reid, (1967), *A Note on the Least Squares Solution of a Band System of Linear Equations by Householder Reductions*, Comput J., **10**, 188–189.

F. Sloboda, (1991), *A projection method of the Cimmino type for linear algebraic systems*, Parallel Computing, **17**, 435–442.

A. J. Wathen, (1989), *An analysis of some element-by-element techniques*, Computational Methods in Applied Mechanics and Engineering, **74**, 271–287.