# Use of Computational Kernels in full and Sparse Linear Solvers, Efficient Code Design on High-Performance RISC Processors[1]

Michel J. Daydé[2] and Iain S. Duff[3]

Technical Report TR/PA/96/47

CERFACS
42 Ave G. Coriolis
31057 Toulouse Cedex
France

## ABSTRACT

We believe that the availability of portable and efficient serial and parallel numerical libraries that can be used as building blocks is extremely important for both simplifying application software development and improving reliability. This is illustrated by considering the solution of full and sparse linear systems. We describe successive layers of computational kernels such as the BLAS, the sparse BLAS, blocked algorithms for factorizing full systems, direct and iterative methods for sparse linear systems.
We also show how the architecture of the today's powerful RISC processors may influence efficient code design.

**Keywords:** Level 3 BLAS, matrix-matrix kernels, RISC processors, blocked Eskow-Schnabel, parallel BLAS, sparse direct methods, solution linear systems, element-by-element preconditioning, sparse BLAS.
**AMS(MOS) subject classifications:** 65F05, 65F50.

[1]Part of this study was funded by Conseil Régional Midi-Pyrénées under project DAE1/RECH/9308020. This paper was presented by the authors in two invited talks at the meeting VECPAR'96 in Porto from 25th to 27th September, 1996.
Also appeared as ENSEEIHT-IRIT Report RT/APO/96/4.

[2]Email: dayde@enseeiht.fr. ENSEEIHT-IRIT, 2 rue Camichel, 31071 Toulouse CEDEX, France.

[3]Email: duff@cerfacs.fr. Also at Atlas Centre, RAL, Oxon OX11 0QX, England.

## Contents

# 1 Introduction

One of the common problems for application scientists is to exploit as efficiently as possible the hardware of high-performance computers (either serial or parallel) without totally rewriting or redesigning existing codes and algorithms. We believe that the availability of portable and efficient serial and parallel numerical libraries that can be used as building blocks is extremely important for both simplifying application software development and improving reliability.

The availability of powerful RISC processors is of major importance in today's market since they are used both in workstations and in the most recent parallel computers. They are usually more efficient than vector processors on scalar applications. The main reason for their success in the marketplace is their very good cost to performance ratio. They are used as a CPU both in workstations and in most of the current MPPs (DEC Alpha in the CRAY T3D, SPARC in the CM5 and PCI CS2, HP PA in the CONVEX EXEMPLAR, and POWER processors in the IBM SP1 and SP2). We report in Table 1.1 the uniprocessor performance of some current RISC processors on the double precision 100-by-100 and 1000-by-1000 LINPACK benchmarks (Dongarra, 1992). We also record their peak performance.

| Computer | LINPACK 100*100 | LINPACK 1000*1000 | Peak performance |
|---|---|---|---|
| DEC 8400 5/300 | 140 | 411 | 600 |
| IBM POWER2-990 | 140 | 254 | 286 |
| HP 9000/755 | 41 | 107 | 200 |
| SGI POWER Challenge | 104 | 261 | 300 |

Table 1.1: Performance in MFlops of RISC processors on the double precision LINPACK benchmarks

We briefly consider the impact of the memory hierarchy on the performance of RISC architectures in Section 2. In Section 3, we show how portable and efficient serial and parallel versions of the Level 3 BLAS can be designed for RISC-based computers using code tuning techniques such as blocking, copying, and loop unrolling. In Section 4, we give an example of a blocked factorization algorithm of special interest for optimization algorithms. This block version of the modified Cholesky factorization of Eskow and Schnabel is designed to make intensive use of the Level 3 BLAS in the same way as other block algorithms. In Section 5, we indicate how BLAS for full matrices can be used within codes for the direct solution of sparse linear equations. In Section 6, we show how an efficient preconditioned conjugate gradient algorithm for symmetric, partially separable, unassembled linear systems can be designed to make use of the previously described computational kernels. We use Element-by-Element preconditioners to exploit the structure of the problems. We demonstrate how a numerical preprocessing step can dramatically improve both the numerical behaviour and the performance of the preconditioners. We describe, in Section 7, an extension of the BLAS for handling sparse matrix operations and indicate its use in the iterative solution of sparse equations. We present some concluding remarks in Section 8.

## 2 Impact of the Memory Hierarchy of RISC Processors on Performance

The ability of the memory to supply data to the processors at a sufficient rate is crucial on most modern computers. This necessitates a complex memory organization, where the memory is usually arranged in a hierarchical manner. The minimization of data transfers between the levels of the memory hierarchy is a key issue for performance (Gallivan, Jalby and Meier, 1987, Gallivan, Jalby, Meier and Sameh, 1988).

Most of the RISC-based architectures use a more complex memory hierarchy than is usually the case for vector processing units. This normally involves one or several levels of cache. Calculations are pipelined over independent scalar operations instead of vector operations. This is why the design of codes for RISC processors may substantially differ from that of code for vector processors. A high reuse of the data located at the highest levels of the memory hierarchy is required for efficient codes for RISC-based architectures. The use of tuned computational kernels that can be used as building blocks is crucial for both simplifying application software development and achieving high performance.

The cache memory is used to mask the memory latency (typically the cache latency is around 1-2 clocks while it is often 10 times higher for the memory). The code performance is high so long as the cache hit ratio is close to 100%. This will happen if the data involved in the calculations can fit in the cache or if the calculations can be organized so that data can be kept in cache and efficiently reused. One of the most commonly used techniques for that purpose is called blocking and an example of this is reported in the following section. Blocking enhances spatial and temporal locality in computations. Unfortunately, blocking is not always sufficient since the cache miss ratio can be dramatically increased in quite an unpredictable way if the memory accesses use a stride greater than 1 (see Bodin and Seznec, 1994).

Some strides are often called *critical* because they generate a very high cache miss ratio (for example, when referencing cache lines that are mapped into the same physical location of the cache). These critical strides obviously depend on the cache management strategy. For example, if the cache line length is equal to four words and the cache is initially empty, the execution of the loop

```
do i=1,n,4
```

```
            temp = temp + a(i)
        enddo
```

will cause a cache miss on each read of a(i), assuming that a(i) is one word.

Copying blocks of data (for example submatrices) that are heavily reused may help to improve memory and cache accesses (by avoiding critical strides for example). However, since such copying may induce a large overhead, it is not always a viable technique. We illustrate the use of copying in our blocked implementation of the BLAS in Section 3. We note that blocking and copying are also very useful in limiting the cost of memory paging.

## 3 The BLAS Computational Kernels

As we have previously discussed, it is very important to use standard building blocks in application codes. They are extremely useful for simplifying the design of codes while guaranteeing portability and efficiency. 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 Basic 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) that include kernels like the matrix-matrix multiply routine _GEMM. Indeed, in Daydé, Duff and Petitet (1994*a*) 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. We consider the performance and the implementation of _GEMM here and show, in Section 3.2, how it can be used to design the other Level 3 BLAS kernels on one example: the symmetric rank-k update _SYRK. The effect of using BLAS in the solution of linear equations, first when the coefficient matrix is full, then when it is sparse will be discussed respectively in Sections 4 and 5.

A tuned manufacturer-supplied version of the BLAS is today available on most high-performance computers and, in cases when it is not provided, a standard Fortran implementation is available on the **netlib** electronic server (Dongarra and Grosse, 1987).

### 3.1 Design of a Fast _GEMM for RISC Processors

We have developed a set of Level 3 BLAS computational kernels in single and double precision for efficient implementation on RISC processors (Daydé and Duff, 1996). This version of the Level 3 BLAS is an evolution of the one described by Daydé et al. (1994*a*) for MIMD vector multiprocessors. They report on experiments on a range of computers (ALLIANT, CONVEX, IBM and CRAY) and demonstrate the efficiency of their approach whenever a tuned version of the matrix-matrix multiplication routine _GEMM is available.

Our basic idea for efficient implementation of the BLAS on RISC processors is to express all the Level 3 BLAS kernels in terms of subkernels on submatrices that involve either _GEMM operations or operations involving triangular submatrices. Additionally, all the calculations on blocks are performed using tuned Fortran codes with loop-unrolling. Copying is occasionally used. Of course, the relative efficiency of this approach depends on the availability of a highly tuned _GEMM kernel.

This approach is relatively independent of the computer: only the block size parameter, here called NB, and in some cases the loop-unrolling depth should be tuned according to the characteristics of the target machine. NB is determined by the size of the cache and the loop-unrolling depth from the number of scalar registers.

We describe only the blocked implementation of _GEMM in this subsection. The other kernels are designed in the same way and we consider, as an example, _SYRK in Section 3.2. Further details can be found in Daydé and Duff (1996).

_GEMM performs one of the matrix-matrix operations

$$\mathbf{C}{=}\alpha \text{ op}(\mathbf{A}) \text{ op}(\mathbf{B}){+}\beta\mathbf{C},$$

where $\alpha$ and $\beta$ are scalars, $\mathbf{A}$ and $\mathbf{B}$ are rectangular matrices of dimensions m$\times$k and k$\times$n, respectively, $\mathbf{C}$ is a m $\times$ n matrix, and op($\mathbf{A}$) is $\mathbf{A}$ or $\mathbf{A}^t$.

We consider the case corresponding to op equal to "No transpose" in both cases and block the computation as:

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,2} & C_{2,2} \end{pmatrix} = \alpha \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} + \beta \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

_GEMM can then obviously be organized in terms of a succession of matrix-matrix multiplications on submatrices as follows:

1. $C_{1,1} \leftarrow \beta C_{1,1} + \alpha A_{1,1} B_{1,1}$ (_GEMM)
2. $C_{1,1} \leftarrow C_{1,1} + \alpha A_{1,2} B_{2,1}$ (_GEMM)
3. $C_{1,2} \leftarrow \beta C_{1,2} + \alpha A_{1,1} B_{1,2}$ (_GEMM)
4. $C_{1,2} \leftarrow C_{1,2} + \alpha A_{1,2} B_{2,2}$ (_GEMM)
5. $C_{2,1} \leftarrow \beta C_{2,1} + \alpha A_{2,1} B_{1,1}$ (_GEMM)
6. $C_{2,1} \leftarrow C_{2,1} + \alpha A_{2,2} B_{2,1}$ (_GEMM)
7. $C_{2,2} \leftarrow \beta C_{2,2} + \alpha A_{2,1} B_{1,2}$ (_GEMM)
8. $C_{2,2} \leftarrow C_{2,2} + \alpha A_{2,2} B_{2,2}$ (_GEMM)

The ordering of these eight computational steps is determined by consideration of the efficient reuse of data held in cache. We have decided to reuse the submatrices of **A** as much as possible and we perform all operations involving a submatrix of **A** before moving to another one (see Figure 3.1). For our simple example, it means that we perform the calculations in the order: Step 1, Step 3, Step 5, Step 7, Step 2, Step 4, Step 6, and Step 8. This approach is similar to that used by Dongarra, Mayes and Radicati di Brozolo (1991*b*). In practice, NB is usually chosen so that all the submatrices of **A**, **B**, and **C** required for each submultiplication fit in the largest on-chip cache. On some machines, access to off-chip caches has so low latency that we can improve performance by using a larger block size. This is true, for example, on the SGI Power Challenge. Since all the computational kernels call _GEMM, the block size NB is always determined as the most appropriate block size for _GEMM, that is, NB is the largest even integer such that

$$3(NB)^2 prec < CS,$$

where *prec* is the number of bytes corresponding the precision used (4 bytes for single precision and 8 bytes for double precision in IEEE format) and $CS$ is the cache size in bytes. We choose an even integer to facilitate loop-unrolling. For example with a 64Kbytes cache, NB is set to 52 using 64-bit arithmetic.

Part of the double precision blocked code is shown in Figure 3.1. Its main features are the following:

- The multiplication of **C** by $\beta$ is performed before all other calculations.
- The submatrix of **A** is multiplied by $\alpha$ and transposed into array $AA$ to avoid non-unit strides because of access by rows in the innermost loops of the calculations. These are organized in such a way that $AA$ is kept in cache as long as required.

We use two tuned Fortran codes to perform calculations on submatrices (see Figure 3.2): DGEMML2X2 is a tuned code for performing matrix-matrix multiplication on square matrices of even order; and DGEMML is a tuned code that includes additional tests over DGEMML2X2 to handle matrices with odd order. It is occasionally slightly less efficient than DGEMML2X2.

We have used two versions for all the tuned codes: the TRIADIC option for computers where triadic operations are either supported in the hardware (for example the floating-point multiply-and-add on IBM RS/6000) or are efficiently compiled, and the NOTRIADIC option for other computers. The use of triadic operations should not normally degrade the performance severely on processors that do not support these operations since efficient code generation can transform them into dyadic operations. However, in early versions of SPARC compilers, we saw that there was sometimes such a degradation. Thus

```
*
*     Form C := beta*C
*
      IF( BETA.EQ.ZERO )THEN
          DO 20  J = 1, N
             DO 10  I = 1, M
                C( I, J ) = ZERO
   10        CONTINUE
   20     CONTINUE
      ELSE
           DO 40  J = 1, N
              DO 30  I = 1, M
                 C( I, J ) = BETA*C( I, J )
   30         CONTINUE
   40      CONTINUE
      END IF
*
*     Form  C := alpha*A*B + beta*C.
*
      DO 70  L = 1, K, NB
         LB = MIN( K - L + 1, NB )
         DO 60  I = 1, M, NB
            IB = MIN( M - I + 1, NB )
            DO II = I, I + IB - 1
               DO LL = L, L + LB - 1
                  AA(LL-L+1,II-I+1)=ALPHA*A(II,LL)
               ENDDO
            ENDDO
            DO 50  J = 1, N, NB
               JB = MIN( N - J + 1, NB )
*
*  Perform multiplication on submatrices
*
               IF ((MOD(IB,2).EQ.0).AND.(MOD(JB,2).EQ.0)) THEN
                  CALL DGEMML2X2(IB,JB,LB,AA,NB,B(L,J),LDB,C(I,J),LDC)
               ELSE
                  CALL DGEMML(IB,JB,LB,AA,NB,B(L,J),LDB,C(I,J),LDC)
               END IF
 50         CONTINUE
 60      CONTINUE
 70   CONTINUE
```

Figure 3.1: Part of the blocked code for DGEMM

```
*
*          C := alpha*A*B + C.
*
           DO 70  J = 1, N, 2
              DO 60  I = 1, M, 2
                 T11 = C(I,J)
                 T21 = C(I+1,J)
                 T12 = C(I,J+1)
                 T22 = C(I+1,J+1)
                 DO 50  L = 1, K
                    B1 = B(L,J)
                    B2 = B(L,J+1)
                    A1 = A(L,I)
                    A2 = A(L,I+1)
                    T11 = T11 + B1*A1
                    T21 = T21 + B1*A2
                    T12 = T12 + B2*A1
                    T22 = T22 + B2*A2
   50            CONTINUE
                 C(I,J) = T11
                 C(I+1,J) = T21
                 C(I,J+1) = T12
                 C(I+1,J+1) = T22
   60         CONTINUE
   70      CONTINUE
```

Figure 3.2: Part of the tuned code for DGEMML2X2 (TRIADIC option)

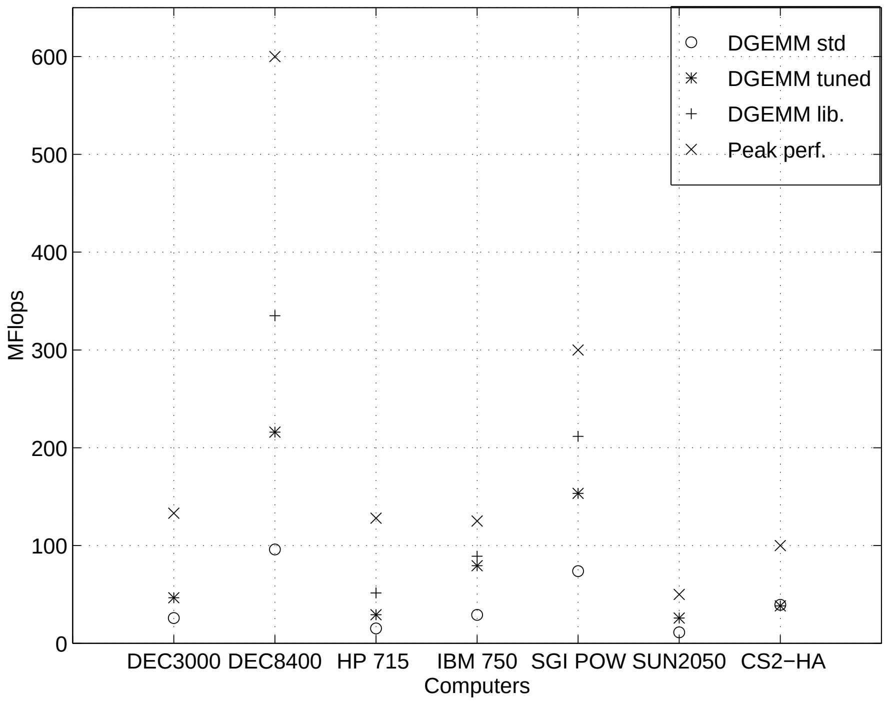

Figure 3.3: Average performance of DGEMM from RISC BLAS ("No transpose", "No transpose").

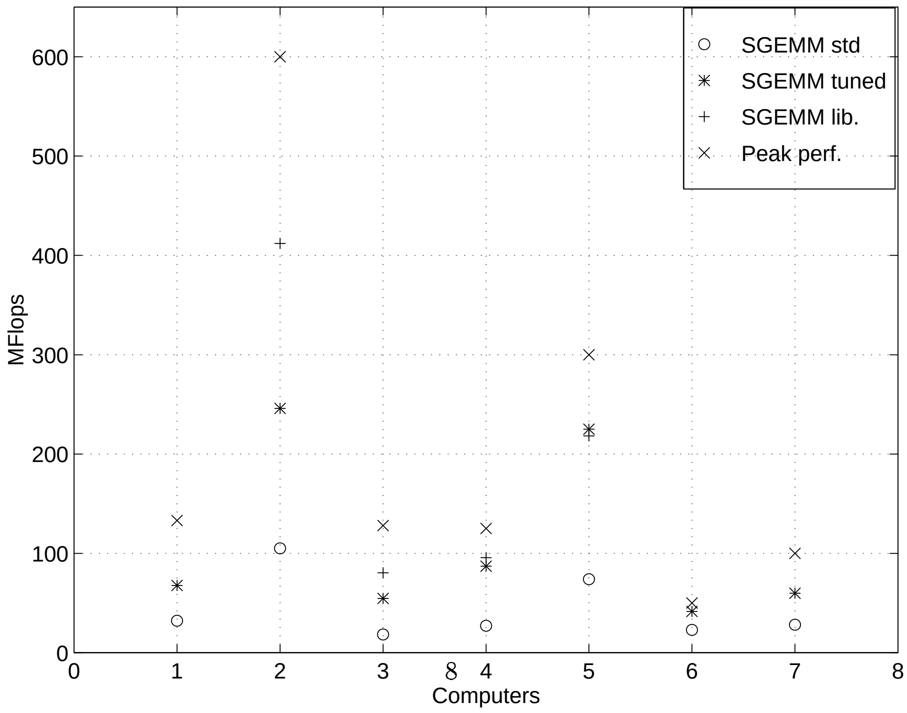

Figure 3.4: Average performance of SGEMM from RISC BLAS ("No transpose", "No transpose").

we prefer to offer both options. Part of the tuned code for DGEMML2X2 using the TRIADIC options is shown in Figure 3.2.

In Figures 3.3 and 3.4, we illustrate respectively the double and single precision average performance of the standard and the blocked versions of _GEMM (averaged over square matrices of order 32, 64, 96, and 128 when **A** and **B** are not transposed). We also include the peak performance of the computer and the performance of the manufacturer-supplied version when available to us. This tuned subset of the BLAS – called the RISC BLAS – is publically available via anonymous ftp at ftp.enseeiht.fr in pub/numerique/BLAS/RISC.

This blocked implementation of _GEMM 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).

## 3.2 Use of _GEMM in Designing the Kernel _SYRK

We now consider the use of _GEMM in designing other kernels as in Daydé et al. (1994*a*). We use the kernel _SYRK to illustrate this.

_SYRK performs one of the symmetric rank-k operations:

$$\mathbf{C}{=}\alpha\mathbf{A}\mathbf{A}^{\mathbf{t}}{+}\beta\mathbf{C}, \text{ or } \mathbf{C}{=}\alpha\mathbf{A}^{\mathbf{t}}\mathbf{A}{+}\beta\mathbf{C}$$

where $\alpha$ and $\beta$ are scalars, **C** is an n $\times$ n symmetric matrix (only the upper or lower triangular parts are updated), and **A** is a n $\times$ k matrix in the first case and a k $\times$ n matrix in the second case.

We consider the case corresponding to $\mathbf{C}{=}\alpha\mathbf{A}\mathbf{A}^{\mathbf{t}}{+}\beta\mathbf{C}$ where only upper triangular part of **C** is updated (that is, "Upper", and "No transpose"), and we block the computation as

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ 0 & C_{2,2} \end{pmatrix} = \alpha \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} A^t_{1,1} & A^t_{2,1} \\ A^t_{1,2} & A^t_{2,2} \end{pmatrix} + \beta \begin{pmatrix} C_{1,1} & C_{1,2} \\ 0 & C_{2,2} \end{pmatrix}.$$

1. $C_{1,1} \leftarrow \beta C_{1,1} + \alpha A_{1,1} A^t_{1,1}$ (_SYRK)
2. $C_{1,1} \leftarrow C_{1,1} + \alpha A_{1,2} A^t_{1,2}$ (_SYRK)
3. $C_{1,2} \leftarrow \beta C_{1,2} + \alpha A_{1,1} A^t_{2,1}$ (_GEMM)
4. $C_{1,2} \leftarrow C_{1,2} + \alpha A_{1,2} A^t_{2,2}$ (_GEMM)
5. $C_{2,2} \leftarrow \beta C_{2,2} + \alpha A_{2,1} A^t_{2,1}$ (_SYRK)
6. $C_{2,2} \leftarrow C_{2,2} + \alpha A_{2,2} A^t_{2,2}$ (_SYRK)

The symmetric rank-k update is expressed as a sequence of _SYRK for updating the submatrices $C_{i,i}$ and _GEMM for the other blocks. The updates of the submatrices of **C** can be performed independently. The _GEMM updates of off-diagonal blocks can be combined. We note that, at the price of extra operations, we could perform the update of the diagonal blocks of **C** using _GEMM instead of _SYRK.

Part of the corresponding blocked code is shown in Figure 3.5. We note that it is more efficient to perform the multiplication of matrix $C$ by $\beta$ before calling _GEMM rather than performing this multiplication within _GEMM. The codes for DSYRKL2X2 and DSYRKL are designed using loop unrolling and copying.

In Figures 3.6 and 3.7, we illustrate respectively the double and single precision average performance of the standard and the blocked versions of _SYRK (averaged over square matrices of order 32, 64, 96, and 128 for the case "Upper" and "No transpose"). We also include the peak performance of the computer and the performance of the manufacturer-supplied version when available to us.

For this kernel, our gains over using standard BLAS are significant, usually by a factor of close to two. We consistently outperform the vendor code on the SGI by a significant amount in single precision. Our blocked code is substantially better than the vendor kernel on the DEC 8400 and would be even faster if we used the vendor-supplied _GEMM.

## 3.3 Parallel Versions of the BLAS

The increased granularity of higher Level BLAS also allows more efficient parallelization because of a reduction in the synchronization overheads (Daydé et al., 1994*a* and Amestoy, Daydé, Duff and Morère, 1995). A parallel version of the Level 3 BLAS is easily obtained using the loop-level parallelism available on most of the shared and virtual shared memory multiprocessors.

We have developed parallel versions of the Level 3 BLAS for two virtual shared memory computers: the BBN TC2000 and the KSR1. On matrices of order 1536, our SGEMM code executes at 150 Mflops on 24 processors on the BBN TC2000, and it is possible to obtain 1320 Mflops with 72 processors on the KSR1 for matrices of order 768 (Amestoy et al., 1995). The RISC BLAS are used as the tuned serial codes.

Two additional libraries have been designed in order to make a parallel BLAS available on message-passing systems: the BLACS (Basic Linear Algebra Communication Subprograms, see Dongarra and Whaley, 1995) that are used as a communication layer (on top of message passing libraries such as PVM, NX, MPI, CMMD,..), and the PBLAS (Parallel Basic Linear Algebra Subprograms, see Choi, Dongarra, Ostrouchov, Petitet, Walker and Whaley, 1995*b*).

We indicate, in Figure 3.8, the performance of the parallel matrix-matrix product from PBLAS (we consider both the single and double precision kernels, that is PSGEMM and PDGEMM respectively) on the MEIKO CS2-HA installed

```
            DO 130, I = 1, N,NB
               NB_LIG_C=MIN(NB,N-I+1)
*
*       Multiplication of diagonal block of C
*

                IF (BETA.EQ.ZERO) THEN
                   DO J = 1, NB_LIG_C
                      DO II = 1, J
                         C(II+I-1,J+I-1) = ZERO
                      ENDDO
                   ENDDO
                ELSE
                   DO J = 1, NB_LIG_C
                      DO II = 1, J
                         C(II+I-1,J+I-1) = BETA*C(II+I-1,J+I-1)
                      ENDDO
                   ENDDO
                END IF

                DO 90, L=1,K,NB

                NB_COL_A=MIN(NB,K-L+1)

              IF ((MOD(NB_LIG_C,2).EQ.0).AND.(MOD(NB_COL_A,2).EQ.0))THEN
                        CALL DSYRKL2X2(NB_LIG_C,NB_COL_A,ALPHA,
     $                                A(I,L),LDA,ONE,C(I,I),LDC)
                ELSE
                        CALL DSYRKL(NB_LIG_C,NB_COL_A,ALPHA,
     $                                A(I,L),LDA,ONE,C(I,I),LDC)
                END IF

90              CONTINUE

                NB_COL_C=N-NB_LIG_C-I+1
                NB_COL_A=K

                 CALL DGEMM('N','T',NB_LIG_C,NB_COL_C,NB_COL_A,
     $                      ALPHA,A(I,1),LDA,A(I+NB_LIG_C,1),LDA,
     $                      BETA, C(I,I+NB_LIG_C),LDC)

130        CONTINUE
```

Figure 3.5: Part of the blocked code for DSYRK

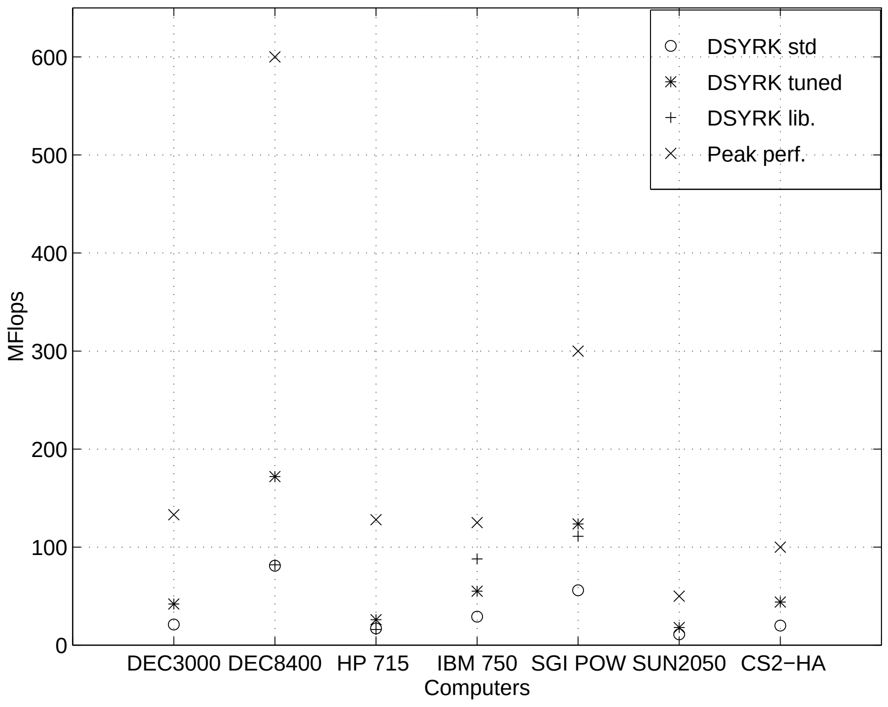

Figure 3.6: Average performance of DSYRK from RISC BLAS ("Upper", "No transpose").

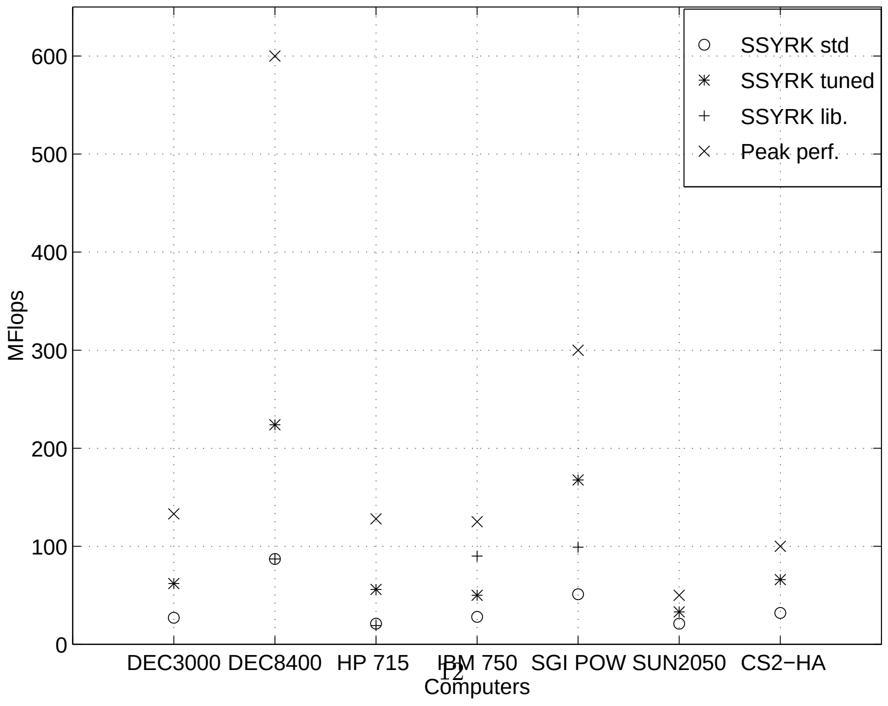

Figure 3.7: Average performance of SSYRK from RISC BLAS ("Upper", "No transpose").

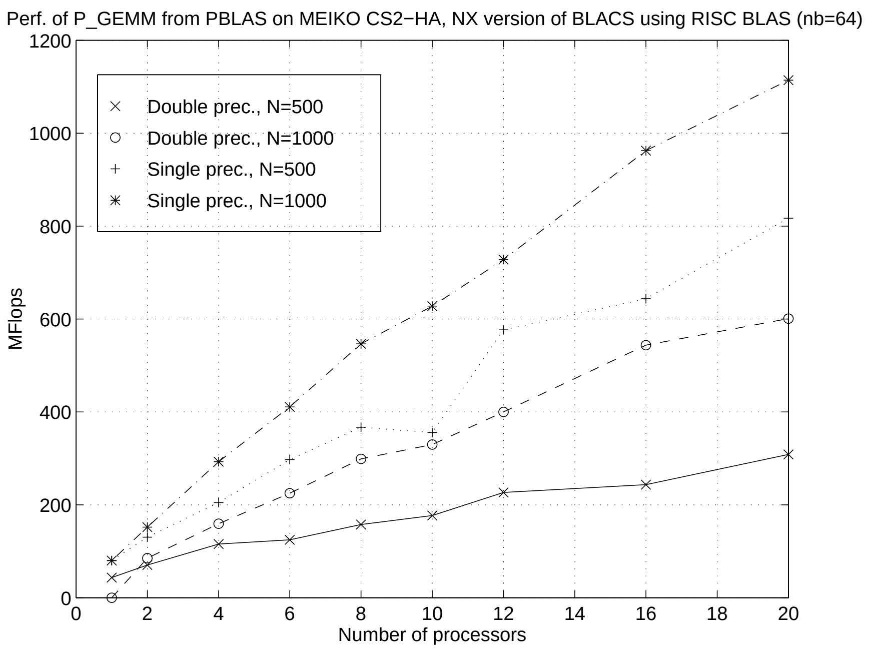

Figure 3.8: Performance of parallel matrix-matrix product from PBLAS on MEIKO CS2-HA

at CERFACS, using square matrices of order 500 and 1000. The processor is a 100 Mhz HyperSPARC of peak performance equal to 100 MFlops. We use the version of BLACS on top of the NX message passing library. The tuned serial BLAS used is the RISC BLAS.

# 4 Solution of Full Linear Systems

## 4.1 Introduction

The BLAS can be used successfully in designing codes for the solution of the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}. \tag{4.1}$$

when $\mathbf{A}$ is full.

The LAPACK library (Anderson, Bai, Bischof, Demmel, Dongarra, DuCroz, Greenbaum, Hammarling, McKenney, Ostrouchov and Sorensen, 1992) uses block algorithms as much as possible to take advantage of the higher level of BLAS. LAPACK contains subroutines to solve systems of linear equations, linear least-squares problems, eigenvalue problems, and singular value problems. It is designed to give high efficiency on vector processors, RISC-based computers, and shared memory multiprocessors.

## 4.2 Use of Parallel BLAS

We have used parallel versions of the BLAS mentioned in Section 3.3 to exploit, in a transparent manner, parallelism in codes from the LAPACK Library on shared and virtual shared memory computers (see Daydé and Duff, 1991 and Amestoy et al., 1995).

The ScaLAPACK library (Choi, Demmel, Dhillon, Dongarra, Ostrouchov, Petitet, Stanley, Walker and Whaley, 1995*a*) is an extension of LAPACK for distributed memory computers. It illustrates the importance of building blocks for reusing existing software as much as possible in the development of portable and efficient codes. ScaLAPACK is based on the BLAS, LAPACK, PBLAS and BLACS libraries. We show, in Figure 4.1, the performance of the LU, Cholesky and QR factorizations from ScaLAPACK version 1.0 using both single and double precision on relatively small matrices of order 1500 on the MEIKO CS2-HA at CERFACS.

## 4.3 Blocked Eskow-Schnabel Modified Cholesky Factorization

The modified Cholesky factorization modifies an indefinite matrix to obtain a Cholesky factorization of a nearby positive definite matrix. It is an important

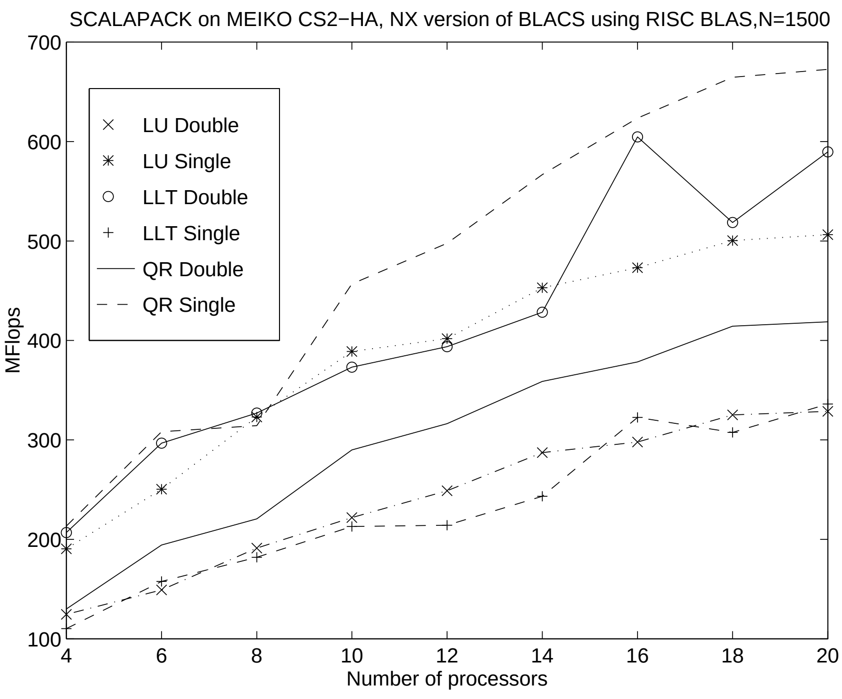

Figure 4.1: Performance of full matrix factorizations from ScaLAPACK on MEIKO CS2-HA

computational kernel in optimization. It was introduced by Gill and Murray (1974) and improved by Gill, Murray and Wright (1981) and more recently by Eskow and Schnabel (1991*b*, 1991*a*). It is particularly useful in optimization algorithms –both in unconstrained and constrained optimization (Gill et al., 1981)– to generate a descent direction when the Hessian matrix is not sufficiently positive-definite. It is also used in some trust region techniques (see Dennis and Schnabel, 1983, Schnabel, Koontz and Weiss, 1985), in the LANCELOT package (Conn, Gould and Toint, 1992), and in sparse preconditioners (Schlick, 1993). It has also been used to build Element-by-Element preconditioners for large scale optimization applications (Daydé, L'Excellent and Gould, 1994*b*, 1995).

We refer here to the Eskow-Schnabel Modified Cholesky Factorization. Let $\mathbf{A}$ be a symmetric $n$-by-$n$, not necessarily positive-definite matrix, then the Eskow-Schnabel factorization computes:

$$\mathbf{P}^T(\mathbf{A}+\mathbf{E})\mathbf{P} = \mathbf{L}\mathbf{L}^T$$

where $\mathbf{P}$ is a permutation matrix and $\mathbf{E}$ is the $n$-by-$n$ zero matrix if $\mathbf{A}$ is sufficiently positive definite. Otherwise $\mathbf{E}$ is a non-negative diagonal matrix chosen so that $\mathbf{A}+\mathbf{E}$ is sufficiently positive-definite. There is no need to know *a priori* if $\mathbf{A}$ is positive-definite and the matrix $\mathbf{E}$ is determined during the factorization process. The Eskow-Schnabel Modified Cholesky Factorization exhibits better properties in terms of computational costs and an *a priori* upper bound on $\|\mathbf{E}\|_\infty$ than those described by Gill and Murray (1974) and Gill et al. (1981).

### 4.3.1 The Eskow-Schnabel Modified Cholesky Factorization.

The Eskow-Schnabel Modified Cholesky factorization is based on a simple idea, namely that once an approximation of the most negative eigenvalue of $\mathbf{A}$ is determined (using the Gerschgorin circle theorem bounds), it is easy to compute $\mathbf{E}$ as a diagonal matrix that shifts the most negative eigenvalue towards a sufficiently positive value. Of course, it is desirable that the eigenvalues be shifted towards the positive values by a quantity that is not substantially greater than the most negative eigenvalue of $\mathbf{A}$.

The algorithm for the Modified Cholesky Factorization is organized in two phases. The first phase corresponds to a standard Cholesky factorization. The algorithm switches to a second phase to perform a modified factorization when the matrix is detected to be not sufficiently positive-definite. Its main characteristic is that diagonal pivoting is used based on the maximum diagonal element. When the matrix is found to be not positive-definite (some diagonal element becomes non-positive), an approximation of the most negative eigenvalue of $\mathbf{A}$ is computed using the Gerschgorin Circle Theorem bounds in order to determine the amount to add to the diagonal of $\mathbf{A}$. It is exactly the Cholesky factorization if $\mathbf{A}$ is sufficiently positive-definite, and in that case the number of flops of the modified factorization is the same as standard Cholesky

($\frac{n^3}{3}$ flops, plus order of $n$ operations to precalculate the new diagonal entries). When $\mathbf{A}$ is not sufficiently positive-definite, the cost of this factorization is at most $2n^2$ additions and $\frac{n^2}{2}$ multiplications greater than the standard Cholesky factorization.

The blocked version of the modified Cholesky factorization performs exactly the same factorization as the original non-blocked one. It is designed in the same way as other block algorithms (Anderson et al., 1992, Daydé and Duff, 1989, Daydé and Duff, 1991, and Dongarra, Duff, Sorensen and van der Vorst, 1991*a*). It is organized using block columns of the matrix. At the $k$ th step, the block column $k$ is factorized using Level 1 and Level 2 BLAS with an unblocked algorithm derived from the standard Eskow-Schnabel factorization. The subsequent updates are effected using Level 3 BLAS. Of course the diagonal pivoting and update of Gerschgorin bounds complicate things since this implies that the diagonal values and the bound estimates must be updated after each column factorization. Also, the switch from Phase 1 to Phase 2 may imply stopping the factorization of a block column to perform all updates resulting from the last steps in Phase 1 before moving to Phase 2. More details can be found in Daydé (1996).

### 4.3.2 Numerical Experiments.

We report on experiments on 3 types of full matrices built using the **genmat** procedure available in the software from Eskow and Schnabel. This generates a random matrix that has eigenvalues in a range of prescribed values: matrix of type 1 has eigenvalues in the range -1. to 100, matrix of type 2 has eigenvalues in the range -10000. to -1, and matrix of type 3 has eigenvalues in the range 1. to 10.

We show, in Table 4.1, the performance of the Level 1 and Level 2 BLAS version of the Modified Cholesky factorization (called **dsymd2**) and the performance of the block version (called **dsymdf**) on two RISC workstations: the HP 715/64 and the IBM RS/6000-750. We give the performance achieved using different block sizes (16, 32, and 64) and for matrices of order 100 and 500. We also give the peak performance in MFlops of each computer in parentheses. We use the manufacturer-supplied BLAS on these computers. We observe that the blocked factorization is at least twice as fast as the unblocked one. The codes described here are available using ftp anonymous to ftp.enseeiht.fr in directory pub/numerique/MODCHOL.

| Computer | Order | Type | **dsymd2** | **dsymdf** | | |
|---|---|---|---|---|---|---|
| | | | | 16 | 32 | 64 |
| HP 715/64 | 100 | 1 | 8.3 | 16.7 | 16.7 | 33.3 |
| | | 2 | 8.3 | 11.1 | 16.7 | 33.3 |
| (128 MFlops) | | 3 | 8.3 | 16.7 | 16.7 | 16.7 |
| | 500 | 1 | 4.2 | 17.7 | 18.4 | 14.9 |
| | | 2 | 4.2 | 17.6 | 18.3 | 14.8 |
| | | 3 | 4.2 | 17.8 | 18.5 | 14.9 |
| IBM RS/6000-750 | 100 | 1 | 16.7 | 33.3 | 16.7 | 33.3 |
| | | 2 | 16.7 | 33.3 | 33.3 | 33.3 |
| (125 MFlops) | | 3 | 16.7 | 33.3 | 33.3 | 16.7 |
| | 500 | 1 | 19.3 | 46.8 | 43.9 | 46.8 |
| | | 2 | 20.0 | 44.8 | 45.8 | 41.3 |
| | | 3 | 19.2 | 47.3 | 48.4 | 43.8 |

Table 4.1: Performance in MFlops of the double precision Modified Cholesky Factorization.

## 5 Use of Level 3 BLAS in the Direct Solution of Sparse Linear Systems

We refer to the studies of Amestoy and Duff (Amestoy and Duff, 1989,Amestoy, 1991,Amestoy and Duff, 1993,Amestoy et al., 1995) on the sparse LU factorization of square matrices on shared memory multiprocessors. They use a multifrontal approach for the factorization. Further background on this approach can be obtained from the original papers by Duff and Reid (1983, 1984).

In a multifrontal method, the sparse factorization proceeds by a sequence of factorizations on small full matrices, called frontal matrices. The ordering for the sequence of computations and the frontal matrices are determined by a computational tree, called assembly tree, where each node represents a full matrix factorization and each edge the transfer of data from child to parent node. This assembly tree is determined from the sparsity pattern of the matrix and from a reordering that reduces the fill-in during the numerical factorization (such as the minimum degree ordering that we use here). During the numerical factorization, eliminations at any node can proceed as soon as those at the child nodes have completed and the resulting contributions from the children have been summed (assembled) with data at the parent node. This is the only synchronization that is required and means that operations at nodes that are not ancestors or dependants are completely independent. The parallelism resulting from this observation will be referred to as tree parallelism.

We note that the factorization at each node is done using full linear algebra and direct addressing so that factorization computations within a node can use the BLAS. All the indirect addressing is confined to the assembly process.

Amestoy and Duff have developed a parallel multifrontal code for the solution of symmetrically structured unsymmetric equations (Amestoy and Duff 1989, Amestoy 1991, Amestoy and Duff 1993, Amestoy et al. 1995). Recently, a library version of the experimental code called MUPS, has been included in Release 12 of the Harwell Subroutine Library (HSL, 1996). This HSL code is called `MA41`, and the results in this section are from runs with this code.

During the LU factorization, if only the tree parallelism is exploited, 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 as the computation proceeds 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 assembly tree. It is thus necessary to obtain further parallelism within the large nodes near the root of the tree by using parallel versions of the BLAS in the factorizations within the nodes. Amestoy and Duff call this node parallelism. When both tree and node parallelism are combined the situation becomes much more encouraging.

In Table 5.1, we show typical performance of `MA41` for a range of RISC-based computers. The RISC BLAS is used on the DEC and the MEIKO and the manufacturer-supplied one in the other cases. A medium size sparse matrix, BCSSTK15 from the Harwell-Boeing set (Duff, Grimes and Lewis, 1992), is used in this table. This is a matrix of order 3948 with 117816 nonzeros from a structural analysis application. A minimum degree ordering is used and the number of floating-point operations for the factorization is 443 million. The performance achieved by `MA41` is usually more than 60% of that of DGEMM.

|  | Peak perf. | DGEMM | `MA41` |
|---|---|---|---|
| Computer | Mflops | Mflops | Mflops |
| DEC 3000/400-AXP | 133 | 49 | 34 |
| HP 715/64 | 128 | 55 | 30 |
| IBM RS6000/750 | 125 | 101 | 64 |
| IBM SP2 (thin node) | 266 | 213 | 122 |
| MEIKO CS2-HA | 100 | 43 | 31 |

Table 5.1: Performance summary of the multifrontal LU factorization `MA41` on matrix BCSSTK15 on a range of RISC processors. The average performance of DGEMM is also shown (averaged over square matrices of order 32, 64, 96, and 128).

Although we have concentrated in this section on the use of higher level BLAS in a multifrontal method designed for symmetrically structured sparse matrices, the use of such computational kernels in sparse direct codes is now very widespread and is arguably the main contributor to the efficiency of modern codes for the direct solution of sparse equations. They have also been used in, for example, the multifrontal factorization of unsymmetric matrices (Davis and Duff 1993) and in supernodal codes for unsymmetric problems (Demmel, Eisenstat, Gilbert, Li and Liu 1995). A fuller discussion on the use of high level computational kernels in sparse direct methods can be found in Duff (1996).

## 6 Solution of Partially Separable Linear Systems Using Element-by-Element Preconditioners

### 6.1 Introduction

We study the solution of large unassembled partially separable systems by methods that aim to exploit their inherent structure. In particular, we consider the linear systems that arise from the minimization of the partially separable (Griewank and Toint, 1982) objective function

$$f(\mathbf{x}) = \sum_{i=1}^{p} f_i(\mathbf{x}^i), \tag{6.1}$$

where each set of *local* variables, $\mathbf{x}^i \in \Re^{n_i}$, is a subset of the *global* variables, $\mathbf{x} \in \Re^n$, and $n_i \ll n$.

In unconstrained optimization, we often require an (approximate) solution, $\mathbf{d}$, to the Newton equations

$$\nabla_{xx} f(\mathbf{x})\mathbf{d} = -\nabla_x f(\mathbf{x}) \tag{6.2}$$

When $f$ has the form (6.1), equation (6.2) can be expressed as

$$\left( \sum_{i=1}^{p} \nabla_{xx} f_i(\mathbf{x}^i) \right) \mathbf{d} = -\sum_{i=1}^{p} \nabla_x f_i(\mathbf{x}^i). \tag{6.3}$$

The Hessian matrix of each $f_i$ is a low-rank, sparse matrix. The overall Hessian is thus frequently also sparse. Putting this in a more general context, we consider the solution of structured systems of linear equations of the form

$$\mathbf{Ax} = \mathbf{b}, \tag{6.4}$$

where

$$\mathbf{A} = \sum_{i=1}^{p} \mathbf{A}_i. \tag{6.5}$$

and $\mathbf{A}$ is large and normally positive definite. Similar linear systems arise when solving constrained optimization problems using augmented Lagrangian methods, and, of course, when using finite-element methods to solve elliptic partial differential equations.

Although direct methods may be appropriate for solving (6.4), we consider here the use of the conjugate gradient method combined with Element-by-Element preconditioners. Additionally, we show that amalgamating elements before constructing such a preconditioner can dramatically improve the speed and numerical behaviour of the method.

## 6.2 Element-by-Element Preconditioners

Element-By-Element (EBE) preconditioners were introduced by Hughes, Levit and Winget (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). These preconditioners have some nice features. They can be computed element-wise and most of them do not require assembly. Furthermore, they permit efficient parallelization.

We describe here only the EBE preconditioner since, in our experience, this is one of the most promising. Further details can be found in Daydé et al. (1994*b*).

Assuming that $\mathbf{A}$ is positive definite, we may rewrite $\mathbf{A}$ as:

$$\mathbf{A} = \sum_{i=1}^{p} \mathbf{M}_i + \sum_{i=1}^{p} (\mathbf{A}_i - \mathbf{M}_i) = \mathbf{M} + \sum_{i=1}^{p} (\mathbf{A}_i - \mathbf{M}_i), \tag{6.6}$$

where $\mathbf{M}_i = diag(\mathbf{A}_i)$ and $\mathbf{M} = \sum_{i=1}^{p} \mathbf{M}_i$. Now, let $\mathbf{M} = \mathbf{L}_M \mathbf{L}_M^T$ be the Cholesky factorization of $\mathbf{M}$ ($\mathbf{L}_M$ is simply a diagonal matrix). Then,

$$\mathbf{A} = \mathbf{L}_M \left( \mathbf{I} + \sum_{i=1}^{p} \mathbf{L}_M^{-1} (\mathbf{A}_i - \mathbf{M}_i) \mathbf{L}_M^{-T} \right) \mathbf{L}_M^T = \mathbf{L}_M \left( \mathbf{I} + \sum_{i=1}^{p} \mathbf{E}_i \right) \mathbf{L}_M^T, \tag{6.7}$$

where $\mathbf{E}_i = \mathbf{L}_M^{-1} (\mathbf{A}_i - \mathbf{M}_i) \mathbf{L}_M^{-T}$. Using the approximation $\mathbf{I} + \sum_{i=1}^{p} \mathbf{E}_i \approx \prod_{i=1}^{p} (\mathbf{I} + \mathbf{E}_i)$, we obtain:

$$\mathbf{A} \approx \mathbf{L}_M \prod_{i=1}^{p} (\mathbf{I} + \mathbf{E}_i) \mathbf{L}_M^T. \tag{6.8}$$

A further approximation gives the EBE preconditioner

$$P_{EBE} = \mathbf{L}_M \left( \prod_{i=1}^{p} \mathbf{L}_i \right) \left( \prod_{i=1}^{p} \mathbf{D}_i \right) \left( \prod_{i=p}^{1} \mathbf{L}_i^T \right) \mathbf{L}_M^T, \tag{6.9}$$

where the $\mathbf{L}_i$ and $\mathbf{D}_i$ factors come from the $\mathbf{LDL}^T$ factorization of the matrices $\mathbf{I} + \mathbf{E}_i$ (also known as the Winget decomposition).

Clearly, the efficiency of the EBE preconditioner depends on the the partitioning of the initial matrix and on the magnitude of the off-diagonal elements of the elementary matrices. As the decomposition of $\mathbf{A}$ is, in general, not unique, different decompositions may significantly affect the performance of the preconditioner.

## 6.3 Preprocessing of Unassembled Linear Systems

The effectiveness of the preconditioner depends crucially on the overlap between elements. Daydé et al. (1994*b*) show that amalgamating elements before constructing such a preconditioner can dramatically improve the speed and numerical behaviour of the method. The amalgamation process typically reduces the overlap between elements, and it is this that leads to improvements in performance.

Our preprocessing step consists of grouping the elements into sets, assembling the elements within each set into a *super-element*, and then applying an Element-by-Element technique to the super-elements instead of the original $\mathbf{A}_i$. Although, as we have already said, we focus on the use of the EBE preconditioner, most of the conclusions are true for the other ones.

### 6.3.1 Amalgamation Algorithm.

A variety of amalgamation techniques are considered in detail by Daydé et al. (1994*b*). Here, we only consider the most successful of these.

Let $\mathcal{G}_i$ denote an element, $\mathcal{V}_i$ denote the set of indices of variables used by the element $\mathcal{G}_i$, and $|\mathcal{V}_i|$ denote the cardinal of $\mathcal{V}_i$. Let $tim(i)$ refer to the time spent:

- in a matrix-vector product of order $i$ for the diagonal (DIAG) preconditioner (strategy **amalg1** in Table 6.2); or
- in a matrix-vector product and in two triangular solves of order $i$ for the EBE preconditioner (strategy **amalg2** in Table 6.2).

The amalgamation process we have used computes the *benefit*

$$b(\mathcal{G}_i, \mathcal{G}_j) = tim(|\mathcal{V}_i|) + tim(|\mathcal{V}_j|) - tim(|\mathcal{V}_i \cup \mathcal{V}_j|), \qquad (6.10)$$

for all pairs of elements and amalgamates the pair with the largest benefit so long as it is larger than a *threshold* value.

We note that $tim(i)$ only depends on $i$ and can be computed once and for all. These machine-dependent costs are stored in files and determined during the installation of the software. If we are only interested in reducing the time per iteration, the best value for the threshold would be zero, but negative values will result in further amalgamations and, hence, possibly better preconditioners.

## 6.4 Numerical Experiments

We describe, in Table 6.1, a set of test matrices that will be used in our experiments. $n$ is the order of the matrices, $p$ the number of elements, and $\kappa$ the condition number. The matrices come either from the Harwell-Boeing collection, see Duff et al., 1992 (CEGB2802), or are problems in SIF format from the CUTE collection, see Bongartz, Conn, Gould and Toint, 1993 (CBRATU3D, NOBNDTOR, and NET3).

The pattern of CEGB2802 arises from a structural engineering problem, NOBNDTOR is a quadratic elastic torsion problem arising from an obstacle problem on a square, NET3 is a very ill-conditioned example from the optimization of a high-pressure gas network, and CBRATU3D is obtained by discretizing a complex 3D PDE problem in a cubic region.

| Problem name | $n$ | $p$ | Min elt size | Max elt size | Mean elt size | Degree of overlap | $\kappa$ |
|---|---|---|---|---|---|---|---|
| CBRATU3D | 4394 | 4394 | 5 | 8 | 7.5 | 7.5 | $3.4 \times 10^1$ |
| CEGB2802 | 2694 | 108 | 42 | 60 | 58.7 | 2.4 | $5.7 \times 10^4$ |
| NET3 | 512 | 531 | 1 | 6 | 2.6 | 2.7 | $2.4 \times 10^9$ |
| NOBNDTOR | 480 | 562 | 1 | 5 | 4.2 | 4.9 | $1.8 \times 10^2$ |

Table 6.1: Summary of the characteristics of each test problem

One important characteristic is the degree of overlap. It is defined as the average number of elements sharing each variable and is an indicator as to how well Element-by-Element preconditioners will behave.

In practice, we use a Modified Cholesky factorization close to the one described in Section 4.3 to guarantee that the preconditioners are positive definite.

Detailed results of amalgamation for various preconditioners can be found in Daydé et al. (1994*b*, 1995, and 1996). These results indicate that Element-by-Element preconditioners are effective in terms of the numbers of iterations required and the clustering of eigenvalues of the preconditioned Hessian, particularly if the overlap between blocks is small. But, except for ill-conditioned problems, EBE is not significantly more efficient than diagonal preconditioning. One reason is because of the structure of the elements. When there is low overlap, EBE appears much more efficient than diagonal preconditioning. Amalgamating elements may reduce the number of iterations by decreasing the degree of overlap in the new partition.

We show, in Table 6.2, the effect of amalgamating elements for diagonal and EBE preconditioning with a threshold equal to 0 for runs on an HP 715/64. The solution time, $t_{sol}$, includes the time for computing the preconditioner and the time for iterating to convergence. The time for computing the preconditioners (both diagonal and EBE) is negligible on all the problems (less than 0.03 seconds) except when using EBE on CBRATU3D and CGEB2002 where it is around 0.5 and 0.7 seconds respectively.

Large gains in execution time are obtained using amalgamation when the elements are initially small and overlap significantly. With amalgamation, EBE is more efficient than diagonal preconditioning as soon as the problem is sufficiently hard to solve. For diagonal preconditioning, the gains from amalgamation are due to the better execution rates whereas, for EBE, the gains are due both to a better execution rate and a smaller number of iterations.

The amalgamation procedure is currently rather costly, but it is hoped that good heuristics will decrease this preprocessing cost with roughly the same effect.

| Problem | Amalg. | Amalg. time | $p$ | Avg size elt | Diagonal | | | EBE | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | $\#its$ | $t_{sol}$ | MF | $\#its$ | $t_{sol}$ | MF |
| CBRATU3D | **none** | | 4394 | 7.5 | 53 | 3.5 | 9.5 | 20 | 5.1 | 4.6 |
| | **amalg1** | 4.9 | 3400 | 9.1 | 53 | 3.5 | 10.7 | 23 | 5.6 | 5.6 |
| | **amalg2** | 8.0 | 1950 | 14.0 | 53 | 3.8 | 13.0 | 24 | 5.8 | 7.6 |
| CEGB2802 | **none** | | 108 | 58.7 | 661 | 31.3 | 16.7 | 120 | 15.6 | 12.4 |
| | **amalg1** | 0.2 | 108 | 58.7 | 661 | 31.5 | 16.6 | 120 | 16.0 | 12.1 |
| | **amalg2** | 0.2 | 106 | 59.4 | 660 | 31.7 | 16.5 | 111 | 14.6 | 12.3 |
| NET3 | **none** | | 538 | 2.6 | 1559 | 5.2 | 4.9 | 723 | 8.7 | 1.9 |
| | **amalg1** | 0.1 | 143 | 5.6 | 1580 | 3.2 | 8.8 | 268 | 1.5 | 4.9 |
| | **amalg2** | 0.2 | 57 | 11.1 | 1668 | 3.2 | 12.2 | 216 | 1.0 | 8.7 |
| NOBNDTOR | **none** | | 562 | 4.2 | 68 | 0.3 | 7.2 | 30 | 0.5 | 3.0 |
| | **amalg1** | 0.2 | 107 | 11.7 | 68 | 0.2 | 13.4 | 36 | 0.3 | 8.2 |
| | **amalg2** | 0.3 | 57 | 17.3 | 68 | 0.3 | 15.3 | 32 | 0.3 | 9.8 |

Table 6.2: Comparison of DIAG and EBE preconditioners without amalgamation and with amalgamation strategies **amalg1** and **amalg2** on HP 715/64. Amalg. time is the time for performing the symbolic amalgamation plus the numerical assembly of the super-elements. $p$ is the number of elements and MF is the performance expressed in Mflops.

As we may have to solve many systems with the same structure in the course of a nonlinear optimization calculation, a good preprocessing step may pay handsome dividends in the long run.

### 6.4.1 Use of Sparse Storage.

When using such an algorithm, the super-elements are stored as full matrices and thus some zeros may be explicitly stored. The density of the super-elements depends on the structure of the initial elements and, because of the dependence of the amalgamation on $tim(.)$, also on the target computer. For example, on an ALLIANT FX/80 vector computer using a threshold equal to zero, we typically obtain elements with a density around 0.3, which is quite large for the elements to be treated as sparse. On RISC architectures, the density of the elements obtained can be larger, since the amalgamation process is stopped earlier (long vectors are not required for efficiency as on a vector processor). However, we have suggested that continued amalgamation is often beneficial to the quality of the preconditioner, and this results in a reduction in the density of the super-elements. Thus it may well be advantageous to use a sparse representation of the super-elements. We define the average density to be the ratio

$$d = \frac{\sum_{i=1}^{p} nz_i}{\sum_{i=1}^{p} s_i^2}, \tag{6.11}$$

where $p$ is the number of elements, $nz_i$ is the number of nonzero entries in element $i$, and $s_i$ is the number of variables in the element $i$.

In Table 6.3, we show the construction time and the time spent in the solves of the EBE preconditioner using both full and sparse storage of the super-elements for the test problem NET3. As matrix-vector products and dot-products are performed in the same way in both cases — using the full initial elements, the construction time and the time spent in the solves are the more meaningful parameters to compare. When using sparse storage, the factorization of the Winget decomposition of each super-element is obtained using the sparse symmetric solver MA27 (Duff and Reid, 1983) from the Harwell Subroutine Library. We compare two different orderings for the elimination of the variables: the minimum degree ordering and the natural ordering of the elements. Except for very small and full elementary matrices, it appears that minimum degree is always better.

| | | | | Dense storage | | | Sparse storage | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Min. deg. | | Nat. ord. | |
| Thresh. | Nb elt | Avg size | Avg dens. | # its | Cons. time | Time solve | # its | Cons. time | Time solve | Cons. time | Time solve |
| 0.01 | 465 | 2.7 | 1.00 | 684 | 0.01 | 3.58 | 675 | 0.14 | 6.49 | 0.12 | 6.49 |
| 0.0 | 104 | 7.0 | 0.51 | 243 | 0.02 | 0.76 | 231 | 0.07 | 1.29 | 0.06 | 1.29 |
| -0.00004 | 52 | 11.9 | 0.34 | 187 | 0.02 | 0.75 | 183 | 0.07 | 0.87 | 0.05 | 1.02 |
| -0.00012 | 39 | 15.1 | 0.26 | 142 | 0.02 | 0.63 | 138 | 0.06 | 0.63 | 0.06 | 0.79 |
| -0.0002 | 29 | 19.6 | 0.19 | 123 | 0.02 | 0.69 | 122 | 0.06 | 0.54 | 0.07 | 0.76 |
| -0.001 | 19 | 28.7 | 0.11 | 87 | 0.04 | 0.71 | 84 | 0.06 | 0.36 | 0.08 | 0.64 |
| -0.004 | 13 | 40.9 | 0.06 | 57 | 0.07 | 0.87 | 58 | 0.06 | 0.23 | 0.16 | 0.63 |
| -0.02 | 9 | 57.6 | 0.02 | 17 | 0.22 | 0.65 | 16 | 0.06 | 0.08 | 0.77 | 0.41 |
| -1000.0 | 8 | 64.0 | 0.01 | 1 | 0.66 | 0.14 | 1 | 0.07 | 0.02 | 2.35 | 0.06 |

Table 6.3: Results of amalgamation obtained for the problem NET3 using different thresholds on a SPARC-10 workstation using full and sparse kernels in the the factorizations and triangular solves.

The sparse storage becomes more efficient than full storage for threshold values smaller than -0.00012, which corresponds to a density smaller than 0.26. This is especially true for the triangular systems for which there is less overhead using sparse storage than for the factorization.

If we consider a vector computer, such as the ALLIANT FX/80, amalgamation is useful with full super-element storage. But again, there is a significant overhead in using sparse elements since the number of iterations is initially small and not significantly reduced by amalgamation, and because of the large amount of fill-in during the factorizations. Therefore, in this case, we prefer to use long vectors rather than sparse elements.

When solving a well conditioned problem, sparse elements do not appear to be very useful. For an ill-conditioned problem, amalgamation often reduces the number of iterations. Sparse elements can then be effective provided a lot of amalgamation occurs. We believe that the main use of sparse super-elements is in those large scale ill-conditioned problems for which direct factorization gives rise to too much fill-in.

Element-by-Element preconditioners may be extremely effective for sparse structured systems of linear equations that arise in partial differential equations and partially separable nonlinear optimization applications. Furthermore, they seem to offer great possibilities of vectorization/parallelization on multiprocessor architectures. They also allow for the use of efficient computational kernels such as the BLAS and blocked factorizations. We believe that further experimentation is necessary to assess the full potential of the methods. The amalgamation technique can also be applied to other Element-by-Element preconditioners or to block methods.

## 7 The Sparse BLAS

Dodson, Grimes and Lewis (1991) proposed a sparse extension of the BLAS some years ago, but this only considered the Level 1 BLAS for vector-vector operations, such as a sparse SAXPY. In keeping with the theme of this paper, we do not discuss these kernels here but concentrate instead on proposals for sparse extensions to higher level BLAS that offer the promise of providing efficient building blocks in a similar fashion to the full kernels discussed earlier.

It is important to stress at the outset that such kernels are not designed for use within sparse direct codes. Indeed it is firmly our belief that the most efficient way to design sparse direct codes is to remove all indirect addressing from the innermost loop and to use full BLAS for the actual elimination operations as we indicated in Section 5. Instead, we envisage the most common use of the sparse BLAS kernels in the iterative solution of sparse equations, and we illustrate such use in Section 7.2.

### 7.1 Definition of Sparse BLAS

We concentrate here on the **User** Level definition of the higher level sparse BLAS as defined by Duff, Marrone, Radicati and Vittoli (1995). The actual implementation for a particular data structure on a particular architecture would be performed using the lower level toolkit codes of Carney, Heroux and Li (1993).

The proposal of Duff et al. (1995) defines standard interfaces for the following functions:

(1) a routine for performing the product of a sparse and a dense matrix,

(2) a routine for solving a sparse upper or lower triangular system of linear equations for a matrix of right-hand sides,

(3) a routine to check the input data, to transform from one sparse format to another, and to scale a sparse matrix, and

(4) a routine to permute the columns of a sparse matrix and a routine to permute the rows of a full matrix.

We note that (1) and (2) define an extension of the Level 3 BLAS, but they include operations on vectors as a trivial subset. These may be coded separately at the machine dependent level.

The data preprocessing routine (3) is essential to this proposal. It is intended that this routine be called before the body of the computation. The interface is designed to accept many different data formats and produce many others. In particular, it can interrogate the machine it is running on and transform the data into a format that is particularly suited for that machine.

Many algorithms require the permutation of matrices. Additionally, some efficient implementations of sparse matrix-vector products, and of the solution of sparse triangular systems on vector or parallel processors, require the vectors to be reordered. If high efficiency is required, it is necessary to avoid explicit vector permutations in the inner loops and, to enable this, routines have been added (4) to permute sparse matrices and full matrices appropriately. The permutation routines can also be called outside the body of the computation in order to increase efficiency by avoiding permutations within the main loop of the algorithm. This facility is discussed more by Duff et al. (1995).

Although the routines in (3) and (4) are an integral and important part of the proposal, we will here confine ourselves to a further discussion of the routines in (1) and (2).

The main additional issue for the sparse case over the dense one lies in the data format used for the matrices. In the sparse case, there are many different formats which are chosen as natural for the application, for compactness, for clarity, or for efficiency. Indeed, the data structure used may change depending on which criterion is emphasized.

In a Fortran 77 environment, we choose to represent the sparse matrix using no less than six arrays in order to accommodate most commonly used data formats. The principal arrays are the real entries and two integer arrays which may, for example, hold the row and column indices of the respective entries if a coordinate storage scheme were being used. In the following, these arrays are designated by A, IA1, and IA2. A character string, FIDA, indicates the storage scheme being used (for example, FIDA = "COO" for coordinate format) and a character array, DESCRA, gives attributes of the matrix (for example, symmetry, triangularity). Finally, a further short integer array, INFOA, supplies further information concerning the matrix, for example the number of entries in the case of the coordinate scheme. In the Fortran 90 environment, also defined by Duff et al. (1995), all these arrays are included in a derived data type to which is also added left and right permutation arrays, PL and PR, respectively.

This Fortran 90 derived data type is shown in Figure 7.1. The sparse matrix has order M by K.

```
MODULE TYPESP
   TYPE SPMAT
      INTEGER M,K
      CHARACTER*5 FIDA
      CHARACTER*1 DESCRA(10)
      INTEGER     INFOA(10)
      DOUBLE PRECISION,POINTER :: A(:)
      INTEGER,POINTER :: IA1(:),IA2(:),PL(:),PR(:)
   END TYPE SPMAT
END MODULE TYPESP
```

Figure 7.1: Fortran 90 derived data type for sparse matrices

With this definition of a sparse matrix, the routines for the matrix-matrix products in (1), which are defined by

- $C \leftarrow \alpha P_R A P_C B + \beta C$
- $C \leftarrow \alpha P_R A^T P_C B + \beta C$

and for solving triangular systems of equations with multiple right-hand sides in (2), which are defined by

- $C \leftarrow \alpha D P_R T^{-1} P_C B + \beta C$
- $C \leftarrow \alpha D P_R T^{-T} P_C B + \beta C$
- $C \leftarrow \alpha P_R T^{-1} P_C D B + \beta C$
- $C \leftarrow \alpha P_R T^{-T} P_C D B + \beta C$

where

- $A$ is a sparse matrix
- $T$ is a triangular sparse matrix
- $B$ and $C$ are dense matrices
- $D$ is a diagonal matrix
- $P_R$ and $P_C$ are permutation matrices
- $\alpha$ and $\beta$ are scalars,

are as shown in Figures 7.2 and 7.3 respectively.

```
_CSMM (TRANS, M, N, K, ALPHA, PR, FIDA, DESCRA, A, IA1, IA2, INFOA, PC,
       B, LDB, BETA, C, LDC, WORK, LWORK, IERROR)
```

| TRANS = 'N' | TRANS = 'T' |
|---|---|
| $C \leftarrow \alpha P_R\ A\ P_C\ B + \beta C$ | $C \leftarrow \alpha P_R\ A^T\ P_C\ B + \beta C$ |

Figure 7.2: _CSMM, sparse matrix times dense matrix kernel

```
_CSSM(TRANS, M, N, ALPHA, UNITD, D, PR, FIDT, DESCRT, T, IT1, IT2, INFOT,
      PC, B, LDB, BETA, C, LDC, WORK, LWORK, IERROR)
```

| | TRANS = 'N' | TRANS = 'T' |
|---|---|---|
| UNITD = 'U' | $C \leftarrow \alpha\ P_R\ T^{-1}\ P_C\ B + \beta C$ | $C \leftarrow \alpha\ P_R\ T^{-T}\ P_C\ B + \beta C$ |
| UNITD = 'L' | $C \leftarrow \alpha P_R\ D\ T^{-1}\ P_C\ B + \beta C$ | $C \leftarrow \alpha P_R\ D\ T^{-T}\ P_C\ B + \beta C$ |
| UNITD = 'R' | $C \leftarrow \alpha P_R\ T^{-1}\ D\ P_C\ B + \beta C$ | $C \leftarrow \alpha P_R\ T^{-T}\ D\ P_C\ B + \beta C$ |
| UNITD = 'B' | $C \leftarrow \alpha P_R D^{\frac{1}{2}} T^{-1} D^{\frac{1}{2}} P_C B + \beta C$ | $C \leftarrow \alpha P_R D^{\frac{1}{2}} T^{-T} D^{\frac{1}{2}} P_C B + \beta C$ |

Figure 7.3: _CSSM, solution of sparse triangular systems of equations

### 7.2 Use in Iterative Methods

As we mentioned earlier, the primary reason for the design of the sparse high level BLAS is for use in the iterative solution of sparse linear equations. We note that our proposal will fit equally well whether the iterative software performs a call to a matrix-vector multiply routine or whether reverse communication is used since in either case a call can be made to a given sparse matrix-full matrix multiplication routine; the call is made by the routine in the former case and by the user in the latter.

We feel that the best interface for iterative solvers, particularly for flexibility and efficiency is to use reverse communication so that the typical use of our sparse BLAS within an iterative solver would be as indicated in the skeleton code in Figure 7.4.

## 8 Conclusion

The studies described in this paper demonstrate how portable and efficient mathematical software can be designed on high performance computers by making heavy use of computational kernels. The main computational kernels that we consider are the Level 3 BLAS, and we show how they can be used, not only in the solution of full systems of linear equations but also in the direct solution of sparse equations. For the iterative solution of sparse equations, we show how advantage can be taken of full blocks within an Element-by-Element preconditioner and how an extension of the BLAS for sparse matrices can be used in the iterative solution code.

**Acknowledgements**
The authors are very grateful to Patrick Amestoy and Jérome Décamps for their support in getting some of the results displayed in the paper.

```
C  Perform an iteration of the BiConjugate Gradient  method

        IFLAG = 0
      DO 10 ITER=1, MAXIT

C  Call to solve routine
        CALL SOLVER(IFLAG, ......

C  Successful termination
        IF (IFLAG.EQ.1) THEN
          GO TO 20
        END IF

C  Error return
        IF (IFLAG.LT.0) THEN
          GO TO 30
        END IF

        IF (IFLAG.EQ.2) THEN
C  Perform the matrix-vector product
          CALL DCSMM(.....
          GO TO 10
        END IF

        IF (IFLAG.EQ.2) THEN
C  Perform the preconditioning operation
          CALL DCSSM(.....
          GO TO 10
        END IF

  10  CONTINUE

C  Code to handle successful termination
  20  ....

C  Code executed if error returns
  30  ....
```

Figure 7.4: Use of kernels in iterative solution of sparse equations

## References

Amestoy, P. R. (1991), Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment, Phd thesis, Institut National Polytechnique de Toulouse. Available as CERFACS report TH/PA/91/2.

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

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

Amestoy, P. R., Daydé, M. J., Duff, I. S. and Morère, P. (1995), 'Linear algebra calculations on a virtual shared memory computer', *Int Journal of High Speed Computing* **7**, 21–43.

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S. and Sorensen, D. (1992), *LAPACK Users' Guide.*, SIAM.

Bodin, F. and Seznec, A. (1994), Cache organization influence on loop blocking, Technical Report 803, IRISA, Rennes, France.

Bongartz, I., Conn, A. R., Gould, N. I. M. and Toint, P. L. (1993), CUTE: Constrained and Unconstrained Testing Environment, Technical Report TR/PA/93/10, CERFACS, Toulouse, France.

Carney, S., Heroux, M. A. and Li, G. (1993), A proposal for a sparse BLAS toolkit, Technical Report TR/PA/92/90 (Revised), CERFACS, Toulouse, France.

Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D. and Whaley, R. C. (1995*a*), ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance, Technical Report LAPACK Working Note 95, CS-95-283, University of Tennessee.

Choi, J., Dongarra, J., Ostrouchov, S., Petitet, A., Walker, D. and Whaley, R. C. (1995*b*), A proposal for a set of parallel basic linear algebra subprograms, Technical Report LAPACK Working Note 100, CS-95-283, University of Tennessee.

Conn, A. R., Gould, N. I. M. and Toint, P. L. (1992), LANCELOT*: a Fortran package for large-scale nonlinear optimization (Release A)*, number 17 *in* 'Springer Series in Computational Mathematics', Springer Verlag, Heidelberg, Berlin, New York.

Davis, T. A. and Duff, I. S. (1993), An unsymmetric-pattern multifrontal method for sparse LU factorization, Technical Report RAL 93-036, Rutherford Appleton Laboratory.

Daydé, M. J. (1996), A block version of the eskow-schnabel modified cholesky factorization, Technical Report RT/APO/95/8, ENSEEIHT-IRIT.

Daydé, M. J. and Duff, I. S. (1989), 'Level 3 BLAS in LU factorization on the CRAY-2, ETA-10P and IBM 3090-200/VF', *Int. J. of Supercomputer Applics.* **3**, 40–70.

Daydé, M. J. and Duff, I. S. (1991), 'Use of level 3 BLAS in LU factorization in a multiprocessing environment on three vector multiprocessors, the ALLIANT FX/80, the CRAY-2, and the IBM 3090/VF', *Int. J. of Supercomputer Applics.* **5**, 92–110.

Daydé, M. J. and Duff, I. S. (1996), A block implementation of level 3 BLAS for RISC processors, Technical Report RT/APO/96/1, ENSEEIHT-IRIT.

Daydé, M. J., Duff, I. S. and Petitet, A. (1994*a*), 'A parallel block implementation of Level 3 BLAS kernels for MIMD vector processors', *ACM Transactions on Mathematical Software* **20**, 178–193.

Daydé, M. J., L'Excellent, J. Y. and Gould, N. I. M. (1994*b*), 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, to appear in SIAM Journal on Scientific Computing.

Daydé, M. J., L'Excellent, J. Y. and Gould, N. I. M. (1995), Solution of structured systems of linear equations using element-by-element preconditioners, *in* 'Proceedings 2nd IMACS International Symposium on Iterative Methods in Linear Algebra', pp. 181–190. also ENSEEIHT-IRIT Technical Report, RT/APO/95/1.

Daydé, M. J., L'Excellent, J. Y. and Gould, N. I. M. (1996), Preprocessing of sparse unassembled linear systems for efficient solution using element-by-element preconditioners, *in* 'Proceedings of Euro-Par 96, Lyon'. To appear.

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

Dennis, J. and Schnabel, R. (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice Hall, Englewood Cliffs, N.J.

Dodson, D. S., Grimes, R. G. and Lewis, J. G. (1991), 'Sparse extensions to the Fortran Basic Linear Algebra Subprograms', *ACM Transactions on Mathematical Software* **17**, 253–263.

Dongarra, J. and Whaley, R. C. (1995), A users' guide to the blacs, Technical Report CS-95-281, University of Tennessee, Knoxville, Tennessee, USA.

Dongarra, J. J. (1992), Performance of various computers using standard linear algebra software, Technical Report CS-89-85, University of Tennessee, Knoxville, Tennessee, USA.

Dongarra, J. J. and Grosse, E. (1987), 'Distribution of mathematical software via electronic mail', *Comm. ACM* **30**, 403–407.

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

Dongarra, J. J., Duff, I. S., Sorensen, D. C. and van der Vorst, H. A. (1991*a*), *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia.

Dongarra, J. J., Mayes, P. and Radicati di Brozolo, G. (1991*b*), Lapack working note 28 : The IBM RISC System/6000 and linear algebra operations, Technical Report CS-91-130, University of Tennessee.

Duff, I. S. (1996), Sparse numerical linear algebra: direct methods and preconditioning, Technical Report RAL 96-047, Rutherford Appleton Laboratory. Also CERFACS Report TR-PA-96-22.

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

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

Duff, I. S., Grimes, R. G. and Lewis, J. G. (1992), Users' guide for the Harwell-Boeing sparse matrix collection (Release I), Technical Report RAL 92-086, Rutherford Appleton Laboratory.

Duff, I. S., Marrone, M., Radicati, G. and Vittoli, C. (1995), A set of Level 3 Basic Linear Algebra Subprograms for sparse matrices, Technical Report TR-RAL-95-049, RAL.

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

Eskow, E. and Schnabel, R. B. (1991*a*), ‘Algorithm 695: Software for a new modified cholesky factorization’, *ACM Transactions on Mathematical Software* **17**, 306–312.

Eskow, E. and Schnabel, R. B. (1991*b*), ‘A new modified cholesky factorization’, *SIAM Journal on Scientific and Statistical Computing* **11**, 1136–1158.

Gallivan, K., Jalby, W. and Meier, U. (1987), ‘The use of blas3 in linear algebra on a parallel processor with a hierarchical memory’, *SIAM J. Sci. Stat. Comput.* **8**, 1079–1084. Timely communications.

Gallivan, K., Jalby, W., Meier, U. and Sameh, A. (1988), ‘Impact of hierarchical memory systems on linear algebra algorithm design’, *Int Journal of Supercomputer Applications* **2**(1), 12–48.

Gill, P. and Murray, W. (1974), ‘Newton-type methods for unconstrained and linearly constrained optimization’, *Mathematical Programming* **28**, 311–350.

Gill, P., Murray, W. and Wright, M. (1981), *Practical Optimization*, Academic Press, London and New York.

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

HSL (1996), *Harwell Subroutine Library. A Catalogue of Subroutines (Release 12)*, AEA Technology, Harwell Laboratory, Oxfordshire, England. For information concerning HSL contact: Dr Scott Roberts, AEA Technology, 552 Harwell, Didcot, Oxon OX11 0RA, England (tel: +44-1235-434714, fax: +44-1235-434136, email: Scott.Roberts@aeat.co.uk).

Hughes, T. J. R., Ferencz, R. M. and Hallquits, J. O. (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.

Hughes, T. J. R., Levit, I. and Winget, J. (1983), ‘An element-by-element solution algorithm for problems of structural and solid mechanics’, *Compututational Methods in Applied Mechanics and Engineering* **36**, 241–254.

Kågström, B., Ling, P. and Loan, C. V. (1993), Portable high performance GEMM-based Level-3 BLAS, *in* ‘Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing’, SIAM, pp. 339–346.

L'Excellent, J. Y. (1995), Utilisation de préconditionneurs élément-par-élément pour la résolution de problèmes d'optimisation de grande taille, PhD thesis, INPT-ENSEEIHT.

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

Schlick, T. (1993), 'Modified Cholesky factorizations for sparse preconditioners', *SIAM Journal on Scientific and Statistical Computing* **14**, 424–445.

Schnabel, R. B., Koontz, J. E. and Weiss, B. E. (1985), 'A modular system of algorithms for unconstrained minimization', *ACM Transactions on Mathematical Software* **11**, 419–440.

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