Parallel Computing 11 (1989) 223–239
North-Holland

Parallel conjugate gradient-like algorithms for solving sparse nonsymmetric linear systems on a vector multiprocessor

G. RADICATI di BROZOLO and Y. ROBERT *
IBM ECSEC, via Giorgione 159, 00147 Roma, Italy

Received November 1987

Abstract. We describe the parallel implementation on a vector multiprocessor of two extensions of the preconditioned conjugate gradient algorithm for nonsymmetric systems: the conjugate gradient squared algorithm (CGS) and the generalized minimal residual algorithm GMRES(k). For both methods, we consider preconditioning by a diagonal matrix and by an incomplete LU factorization.

The uniprocessor implementation of CGS and GMRES(k) is based on a general sparse matrix representation to deal with matrices with an irregular sparsity structure (the ITPACKV format).

The parallelization of the non-preconditioned versions is straightforward and leads to very good speedups. The parallelization of the ILU preconditioned versions is more challenging. We describe two parallel preconditioners:

- we compute the global ILU factorization and partition it into one block per processor,
- we compute in parallel local partial factorizations for each block of the matrix.

The blocks can overlap, and this partitioning is not restricted to matrices with a special sparsity structure.

In the iteration loop the information between blocks is exchanged at each synchronization point, through the matrix-vector product and through the overlap between blocks. We discuss how the size of the overlap influences the efficiency.

Our experiments, using up to 6 processors, show that the best strategy is to compute in parallal a local ILU factorization on slightly overlapping blocks.

Keywords. Nonsymmetric systems, iterative methods, conjugate gradient, vector processor, parallel processor, shared memory.

1. Introduction

Let $A\,u = b$ be a linear system of equations, where A is a sparse nonsingular matrix of size $m \times m$ and b a vector with m components. If A is symmetric positive definite, the preconditioned conjugate gradient (PCG) algorithm is one of the most efficient iterative method for solving the system (see, among others, [10,10b,12,16,18]). For the nonsymmetric case, the situation is not so well established. Although several extensions of the conjugate gradient algorithm to deal with nonsymmetric matrices have been proposed (e.g. [2,7]), none emerges as a clear winner [26].

In this report, we describe the implementation on a vector multiprocessor of two promising extensions of the PCG algorithm for nonsymmetric systems: the conjugate gradient squared

* On leave from CNRS, Laboratoire TIM3, INPG Grenoble, France

algorithm (CGS) proposed by Sonneweld *et al.* [28], and the generalized minimal residual algorithm GMRES(k) introduced by Saad and Schultz [26]. For both methods, we consider preconditioning by a diagonal matrix and preconditioning by an incomplete LU factorization.

We first briefly describe the uniprocessor implementation of CGS and GMRES(k). It is based on a general sparse matrix representation to deal with matrices with an irregular sparsity structure (the ITPACKV format [13]).

We then describe a parallel implementation of the two methods. Our target machine is the IBM 3090-600E VF [4,32], but most of our conclusions would remain valid on any vector multiprocessor with a shared memory and where the cost of forking parallel tasks and joining at synchronization barriers is of the order of a few thousand floating-point operations (e.g. multitasking on the CRAY X-MP).

The parallelization of the diagonally preconditioned versions is straightforward: we basically split the work equally among the processors (see [27] for a similar approach with the standard CG algorithm). The parallelization of the ILU preconditioned versions is more challenging. We describe two parallel preconditioners derived from the original incomplete factorization technique (which is entirely sequential):

- we compute the global ILU factorization and partition it into one block per processor,
- we compute in parallel local partial factorizations for each block of the matrix.

The blocks can overlap, and this partitioning is not restricted to matrices with a special sparsity structure.

In the iteration loop the information between blocks is exchanged at each synchronization point, through the matrix-vector product and through the overlap between blocks. We discuss how the size of the overlap influences the efficiency.

Our numerical experiments, using up to 6 processors, show that the best strategy is to compute in parallel a local ILU factorization on slightly overlapping blocks. We use analytical matrices so as to make the experiences easily reproducable.

Reporting very good speedups for both diagonal and ILU preconditioning, we conclude that for vector multiprocessors such as the IBM 3090, it is not necessary to recast CG-like algorithms in terms of block algorithms, contrarily to implementations on hypercubes multiprocessors [3,22].

Domain decomposition techniques also lead to parallel implementations of CG-like algorithms (see [6,11,20] among others). We point out that the preconditioning techniques we describe here can be applied to a wide class of matrices, regardless of the original physical meaning of the problem.

2. The CGS and GMRES(k) algorithms

In this section, we describe the CGS and GMRES(k) algorithms. We divide each procedure into computational kernels making clear where the synchronization points will take place in a parallel execution.

2.1. The CGS algorithm

The conjugate-gradient squared (CGS) algorithm has been recently introduced by Sonneveld et al. [28]. The algorithm is derived from the bi-conjugate gradient algorithm (BCG) proposed by Fletcher [8]. As BCG, in exact arithmetic CGS converges to the correct solution in at most m iterations provided that it does not break down. CGS has several advantages over BCG: the transpose of the coefficient matrix is not needed, the algorithm is theoretically ensured to converge whenever BCG does, and it generally converges faster (see [23,28]).

We give below the CGS algorithm. We have slightly modified the original algorithm of [28] in order to optimize the use of vector registers. K is the preconditioning matrix; u_0 is the initial guess for the solution, and w is a work vector.

CGS procedure

- Initialization
  - Kernel 1
    $w = b - Au_0$
    $r_0 = K^{-1}w$
    $q = r_0$
    $e_0 = (q, r_0)$
  - $\beta_0 = 0$
- For $i = 0, 1, \ldots,$ until convergence, do
  - Kernel 2
    $g_i = r_i + 2\beta_i h_i + \beta_i^2 g_{i-1}$
  - Kernel 3
    $w = K^{-1}Ag_i$
    $\sigma = (q, w)$
  - $\alpha_i = e_i/\sigma$
  - Kernel 4
    $h_{i+1} = (r_i + \beta_i h_i) + \alpha_i K^{-1}Ag_i$
    $p_i = (r_i + \beta_i h_i) + h_{i+1}$
    $u_{i+1} = u_i + \alpha_i p_i$
    $eu_{i+1} = \| u_{i+1} \|$
  - Kernel 5
    $w = K^{-1}Ap_i$
    $r_{i+1} = r_i - \alpha_i w$
    $e_{i+1} = (q, r_{i+1})$
    $err_{i+1} = \| r_{i+1} \|$
  - $\beta_{i+1} = e_{i+1}/e_i$
  - end do

The stopping criterion is as follows: we stop the iteration when the ratio err_i/eu_i of the norm of the residual over the norm of the solution vector is smaller than a precision fixed by the user.

The procedure is divided into computational kernels, which are executed in parallel. Each of the kernel is followed by a synchronization point. We describe later on how we parallelize the execution of each kernel.

In most kernels a scalar product is computed, which is needed before any work can be done in the next kernel: this is a bottleneck which requires a barrier where all processors will synchronize. Meurant [19] describes how to remove a synchronization point in the standard CG algorithm by precomputing one of the scalar products. We do not know whether something similar can be done for CGS. In Kernel 2 the whole of vector g will be needed by all processors in the next matrix-vector product, hence again a synchronization point.

2.2. The GMRES(k) algorithm

In the GMRES algorithm [26], an orthonormal basis of directions is formed and the solution is given by a linear combination of the basis vectors so as to minimize the norm of the residual in the subspace spanned by the basis. Because storing all the previous directions is very costly, in practice a truncated version GMRES(k) is used. In this algorithm, the process is restarted

every k steps. A nice theoretical property of GMRES(k) is that convergence is ensured if k is chosen large enough.

Here again, we have modified the original procedure [26] in order to increase the granularity of the computational blocks. Note that the norm of the residual can be estimated at each iteration without explicitly computating the residual vector. In the following procedure, c, s and rs are vectors of size $k+1$ and H is a $(k+1)\times(k+1)$ matrix; K is the preconditioning matrix, and u_0 is the initial guess for the solution.

GMRES(k) procedure

- Outer loop: for $iout = 0$ step $k+1$ until convergence, do
  - Kernel 1
    $w = b - Au_{iout}$
    $v_1 = K^{-1}w$
    $h_{1,1} = \| v_1 \|$
  - $err = rs_1 = h_{11}$
  - Inner loop: for $j = 1$ to k do
    1. Kernel 2
       $w = Av_j/h_{j,j}$
       $v_{j+1} = K^{-1}w$
       $h_{1,j+1} = (v_1, v_{j+1})$
    2. for $i = 1$ to $j-1$ do
       - Kernel 3
         $v_{j+1} = v_{j+1} - h_{i,j+1}v_i$
         $h_{i+1,j+1} = (v_{i+1}, v_{j+1})$
    3. $h_{j,j+1} = h_{j,j+1}/h_{j,j}$
    4. Kernel 4
       $v_j = v_j/h_{j,j}$
       $v_{j+1} = v_{j+1} - h_{j,j+1}v_j$
       $h_{j+1,j+1} = (v_{j+1}, v_{j+1})$
    5. $h_{j+1,j+1} = \sqrt{h_{j+1,j+1}}$
    6. perform some scalar computations:
       - for $i = 1$ to $j-1$ do
         $$\begin{bmatrix} h_{i,j+1} \\ h_{i+1,j+1} \end{bmatrix} = \begin{bmatrix} c_i & s_i \\ -s_i & c_i \end{bmatrix} \begin{bmatrix} h_{i,j+1} \\ h_{i+1,j+1} \end{bmatrix}$$
       - $\gamma = (h_{j,j+1}^2 + h_{j+1,j+1}^2)^{1/2}$: $c_j = h_{j,j+1}/\gamma$; $s_j = h_{j+1,j+1}/\gamma$
       - $rs_{j+1} = -s_j \times rs_j$; $err = |rs_{j+1}|$
       - $rs_j = c_j \times rs_j$
       - $h_{j,j+1} = c_j \times h_{j,j+1} + s_j \times h_{j+1,j+1}$
       - solve the upper triangular system
         $$\begin{bmatrix} h_{12} & h_{13} & \cdots & h_{1,j+1} \\ & h_{23} & \cdots & h_{2,j+1} \\ & & \cdots & \cdots \\ & & & h_{j,j+1} \end{bmatrix} \begin{bmatrix} rs_1 \\ rs_2 \\ \cdots \\ rs_j \end{bmatrix} = \begin{bmatrix} rs_1 \\ rs_2 \\ \cdots \\ rs_j \end{bmatrix}$$
    7. end inner loop
  - Kernel 5
    $$u_{iout} = u_{iout} + \sum_{j=1}^{k} rs_j \times v_j$$
    $eu = (u, u)$
  - end outer loop

The stopping criterion is the same as for CGS, using the estimate *err* for the norm of the residual.

A few words of explanation are needed here: in the usual implementation of GMRES(k), the vector v_j is scaled by $1/h_{j,j}$ *before* computing $w = Av_j$ (Kernel 2). This requires a synchronization point. In our implementation we have removed this synchronization point at the cost of an extra multiplication on w, which comes nearly for free when w is stored in a vector register (Kernel 2). Vector v_j is then updated in Kernel 4, which explains why the i-loop of Kernel 3 type operations go to $j-1$ rather than j.

3. Sparse matrix representation and matrix-vector product

We briefly describe in this section the internal representation which we adopt to store the coefficient matrix A. Our objective is a representation general enough to store a wide class of matrices and allowing for an efficient vectorization of matrix-vector products.

The standard *row-wise representation* to store a sparse $m \times m$ matrix A is not suited to vector computers: in the matrix-vector product $y = Ax$, the small number of nonzero elements per row does not allow to efficiently vectorize the dot product of a row of A and the gathered components of x.

The use efficiently the vector hardware, we adopt the storage scheme used in ITPACKV [13] which we term *compressed-matrix representation*. Let k be the maximum number of nonzero elements per row of A. The matrix is stored using two rectangular arrays AC and KA with m rows and k columns:

- AC is a real array: each row of AC contains the nonzero elements of the corresponding row of the matrix A. If a row of A has fewer than K elements, the corresponding row in AC is padded with zeros to length k.
- KA is an integer array, which contains the column number of the corresponding elements of AC. If the corresponding element in AC is zero, any index in the range $1 \ldots m$ may be used.

Following is a FORTRAN program to perform the matrix-vector product $y = Ax$ using this storage scheme:

```
                SUBROUTINE MX(M,K,AC,KA,LDA,X,Y)

                REAL*8 AC(LDA,K), X(M), Y(M)
                INTEGER KA(LDA,K)

VECT +-------   DO 20 I=1,M
     |             Y(I) = 0.
     |             DO 10 J=1,K
     |                Y(I) = Y(I) + X(KA(I,J))*AC(I,J)
     |       10    CONTINUE
     |______ 20 CONTINUE
```

Note that arrays AC and KA are accessed in an orderly fashion. Array X, on the contrary, is accessed randomly with addresses specified by KA.

In this algorithm, the vector y is computed by summing k vectors of length m that are the result of an element-by-element product of two vectors. Sections of y are accumulated in a vector register and only stored once at the end of the loop on k.

On a scalar processor, this algorithm does not have an optimal performance because, if some rows have less than k elements, it will result in the execution of a certain amount of multiplications by zero. On a vector processor, using this representation, the matrix-vector multiplication vectorizes with the use of *gather-scatter* type operation to collect the elements of the vector X involved in each element-by-element product. All the vector instruction are executed on long vectors (of length m).

The *compressed-matrix representation* is efficient to store matrices which do not have a widely differing number of nonzero elements on each row, so as not to perform a too important fraction of multiplications by zero. This condition is fulfilled by many sparse matrices arising from the applications. The *compressed-matrix representation* is more general than a representation by diagonals (which leads to the most efficient implementation for structured sparse matrices). It uses the vector hardware efficiently and allows us to represent matrices arising from the discretization of partial differential equations on irregular grids.

4. Parallelization of the non-preconditioned versions

We discuss here the implementation of the non-preconditioned versions of the two algorithms. In fact, we perform a diagonal scaling before iterating. For symmetric positive definite matrices, the choice of $K = D$, where D is the main diagonal of A, seems to be the most effective. In the nonsymmetric case, various schemes are possible. We choose to normalize to unity the main diagonal of the matrix AA^{T}, that is we let

$$K_i = \sqrt{\sum_j a_{ij}^2}$$

such that all the diagonal elements of the matrix $(D^{-1}A)(D^{-1}A)^{\mathrm{T}}$ are equal to 1. In this particular case, we do not have to solve a diagonal system at each iteration step. Rather, we scale the matrix before entering the iteration loop.

In what follows, we always implicitly use this diagonal preconditioning when referring to the CGS and GMRES(k) algorithms, that is, we assume that the diagonal of the matrix AA^{T} has been normalized (in addition to other preconditionings if any).

In the case we discuss here, we just let K be the identity matrix in the procedures. The basic kernels of the two schemes are all of the following type, where x, y, z and q are four vectors of size m:

Kernel A
$y = Ax + \alpha \times z$
$e = (y, q)$

The value e of the scalar product is typically needed before the execution of the next kernel, so that a synchronization point is needed after the execution of Kernel A.

Assume we have p processors. The idea is to split the computation equally among the processors: each processor operates on a horizontal slice of A and computes a section of length $1/p$ of vector y, and the corresponding fraction of the scalar product e. Note that the whole vector x will be read by all the processors. However, since in a typical PDE problem A has a limited bandwidth, read conflicts are unlikely to occur.

After the synchronisation point, we compute the scalar product $e = (y, q)$ as

$$e = \sum_{n=1}^{p} e(n).$$

This example should give a good insight to the parallelization of both CGS and GMRES(k). In the CGS procedure we have one synchronization point at the initialization and four inside the iteration loop, which amounts to two synchronizations per matrix-vector product. In GMRES(k) there are $k + 2 + \frac{1}{2}k(k+1)$ synchronization points every $k+1$ iterations, which amounts to roughly $\frac{1}{2}k + 1$ synchronizations per matrix-vector product.

Finally, we point out that the same techniques can be applied when the algorithms are preconditioned by a polynomial in A of degree 1; see [27] in the case of standard CG.

Here we describe some implementation details: the uninterested reader might skip the end of this section. Assume that m is divisible by p, and let $ns = m/p$ be the number of components of y that each processor will compute. Kernel A will be executed in parallel as follows, where DSPTCH and SYNCRO are the MTF FORTRAN callable routines for forking and joining (see [9, appendix E]). Each of the p processors works on different sections of y, z and q.

```
      DO 10 n = 1, p
         ibeg(n) = (n-1)*ns + 1
         CALL DSPTCH('subA',ns,A(ibeg(n),1),KA(ibeg(n),1),LDA,
     *               X,Y(ibeg(n)),Z(ibeg(n)),Q(ibeg(n)),alpha,e(n))
   10 CONTINUE
      CALL SYNCRO
```

Processor n executes the following routine:

```
      SUBROUTINE subA (L,K,AC,KA,LDA,X,Y,Z,Q,alpha,e)
      ...
VECT +-------    DO 20 I=1,L
     |              ACC = 0.
     |              DO 10 J=1,K
     |                 ACC = ACC + X(KA(I,J))*AC(I,J)
     |          10    CONTINUE
     |              Y(I) = ACC + alpha*Z
     |              e = e +  Y(i)*Q(i)
     |_______   20 CONTINUE
```

We reproduce the code of routine subA to illustrate the reuse of vector registers for computing y and e.

5. Parallelization of the ILU preconditioned versions

Parallelizing the ILU preconditioned versions is more challenging. To our knowledge, this work has been tackled only for special class of matrices (arising from the discretization of PDEs on rectangular grids [6,20].

We first discuss the choice of an ILU-based preconditioner. We then describe the parallelization of the iterative procedure. The basic idea is to give to each processor a sub-block of the ILU factorization. The blocks can overlap.

5.1. Parallel ILU preconditioners

We assume the reader is familiar with the incomplete factorization [15,16], which consists in computing an approximate factorization of A

$$A = LDU - R.$$

The sparsity structure of the factors L and U is the same as that of A: we accept a nonzero entry in a factor only if the corresponding entry in the matrix A is non-zero. In other words, fill-in has been suppressed in the factorization, which in turn is no longer exact. R is a (hopefully) small error matrix, which as far as preconditioning is concerned is not computed: $K = LDU$ is the preconditioning matrix.

Incomplete factorization preconditioning has been successfully used by many authors in the symmetric positive definite case (e.g. [12]). It appears also very attractive in the nonsymmetric case, as reported in [7,14,21,28,29] among others.

The ILU preconditioner is sequential, because at each step of the iterative procedure we have to solve sparse triangular systems like $LDUx = y$, which requires a global knowledge of y. To derive a parallel preconditioner from the original ILU factorization, we split the factorization into blocks that can overlap, and solve in parallel a subsystem for each block.

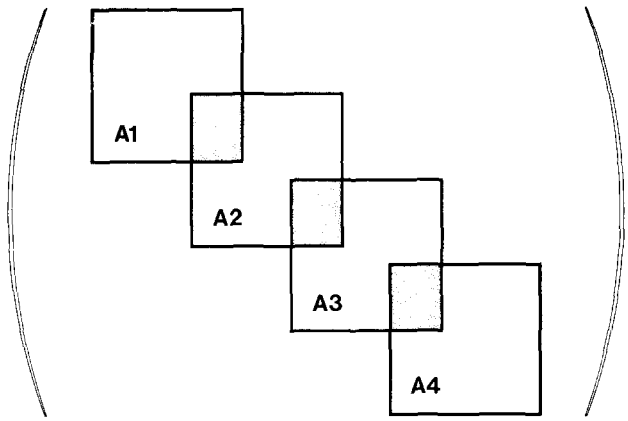

Fig. 1. The partition induced by the ILU preconditioning.

More precisely, if we have p processors, we will have p partial factorizations $L_n D_n U_n$. In Fig. 1, each block A_n is a square matrix of order $nsilu \geqslant m/p$. If $nsilu = m/p$, the blocks do not overlap. Otherwise the overlap nov between two consecutive blocks is equal to

$$nov = (nsilu \times p - m)/(p - 1).$$

We used two different strategies for computing the parallel ILU preconditioners $L_n D_n U_n$:

(1) *Global preconditioner*: the first solution is to compute the global ILU factorization of A: $LDU - R$. Then for each block A_n we only consider those elements of L, D and U that belong to A_n and store them into L_n, D_n and U_n.

(2) *Local preconditioner*: the second solution is to compute in parallel the ILU factorization of all the blocks: $A_n = L_n D_n U_n - R_n$,

Both stategies lead to p partial factorizations $L_n D_n U_n$. To formally define the preconditioning, we must explain what computing $x = K^{-1}w$ means, where w is a vector of size m. We describe this operation as follows:

- split w into p sections w_n according to the partition,
- solve (in parallel) the p systems $L_n D_n U_n x_n = w_n$,
- define x to be equal to the x_n's in the nonoverlapped parts and the average of them in the overlapped parts.

The *local* solution is likely to lead to a worse preconditioner, because less information is propagated downwards. However, the fact that the preconditioning is computed in parallel might pay off for a few extra iterations.

We report numerical experiments with these two strategies in Section 6.

5.2. Parallel implementation

The computational kernels of the preconditioned CGS and GMRES(k) algorithms are of the following type, where x, w, y and q are four vectors of size m:

Kernel B

$w = Ax$
$y = K^{-1}w$
$e = (y, q)$
$f = (y, y)$

We now discuss how to compute this kernel with the parallel ILU preconditioners as discussed above.

If there is no overlap between neighboring blocks, Kernel B can be parallelized easily just as we did for Kernel A. We show in Section 6 that some overlap between blocks leads to a better preconditioner. If there is some overlap between neighboring blocks, y cannot be computed until the computation of w is complete. A simple solution is to compute w in parallel just as we did in Kernel A; then synchronize, solve the systems $L_n D_n U_n$ to compute y in parallel as discussed above; then synchronize again, and proceed to the parallel computation of the scalar products e and f. This simple approach would add two synchronization points per each solve, and would significantly decrease the efficiency of the parallel implementation.

Examining the solution of $y_n = (L_n D_n U_n)^{-1} w$ more carefully, we note that only a section of w is needed in this computation. So we can avoid the first synchronization point by computing $w = Ax$ in slices of size *nsilu* induced by the preconditioner. The slices of A given to the processors overlap, hence the overlapping sections will be computed repeatedly. So we avoid the first synchronization at the cost of some extra arithmetic on the overlapping slices.

Rather than using a work vector w of length m, we use p shorter vectors w_n of length *nsilu*, the size of the blocks in the partition induced by the preconditioner. Each processor n computes its own partial product and stores the result into w_n. Still in parallel and without synchronization, we can solve for the $L_n D_n U_n$ systems. The simplest solution is to overwrite $w_n = L_n D_n U_n w_n$. Then after synchronization we copy the w_n's into y, averaging the values on the overlapped parts.

We now describe how we have implemented the remainder of Kernel B to avoid the second synchronization point after the solve and costly moves of the vectors w_n into y. The idea is the following: when solving the triangular systems, we can already store part of the solution into y, provided that each processor writes on a different part of y. We adopt the following method: processor n stores the last components of its solution in y. The first components, which overlap with processor $n-1$,will overwrite those of w_n (see Fig. 2). We denote this symbolically with

$$L_n D_n U_n (w_n, y) = w_n.$$

After the synchronization point, we compute the final result y by averaging the values on the overlapped parts.

For the parallel computation of the scalar products e and f, we split each block into three parts (see Fig. 2): the part that overlaps with the previous block (region 1), the non-overlapped part (region 2) and the part that overlaps with the next block (region 3). Since e is a linear function of y, we compute a partial scalar product on each region (we denote them by e_1, e_2 and e_3 respectively), and will average the global value after synchronization. For f, we only compute the value f_2 for region 2. The part relative to the overlapped regions will be computed after the synchronization, from the final value of y.

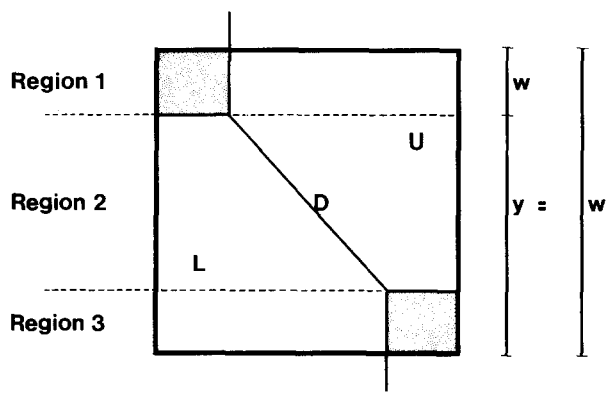

Fig. 2. Scheme of computation inside one block.

Each processor executes the following code:

```
SUBROUTINE subB (L,K,AC,KA,X,Y,Q,W,len,e1,e2,e3,f2)
...
1. Compute the product w = A x:
      CALL MX(L,K,AC,KA,LDA,X,W)
2. Solve the triangular systems: LDU (w,y) = w
      CALL SOLVE
3. Compute e1,e2,e3
      e1 = (q,w) on region 1
      e2 = (q,y) on region 2
      e3 = (q,y) on region 3
4. Compute f2
      f2 = (y,y) on region 2
```

Note that w is a different vector for each processor. Similarly, e_1, e_2, e_3 and f_2 are private variables to each processor. All processors access vector x, but there are few memory contentions due to the sparsity of A. They all write in different sections of y, and store the components corresponding to the top overlap in their vector w.

For the sake of clarity, we have not described the routine SOLVE (and not passed the preconditioner $L_n D_n U_n$ to the subroutine). The matrices L_n and U_n are stored in the standard row-wise format, and SOLVE is just the usual serial routine with an additional flag for storing the result either in w or in y.

Note that the solution of a sparse triangular system of equations cannot, to our knowledge, be vectorized efficiently. Some techniques have been derived for some special cases, assuming a specific sparsity pattern [30] or the presence of many right-hand sides [1,31]. So in our implementation we use the standard *row-wise* representation to store the triangular factors L and U of the incomplete factorization.

After the synchronization point, we compute e as

$$e = \sum_{n=1}^{p} e_2(n) + \tfrac{1}{2}(e_1(n) + e_3(n)).$$

A first part of f is evaluated as

$$f = \sum_{n=1}^{p} f_2(n).$$

We then average y on the overlapped parts and finish the computation of f accordingly.

We outline now the parallel section of the code corresponding to Kernel B. w_n is the nth column of a working matrix W. The lengths of the three regions for processor n are stored in $len(i, n)$, $i = 1, 2, 3$. $len(1, n) = nov$ for each processor except the first one, $len(3, n) = nov$ for each processor except the last one, and $len(1, n) + len(2, n) + len(3, n) = nsilu$ for each processor.

```
      nov = (nsilu*p - m) / (p-1)
      DO 10 n = 1, p
         ibegilu(n) = (n-1)*(ns-nov) + 1
         CALL DSPTCH('subB',nsilu,k,A(ibegilu(n),1),KA(ibegilu(n),1),LDA,
     *               X,Y(ibegilu(n)),Q(ibegilu(n)),W(1,n),len(1,n),
     *               e1(n),e2(n),e3(n),f2(n))
   10 CONTINUE
      CALL SYNCRO
      DO 30 n = 1, p
         ibegilu(n) = (n-1)*(ns-nov) + 1
C---     average Y with W(.,n) on region 1
         DO 20 i = ibegilu(n), ibegilu(n) + len(1,n) - 1
            Y(i) = 0.5 * ( Y(i) + W(i-ibegilu(n)+1,n) )
            f = f + Y(i)*Y(i)
   20    CONTINUE
   30 CONTINUE
```

Note that the code after the synchronization is not executed in parallel.

In summary, we have parallelized the execution of Kernel B without any synchronization inside the Kernel, even when the blocks A_n overlap, at the cost of some extra arithmetic operations on the overlapping regions. In particular, we compute extra elements of the matrix-vector product, extra scalar products, and we have to average the vector y on the overlapping regions. So the extra arithmetic is proportional to the sum of the overlap between the blocks.

The number of synchronizations is the same as in the non-preconditioned versions.

6. Numerical experiments

In this section we report the performance of the parallel implementation of the iterative solvers on two sets of test matrices. We consider how the efficiency of the parallel implementation is influenced by the size of the problem, the choice of the preconditioner (the global vs the local strategy), and the overlap of neighboring blocks. Our goal is to study the efficiency of the parallel implementation rather than comparing CGS and GMRES(k) (see [23] for such a comparison). Accordingly, we report speedups over the uniprocessor implementation rather than timings. We take $k = 10$ in all the experiments with GMRES. The results are not very sensitive to the choice of k.

We considered two nine-diagonal matrices for simplicity. In all the experiments, we generate a random right-hand side and compute the real error to verify the error estimate provided by the algorithms. All times refer to the full procedure, including the computation of the incomplete factorization. All procedures are entirely FORTRAN codes running on an IBM 3090-600 E VF [4,32]. To give a rough estimate, with a uniprocessor implementation we obtain about 17 Mflops when iterating the diagonally preconditioned versions and 8 Mflops when iterating the ILU preconditioned ones.

We consider two nine-diagonal matrices A and B. The first, A, is a diagonal matrix whose nonzero coefficients are constant along the diagonals of indices -181, -180, -179, -1, 0, 1, 179, 180, 181. The second, B, is a diagonal matrix of size 32 400 whose nonzero coefficients are constant along the diagonals of indices -10801, -180, -179, -1, 0, 1, 179, 180, 10801. When we compute the local factorization for the 3, 4, 5 and 6 processors versions, the diagonals -10801 and 10801 do not contribute to the computation, giving a notably different result from the global factorization scheme.

Diagonal structures of these kind are typical of matrices arising from the discretization of a partial differential equation using two-dimensional linear finite elements on a rectangular grid. We chose such matrices so that these experiments could be easily reproduced.

The coefficients along the diagonals are

$$A = (-0.5, -2, -0.5, -1.5, 12, -2.5, -1.5, -2, -1.5),$$

$$B = (-0.5, -2, -0.5, -1.5, 11.3, -2.5, -1.5, -2, -1.5).$$

In Fig. 3 we plot the efficiency of the parallel version of CGS as a function of the size of the problem. In this case the number of operations is the same in the parallel version as in the uniprocessor version. The data clearly shows that the efficiency of the parallel versions increases with the size of the computational granularity. The two processor version reaches a good efficiency for problems of size 5000. To reach a comparable efficiency on six processors the size of the problem has to increase to about 25 000. For problems of such a size the dispatch and synchronization overhead becomes negligible.

In Fig. 4 we compare the efficiency of the parallel implementation of CGS and GMRES as a function of the number of processors for the solution of problem A of size 32 400. As we

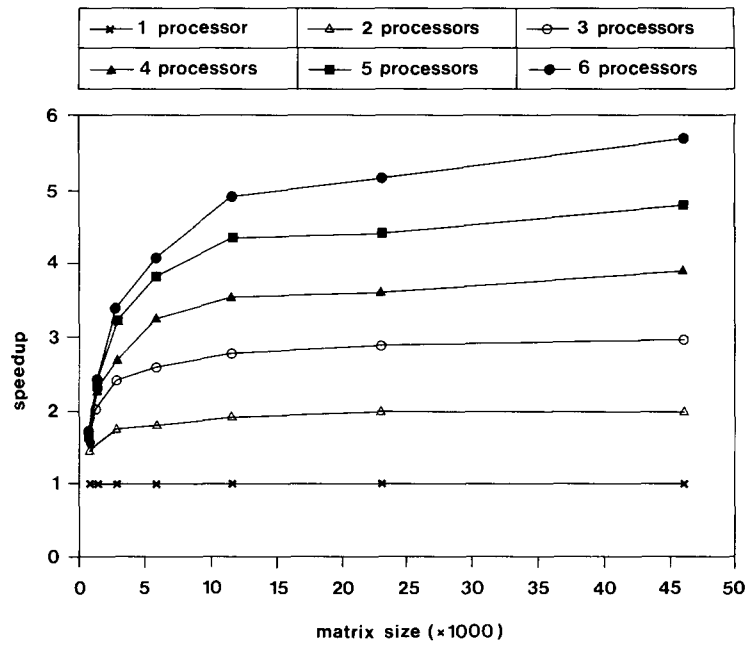

Fig. 3. Speedup of the six parallel versions of CGS as a function of the problem size.

remarked earlier, in CGS we need two synchronization points per matrix-vector product, whereas we need an average of approximately $1 + k/2$ in GMRES(k). The greater cost of the synchronizing and dispatching tasks accounts for the lower efficiency of GMRES(k).

In Fig. 5 we compare the efficiency of different versions of CGS preconditioned by an incomplete factorization (ILU-CGS). We consider the global factorization with no overlap between blocks (*global/no-ovlp*), and with some overlap between blocks (*global/ovlp*); and we

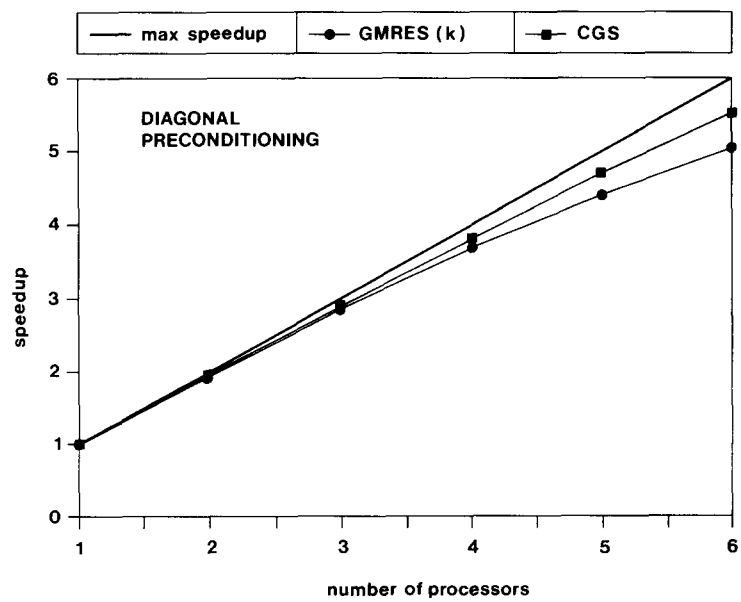

Fig. 4. Speedup of CGS and GMRES(k) as a function of the number of processors ($m = 32400$)

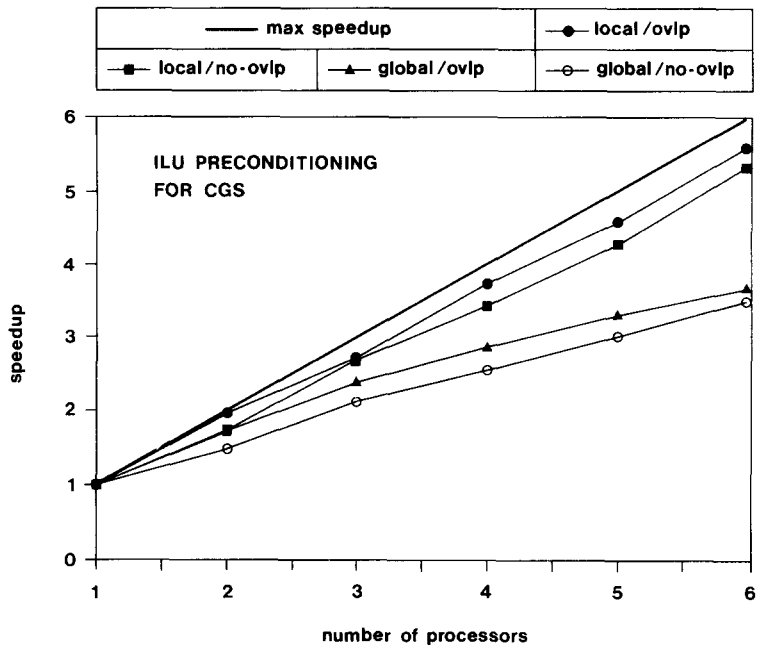

Fig. 5. Speedup of ILU-CGS as a function of the number of processors ($m = 32400$).

consider the local factorization with non-overlapping blocks (*local/no-ovlp*), and with some overlap (*local/ovlp*). In Fig. 6 we plot similar data for GMRES(k). The data shows that the local factorization scheme has a better overall performance. Comparing the global performances of the global vs the local factorization schemes, we note that generally the global

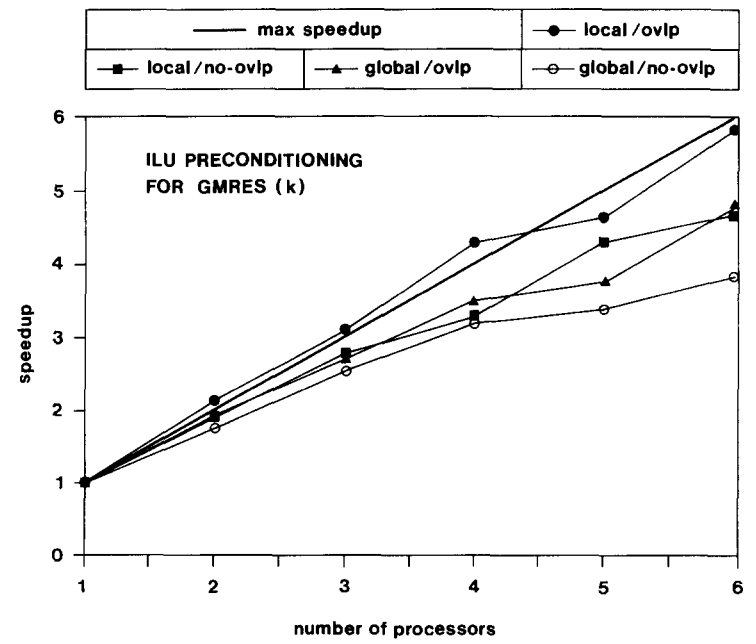

Fig. 6. Speedup of ILU-GMRES(k) as a function of the number of processors ($m = 32400$).

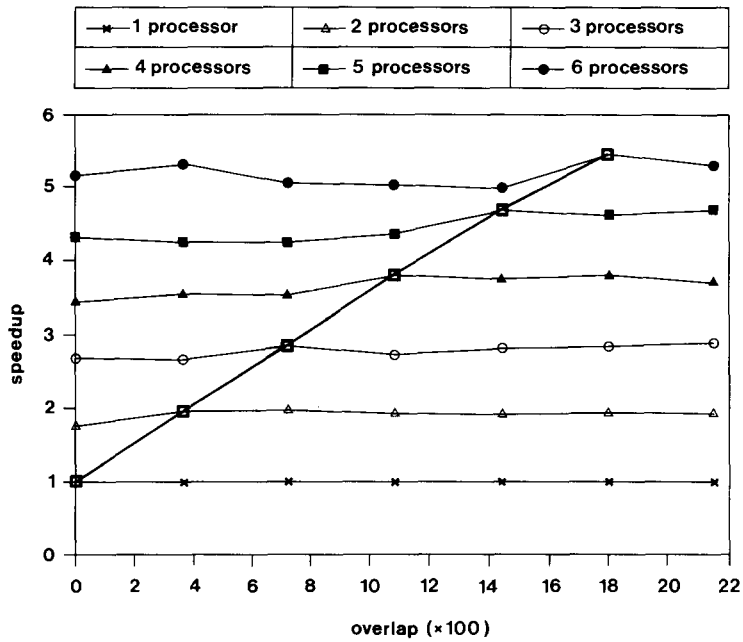

Fig. 7. Speedup of ILU-CGS as a function of the total overlap.

scheme provides a better preconditioner, because the number of iterations is slightly smaller. However, this does not always compensate the cost of having to compute the factorization on one processor. In the example we consider the cost of the factorization is quite large relative to the total cost of the algorithm (10–15%). In the global factorization scheme the factorization has to be computed on one processor while the others remain idle. The figures also show that some overlap between neighboring blocks decreases the global cost of the algorithm. Note that the uniprocessor version of all the algorithms is the same.

In Fig. 7 we consider the efficiency of ILU-CGS, and in Fig. 8 that of ILU-GMRES, as a function of the sum of the overlaps between neighboring blocks, using the parallel local ILU scheme. The amount of extra arithmetic per iteration is proportional to the total overlap. In both CGS and GMRES when the overlap is 0 (i.e. we consider distinct blocks) the number of iteration of the parallel versions is somewhat higher than that of the uniprocessor version which uses a standard incomplete factorization. This means that the parallel preconditioner is not as good as the standard one. The number of iterations in the parallel versions decreases to that of the uniprocessor version as the overlap between blocks increases. So the efficiency of the parallel versions increases to reach a peak when to neighboring blocks overlap—in our examples when two neighboring blocks overlap by roughly 350 elements. The number of iterations remains roughly constant as the overlap increases further, but the efficiency decreases because the amount of floating-point arithmetic increases with the size of the overlap.

In Figs. 7 and 8, the thick lines underline the best performance for each value of p, the number of processors. Note that in each case this corresponds to an overlap of approximately 360 between neighboring blocks. In Fig. 8 (GMRES(k)) we have some efficiencies greater than one, because some of the parallel versions have fewer iterations than their sequential counterpart. Surprisingly enough in this example, the parallel preconditioner is better than the uniprocessor one!

In Fig. 9 we plot of ILU-CGS for matrix B which has a much wider bandwidth than matrix A. The efficiency of the parallel versions is less than in Fig. 7, because the number of iterations

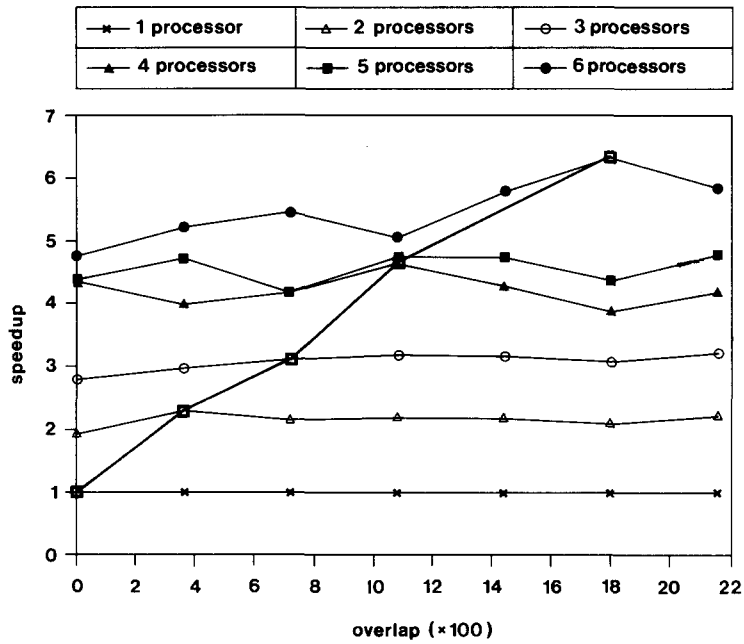

Fig. 8. Speedup of ILU-GMRES(k) as a function of the total overlap.

is somewhat greater than that of the uniprocessor version. Basically, the parallel ILU preconditioner is less accurate than a standard incomplete factorization, because it leaves out a large number of elements of the coefficient matrix.

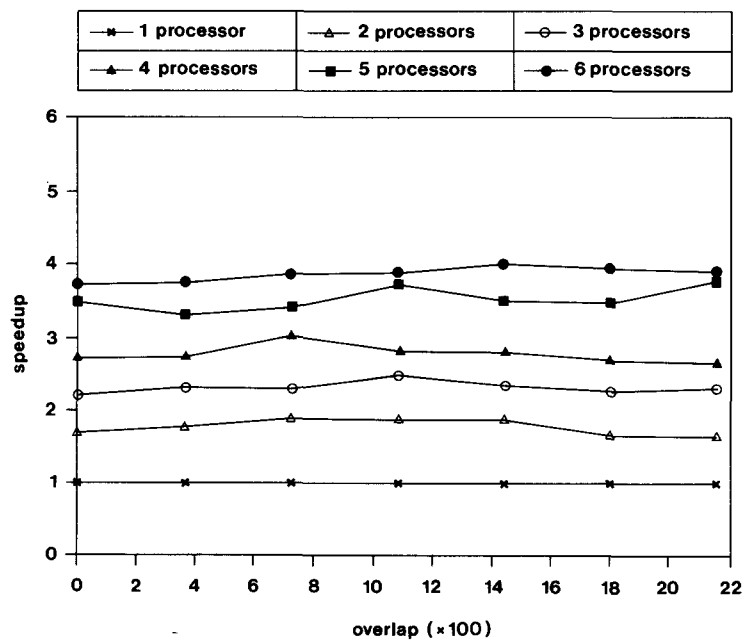

Fig. 9. Speedup of ILU-CGS as a function of the total overlap (wide bandwidth).

7. Conclusion

In this paper, we have discussed the parallel implementation of the CGS and GMRES(k) algorithms on a vector multiprocessor (the IBM 3090-600E VF). For both algorithms we have considered preconditionings by a diagonal matrix and by an incomplete LU factorization. We used the ITPACKV vector format [13] which allows both for an efficient use of the vector hardware and representing a wide class of matrices.

We successfully parallelized large problems, where the impact of communication and synchronization between processors become negligible. Matrices arising from the applications are generally of the order of a few tens of thousands unknowns, which is sufficient to lead to efficient parallelizations.

Parallelizing the diagonally preconditioned versions presents no difficulty. Parallelizing the ILU preconditioned versions is more challenging. In this case, since the ILU preconditioner is entirely sequential, we have to modify it for a parallel implementation. We used a parallel preconditioner which is generally less accurate than a standard incomplete factorization, because the number of iterations increases slightly.

We have discussed two strategies for chosing the preconditioner: either split the original incomplete factorization into blocks (global preconditioner) or directly compute concurrently partial factorizations (local preconditioner). The experiments show that the local strategy is generally the best one.

The partial factorizations should overlap, so that more information is exchanged in the iterative process. The experiments show that a slight overlap is generally sufficient to get a number of iterations close to the uniprocessor execution (in some cases, it is even less!).

As a conclusion, the best strategy, which consists of using a local preconditioner with a small overlap between the blocks, was shown to lead to good speedups, very close to those obtained with a diagonal preconditioning.

References

[1] E. Anderson and Y. Saad, Solving sparse triangular linear systems on parallel computers, Technical Report, CSRD, University of Illinois at Urbana Champaign, 1987.

[2] O. Axelsson, Conjugate gradient type methods for unsymmetric and inconsistent systems of linear equations, *Linear Alg. Appl.* **29** (1980) 1–16.

[3] C. Aykanat and F. Ozguner, Large grain parallel conjugate gradient algorithms on a hypercube multiprocessor, in: S.K. Sahni, ed, *Proc. 1987 International Conference on Parallel Processing* (The Pennsylvania State University Press, Philadelphia, 1987) 641–644.

[4] W. Buchholz, The IBM system/370 vector architecture, *IBM Systems J.* **25** (1) (1986) 51–62.

[5] T.F. Chan, Analysis of preconditioners for domain decomposition, *SIAM J. Numer. Anal.* **24** (2) (1987) 382–390.

[6] A.T. Chronopoulos and C.W. Gear, Implementation of preconditioned s-step conjugate gradient methods a multiprocessor system with memory hierarchy, Technical Report UIUCDCS-R-87-1347, University of Illinois at Urbana Champaign, 1987.

[7] H.C. Elman, Iterative methods for large sparse nonsymmetric systems of linear equations, Research Report 229, Computer Science Department, Yale University, 1982.

[8] R. Fletcher, Conjugate gradient methods for indefinite systems, in: G.A. Watson, ed., Lecture Notes in Mathematics **506** (Springer, Berlin, 1976) 73–89.

[9] VS FORTRAN Version 2 Programming Guide (Release 1.1), Order No SC26-4222-1, available through IBM branch offices, 1986.

[10] G.H. Golub and G.A. Meurant, *Résolution Numérique des Grands Systèmes Lineaires* (Eyrolles, Paris, 1983).

[10b] L.A. Hageman and D.M. Young, *Applied Iterative Methods* (Academic Press, London, 1981).

[11] D.E. Kayes and W.D. Gropp, A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation, *SIAM J. Sci. Statist. Comput.* **8** (2) (1987) 166–202.

[12] D.S. Kershaw, The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations, *J. Comput. Phys.* **26** (1978) 43–65.

[13] D.R. Kincaid, T.C. Oppe, J.R. Respess and D.M. Young, ITPACKV 2C user's guide, Technical Report CNA-191, The University of Texas at Austin, 1984.

[14] A.E. Koniges and D.V. Anderson, ILUBCG2: a preconditioned biconjugate gradient routine for the solution of linear asymmetric matrix equations arising from 9-point discretizations, *Comput. Phys. Comm.* **43** (1987) 297–302.

[15] T.A. Manteuffel, An incomplete factorization technique for positive definite linear systems, *Math. Comp.* **34** (150) (1980) 473–497.

[16] J.A. Meijerink and H.A. Van der Vorst, An iterative solution method for linear systems of which the coefficient matrix is a symmetric *M*-matrix, *Math. Comp.* **31** (137) (1977) 148–162.

[17] J.A. Meijerink and H.A. Van der Vorst, Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems, *J. Comput. Phys.* **44** (1981) 134–155.

[18] R. Melhem, Towards efficient implementation of preconditioned conjugate gradient methods on vector supercomputers, *J. Supercomput. Appl.* **1** (1) (1987) 70–98.

[19] G. Meurant, Multitasking the conjugate gradient on the CRAY XMP-48, Technical Report NA-85-33, Stanford University, 1985.

[20] G. Meurant, Domain decomposition vs block preconditioning, in: R. Glowinski et al., eds., *Proc. 1st International Conference on Domain Decomposition Methods for Partial Differential Equations* (SIAM, Philadelphia, PA, 1987).

[21] Z. Mikic and M. Morse, The use of a preconditioned bi-conjugate gradient method for hybrid plasma stability analysis, *J. Comput. Phys.* **61** (1985) 154–185.

[22] D.P. O'Leary, Parallel implementation of the conjugate gradient algorithm, *Parallel Comput.* **5** (1987) 127–139.

[23] G. Radicati and Y. Robert, A comparison of conjugate gradient-like algorithms for solving sparse nonsymmetric linear systems: a vector and parallel implementation, Technical Report Laboratoire TIM3, Institut National Polytechnique de Grenoble, 1987).

[24] G. Radicati and M. Vitaletti, Sparse linear systems iterative solvers on the IBM 3090 with vector facility, *IBM J. Res. Develop.*, to appear.

[25] Y. Saad and M.H. Schultz, Conjugate gradient-like algorithms for solving nonsymmetric linear systems, *Math. Comp.* **44** (170) (1985) 417–424.

[26] Y. Saad and M.H. Schultz, GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM J. Sci. Statist. Comput.* **7** (3) (1986) 856–869.

[27] M.K. Seager, Parallelizing conjugate gradient for the CRAY X-MP, *Parallel Comput.* **3** (1986) 35–47.

[28] P. Sonneweld, P. Wesseling and P.M. de Zeeuw, Multigrid and conjugate gradient methods as convergence acceleration techniques, in: D.J. Paddon and M. Holstein, eds., *Multigrid Methods for Integral and Differential Equations* (Claredon Press, Oxford, 1985) 117–167.

[29] H.A. Van der Vorst, Iterative solution methods for certain sparse linear systems with a nonsymmetric matrix arising from PDE-problems, *J. Comput. Phys* **44** (1981) 1–19.

[30] H.A. Van der Vorst, The performance of FORTRAN implementations for preconditioned conjugate gradients on vector computers, *Parallel Comput.* **3** (1986) 49–58.

[31] H.A. Van der Vorst, Large tridiagonal and block tridiagonal linear systems on vector and parallel computers, *Parallel Comput.* **5** (1987) 45–54.

[32] S.G. Tucker, The IBM 3090 system: an overview, *IBM Systems J.* **25** (1) (1986) 4–19.