

A PARALLEL HYBRID SPARSE LINEAR SYSTEM SOLVER

K. GALLIVAN,† A. SAMEH† and Z. ZLATEV‡

†Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign,
 305 Talbot Laboratory, 104 South Wright Street, Urbana, IL 61801, U.S.A.

‡Air Pollution Laboratory, Danish Agency of Environmental Protection, Risoe National Laboratory,
 DK-4000 Roskilde, Denmark

(Received 27 April 1990)

Abstract—Consider the system $Ax = b$, where A is a large sparse nonsymmetric matrix. It is assumed that A has no sparsity structure that may be exploited in the solution process, its spectrum may lie on both sides of the imaginary axis and its symmetric part may be indefinite. For such systems direct methods may be both time consuming and storage demanding, while iterative methods may not converge. In this paper, a hybrid method, which attempts to avoid these drawbacks, is proposed. An LU factorization of A that depends on a strategy that drops *small* non-zero elements during the Gaussian elimination process is used as a preconditioner for conjugate gradient-like schemes, ORTHOMIN, GMRES and CGS. Robustness is achieved by altering the drop tolerance and recomputing the preconditioner in the event that the factorization or the iterative method fails. If after a prescribed number of trials the iterative method is still not convergent, then a switch is made to a direct solver. Numerical examples, using matrices from the Harwell-Boeing test matrices, show that this hybrid scheme is often less time consuming and storage demanding than direct solvers, and more robust than iterative methods that depend on preconditioners that depend on classical positional dropping strategies.

1. THE HYBRID ALGORITHM

Consider the system of linear algebraic equations $Ax = b$, where A is a nonsingular, large, sparse and nonsymmetric matrix. We assume also that matrix A is generally sparse (i.e. it has neither any special property, such as symmetry and/or positive definiteness, nor any special pattern, such as bandedness, that can be exploited in the solution of the system). Solving such linear systems may be a rather difficult task. This is so because commonly used direct methods (sparse Gaussian elimination) are too time consuming, and iterative methods whose success depends on the matrix having a definite symmetric part or depends on the spectrum lying on one side of the imaginary axis are not robust enough. Direct methods have the advantage that they normally produce a sufficiently accurate solution, although a direct estimation of the accuracy actually achieved requires additional work. On the other hand, when iterative methods converge sufficiently fast, they require computing time that is several orders of magnitude smaller than that of any direct method. This brief comparison of the main properties of direct methods and iterative methods for the problem at hand shows that the methods of both groups have some advantages and some disadvantages. Therefore it seems worthwhile to design methods that combine the advantages of both groups, while minimizing their disadvantages.

Throughout we assume that sparse Gaussian elimination is the direct method chosen for the solution of $Ax = b$. It is well known that this is the best choice in the case where A is large and generally sparse; see for example Ref. 1 or 2. The arithmetic

operations during stage k ($k = 1, 2, \dots, n-1$) of Gaussian elimination are carried out by the formula

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - a_{ik}^{(k)}(a_{kk}^{(k)})^{-1}a_{kj}^{(k)}, \quad (1)$$

where $i = k+1, k+2, \dots, n$, $j = k+1, k+2, \dots, n$, while $a_{ij}^{(k)} = a_{ij}$ are the elements of matrix A . It is clear that if $a_{ij}^{(k)} = 0$ while neither $a_{ik}^{(k)}$ nor $a_{kj}^{(k)}$ vanish, then a new non-zero element, *fill-in*, is created in position (i, j) . Unfortunately, fill-in does occur when large sparse matrices are factored by Gaussian elimination and this method becomes rather expensive (in terms of time and storage requirements) when many fill-ins are introduced. Therefore, reducing the number of fill-ins is one of the main tasks during the development of sparse matrix codes, at least on sequential and vector computers. Such minimization of fill-in is achieved by adopting a suitable pivoting strategy; see for example Refs 3-5. The amount of fill-in may be large, however, even when a good pivoting strategy is adopted. For such systems direct methods may lose competitiveness with iterative methods if a convergent method can be found for the system. The successful use of iterative methods often depends upon the effective use of *preconditioning*. Preconditioning a system $Ax = b$ involves using the iterative method to solve the related system $M^{-1}Ax = M^{-1}b$, where the preconditioned M is easily invertible and $M^{-1}A \approx I$. The choice of M is an art in itself and depends on the iterative method used, the application from which the system arises, and, for high-performance machines, the architecture on which the algorithm is to execute.⁶

For the purpose of obtaining an approximate factorization of A for preconditioning, while main-

taining efficiency in the sparse Gaussian elimination process, we attempt to reduce further the number of fill-ins as follows (for more details see Ref. 7). Let τ be a parameter that satisfies

$$0 \leq \tau < 1 \quad (2)$$

and let

$$a_i^{(k)} = \max(|a_{i,k+1}^{(k)}|, |a_{i,k+2}^{(k)}|, \dots, |a_{i,n}^{(k)}|), \quad (3)$$

where the elements over which the maximum is taken form the *active* part of row i at stage k . The parameter τ will be called the *drop-tolerance*, because any element at stage k that satisfies

$$|a_{ij}^{(k)}| \leq \tau a_i^{(k)} \quad (4)$$

is *dropped* (removed from the arrays where the non-zero elements, together with their indices, are kept and neglected in the computations after stage k).

It is clear that by choosing a sufficiently large drop-tolerance τ the number of fill-ins can be reduced considerably. This will lead to an approximate factorization stage in which both the computing time and storage requirements are substantially reduced compared to the classical direct methods. It is also clear, however, that there are two difficulties with this approach: (1) there is no guarantee that the factorization will be completed successfully when a large drop-tolerance is specified; and (2) the solution obtained with the factors calculated by using a large drop-tolerance (assuming that the factorization is successfully completed) will normally be inaccurate.

While it is assured [see Eqs (2)–(4)] that not all the elements in the active part of a row will be removed, it is possible that all non-zero elements in the active part of a column will be dropped when the drop-tolerance is large. Therefore, it is useful to enhance the dropping criterion defined by Eqs (2)–(4) by adding an extra requirement that the dropped non-zero element is not the last one in the active part of its column. In this way, structural singularity is assured at least not to appear at the stage under consideration. Although this enhancement does not guarantee that structural singularity will be avoided throughout the procedure, it works rather well in practice.

Once the approximate factors of A are successfully obtained as a preconditioner, an iterative method must be used in an attempt to obtain a good approximation of the solution. Depending on the drop tolerance, however, the iterative method may not converge. Experiments show, however, that conjugate gradient-type methods perform rather satisfactorily in this situation. Three such methods, ORTHOMIN, GMRES and CGS, are used in the experiments.

The improvement made in the dropping criterion and the use of an iterative method (preconditioned with the factors L and U obtained by Gaussian elimination) enhance the chances of solving the system $Ax = b$ with the desired accuracy with reason-

able efficiency. These two steps so far do not guarantee such success, however. A third step is necessary to yield a solution with the accuracy requested by the user when either the factorization is not completed (due to singularity resulting from dropping too many non-zero elements) or when the preconditioned iterative method is not convergent (or converges too slowly). This third step is rather obvious. If either the factorization process fails or the iterative scheme does not converge, then the drop tolerance must be reduced, new approximate factors L and U computed, and the iterative method restarted. This action can be repeated for a prescribed number of trials after which the drop-tolerance is set equal to zero, i.e. switching to a direct method. Let $\tilde{A}(\tau)$ denote the matrix such that, in the absence of rounding errors, $\tilde{A}(\tau) = LU$ is formed by performing an approximate factorization of A using drop-tolerance τ (permutations required to form L and U have been ignored for simplicity of presentation). Note that $\tilde{A}(\tau) \rightarrow A$ as $\tau \rightarrow 0$. The hybrid solver can then be described as follows:

```

DROP TOLERANCE  $\tau$  IS GIVEN
DESIRED ACCURACY  $\epsilon$  IS GIVEN
DO UNTIL (X IS ACCEPTED)
  IF ( $LU = \tilde{A}(\tau)$ ) EXISTS THEN
     $M \leftarrow LU$ 
     $x \leftarrow (LU)^{-1}b$ 
    CALL PCG_TYPE_METHOD ( $M, A, x, b, \epsilon$ )
    IF (NOT CONVERGED OR TOO SLOW) THEN
       $\tau \leftarrow p_1(\tau)$ 
    END IF
  ELSE
     $\tau \leftarrow p_2(\tau)$ 
  END IF
END DO

```

The functions $p_1(\tau)$ and $p_2(\tau)$ are functions that adjust the value of τ given an unsatisfactory performance by the iterative method and an unsuccessful factorization, respectively. The outer loop around the classical form of preconditioning which makes use of the two reduction functions yields a robust algorithm—in the worst case a direct method will eventually be used. By recomputing the preconditioner with smaller τ when the iterative method does not appear to be performing well we avoid the use of a poor preconditioner and the subsequent inefficiency. The adaptive behavior of the algorithm can therefore be used, starting with a relatively large initial τ , to allow the algorithm to find a drop tolerance that is natural for the problem. The early iterations with large τ require some extra time but the fact that many elements are dropped reduces the number of operations performed (significant for a single processor) and provides more opportunity for the creation of parallel pivot sets (important for parallel processors). The effort is usually repaid with rapid convergence of the iterative method and can be very worthwhile if a sequence of problems is to be solved with similar matrices, i.e. those with effective values of τ that are about the same.

Assuming that the above steps are properly incorporated in a code, we must show the robustness and effectiveness of the hybrid. Below we illustrate by numerical examples that (see also Ref. 7):

- (1) this hybrid method is more robust than other preconditioned iterative schemes available in the literature, and often there is no need for recalculating the approximate factors, and in this case the hybrid scheme is much faster than direct methods;
- (2) the global computing time of the hybrid scheme is less than that of direct methods even if one has to recalculate the approximate factors once or twice;
- (3) even in the worst case (when we have to switch to a direct method), the increase of the computing time is not that high.

It is important to carry out the third step (the reduction of τ and refactorization) properly. This step is rather time-consuming and, therefore, should be performed only when either the factorization process with a given drop tolerance fails, or when the iterative method does not converge. In the first case, the third step is activated when the code detects a zero column or row. In the second case, failure of the iterative scheme, it is crucial to determine when to abandon the iterations and to resort to determining new approximate factors with a more stringent drop-tolerance. In the next section we shall show that it is possible to satisfy these two requirements and, thus, it is possible to implement all three steps efficiently.

2. THE STOPPING CRITERIA

Designing stopping criteria for iterative methods applied to systems $Ax = b$, where A is a general matrix, is a critical task. Since Krylov sub-space methods (see above conjugate gradient-type methods) are not guaranteed to converge for general linear systems, the first task is to determine whether the iterates are converging.

The residual vector r_i defined by

$$r_i = b - Ax_i = A(x - x_i), \quad (5)$$

where x_i is the i th iterate, is often used in formulating stopping criteria. It is clear from Eq. (5) that some norm of the residual vector may provide a reliable estimate of the norm of the error if the norm of matrix A (corresponding to the vector norm chosen) is, roughly speaking, of order 1.

If $\|A\|$ is large, a stopping criterion based on the use of $\|r_i\|$ may not detect that the error in the solution, $\|x - x_i\|$, is small. This could be illustrated by taking only one equation with $A = b = 10^{10}$. Assume that the accuracy required is $ACCUR = 10^{-4}$ and that the current approximation x_i is such that $|x - x_i| = 10^{-10}$. Then $|r_i| = 1$ and a stopping criterion based on the use of the residual will be misleading.

If $\|A\|$ is small, a stopping criterion based on the use of $\|r_i\|$ does not yield useful information about the

error of the approximation, $\|x - x_i\|$. To illustrate this, consider again one equation only, this time with $A = b = 10^{-10}$. Let the accuracy required be again defined by $ACCUR = 10^{-4}$. Assuming that the current approximation x_i is such that $|x - x_i| = 10^5$, then $|r_i| = 10^{-5}$ and a stopping criterion based on the use of the residual alone will be misleading.

The examples given above are extreme (and in practice the situation will often be better). Nevertheless, these examples indicate that it is not a good idea to have the matrix A (or some norm of this matrix) involved, directly or indirectly, in the stopping criteria.

An interesting attempt to eliminate the influence of A is made in the so-called simple version of GMRES (the theoretical background of code GMRES is discussed in Ref 8). In this code the iterations are stopped if

$$\frac{\|r_i\|}{\|r_0\|} \leq ACCUR, \quad (6)$$

where $ACCUR$ is the accuracy required by the user. Assume that

$$\|A(x - x_i)\| \approx \|A\| \|x - x_i\|, \quad (7)$$

where $i = 0, 1, \dots$. Under this assumption

$$\frac{\|r_i\|}{\|r_0\|} \approx \frac{\|x - x_i\|}{\|x - x_0\|} \quad (8)$$

and, if the starting approximation x_0 is equal to zero, then an attempt to evaluate the relative error is made by the stopping criterion in GMRES (at the same time eliminating the influence of matrix A). While the assumption under which this criterion works, Eq. (7), is not unrealistic, there will be difficulties when the starting approximation is close to the exact solution. This could be the case, for example, when solving time-dependent problems.

It follows then that it is worthwhile to determine whether or not the iterative process is convergent without using the matrix A in the criterion developed. In many iterative processes the new approximation is obtained from the old one by using a simple transformation:

$$x_{i+1} = x_i + \alpha_i p_i, \quad (9)$$

where $i = 0, 1, \dots$, α_i are constants and p_i are certain correction vectors. Based on this observation, a convergence check with the desired properties can be derived.

Let us assume that Eq. (9) holds for the iterative method selected. This is true for ORTHOMIN (see Ref. 9) and for CGS (see Ref. 10), but not for the GMRES (see Ref. 8). Let us assume also that the computations with the above formula are carried out without rounding errors. Then, if the iterative method under consideration is convergent, the exact solution x of the system $Ax = b$ can be written as

$$x = x_i + \sum_{j=i+1}^{\infty} \alpha_j p_j. \quad (10)$$

and an upper bound on the norm of $x = x_i$ can, under some mild assumptions, be given by

$$\|x - x_i\| \leq |\alpha_{i+1}| \|p_{i+1}\| \left[1 + \sum_{j=i+2}^{\infty} \frac{|\alpha_j| \|p_j\|}{|\alpha_{i+1}| \|p_{i+1}\|} \right]. \quad (11)$$

Consider a real constant *RATE* such that

$$0 < \text{RATE} < 1 \quad (12)$$

and let for $j = -2, -1, 0$

$$\text{RATE}_j = 0.999 \text{ RATE}. \quad (13)$$

Define for $i = 1, 2, \dots$ the quantities RATE_i and MEANRATE_i by the recurrences

$$\text{RATE}_i = \frac{|\alpha_i| \|p_i\|}{|\alpha_{i-1}| \|p_{i-1}\|} \quad (14)$$

and

$$\text{MEANRATE}_i = 0.25(\text{RATE}_i + \text{RATE}_{i-1} + \text{RATE}_{i-2} + \text{RATE}_{i-3}). \quad (15)$$

Consider also an integer variable *NBAD*, which is initially set to zero and is updated at each iteration by

$$\text{NBAD} = \begin{cases} \text{NBAD} + 1 & \text{if } \text{MEANRATE}_i > \text{RATE} \\ 0 & \text{otherwise.} \end{cases} \quad (16)$$

We therefore have adopted the following criterion.

Convergence criterion. The iterative process defined by Eq. (9) is assumed to be converging at a sufficient rate if $\text{NBAD} \leq \sigma$, where σ is some small positive integer (presently in our code, $\sigma = 5$).

Normally much more stringent requirements are imposed in the construction of a convergence criterion. For example, simple iterative refinement (see Ref. 5 or 11) is often stopped when $\text{RATE}_i > \text{RATE}$. Such a criterion is rather restrictive in connection with conjugate gradient-type methods and very often stops the iterative process very early and, what is even more important, unnecessarily (because the process will often converge, although slowly, even when $\text{RATE}_i > \text{RATE}$ is occasionally true).

An attempt to evaluate the rate of convergence, especially for iterative refinement, is made by using the parameter *RATE*. In connection with conjugate gradient-type methods, however, it is often useful to set *RATE* close to 1 (*RATE* = 0.999 is used in our experiments). This is so because conjugate gradient-type methods often converge very slowly at the beginning with remarkable improvements later. In fact, these slow and fast regimes of convergence could occur several times over the course of solving a system. On the other hand, when solving time-dependent problems, for example, it is advisable to keep the value of *RATE* small. The code in such a case will, after several trials and reductions of τ , find some sufficiently accurate preconditioner for which

the convergence rate is fast from the beginning, and use it in several time-steps.

The second task which must be considered is to terminate the iterations when the desired accuracy is achieved. Let us assume that the iterative process converges (according to the convergence criterion given above). Let the stopping criterion be given by $\|x - x_i\| \leq \text{ACCUR}$. Equation (11) requires the obtaining of an estimate of the sum in order to obtain an accuracy check; such an estimate may be obtained as follows. Define

$$\text{MAXRATE}_i = \max(\text{RATE}_i, \text{RATE}_{i-1}, \text{RATE}_{i-2}, \text{RATE}_{i-3}) \quad (17)$$

and let (for some $q = i, i = 1, 2, \dots$)

$$\text{MAXRATE}_q \leq \text{RATE}. \quad (18)$$

If we assume that

$$\text{RATE}_j \leq \text{MAXRATE}_q \text{ for } j = q-3, q-2, q-1, q \quad (19)$$

implies

$$\text{RATE}_j \leq \text{MAXRATE}_q \text{ for } j = q+1, q+2, \dots, \infty, \quad (20)$$

then it can easily be shown that Eq. (11) reduces to

$$\|x - x_q\| \leq \frac{|\alpha_q| \|p_q\|}{1 - \text{MAXRATE}_q}. \quad (21)$$

As a result, we have the following stopping criterion based on the accuracy required.

Acceptability criterion. Let $0 \leq \text{RATE} < 1$ and $\text{ACCUR} \leq 0$ be given real numbers and assume that at the end of some iteration q we have $\text{MAXRATE}_q \leq \text{RATE}$. Then the iterative process is terminated if

$$\frac{|\alpha_q| \|p_q\|}{1 - \text{MAXRATE}_q} \leq \text{ACCUR}. \quad (22)$$

It is clear that one should expect $\|x - x_q\| \leq \text{ACCUR}$ to be satisfied on successful exit when the above acceptability criterion is in use.

We have sketched above the ideas used rather than the more complex termination criteria actually used in our code. For example, in an attempt to enhance the possibility that assumptions made by Eqs (19)–(20) actually hold, we also require that all of the last four values of α_i are close to

$$0.25(\alpha_i + \alpha_{i-1} + \alpha_{i-2} + \alpha_{i-3}). \quad (23)$$

Equation (22) is checked only when this condition is satisfied.

When the ORTHOMIN algorithm is chosen as the iterative method, a special action is carried out at each iteration. The number of vectors in the Krylov subspace is increased by one if $\text{MEANRATE}_i > \text{RATE}$. (The user provides a parameter *MAXCOL* which determines the maximum dimension of the

Krylov subspace allowed.) In this way the code attempts to automatically increase the possibility of achieving convergence. Experiments indicate that this is a very useful device. This is in part due to the fact that a large initial Krylov dimension is a waste of time if the iterative process converges quickly with a subspace containing one or two vectors only; and for any particular problem one normally does not know in advance the optimal number of vectors in the Krylov subspace for which the ORTHOMIN converges (and thus it is better to leave to the code the task of finding a nearly optimal number of vectors for the Krylov subspace). This idea could also be used with other truncated conjugate gradient-type methods. For CGS, of course, such a device cannot be applied. It is yet not clear to us how to apply the idea to methods based on restarting the iterations (such as GMRES).

The convergence criterion, or more precisely the non-termination criterion, is rather lenient. The code will try to carry out the computations in the iterative method as long as possible. This is in some sense justified, because it is rather expensive to stop the iterations unnecessarily since it will require refactorization. On the other hand, the acceptability criterion is rather conservative: the codes usually perform several extra iterations (and compute solutions the accuracy of which is very often greater than the desired accuracy). However, this is typically not an excessive amount of extra work: the cost per iteration is normally very low. This means that we invest some more work in an attempt to prevent an unnecessary application of costly refactorization and to ensure that the accuracy required is achieved.

By checking the norms of the residual vectors and the correction vectors, r_i and $\alpha_i p_i$, respectively, we have observed that, when the matrix A is badly scaled and/or ill-conditioned, these quantities may be small, while the norm of the error vector $x - x_i$ is considerably greater than the accuracy requirement. However, in all such situations that have been observed in our experiments, the stopping criteria indicate correctly that the iterative process must be continued because the accuracy desired has not been achieved.

Finally, we have not discussed the alterations to the termination criteria which introduce the cost of refactorization and the cost per iteration into the decision to reduce τ . This is discussed in Ref. 7.

3. THE CODES

The hybrid algorithm described above has been implemented in a code for the Alliant FX/80 which is based on a modified version of the direct method package for general sparse matrices Y12M,^{2,5,12} and three iterative solvers based on the algorithms ORTHOMIN, GMRES and CGS (see Refs 8, 9 and 10, respectively) adapted to use the approximate factorization and initial guess gener-

ated by Y12M. Stopping criteria based on the ideas discussed in the previous section have been implemented in ORTHOMIN and CGS, while the original stopping criterion is used in GMRES. A moderate amount of optimization for the multi-vector processing capabilities has been performed on the version used to generate the results presented below. For more details concerning the effect of more aggressive optimization and the algorithm/architecture mapping, see Ref. 7.

The input stage of the package has been organized in a user-friendly fashion. The matrix A is input in simple triplet form, i.e. a series of triples (α_{ij}, i, j) in an arbitrary order. The user is not required to order the non-zero elements by rows or by columns nor to count the non-zero elements per each row or column. The code converts this storage scheme into two others required for the computations. The first is a static structure which preserves the original values of A needed for the iterative methods in a way which supports the computational primitives found in that portion of the code. The non-zero elements of matrix A are stored in an array *AORIG*, grouped by rows, i.e. non-zero elements of the first row, then the non-zero elements of the second row, etc. The order of the non-zero elements within a row is arbitrary and there are no free locations between any two rows. The column numbers associated with the elements are kept in an array *CNORIG* in corresponding positions—if a non-zero element a_{ij} is stored in *AORIG*(K), then its column number j is stored in *CNORIG*(K). Pointers into *AORIG* to first and last positions of each row of A are also prepared. This static storage scheme supports the standard vector-vector multiplication primitive (parallel sparse dot products, each of which is vectorized within a processor) required by the iterative method. If a different form of this primitive is used to improve performance of the iterative methods, e.g. those discussed in Ref. 13, this structure must be updated appropriately. A second dynamic structure is required to compute the approximate factorization of $\hat{A}(\tau)$. It must be dynamic due to the elements added through fill-in and removed through dropping. The second structure is initialized by making a copy of the static structure sketched above and a second column-oriented version of the structure of A . In this structure, the row numbers of the non-zero elements are grouped by columns, i.e. the row numbers of the non-zero elements in the first column, then the row numbers of the non-zero elements in the second column, etc. Pointers into this structure to the first and last elements in each column are also kept. The column-oriented structure contains no numerical information; it is used to facilitate the search for a pivot element at each stage.

The version of the code used to generate the results below is based on exploiting the parallelism involved in a single stage of sparse Gaussian

elimination. The major phases of each stage are as follows:

- (1) pivotal search
- (2) interchanges
- (3) symbolic factorization in the column- and row-oriented structures
- (4) numerical rank-1 modification.

In the pivotal search, a set of the *best* candidate rows is assembled (a few rows that contain a relatively small number of non-zero elements; see Ref. 4). This operation can be carried out in a vector-concurrent mode. The simplest way to do this is to search all rows in the active part of the matrix at the stage under consideration. This gives $O(n^2)$ operations. On sequential machines it is better to order the rows in increasing number of non-zero elements in the beginning and then at each stage to update this order: this gives about $O(nq)$ operations, where q is the average number of non-zero elements in the active parts of the pivotal columns. Interestingly, a long series of experiments indicated that the simpler algorithm performs satisfactorily. Our experiments were with matrices of up to order $n = 10^4$. As n increases significantly beyond this point, however, a parallel version of the second algorithm should be preferable.

The interchanges which permute the chosen pivot element α_{ij} to the (k, k) position in the k th stage can be carried out in parallel in a straightforward manner. To see this, consider the row-oriented structure. We must scan each target row (a row that has a non-zero element, in the pivotal column) for elements in columns j and k and relabel them appropriately. The interchanges in each target row are completely independent of the interchanges in the other target rows, and thus can be carried out concurrently. Similar considerations apply to the column-oriented structure.

Once the pivot column and row are identified and permuted a *symbolic factorization* is performed. This calculates the location of fill-ins in the target rows and columns involved in the rank-1 update and updates the dynamic data structure containing $\bar{A}(\tau)$. The advantage of using the symbolic factorization at each stage is that the numerical update of the active portion of the matrix can be carried out using vectorization and concurrency. This is different from the original version of Y12M,² where no symbolic factorization is carried out and the code searches for a place to locate the fill-in immediately when it discovers that a fill-in is to be created. The symbolic factorization is performed sequentially in the code whose results are presented below but it also lends itself to vector and concurrent processing (see Ref. 7 for details).

The main computational primitives required by each step of a preconditioned conjugate gradient-type algorithm are as follows:

- (1) matrix-vector multiplication
- (2) solving systems with triangular matrices

(3) vector triads

(4) norms (inner products) of a vector (two vectors).

The last two operations are simple BLAS1 level primitives (see Ref. 14), and the main implementation question is whether the entire machine should be used to compute them or whether they should be merged with one of the other phases into a more complicated primitive. The first is a BLAS2 operation and is well understood on Alliant-like architectures (see Refs 15 and 16). If the matrix is sparse, then various approaches are possible. For example, the simplest approach is to use parallel sparse dot products, each of which are vectorized as mentioned by Dodson and Lewis.¹⁷ Depending on the matrix A , however, other forms may be more efficient on architectures like the Alliant FX/80 (see Ref. 13).

The second primitive is the source of most of the performance questions for a preconditioned method since it tends to be the most time-consuming. Parallelism and vectorization can be exploited if the structures of the factors L and U are examined before the start of the iterative process in order to create a parallel schedule of row operations, typically by a simple levelization of the computational dependence graphs. Our code uses a modified version of the code developed by Anderson¹⁸ (see also Ref. 19).

4. ORGANIZATION OF THE NUMERICAL EXPERIMENTS

The numerical experiments whose results are presented in the next section were generated using matrices from the well-known Harwell-Boeing set of test matrices prepared by Duff *et al.*^{20,21} The three main purposes of the numerical experiments are as follows.

- (1) We show that the method sketched in the previous sections often performs better than direct methods. Results from Y12M and two other well-known packages, SPARSPAK-C and MA28 (see the works of Østerby and Zlatev,² Duff²² and George and Ng,²³ respectively) are presented, the first modified for the Alliant as discussed above, and the rest using compiler generated parallelism and vectorization only.
- (2) We also show that the method is much more reliable than using the more standard *incomplete LU factorization* preconditioning which is based on positional dropping rather than numerical dropping. A code developed for the Alliant is used.¹⁸
- (3) The reliability of the stopping criteria will be illustrated by showing the accuracy estimated by the code and the accuracy actually achieved.

The matrices selected for the experiments are all unsymmetric. Most have an order greater than 1000 and those with a smaller order are such that they produce a large amount of fill-in when a direct solver is used. Some of the matrices used in our previous report²⁴ are not used here and improvements to

the code used in that work, both to Y12M and to the iterative method portion of the code, have been made. Moreover, the new and more efficient option of MA28, where a pivotal strategy based on the algorithm proposed by Zlatev⁴ is implemented, is used in the present experiments. The parameter *NTOL* in SPARSPAK-C (by which one determines when a row of the matrix should be considered as dense and treated in a special way) was set to 100, while *NTOL* = 25 was used in Ref. 24. However, results with *NTOL* = 25 are also presented in order to demonstrate the fact that this code may sometimes be sensitive to the choice of this parameter.

The drop-tolerance τ was set to 2^{-4} for initially calculating preconditioners for the conjugate gradient methods. An element is dropped from the active part of the matrix if (4) is satisfied after completion of the rank-1 update. The drop-tolerance is reduced by a factor of 2^{-10} when the factorization of $\tilde{A}(\tau)$ does not succeed and by a factor of 2^{-5} when there are difficulties with the convergence of the iterative method. The calculation of several preconditioners may require several such trials and the computing times reported are the sum of the computing times for all trials required for a particular matrix.

Difficulties in the calculation of the preconditioners very often appear because the columns of the matrices are very badly scaled (and the code throws away all non-zero elements in a column). In an attempt to reduce the possibility for appearance of such a situation, we perform an initial scaling of the matrices (the time needed to scale is included in the computing times given). For the matrices tested scaling yields rather good results with regard to avoiding unnecessary reductions of the drop-tolerance. Two extra benefits due to the scaling procedure are also observed. The first is connected with the preservation of the sparsity (when the matrix is scaled the number of candidates for pivots is normally increased and this leads to sparse factors L and U , even if the drop-tolerance is set to zero). The second benefit is connected with the condition number. Very often the condition number of the scaled matrix is smaller than the condition number of the original matrix, and therefore the iterative processes converge faster, as a rule, when scaling is in use.

Some characteristics, including the condition number $\kappa(A)$, of the matrices used are presented in Table 1. We believe that experiments using these matrices are representative. There are matrices that are very sparse (**west2021** has about three elements per row on average) and matrices that have many elements initially (**mc-fe** contains more than 30 elements per row on average). There are both very well conditioned matrices (**steam2**) and very ill-conditioned matrices (**nnc1374**). For some matrices the condition numbers are reduced considerably after application of the scaling procedure. For

Table 1. Matrices used in the experiments

Matrix	Order	Non-zeros	$\kappa(A)$
steam2	600	5660	$3.2\text{E} + 0$
mc-fe	765	24,382	$7.7\text{E} + 1$
sherman2	1000	23,094	$2.6\text{E} + 3$
pores-2	1224	9613	$3.6\text{E} + 5$
nnc1374	1374	8588	$2.0\text{E} + 13$
hwatt-1	1856	11,360	$4.7\text{E} + 3$
hwatt-2	1856	11,550	$3.1\text{E} + 5$
west2021	2021	7310	$4.8\text{E} + 7$
orsreg-1	2205	14,133	$8.2\text{E} + 3$
or678lhs	2529	90,158	$2.1\text{E} + 6$
sherman5	3312	20,793	$6.0\text{E} + 3$
saylr4	3564	22,316	$1.2\text{E} + 7$
gemat11	4929	33,108	$1.6\text{E} + 6$
gemat12	4929	33,044	$1.8\text{E} + 6$
sherman3	5005	20,033	$1.9\text{E} + 5$

example, the condition number estimation for **sherman3** is $6.9\text{E} + 16$ when scaling is not applied. The matrices vary in the amount of fill-in produced (**west2021** produces very little fill-in while **saylr4** produces significantly more). Finally, while results from 15 matrices are presented, the conclusions are based on experiments with several hundred matrices (including some artificially created matrices created using the matrix generators from Ref. 2).

5. EXPERIMENTAL RESULTS

In this section the numerical results obtained by using the test matrices listed in Table 1 are discussed. As mentioned earlier, the conclusions are drawn by using a much larger set of test matrices, but the matrices chosen illustrate correctly the different situations that could arise in practice; situations in which the hybrid methods are very successful or in which the direct methods or the purely iterative methods should be preferred.

5.1. The performance of the direct solvers

The computing times achieved with the three direct solvers, the new version of MA28, SPARSPAK-C with *NTOL* = 1000 and the new version of Y12M (denoted Y12M1), are given in Table 2. The numbers of non-zeros in the factors L and U are given in Table 3 and the accuracy achieved is recorded in Table 4.

Consider first the accuracy results for the direct methods. Surprisingly, the results obtained by SPARSPAK-C, which uses partial pivoting, tend to be less accurate than the results obtained by the other two codes. The cause of this behavior is not clear. It could be due to the fact that an improved generalized Markowitz pivotal strategy is used in MA28 and Y12M1. In this strategy some features of the complete pivoting for dense matrices are applied on a subset of rows of the active part $A^{(k)}$ of matrix A at each stage k of Gaussian elimination. (See Theorem 1 in Ref. 4 or the corresponding theorem

Table 2. Computing times (s) for the direct method codes

Matrix	MA28	SPARSPAK	Y12M1
steam2	16	62	5
mc-fe	41	132	17
sherman2	361	254	107
pores-2	42	22	13
nnc1374	113	50	31
hwatt-1	91	107	43
hwatt-2	99	100	42
west2021	4	13	2
orsreg-1	137	214	65
or678lhs	174	195	75
sherman5	286	141	61
saylr4	353	312	147
gemat11	14	71	13
gemat12	14	72	12
sherman3	237	238	95

Table 3. Non-zero elements in LU for the direct method codes

Matrix	MA28	SPARSPAK	Y12M1
steam2	27,625	40,809	21,207
mc-fe	61,376	59,424	61,419
sherman2	209,648	301,610	177,712
pores-2	66,452	49,913	53,371
nnc1374	123,202	66,599	67,011
hwatt-1	113,711	211,150	117,103
hwatt-2	118,343	200,977	114,771
west2021	14,576	10,574	8958
orsreg-1	151,150	312,650	158,085
or678lhs	146,245	188,606	133,715
sherman5	223,967	229,912	149,593
saylr4	311,379	489,256	308,455
gemat11	52,826	70,630	45,987
gemat12	54,403	72,905	49,118
sherman3	218,315	350,046	210,394

Table 4. Accuracy achieved with the direct method codes

Matrix	MA28	SPARSPAK	Y12M1
steam2	7.1E-13	1.1E-12	1.2E-15
mc-fe	5.3E-15	6.8E-14	2.1E-14
sherman2	1.5E-11	3.2E-7	3.2E-13
pores-2	2.8E-13	8.6E-12	1.5E-12
nnc1374	1.3E-2	2.2E-3	9.1E-5
hwatt-1	1.3E-14	2.5E-15	8.4E-15
hwatt-2	1.2E-14	7.6E-14	1.2E-14
west2021	2.8E-10	7.1E-10	1.2E-11
orsreg-1	1.9E-13	3.1E-13	2.2E-13
or678lhs	2.0E-14	1.4E-8	4.8E-14
sherman5	1.4E-14	8.9E-14	1.4E-14
saylr4	7.9E-11	8.0E-12	5.6E-11
gemat11	6.4E-12	5.2E-11	3.0E-12
gemat12	2.5E-11	1.6E-10	2.0E-12
sherman3	2.6E-13	4.0E-13	8.8E-14

and the numerical results in Ref. 2.) More experiments are needed in order to confirm this conjecture, but it should be noted here that if the option in which the classical Markowitz strategy is used is chosen with MA28, then the results obtained by this code become, in general, less accurate than those obtained by SPARSPAK-C (and thus, by partial

pivoting), as should be expected. Scaling is used with Y12M1 but not with the other codes. As expected, the results show that scaling does not, in general, lead to better accuracy. Recall that the motivation for scaling was to facilitate a more effective choice of drop-tolerance.

Comparing the number of non-zero elements in L and U , it is seen that as a rule this number is considerably greater for SPARSPAK-C. Of course, this should be expected: it is well-known that the pivotal strategies of the Markowitz will normally preserve the sparsity better than partial pivoting. The reason that the numbers of non-zeros in L and U for Y12M1 tend to be smaller than those for MA28 is probably due to the fact that scaling is used with the former package and as a result the set of candidates for pivots for the scaled matrix tend to be, at each stage k , greater than that for the original matrix. However, it should be noted that the differences are not very large.

The computing times for Y12M1 are much smaller than those for MA28 and SPARSPAK-C. This is expected since Y12M1 has been somewhat modified to exploit parallel and vector processing while the others rely only upon the restructuring compiler. The poor performance of the latter codes indicates the inability of restructuring compilers to achieve much with standard sequential sparse solvers. If no optimization is used, i.e. both vectorization and concurrency are suppressed, then the computing times for Y12M1 become two to four times greater. The results of experiments with concurrency and vectorization suppressed on Y12M1, but not for MA28 or SPARSPAK-C, are given in Table 5. The results for MA28 in this table are obtained with the old pivotal strategy (the classical Markowitz). Comparing the results with those in Table 2, one can see the efficiency of the pivotal strategy based on Theorem 1 in Ref. 4. The newer pivotal strategy also often gives better accuracy and a smaller number of non-zeros in L and U .

The effects of the choice of $NTOL$ in SPARSPAK-C are also seen in Table 5. The results there were obtained with $NTOL = 25$ while those in Tables 1-3 were obtained with $NTOL = 100$. The

Table 5. Computing times (s) on one processor for the direct method codes

Matrix	MA28	SPARSPAK	Y12M1
sherman2	580	1052	362
pores-2	61	22	39
nnc1374	224	42	62
hwatt-1	437	107	129
hwatt-2	406	100	127
west2021	31	13	8
orsreg-1	140	197	195
sherman5	361	123	204
saylr4	1147	293	455
sherman3	847	215	309

results indicate that SPARSPAK-C may be very sensitive to the choice of $NTOL$. For the matrix **sherman2** the use of $NTOL = 25$ instead of $NTOL = 100$ gives a factor of 4 increase in computing time and a factor of 3 increase in the number of non-zeros in L and U . There is, on the other hand, a corresponding improvement in the accuracy achieved (the max-norm of the error-vector was $1.9E-10$). Various experiments with SPARSPAK-C were run with $NTOL$ equal to 25, 50 and 100 and the overall performance was best with $NTOL = 100$.

5.2. The performance of the hybrid algorithm

Table 6 shows the computing times achieved by the hybrid algorithm using preconditioned ORTHOMIN as the iterative method and by Y12M1. Recall that the initial value of the drop-tolerance is $\tau = 2^{-4}$ and it is reduced by a factor of 2^{-10} when the factorization fails and by a factor 2^{-5} when the iterative process does not converge fast enough. The numbers of trial factorizations are also given in this table.

The results shown, together with many other experiments, admit several conclusions. It is seen that the reduction in computing time can be greater than one order of magnitude (see the results for **sherman2** and **saylr4**). For very ill-conditioned problems (e.g. **nnc1374**), or for problems that stay very sparse during the factorization (e.g. **west2021**, **gemat11** and **gemat12**), direct methods (Y12M1) are superior. The hybrid tends to perform best for large time-consuming problems, i.e. precisely for problems where improving the performance is most wanted. This is due to the fact that such problems are normally time-consuming due to a large amount of fill-in during the factorization, many of which are dropped when a positive value for the drop-tolerance is used. If the problem solved is time-consuming, then the hybrid can be better than direct methods (Y12M1) even if more than one trial is required to

Table 6. Computing times (s) for Y12M1 and ORTHOMIN hybrid (iterations)

Matrix	Y12M1	ORTMIN	Trials
steam2	5	1(3)	1
mc-fe	17	3(7)	1
sherman2	107	9(4)	2
pores-2	13	4(64)	1
nnc1374	31	35(3)	5
hwatt-1	43	6(20)	1
hwatt-2	42	7(38)	1
west2021	2	5(3)	2
orsreg-1	65	5(38)	1
or678lhs	75	10(13)	1
sherman5	61	7(21)	1
saylr4	147	9(46)	1
gemat11	13	20(5)	2
gemat12	12	20(5)	2
sherman3	95	21(73)	1

Table 7. Non-zeros in LU for Y12M1 and ORTHOMIN hybrid

Matrix	NZ	Y12M1	ORTMIN
steam2	5660	21,207	1050
mc-fe	24,382	61,419	6948
sherman2	23,094	177,712	19,680
pores-2	9613	53,371	4665
nnc1374	8588	67,011	36,772
hwatt-1	11,360	117,103	13,857
hwatt-2	11,550	114,771	13,979
west2021	7310	8958	8670
orsreg-1	14,133	158,085	14,133
or678lhs	90,158	133,715	9682
sherman5	20,793	149,593	12,810
saylr4	22,316	308,455	9915
gemat11	33,108	45,987	44,876
gemat12	33,044	49,118	45,096
sherman3	20,033	210,394	16,384

compute a successful preconditioner (see the results for **sherman2**).

Table 7 lists the number of non-zero elements in the factors L and U for the Y12M1 and for the hybrid. In order to facilitate the comparison, we also list the numbers of non-zero elements, NZ , in the original matrices. It is seen that the number of non-zero elements may be reduced more than 10 times when a positive drop-tolerance is used. It is also seen that the number of non-zero elements in the factors L and U may be smaller than NZ when using the hybrid. This is due to the fact that the code performs a scan of the non-zero elements before the start of the factorization of $\hat{A}(\tau)$ and removes all non-zero elements that satisfy the drop-tolerance relation. The fact that the number of non-zero elements could be kept very small also allows us to solve on the Alliant some very large problems that cannot be solved by Y12M1. An example of such a problem is the largest problem in the Harwell-Boeing set, **bcsstk32**, which is a system of order 44,609 with 2,014,701 non-zeros when the symmetry is not exploited.

In Table 8 the accuracy achieved by Y12M1 is compared with that achieved by the hybrid method. The column labeled "Actual" is the true error in the solution computed by the hybrid and the column labeled "Evaluated" is the error which the preconditioned ORTHOMIN code thought it had achieved. It is seen that Y12M1 gives greater accuracy; however, more important here is the fact that the accuracy requirement imposed in all runs (to calculate a solution such that the norm of the solution vector is smaller than 10^{-4}) is always achieved. The error estimates calculated by the code are, as a rule, less than the actual errors found (this shows that the acceptability criterion in the code is rather cautious). If the number of iterations is small, then the accuracy achieved is much better than the required accuracy, because the code studies the behavior of certain parameters during several successive iterations in order to decide whether the computations should be

Table 8. Accuracy for Y12M1 and ORTHOMIN hybrid (iterations)

Matrix	Y12M1	Actual	Evaluated
steam2	1.2E-15	7.2E-16	9.5E-8(3)
mc-fe	2.1E-14	5.1E-7	2.1E-5(7)
sherman2	3.2E-13	2.7E-13	1.8E-9(4)
pores-2	1.5E-12	3.9E-8	1.1E-6(64)
nnc1374	9.1E-5	6.1E-7	3.1E-8(3)
hwatt-1	8.4E-15	1.6E-5	8.3E-5(20)
hwatt-2	1.2E-14	1.1E-6	4.9E-5(38)
west2021	1.2E-11	7.3E-6	5.0E-5(3)
orsreg-1	2.2E-13	5.8E-7	9.9E-5(38)
or678lhs	4.8E-14	8.7E-6	9.1E-5(13)
sherman5	1.4E-14	5.0E-7	2.7E-6(21)
saylr4	5.6E-11	5.7E-7	9.4E-6(46)
gemat11	3.0E-12	4.9E-11	1.1E-8(5)
gemat12	2.0E-12	4.9E-9	6.7E-7(5)
sherman3	8.8E-14	2.2E-7	6.2E-5(73)

stopped or not and, if the convergence rate is very fast, then the accuracy achieved when this study is completed is usually greater than the required accuracy. If the iterative process converges slowly, then normally the accuracy achieved is only slightly better than the required accuracy. This means that if the iterative process is time-consuming, then the code does not carry out many extra iterations before the decision to stop the calculations.

Three hybrid solvers, based on ORTHOMIN, GMRES and CGS, respectively, are compared with regard to the computing time in Table 9. While it is too early to draw any final conclusions since the work with ORTHOMIN is more advanced than that with the other two solvers, it is seen that for any of the three methods there are problems for which it performs best. However, note that the differences are not very big and will probably become smaller when GMRES and CGS are optimized. (Their performances will undoubtedly improve with tuning.)

The three conjugate gradient-type methods, ORTHOMIN, GMRES and CGS, have also been used as pure iterative methods. If the convergence rate is fast, then this is a good choice (both computing time and storage being saved). However, for many of the tested problems the pure iterative methods converge very slowly or do not converge at all. Of course, this should be expected: the theory of

these methods tells us that convergence is guaranteed for special matrices only, e.g. for matrices whose symmetric part is positive definite.

The use of iterative refinement with approximate factors L and U generated via numerical dropping was proposed in Ref. 25. It was used successfully in the numerical treatment of some large problems arising in nuclear magnetic resonance spectroscopy (see e.g. Ref. 5). Since 1983, iterative refinement has also been implemented in other sparse matrix codes (as, for example, in MA28; see Ref. 1). Our experiments show that even with the improvement proposed here and in Ref. 24, e.g. the replacement of the absolute drop-tolerance with a relative one, the performance of iterative refinement is often inferior to that of the hybrid using CG-type methods in the sense that the drop-tolerance needed to obtain a convergent process when the iterative refinement is used is normally less than that needed to obtain a convergent iterative process when the preconditioned CG-type methods are used. This leads to using more time to calculate the preconditioners and to using larger total time to solve the problem. The fact that the cost per iteration for the iterative refinement is less than the cost per iteration for the preconditioned CG-type methods is normally not sufficient to compensate for the increase of time for calculating the preconditioners with a smaller drop-tolerance. Of course, for those few problems where iterative refinement solves the problem with the same drop-tolerance and with a similar number of iterations as the preconditioned CG-type method, then it is preferable.

5.3. The performance of GMRES with ILU preconditioning

The code of Anderson,¹⁸ developed for the Alliant and in which GMRES with preconditioning by an incomplete LU is used, has been compared with the proposed hybrids based on numerical dropping. The incomplete LU (ILU) preconditioner calculates an approximate LU factorization based on *positional dropping*, i.e. fill-in elements are dropped when they appear in inconvenient places. Typically, no fill-in is tolerated and only non-zero elements of the original matrix are modified during the factorization. Moreover, no pivoting is carried out in the code (thus, if the matrix treated has a zero on the main diagonal the method will fail to complete the Gaussian elimination).

It is seen from Table 10 that GMRES with ILU preconditioning sometimes fails as expected. It is also seen that the method is sometimes more expensive than the preconditioned ORTHOMIN (this should also be expected: if the discarded fill-ins are large, then the preconditioners L and U may be very crude and too many iterations may be needed to obtain the accuracy required). However, when the iterative process converges sufficiently fast, the

Table 9. Computing times (s) for three hybrids (iterations)

Matrix	ORTMIN	GMRES	CGS
sherman2	9(4)	7(7)	8(5)
pores-2	4(64)	10(3)	12(3)
nnc1374	35(3)	38(1)	30(1)
hwatt-1	6(20)	9(18)	12(16)
hwatt-2	7(38)	10(21)	13(19)
west2021	5(3)	5(2)	5(1)
orsreg-1	5(38)	7(25)	9(19)
sherman5	7(21)	8(16)	14(14)
saylr4	9(46)	16(92)	15(38)
sherman3	21(73)	35(137)	29(54)

Table 10. Computing times (s) for ORTHOMIN hybrid and GMRES-ILU (iterations)

Matrix	ORTMIN	GMRESwith ILU
steam2	1(3)	0.5(2)
mc-fe	3(7)	1.4(8)
sherman2	9(4)	0.5(17)
pores-2	4(64)	1.9(114)
nnc1374	35(3)	Failed
hwatt-1	6(20)	2.5(111)
hwatt-2	7(38)	5.7(260)
west2021	5(3)	Failed
orsreg-1	5(38)	2.2(79)
or678lhs	10(13)	Failed
sherman5	7(21)	3.4(99)
saylr4	9(46)	23.1(553)
gemat11	20(5)	Failed
gemat12	20(5)	Failed
sherman3	21(73)	32.4(684)

GMRES with an incomplete LU is normally the best choice since there is no fill-in and the *a priori* knowledge of the non-zero locations which must be updated allows the complete suppression of much of the non-numerical work present in most direct method codes for general sparse systems such as symbolic factorization and dynamic data structures.

In Anderson's code the stopping criterion is based on Eq. (6), as in the original GMRES. No attempt to investigate whether the method converges or not is made: the code carries out the computations until either Eq. (6) is satisfied or the maximal allowed number of iterations prescribed by the user, MAXIT, is reached. The attempt to carry out the computations with an accuracy requirement of 10^{-4} (as for the preconditioned ORTHOMIN) was not successful: the code yielded solutions with poor accuracy in all but one case, **steam2**. It is recommended in the code to use an accuracy requirement given by $ACCUR = 10^{-8}$. Even then, the code often returns solutions with poorer accuracy. The results obtained with accuracy requirements of 10^{-4} , 10^{-8} and 10^{-10} are displayed in Table 11. The fact that the

GMRES-ILU code does not try to determine whether the process is convergent or not can result in many unnecessary iterations, especially when the ILU preconditioner is not sufficient to make the method converge for a problem. This situation occurred for the matrix **gre-1107**. The code performed MAXIT = 3000 iterations and returned a wrong result.

The results given in Tables 8 and 11 indicate that the acceptability test proposed above seems to provide a more robust way of evaluating termination of an iteration than does using the residual vectors. It is also seen from Table 11 that the attempt to eliminate the influence of matrix A on the stopping criterion by using Eq. (6) is not always reliable [probably because the assumption (7) is not always satisfied]. Of course, our proposed criteria are also based on heuristics and many more experiments, and perhaps some improvements are needed to estimate their reliability in general. For the set of problems tested, however, they seem to be much more reliable than other proposed criteria. It should also be emphasized that this behavior of false convergence is not peculiar to GMRES-ILU. Small residuals and/or correction vectors have been observed in the runs with the preconditioned ORTHOMIN. However, the proposed stopping criteria were not satisfied and the failure of the method was avoided. The test-matrices from the Harwell-Boeing set for which this happened are **gre-1107** and **gaff1104** (some results concerning runs with these matrices are presented in our previous work²⁴). Examples like this demonstrate that the code has been able to make the crucial decisions

- (1) to stop the iterative process
- (2) to reduce the drop-tolerance
- (3) to repeat the computations with recalculated preconditioners

in a difficult situation where the norms of the correction vectors are smaller than the accuracy required. It is necessary to emphasize that, in these two examples, continuing the iterations does not improve the situation: the iterative process simply does not converge (in spite of the smallness of the correction vectors, and the behavior of the residual vectors is similar). It should also be mentioned that the matrices involved in these examples are rather ill-conditioned. Checks of the correction norms, as well as the residual norms, have clearly shown that stopping criteria based on these norms often produce solutions that are not sufficiently accurate when the matrices involved are ill-conditioned and/or badly scaled.

6. COMMENTS ON FURTHER ENHANCEMENTS

As noted earlier, the results presented above were for codes whose parallelism has been generated via moderate restructuring by hand to the sequential code and a restructuring compiler. The improvement due to the hand tuning can be seen by comparing the

Table 11. Accuracy of GMRES-ILU when three accuracy requirements are imposed (iterations)

Matrix	$\epsilon = 1.0E-4$	$\epsilon = 1.0E-8$	$\epsilon = 1.0E-10$
steam2	1.6E-7(1)	1.7E-9(2)	1.7E-9(2)
mc-fe	1.7E+2(2)	1.8E-2(6)	3.2E-4(8)
sherman2	5.4E-2(7)	2.6E-6(14)	2.2E-8(17)
pores-2	2.9E-2(49)	1.8E-6(97)	3.4E-8(114)
nnc1374	Failed	Failed	Failed
hwatt-1	5.6E-1(1)	8.8E-2(5)	1.2E-7(111)
hwatt-2	8.6E-1(9)	3.1E-2(30)	1.5E-7(260)
west2021	Failed	Failed	Failed
orsreg-1	2.9E-2(23)	2.8E-7(63)	2.7E-9(79)
or678lhs	Failed	Failed	Failed
sherman5	1.2E-2(17)	1.3E-6(79)	1.1E-8(99)
saylr4	1.8E-2(17)	5.3E-6(429)	7.1E-8(553)
gemat11	Failed	Failed	Failed
gemat12	Failed	Failed	Failed
sherman3	4.7E-2(19)	5.6E-4(470)	1.3E-7(684)

results presented here to those in Ref. 24. It should not be surprising, therefore, that with more intense tuning the performance of both the calculation of the preconditioner and the iterative method can be improved considerably. This has been demonstrated for the positional dropping GMRES-ILU code developed by Anderson for the Alliant FX-series.¹⁸ As mentioned earlier, Wijshoff has also studied the architecture/algorithm mapping of sparse primitives, in particular a sparse matrix multiplied by one or more dense vectors, that are of interest for the iterative method portion of the code on multivector processors.¹³ The effect of applying these performance enhancements to the iterative method portion of the code is discussed in Ref. 7.

The improvement of the performance of the general sparse factorization portion of the algorithm is more difficult, but certainly possible. For example, changing the way in which the code handles the symbolic factorization portion of the rank-1 update further improves performance. Table 12 compares the performance of the version of Y12M used in Ref. 24 to one with the symbolic factorization changes executing in direct method mode, i.e. $\tau = 0$, for some additional Harwell-Boeing matrices. Comparing these results to some in Table 2 also indicates how much the incorporation of the factorization changes into Y12M1 could improve its performance.

It is well known that for machines with hierarchical memory systems dense factorization algorithms must be written in terms of BLAS3 constructs in order to achieve high performance.²⁶ Furthermore, on such machines the discrepancy in the performance of general sparse solvers and dense solvers is considerable. Therefore, the appropriate use of a switch to a dense solver during sparse factorization can also contribute to improved performance. Indeed, on a machine like the Alliant FX/80, for many of the

Table 13. Computing times (s) with the addition of a switch to dense factorization code

Matrix	Time
jpwh-991	2.4
orsirr	2.8
sherman2	4.9
gaff1104	3.9
gre-1107	2.8
pores-2	3.3
hwatt-1	5.7
hwatt-2	5.7
orsreg-1	9.0
sherman5	8.1
saylr4	23.4
sherman3	16.2

Harwell-Boeing matrices a well-implemented rank-1-based code with a dense switch will yield just as significant a performance improvement as codes based on more complex parallel pivot strategies. Table 13 shows the computing time for some of the Harwell-Boeing matrices which benefit from the switch to dense factorization routines. Additional performance improvements are possible by the careful consideration of the use of the memory hierarchy for both rank-1 and parallel pivot versions of the code and by exploiting information gained in factorizations with larger values of τ when updating of the drop-tolerance is required. (See Ref. 7 for details.)

Acknowledgements—This work was supported in part by the NSF under Grants No. NSF MIP-8410110 and No. CCR-8718942, the Department of Energy under Grant No. DOE-DE-FG02-85ER25001, and AT&T Corp. under Grant No. AT&T-AFFL-67-SAMEH.

REFERENCES

1. I. Duff, A. Erisman and J. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, 1986.
2. O. Østerby and Z. Zlatev, *Direct Methods for Sparse Matrices*, Springer, Berlin, 1983.
3. I. Duff and J. Reid, "A comparison of sparsity orderings for obtaining a pivotal sequence in Gaussian elimination," *Journal of the Institute of Mathematics and its Applications* **14**, 281–291 (1974).
4. Z. Zlatev, "On some pivotal strategies in Gaussian elimination by sparse technique," *SIAM Journal of Numerical Analysis* **17**, 18–30 (1980).
5. Z. Zlatev, "Sparse matrix technique for general matrices: pivotal strategies, decompositions and applications in ODE software," in *Sparsity and its Applications* (edited by D. Evans), pp. 185–228, Cambridge University Press, Cambridge, 1985.
6. U. Meier and A. Sameh, "The behavior of conjugate gradient algorithms on a multivector processor with a hierarchical memory," *Journal of Computational and Applied Mathematics* **24**, 13–32 (1988).
7. K. Gallivan, A. Sameh and Z. Zlatev, "A robust parallel linear system solver," Report No. 984, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1990.
8. Y. Saad and M. Schultz, "GMRES: a generalized minimal residual algorithm for solving nonsymmetric

Table 12. Computing times (s) after symbolic factorization alteration

Matrix	Old	New
pde9511	5	2.5
jpwh-991	25	7.0
sherman1	5	2.4
orsirr	18	5.4
sherman2	199	32.1
gaff1104	27	9.4
sherman4	4	1.5
gre-1107	15	5.0
pores-2	18	5.9
mahistlh	4	2.1
nnc1374	39	4.8
hwatt-1	58	15.8
hwatt-2	57	15.5
west2021	4	2.2
orsreg-1	90	22.7
sherman5	100	23.5
saylr4	197	52.6
sherman3	147	35.3

- linear systems," *SIAM Journal of Scientific and Statistical Computing* **7**, 856–869 (1986).
9. S. Eisenstat, H. Elman and M. Schultz, "Variational methods for nonsymmetric systems of linear equations," *SIAM Journal of Numerical Analysis* **20**, 345–357 (1983).
 10. P. Sonneveld, "CGS, a fast Lanczos-type solver for nonsymmetric linear systems," *SIAM Journal of Scientific and Statistical Computing* **10**, 36–52 (1989).
 11. Å. Björck and Z. Zlatev, "Exploiting the separability in the solution of linear ordinary differential equations," *Computers and Mathematics with Applications* (to appear).
 12. Z. Zlatev, "Survey of the advances of exploiting the sparsity in the solution of large problems," *Journal of Computational and Applied Mathematics* **20**, 83–105 (1987).
 13. H. Wijshoff, "Implementing sparse BLAS primitives on concurrent/vector processors: a case study," Report No. 843, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1989.
 14. C. Lawson, R. Hanson, O. Kincaid and F. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM Transactions on Mathematical Software* **7**, 308–323 (1979).
 15. J. Dongarra, J. Du Croz, S. Hammarling and R. Hanson, "A proposal for an extended set FORTRAN basic linear algebra subprograms. *ACM SIGNUM Newsletter* **20**, 1–18 (1985).
 16. J. Dongarra and S. Eisenstat, "Squeezing the most of an algorithm in CRAY FORTRAN," *ACM Transactions on Mathematical Software* **10**, 219–230 (1984).
 17. D. Dodson and J. Lewis, "Proposed sparse extension to the basic linear algebra subprograms," *ACM SIGNUM Newsletter* **20**, 22–25 (1985).
 18. E. Anderson, "Parallel implementation of preconditioned conjugate gradient methods for solving sparse systems of linear equations," Report No. 805, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1988.
 19. E. Anderson and Y. Saad, "Preconditioned conjugate gradient methods for general sparse matrices on shared memory machines," in *Parallel Processing for Scientific Computing* (Edited by G. Rodrigue), pp. 88–92, Society for Industrial and Applied Mathematics, Philadelphia, 1989.
 20. I. Duff, R. Grimes and J. Lewis, "Sparse matrix test problems," *ACM SIGNUM Newsletter* **17**, 22–27 (1982).
 21. I. Duff, R. Grimes and U. Lewis, "Sparse matrix test problems," *ACM Transactions on Mathematical Software* **15**, 1–14 (1989).
 22. I. Duff, "MA28: a set of FORTRAN subroutines for sparse unsymmetric linear equations," Report No. R8730, AERE Harwell Laboratory, Harwell, U.K., 1977.
 23. J. George and E. Ng, "An implementation of Gaussian elimination with partial pivoting for sparse systems," *SIAM Journal of Scientific and Statistical Computing* **6**, 390–405 (1985).
 24. K. Gallivan, A. Sameh and Z. Zlatev, "Solving general sparse linear systems using conjugate gradient-type methods," Proceedings of 1990 International Conference on Supercomputing, Amsterdam, ACM Press, 1990, pp. 132–139.
 25. Z. Zlatev, "Use of iterative refinement in the solution of sparse linear systems," *SIAM Journal of Numerical Analysis* **19**, 381–399 (1982).
 26. K. Gallivan, W. Jalby, U. Meier and A. Sameh, "Impact of hierarchical memory systems on linear algebra algorithm design," *International Journal of Supercomputer Applications* **2**, 12–48 (1988).