

# Solving General Sparse Linear Systems Using Conjugate Gradient-type Methods

K. Gallivan, A. Sameh and Z. Zlatev  
Center for Supercomputing Research and Development  
University of Illinois at Urbana-Champaign

## Abstract

The problem of finding an approximation of  $x = A^\dagger b$  (where  $A^\dagger$  is the pseudo-inverse of  $A \in \mathbb{R}^{m \times n}$  with  $m \geq n$  and  $\text{rank}(A) = n$ ) is discussed. It is assumed that  $A$  is sparse but has neither a special pattern (as bandedness) nor a special property (as symmetry or positive definiteness). In this paper it is shown that preconditioners obtained by neglecting small elements during the decomposition of  $A$  into easily invertible matrices can be used efficiently with conjugate gradient-type methods if an adaptive strategy for deciding when an element is small is implemented. The resulting preconditioned methods are often better than the corresponding direct and pure iterative methods or those based on preconditionings in which elements are neglected when they appear in a predefined set of positions in the matrix, i.e. positional rather than numerical dropping. Numerical results are given to illustrate the performance of the CG-type methods preconditioned via numerical dropping.

## 1 The problem

Consider the problem of finding  $x \in \mathbb{R}^{n \times 1}$  from  $x = A^\dagger b$ , where  $A \in \mathbb{R}^{m \times n}$ ,  $b \in \mathbb{R}^{m \times 1}$ ,  $m \geq n$ , and  $\text{rank}(A) = n$ . If  $m > n$  a linear least squares problem (LLSP) must be solved and when  $m = n$  a system of linear algebraic equations (LAE's) when  $m = n$  must be solved. If  $\text{rank}(A) < n$ , or even if  $\text{rank}(A) = n$ , but  $A$  is close to a rank-deficient matrix, then special methods, such as those based on the singular value decomposition ([15]), are required. Such problems will not be considered. The matrix  $A$  is assumed to be a general sparse matrix, i.e.,

no special algebraic properties, e.g., symmetric positive definiteness, nor special sparsity patterns, e.g., bandedness, are assumed.

There are, of course, two basic classes of methods for these problems: direct and iterative. Direct methods are reasonably robust for general matrices and usually produce sufficiently accurate solutions (although extra work is usually required to estimate the accuracy). For general sparse matrices, these methods can be difficult to map efficiently to parallel processors and tend not to fully exploit the computational capability of the architectures (compared, say, to a direct method for dense problems). For matrices with special algebraic properties, such as positive definiteness, iterative methods whose convergence is assured may be used. These methods tend to map more easily to parallel architectures than direct methods but they also often do not fully exploit the computational capabilities of the architecture. When these methods converge at a sufficient rate, are often more efficient over all than direct methods. The problems with applying them to general sparse problems are that convergence is no longer assured and when convergence occurs it may be at an abysmal rate.

The basic question for a general sparse matrix  $A$  is then: Is it possible to design a method for solving  $x = A^\dagger b$  which preserves the advantages of the iterative methods and robustly computes an acceptable solution like the direct methods? In this paper we explore one approach to developing such a method based on the use of conjugate gradient-type (CG-type) algorithms with preconditioning for unsymmetric systems; for example, see [2, 3].

## 2 Preconditioning

As noted above, the basic problems with iterative methods are assuring convergence and a sufficiently fast rate of convergence. For problems with special properties that guarantee convergence, the rate of convergence is

often accelerated by using preconditioning or acceleration parameters [16].

Preconditioning a system  $Ax = b$  involves using the iterative method to solve the related system  $M^{-1}A = b$  where the preconditioner  $M$  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 [18].

One of the most popular preconditioning strategies is based on *incomplete* factorization of  $A$  (ILU) [19]. During factorization, elements are dropped based on their position in the matrix. Typically, the elements are dropped if they are not in a position where a nonzero element occurs in the original matrix  $A$ , i.e. all fill-in are ignored. This can be generalized so that fill-in is kept up to some predetermined number of levels, e.g., fill-in caused by elements in positions that are nonzero in  $A$  are allowed but not fill-in caused by other fill-in elements. The existence of the incomplete factorizations and the convergence of some associated iterative methods has been shown for M-matrices and Stieltjes matrices. Existence and convergence are, of course, not guaranteed when positional dropping is applied to a general matrix. Further, when it is applied and the factorization does not exist or the method does not converge it is not obvious how to alter the method to make existence or convergence more likely.

Østerby and Zlatev suggested the use of numerical dropping to make direct methods more efficient, i.e. elements were dropped from the factorization when found to be *suitably small* [20, 26]. An absolute tolerance was used to determine smallness and a simple iterative refinement technique was used to improve the accuracy of the solution. In this paper, we explore the use of *approximate* LU and QR factorization preconditioning based on a more aggressive use of numerical dropping. The preconditioning is applied to ORTHOMIN, [8], GMRES, [21], and CGS [17, 22] for  $m = n$ , and to CG for  $m > n$ .

### 3 Approximate factorization preconditioning

If  $\text{rank}(A) = n$ , then  $x = A^\dagger b$  reduces to a system of LAE's:  $Bx = c$ , where  $B = A$  and  $c = b$  if  $m = n$ ; and  $B = A^T A$  and  $c = A^T b$  if  $m > n$ . Let the *drop tolerance*,  $\tau \geq 0$ , be given. (For more details concerning the idea of using  $\tau$  see [20]). Suppose we have an algorithm which produces a matrix  $\tilde{A}(\tau)$  such that, in the absence of rounding errors,  $\tilde{A}(\tau) \rightarrow A$  as  $\tau \rightarrow 0$ . The matrix  $\tilde{A}(\tau)$

can be used to obtain a preconditioned system  $Cy = d$  from  $Bx = c$ .

If  $m = n$ , then  $\tilde{A}(\tau) = LU$  is formed by performing an approximate factorization of  $A$  using drop tolerance  $\tau$  (permutations required to form  $L$  and  $U$  have been ignored for simplicity of presentation). The preconditioned system is then  $(LU)^{-1}Ax = (LU)^{-1}b$  which is solved via one of the iterative methods mentioned above. If  $m > n$ , an approximate orthogonal factorization using drop tolerance  $\tau$  is formed based on the QDR decomposition found by Gentleman's form of plane rotations [11, 12]. CG is then used to solve the preconditioned system  $Cy = d$ , where  $C = D^{-1}(R^T)^{-1}A^T A R^{-1}D^{-1}$ ,  $y = DRx$ , and  $d = D^{-1}(R^T)^{-1}A^T b$ .

Of course, for a general matrix  $A$  and a given  $\tau$  there is no guarantee that the factorizations exists and if they that the iterative methods will converge. Hence, the process above is used as a single major step in our solver. If the decomposition does not exist  $\tau$  is reduced and the factorization is restarted. Similarly if the preconditioned iteration does not converge rapidly enough  $\tau$  is reduced and a new preconditioner is computed. The algorithm for the case  $m = n$  can be summarized:

```

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
       $\tau \leftarrow \rho_1(\tau)$ 
    END IF
  ELSE
     $\tau \leftarrow \rho_2(\tau)$ 
  END IF
END DO

```

The functions  $\rho_1(\tau)$  and  $\rho_2(\tau)$  are functions that adjust the value of  $\tau$  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  $\tau$  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  $\tau$ , to allow the algorithm to find a drop tolerance that is natural for the problem. The early iterations with large  $\tau$  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  $\tau$  that are about the same. A similar algorithm can be used for  $m > n$ .

The use of  $\tau > 0$  is based on the following heuristics. Assume that  $m = n$ ,  $E = A - LU$  and  $\tau$  is chosen so  $\|A^{-1}E\| < 1$ . Then  $(LU)^{-1} = (I + F)A^{-1}$  where  $\|F\| \leq \|A^{-1}E\|/(1 - \|A^{-1}E\|)$  ([23, p. 188]). Thus,  $C = I + F$  is a perturbation of the identity matrix. If  $E = \theta\tau A$  with  $0 < \theta < 1$  (which is an analog to the assumption  $E = \varepsilon A$  made in [23]), then  $\|F\| \leq \theta\tau/(1 - \theta\tau)$  and  $\|F\|$  can be made arbitrarily small by choosing  $\tau$  small.

Of course, the two factorization methods and iterative methods used here are not the only possibilities. Any factorization that decomposes  $A$  into easily invertible matrices can be applied as well as other iterative methods such as ORTHODIR and biconjugate gradients [2].

## 4 Dropping procedures

In order to perform the approximate factorization we need a non-decreasing (in  $\tau$ ) function  $f(\tau, i, j, k) \geq 0$ , whose arguments are the drop-tolerance, the row and column numbers of the nonzero under consideration and the stage of the computational process. Then  $a_{ij}^{(k)}$  is kept and updated as long as  $|a_{ij}^{(k)}| > f$ . If  $|a_{ij}^{(k)}| \leq f$ , then it is removed and not used any more ("it is dropped"). One can apply this test to the entire active portion of the matrix at stage  $k$  or it can be combined with some positional restrictions such as only consider the pivot row and column to be used in the next stage. If such a strategy is used modifications to the Markowitz count can also be made for use in choosing the next pivot, i.e. the counts may only consider elements that satisfy the test based on  $f$ .

The simplest and least effective function  $f$  is based on an absolute drop tolerance, i.e.,  $f$  depends only on  $\tau$ . The function is defined

$$f(\tau, i, j, k) = \tau. \quad (1)$$

Such dropping requires almost no extra work. To facilitate the choice of  $\tau$ , one can scale  $A$  and then set  $\tau = c * \min(a_{i,*}), i = 1(1)m$  or  $\tau = c * \min(a_{*,j}), j = 1(1)n$ , where  $0 < c < 1$  is a constant,  $a_{i,*} = \max(|a_{ik}|, |a_{i,k+1}|, \dots, |a_{in}|)$  and  $a_{*,j}$  is similarly defined as the maximum in the  $j$ -th column. Column and/or row equilibration is often a successful scaling when  $A$

is not very ill-conditioned. Only column scaling is allowed for LLSP's. When using approach (1) the code can remove all of the nonzeros from the active part of a row or column resulting in structural singularity. This is not necessarily a problem since it can be handled by reducing  $\tau$  but it can be prevented from happening or at least made less likely.

Such structural singularity will not occur at stage  $k$  if the dropping is based on

$$f(\tau, i, j, k) = \tau * \min(a_{i,*}^{(k)}, a_{*,j}^{(k)}), \quad (2)$$

where  $a_{i,*}^{(k)} = \max(|a_{ik}^{(k)}|, |a_{i,k+1}^{(k)}|, \dots, |a_{in}^{(k)}|)$  and  $a_{*,j}^{(k)}$  is the maximum in the active part of the  $j$ -th column. Since the nonzeros are typically ordered either by rows or by columns, it is not expensive to compute either  $a_{i,*}^{(k)}$  or  $a_{*,j}^{(k)}$ , but not both. Therefore, approach (2) is rather costly.

A compromise between (2), which is computationally expensive, and (1), which is cheap but may force us to repeat the computations with a smaller  $\tau$  in situations where (2) does not, is the choice

$$f(\tau, i, j, k) = \tau * a_{i,*}^{(k)}, \quad (3)$$

when the nonzeros are ordered by rows (otherwise  $a_{i,*}^{(k)}$  should be replaced by  $a_{*,j}^{(k)}$ ). Compared with (1), (3) requires some extra work (to calculate  $a_{i,*}^{(k)}$ ). The fact that  $\tau < 1$  ensures that not all elements in the active part of a row will be dropped. Structural singularity at stage  $k$  can appear, however, when all elements in the active part of a column are removed. Thus, (3) is not as reliable as (2), but is cheaper.

Approach (3) may be improved at the cost of some extra work. An element  $a_{ij}^{(k)}$  could be held, even when (3) is satisfied, if it is the last element in the active part of its column. The extra work is due not only to the extra check needed to decide whether the element is the last one in the active part of its column or not, but also because some small elements are to be kept and updated. A similar approach can be applied when the nonzeros are ordered by columns.

Structural singularity of the remaining active part of the matrix caused by two rows (columns) having a single element in the same column (row) can also be avoided when applying numerical dropping. One need only keep track of the rows and columns which have exactly one element. When an element is dropped from the active part of the matrix and produces a new single element row (column) a single read can be performed to determine if there is another row with its singleton in the same column.

Matrix	$n$	Nonzeros	$\kappa(A)$
pde-9511	961	4681	1.35E+ 2
west0989	989	3537	3.10E+12
jpwh-991	991	6027	2.56E+ 2
sherman1	1000	3750	4.64E+ 3
orsirr-1	1030	6858	9.94E+ 6
sherman2	1080	23094	6.25E+11
gaff1104	1104	16056	2.59E+11
sherman4	1104	3786	2.32E+ 3
gre-1107	1107	5664	2.02E+ 7
pores-2	1224	9613	3.69E+ 9
mahistlh	1258	7682	2.48E+12
nnc1374	1374	8606	2.07E+15
west1505	1505	5414	3.87E+12
hwatt-1	1856	11360	4.31E+ 9
hwatt-2	1856	11550	1.08E+12
west2021	2021	7353	3.27E+12
orsreg-1	2205	14133	2.22E+ 5
sherman5	3312	20793	4.21E+ 3
saylr4	3564	22316	7.59E+ 6
sherman3	5005	20033	6.90E+16

Table 1: Harwell-Boeing matrices used in  $LU$  experiments.

There seem to be no satisfactorily simple ways of protecting against numerical dropping producing a numerically singular active portion of the matrix; there are only ways of recovering once it happens one of which is the adaptation of  $\tau$  used here.

## 5 Linear algebraic equations

Three CG-type methods, ORTHOMIN [8], GMRES [21] and CGS [17, 22], have been tested<sup>1</sup>. The preconditioners are obtained using the sparse solver Y12M [20, 26]. Some results are reported here for which test matrices from the Harwell-Boeing set of test problems are used [7]. Some characteristics of these matrices are given in Table 1. The conclusions presented, however, are based on results found with several hundred matrices from [7] and [20]. Condition number estimates are calculated by Y12M via the estimator from [4] modified for sparse matrices [29].

Y12M was chosen because it provides a device for dropping, and because with some minor modifications the restructuring compiler was able to produce results superior to those it achieved for MA28 [5, 6] and SPARSPAK-C [14] (Table 2)<sup>2</sup>. This altered version of

Matrix	MA28	SPAK-C	Y12M	OMIN
pde-9511	32	13	5	3 ( 10,1)
west0989	7	9	1	2 ( 2,3)
jpwh-991	99	68	25	4 ( 9,1)
sherman1	14	15	5	3 ( 13,1)
orsirr-1	33	47	18	3 ( 31,1)
sherman2	580	1052	199	7 ( 1,2)
gaff1104	105	39	27	72 ( 1,4)
sherman4	11	11	4	3 ( 19,1)
gre-1107	50	49	16	39 ( 28,3)
pores-2	61	22	18	10 ( 4,2)
mahistlh	6	17	4	7 ( 1,3)
nnc1374	224	23	39	43 ( 1,6)
west1505	17	10	2	4 ( 2,3)
hwatt-1	437	107	58	11 ( 33,1)
hwatt-2	406	100	57	15 ( 53,1)
west2021	31	13	4	6 ( 4,4)
orsreg-1	140	189	90	8 ( 29,1)
sherman5	361	116	100	27 ( 9,2)
saylr4	1147	290	197	15 ( 30,1)
sherman3	847	215	147	50 (112,1)

Table 2: Computing times in seconds on an Alliant FX/80. (Iterations and number of trials are given in parentheses.)

Y12M is slightly faster than the original version on one processor and on serial machines the altered version is competitive with SPARSPAK-C (SPARSPAK-C tends to be slightly faster). The default pivotal strategy of MA28 has been used in these tests. There is an option in MA28 based on the pivotal strategy in [25], which performs much better, e.g., the computing time for *saylr4* is reduced from 1147 to 354 when the default pivotal strategy is replaced (the Y12M, however, time is 147). On serial machines the new version of Y12M is competitive with MA28 with the pivotal strategy of [25] (and better than MA28 with its default pivotal strategy). The results in the first three columns of Table 2 illustrate that on the Alliant the preconditioned CG-type methods should be compared with the altered Y12M. On serial machines the preconditioned CG-type methods must be compared with the best for the particular matrix direct solver (no code, among these three, performs best for all matrices). For a discussion of more substantial changes to enhance the performance of Y12M via parallelism see below and for much more detail see [10].

For the CG-type methods  $\|x - x_i\|/\|x\| \leq 10^{-4}$  is required in order for the method to be considered successful (where  $x_i$  is the accepted approximation). This requirement has been satisfied by ORTHOMIN for all problems. The direct codes give better accuracy for all problems except *nnc1374* (the error for *nnc1374* is of

<sup>1</sup>Note that ORTHOMIN is labeled OMIN and GMRES is labeled GMR in the tables due to space constraints.

<sup>2</sup>SPARSPAK-C is labeled SPAK-C in the tables

Matrix	CGS	OMIN	GMR	GMR+ILU
pde-9511	6	10	8	34
west0989	1	2	2	Failed
jpwh-991	6	9	8	25
sherman1	12	13	13	99
orsirr-1	24	31	33	70
sherman2	5	1	7	17
gaff1104	1	1	1	Failed
sherman4	14	19	16	105
gre-1107	45	28	6	Failed
pores-2	3	4	3	114
mahistlh	1	1	1	Failed
nnc1374	1	1	1	Failed
west1505	1	2	2	Failed
hwatt-1	16	33	18	111
hwatt-2	19	53	21	260
west2021	1	4	2	Failed
orsreg-1	19	29	25	79
sherman5	14	9	16	99
saylr4	38	30	92	553
sherman3	54	112	137	684

Table 3: Total number of iterations for the CG-type methods and GMRES+ILU.

order 0.01). A similar relative test on the correction at each step was used in conjunction with tests on the residual size to determine termination of the iterative method. The right-hand sides are generated so that  $\mathbf{x} = (1, 1, \dots, 1)^T$ . The results of more detailed experiments can be found in [10].

We use (3) with an initial  $\tau = 2^{-5}$  when dropping to produce the preconditioner for the CG-type methods. When the factorization fails to produce a preconditioner  $\tau$  is multiplied by  $2^{-10}$ . If the iterative method does not converge (or converges slowly)  $\tau$  is multiplied by  $2^{-5}$ . In both cases, the factorization is repeated with the new  $\tau$ . Neither initial scaling of  $A$  nor the enhancements mentioned to reduce the possibility of structural singularity mentioned earlier are used in the results reported here. (The tests are therefore very conservative in the amount of time taken to perform refactorization with different  $\tau$  values to produce a preconditioner.) The numbers of trials (factorizations and attempted use of a CG-type method with a particular  $\tau$ ) are given in Table 2 along with the total number of iterations. The times are sums of the times spent for all trials.

The approximate factorization preconditioned ORTHOMIN tends to perform much better than MA28 and SPARSPAK-C as expected. The more efficient direct solver Y12M is better than ORTHOMIN if  $A$  is very sparse and stays very sparse, and/or if several trials are needed. For some problems, however, ORTHOMIN per-

forms better even if many trials are required. For all of the computationally expensive problems ORTHOMIN performs very well, e.g., for *sherman2* it is 28 times faster than Y12M. As hoped, the storage is normally reduced when the preconditioned ORTHOMIN is used. For example, when *saylr4* is solved by Y12M the length of the large arrays has to be greater than 308630, while for ORTHOMIN the length is only 22316. For positional dropping strategies, such a bound on space is known a priori. Indeed, the amount of space available can be one of the key considerations when choosing the level of fill-in tolerated in such schemes. In the case of numerical dropping, no such a priori bound exists but empirical evidence indicates that savings are realized in practice.

The total number of iterations required for three preconditioned CG-type methods and from GMRES+ILU (GMRES preconditioned by an incomplete factorization; no fill-in allowed, no pivoting used) are given in Table 3. The computing times for the three CG-type methods are all comparable and the tendency (for the whole set of test-matrices) is the same: the preconditioned CG-type methods perform, as a rule, better than direct methods. They are also considerably more robust than GMRES+ILU for these general problems in terms of failures and number of iterations. The version of GMRES+ILU was optimized for the Alliant by Anderson ([1]) to a much greater degree than the compiler-based optimizations used for the CG-type methods and its preconditioner Y12M. For a detailed timing comparison of GMRES+ILU and a more substantially tuned version of the CG-type codes see [10]. (These results will also be presented at the conference.)

CGS, ORTHOMIN and GMRES were also run as pure iterative methods, i.e. no preconditioning, but the failure rate was greater than 50. Iterative refinement (IR), [26], was also used. Comparisons indicate that sometimes it either converges slowly or does not converge for drop-tolerances for which the CG-type methods converge sufficiently fast. However, when IR could successfully be used with the same  $\tau$  as that for the CG-type methods, it tends to be slightly more efficient (the work per iteration is smaller).

## 6 Linear least squares problems

In this section the results of using CG preconditioned with approximate orthogonal factorizations based on the codes in [27] and [28] to solve linear least squares problems are presented. Column equilibration is applied to  $A$  to facilitate the choice of  $\tau$  and dropping is carried out using approach (1). The use of (1) with orthogonal transformations performs better than it does with Gaussian elimination. This may be due to the fact that

Matrix	Rows	Columns	Nonzeros	CG
well1033	1033	320	4732	Failed
well1850	1850	712	8758	Failed
illc1033	1033	320	4732	Failed
illc1850	1850	712	8758	Failed
abb313	313	176	1557	Failed
ash219	219	85	438	.1 (15)
ash331	331	104	662	.2 (14)
ash608	608	188	1216	.3 (16)
ash958	958	292	1916	.5 (15)

Table 4: Harwell-Boeing matrices used for least squares tests and CG time in seconds (iterations).

orthogonal transformations preserve the column norms. In general, however, it is most likely preferable to apply (3). The same accuracy requirement as that used to determine the success of the  $LU$  preconditioning is used. The initial value  $\tau$  is taken to be  $2^{-4}$  (found experimentally using a large set of test problems). When necessary  $\tau$  is updated using the same factors as in the previous section.

Nine matrices with  $m > n$  taken from the Harwell-Boeing test set are used in the experiments. Table 4) lists some of the characteristics of the matrices and their pure CG execution time and number of iterations. While the square unsymmetric matrices in the Harwell-Boeing test set form a representative subset, this is not true when  $m > n$ . The matrices in Table 4 are rather sparse and they stay sparse during the orthogonal factorization process. This is not typical; it is well known that normally a significant amount of fill-in appear during the orthogonal decomposition. Therefore, matrices from [20] are also used. These synthetic test matrices depend on five parameters:  $m, n, c, r, \alpha$  (by which one can vary the number of rows, the number of columns, the sparsity pattern, the number of nonzero and their magnitude respectively). Four of the parameters are fixed ( $m = 500$ ,  $n = 250$ ,  $c = 100$  and  $\alpha = 32.0$ ), while  $r$  is varied. The number of nonzeros ( $NZ$ ) increases with  $r$  ( $NZ = rm + 110$ ). Matrices that create significant amounts of fill-in are usually produced when  $r$  becomes large. Thus, we can study the codes when the orthogonal decomposition suffers from heavy fill-in.

Results are given in Table 5 and Table 6. Two direct codes, SPARSPAK-B [13] and LLSS01-DS with  $\tau = 0.0$  [27], an IR code, LLSS01-IR [27], preconditioned CG, LLSS02 [28], and pure CG are used<sup>3</sup>. Pure CG failed to converge for all problems from [20]. Conclusions similar to the previous section can be drawn from the numerical results (including others not presented in the tables). It

<sup>3</sup>These codes are referred to in the table as SPAK-B, LS-DS, LS-IR, LS-CG and CG respectively.

Matrix	SPAK-B	LS-DS	LS-IR	LS-CG
well1033	12	2	Failed	2 (13)
well1850	25	11	Failed	7 (11)
illc1033	12	2	4 (5)	4 (3)
illc1850	25	11	11 (16)	10 (7)
abb313	6	2	2 (24)	1 (10)
ash219	4	0.6	0.3 (7)	0.3 (4)
ash331	5	1.2	0.4 (6)	0.4 (4)
ash608	7	4.3	0.8 (7)	0.8 (5)
ash958	11	10	1.2 (5)	1.2 (5)

Table 5: Computing times in seconds (iterations) on an Alliant FX/80 for Harwell-Boeing matrices.

r	SPAK-B	LS-DS	LS-IR	LS-CG
10	36	72	1.4 (19)	1.1 (8)
20	61	107	1.5 (9)	1.4 (7)
30	73	144	2.0 (13)	1.8 (8)
40	101	181	2.2 (22)	1.6 (8)
50	128	208	2.1 (14)	1.8 (8)
60	136	214	3.6 (34)	2.4 (10)
70	141	257	Failed	6.6 (11)
80	141	274	Failed	7.8 (12)
90	150	282	Failed	9.6 (13)
100	159	268	Failed	9.5 (14)

Table 6: Computing times in seconds (iterations) on Alliant FX/80 for matrices from [20].

is not necessary to repeat them here. We can conclude, however, that the use of the numerical dropping to produce approximate factorization preconditioners for the least squares problem reduced the computing time required to solve the problems by several orders of magnitude.

## 7 Comments on further enhancements

As noted earlier, the results presented above were for codes whose parallelism had been generated via minor changes to the sequential code and a restructuring compiler. It should not be surprising that with more intense tuning the performance of both of 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 [1]. Wijshoff has also study 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 pro-

Matrix	Old	New
pde9511	5	2.5
jpwh-991	25	7.0
sherman1	5	2.4
orsirr	18	5.4
sherman2	199	32.1
gaffl104	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

Table 7: Computing time in seconds after symbolic factorization alteration.

cessors [24]. The effect of applying these performance enhancements to the iterative method portion of the code is discussed in [10].

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 7 compares the performance of the version of Y12M used in the previous sections to one with the further changes executing in direct method mode, i.e.,  $\tau = 0$ .

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 [9]. 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 Harwell-Boeing matrices a well-implemented rank-1-based code with a dense switch will yield just as significant performance improvement as codes based on more complex parallel pivots strategies. Table 8 shows the computing time for some of the matrices which benefit from the switch to dense factorization routines. Additional performance improvements are possible by the careful consideration

Matrix	Time
jpwh-991	2.4
orsirr	2.8
sherman2	4.9
gaffl104	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

Table 8: Computing time in seconds with the addition of a switch to dense factorization code.

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  $\tau$  when updating the drop tolerance is required.. See [10] for more details.

## Acknowledgements

This work was supported in part by the NSF under Grants No. NSF MIP-8410110 and No. CCR-8717942, 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] E. C. 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 at Urbana-Champaign, 1988.
- [2] O. Axelsson, *A survey of preconditioned iterative methods for linear systems of algebraic equations*, BIT, 25(1985), pp. 166-187.
- [3] O. Axelsson and V. A. Barker, *Finite element solutions of boundary value problems*, Academic Press, New York, 1984.
- [4] J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK: Users' Guide*, SIAM, Philadelphia, 1979.

- [5] I. S. Duff, *MA28: a set of FORTRAN subroutines for sparse unsymmetric linear equations*, Report No. R8730 A.E.R.E., Harwell, England, 1977.
- [6] I. S. Duff, A. M. Erisman and J. K. Reid, *Direct methods for sparse matrices*, Oxford University Press, Oxford-London, 1986.
- [7] I. S. Duff, R. G. Grimes and J. G. Lewis, *Sparse matrix test problems*, ACM Trans. Math. Software, 15(1989), pp.1-14.
- [8] S. C. Eisenstat, H. C. Elman and M. H. Schultz, *Variational methods for nonsymmetric systems of linear equations*, SIAM J. Numer. Anal., 20(1983), pp. 345-357.
- [9] K. Gallivan, W. Jalby, U. Meier, and A. Sameh, *Impact of hierarchical memory systems on linear algebra algorithm design*, Intl. J. Supercomputer Appl., 2(1988), pp. 12-48.
- [10] 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 at Urbana-Champaign, 1990.
- [11] W. M. Gentleman, *Least squares computations by Givens transformations without square roots*, J. Inst. Math. Applies., 12(1973), pp. 329-336.
- [12] W. M. Gentleman, *Error analysis of QR by Givens transformations*, Lin. Alg. Appl., 10(1975), pp. 189-197.
- [13] J. A. George and M. T. Heath, *Solution of sparse linear least squares problems using Givens rotations*, Lin. Alg. Appl., 34(1980), pp. 69-83.
- [14] J. A. George and E. Ng, *An implementation of Gaussian elimination with partial pivoting for sparse systems*, SIAM J. Sci. Statist. Comput., 6(1985), pp. 390-405.
- [15] G. Golub and C. F. Van Loan, *Matrix Computations*, The John Hopkins University Press, Baltimore, 1983.
- [16] L. A. Hageman and D. M. Young, *Applied Iterative Methods*, Academic Press, New York, 1981.
- [17] E. F. Kaasschieter, *The solution of nonsymmetric linear systems by bi-conjugate gradients or conjugate gradients squared*, Report No. 86-21. Department of Mathematics and Informatics, Delft University of Technology, Delft, Netherlands, 1986.
- [18] U. Meier and A. Sameh, *The behavior of conjugate gradient algorithms on a multivector processor with a hierarchical memory*, Jour. Comp. Appl. Math., 24(1988), pp. 13-32.
- [19] 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(1977), pp. 148-162.
- [20] O. Østerby and Z. Zlatev, *Direct methods for sparse matrices*, Springer, Berlin, 1983.
- [21] Y. Saad and M. H. Schultz, *GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 7(1986), pp. 856-869.
- [22] P. Sonneveld, *CGS, a fast Lanczos-type solver for nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 10(1989), pp. 36-52.
- [23] G. W. Stewart, *Introduction to matrix computations*, Academic Press, New York, 1973.
- [24] 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 at Urbana-Champaign, 1989.
- [25] Z. Zlatev, *On some pivotal strategies in Gaussian elimination by sparse technique*, SIAM J. Numer. Anal., 17(1980), pp. 18-30.
- [26] Z. Zlatev, *Use of iterative refinement in the solution of sparse linear systems*, SIAM J. Numer. Anal., 19(1982), pp. 381-399.
- [27] Z. Zlatev, *Comparison of two pivotal strategies in sparse plane rotations*, Comput. Math. Appl., 8(1982), pp. 119-135.
- [28] Z. Zlatev and H.B. Nielsen, *Solving large and sparse linear least-squares problems by conjugate gradient algorithms*, Comput. Math. Appl., 15(1988), pp. 185-202.
- [29] Z. Zlatev, J. Wasniewski and K. Schaumburg, *Condition number estimators in a sparse matrix software*, SIAM J. Sci. Statist. Comput., 7(1986), pp. 1175-1186.