RC24598 (W0807-036) July 7, 2008
Mathematics

IBM Research Report

Adaptive Techniques for Improving the Performance of Incomplete Factorization Preconditioning

Anshul Gupta
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Thomas George
Department of Computer Science
Texas A&M University
College Station, TX 77843

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

ADAPTIVE TECHNIQUES FOR IMPROVING THE PERFORMANCE OF INCOMPLETE FACTORIZATION PRECONDITIONING

ANSHUL GUPTA* AND THOMAS GEORGE†

Abstract. Three techniques for improving the robustness and performance of iterative solvers for sparse systems with symmetric positive definite or mildly indefinite coefficient matrices are introduced. The primary contribution is new block algorithms for incomplete factorization that result in an improvement in the performance of both the preconditioner generation and the iterative solution phases. One of the algorithms applies to matrices that have a natural block structure in their original form, and the other one applies to matrices without natural dense blocks. Additionally, two relatively simple but highly effective techniques are introduced. These include selecting the solver based on the definiteness properties of the preconditioner and automatic selection and tuning of incomplete factorization parameters. All three techniques have adaptive components; i.e., the preconditioner-solver combination chooses parameters or algorithmic components based on the properties of the coefficient matrix and its incomplete factors. Two of the three techniques are applicable to incomplete LU factorization for unsymmetric systems as well.

Key words. sparse solvers, iterative methods, preconditioning, incomplete factorization

AMS subject classifications. 65F10, 65F50

1. Introduction. Incomplete factorization methods have long been used to derive preconditioners for Krylov subspace methods to solve large sparse systems of linear equations [7, 56]. Like many preconditioning methods, incomplete factorization has its share of drawbacks. In this paper, we address three weaknesses of incomplete Cholesky factorization preconditioners and offer solutions to mitigate these problems. All three techniques introduced in the paper have adaptive components; i.e., the preconditioner-solver combination chooses parameters or algorithmic components based on the properties of the coefficient matrix and its incomplete factors. While we focus our discussion on incomplete Cholesky factorization for the iterative solution of symmetric positive definite (SPD) systems, two of the three problems and their mitigation methods presented in this paper are also relevant to incomplete LU factorization for the iterative solution of general sparse systems.

The first problem that we address is related to performance. A typical implementation of complete sparse Cholesky factorization [35] can realize a fairly respectable fraction of the peak performance of a machine. There are two main reasons for this. First, the numerical factorization is preceded by a symbolic factorization phase that computes the static structure of the factors. Secondly, supernodal [30] and multifrontal [25, 46] techniques ensure that practically all numerical computation is performed by cache-friendly level 2 and level 3 basic linear algebra subprograms (BLAS) [20, 21]. A-priori symbolic factorization cannot be used in incomplete factorization when a dropping strategy based on the values of the factor entries is used. As a result, for many problems, a direct solution turns out to be faster than an iterative solution based on incomplete factorization, even when complete factorization consumes significantly more memory than incomplete factorization [31, 32]. Dense blocks have been used successfully in the past [12, 27, 37, 41, 42, 53] to enhance the performance of incomplete factorization preconditioners. However, with a few exceptions [37, 53],

*Mathematical Sciences Department, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598 (anshul@watson.ibm.com).

†Department of Computer Science, Texas A&M University, College Station, TX 77843 (tgeorge@cs.tamu.edu).

these block algorithms have been applied to relatively simpler preconditioners, and only for matrices that have a naturally occurring block structure. We present an incomplete factorization algorithm that not only detects and uses dense or near dense blocks as they emerge in the incomplete factors during preconditioner generation, but also promotes the creation of such blocks by a local permutation scheme. Thus, this algorithm makes the use of higher level BLAS possible during the incomplete factorization of even those matrices that have poor or no block structure to begin with. In addition, the solution phase also benefits from blocking.

The second problem that we address is the breakdown of the Cholesky process due to negative or near zero pivots. Traditional methods to address this problem fall into two categories [7]: *preemptive methods*, which either modify the matrix or the factorization method to ensure that breakdown does not occur, and *reactive methods*, which either apply some sort of local perturbation or roll back the computation upon encountering an unsuitable pivot and restart with modified parameters until the factorization succeeds. Preemptive methods unnecessarily increase the cost and error of incomplete factorization for those matrices for which the standard incomplete factorization process may have succeeded. Reactive methods that roll back the computation are too costly and the ones that apply a local perturbation are too error prone. Our approach, which we show to work well in practice, involves switching to complete LDL^T factorization for the troublesome portions of the matrix and dropping the onerous requirement that the preconditioner be positive definite. During the solution phase, the conjugate gradient method is used if the preconditioner is positive definite and GMRES [57] or symmetric QMR [28] is used if it is indefinite.

The third problem that we address is that the performance and effectiveness of incomplete factorization is not only problem dependent, but is also highly sensitive to parameters such as ordering, drop-tolerance, fill-factor, or level of fill [31]. Moreover, many applications require the solution of a series of systems with the coefficient matrices changing gradually and the set of parameters that are best for the first system may not be suitable for the later ones. To address these issues, we built features in both the incomplete factorization algorithm and its software implementation that minimize the effort required by the user to select appropriate parameters. The solver not only offers a degree of automation in initial parameter selection, but also monitors the relative computation performed in the preconditioner generation and the solution phases to continuously adjust the drop-tolerance and fill-factor of incomplete factorization. The fill-factor is also used flexibly to allow less diagonally dominant columns to retain more nonzeros than the more diagonally dominant ones, thus reducing the sensitivity to the original selection.

In the paper, wherever practical and useful, we present experimental results on matrices derived from real applications to demonstrate the benefits of the techniques introduced in the paper. We also present experimental results to show that the robustness and performance of the resulting linear solver, which we shall refer to as WSMP (Watson Sparse Matrix Package) ICT, is highly competitive with that of well known iterative solver packages such as PETSc [5], Trilinos [39], Hypre [26], and ILUPACK [51] and that the techniques introduced in this paper do succeed in bridging the robustness and performance gap with direct solvers. Table 1.1 lists the matrices, along with some of their characteristics, that we used in our experiments. The column labeled *BlkSz* contains the average size of row and column blocks with nearly identical[1] structure. The column labeled *Dim* indicates the dimension of the

[1]Please refer to section 2 for more details.

Matrix	N	NNZ	BlkSz	Dim	DD	Application
af_shell7	504855	17588875	5.00	2	No	Sheet metal forming
audikw_1	943695	77651847	3.02	3	No	Crankshaft modeling
bcsstk25	15439	252241	1.33	3	No	Skyscraper simulation
bmwcra_1	148770	10644002	3.00	3	No	Crankshaft modeling
bst-1	1017397	74144859	2.99	2+	No	Structural analysis
bst-2	384012	28069776	3.02	3	No	Structural analysis
cfd1	70656	1828364	1.00	3	No	CFD Pressure matrix
cfd2	123440	3087898	1.00	3	No	CFD Pressure matrix
conti20	20341	1854361	3.02	2+	No	Structural analysis
dfi1	89616	1474304	1.88	U	No	Linear programming
fm90153	90153	5629095	9.00	2	No	Sheet metal forming
garybig	42459173	238142243	1.13	2+	Yes	Circuit simulation
hood	220542	10768436	7.00	2	No	Automotive
inline_1	503712	36816342	3.05	2	No	Structural engineering
kyushu	990692	26268136	1.00	3	No	Structural engineering
ldoor	952203	46522475	6.96	2	No	Structural analysis
minsurfo	40806	203622	1.00	2	Yes	Minimum surface problem
msdoor	415863	20240935	6.91	2	No	Structural analysis
mstamp-2c	902289	70925391	3.00	3	No	Metal stamping
nastran-b	1508088	111614436	3.04	3	No	Structural analysis
nd24k	72000	28715634	2.42	3	No	3d mesh (ND problem set)
oilpan	73752	3597188	7.00	2	No	Structural analysis
prblc_fem	525825	3674625	1.00	3	Yes	CFD Convection-diffusion
qa8fk	66127	1660579	1.00	3	Yes	Acoustics (stiffness)
qa8fm	66127	1660579	1.00	3	No	Acoustics (mass)
pga-rem-1	5978665	29640547	1.00	2	Yes	Power network analysis
pga-rem-2	1480825	7223497	1.00	2	Yes	Power network analysis
ship_003	121728	8086034	6.00	3	No	Structural analysis
shipsec5	179860	10113096	5.91	2	No	Structural analysis
torso	201142	3161120	1.00	3	No	Human torso modeling

TABLE 1.1

SPD test matrices with their order (N), number of nonzeros (NNZ), average clique size (BlkSz), dimension of the physical problem (Dim), diagonal dominance (DD), and application area. A 2+ in the DIM column indicates that the physical domain has a small constant third dimension, and a U in this column indicates unknown dimension.

physical problem that the matrix is derived from. The column labeled *DD* indicates whether or not the matrix is diagonally dominant. Most of the matrices are obtained from the University of Florida sparse matrix collection [18]. The remaining ones are from some of the applications that currently use WSMP direct solver [35]. All experiments were performed on a single CPU of a Power 5+ system running AIX with 16 GB of memory. In all our experiments, the right hand side vector b of the sparse system $Ax = B$ to be solved is set such that the solution x is all ones. A maximum of 1000 iterations were permitted in any experiment, and were terminated when the relative residual norm dropped below 10^{-8}.

The remainder of the paper is organized as follows. In section 2, we describe our block incomplete factorization algorithm in detail and experimentally demonstrate the effectiveness of the blocking scheme. In section 3, we discuss breakdown of incomplete Cholesky and a reasonably effective way of recovering from it. In section 4, we discuss selection and tuning of parameters for incomplete Cholesky factorization. Section 5 contains experimental comparison of the our preconditioner with several other precon-

ditioners from PETSc, Trilinos, Hypre, and ILUPACK. Section 6 contains concluding remarks.

2. Incomplete Factorization with Blocking. It is a well known fact that the nature of computations in a typical sparse iterative solver is such that its CPU utilization on modern cache-based microprocessors is quite poor. This problem of poor CPU utilization (relative to the CPU utilization of a well-implemented direct solver) is evident in varying degrees for almost all preconditioners [31]. In the context of incomplete factorization preconditioners, the primary culprits are indirect addressing and lack of data reuse, both during the preconditioner generation and solution phases. Performing sparse operations on dense blocks instead of individual elements during incomplete factorization and the solution phases can potentially reduce both these overheads.

2.1. Motivation for Block Incomplete Factorization. Figure 2.1(a) compares the preconditioner generation speeds of a few iterative solvers with that of complete factorization of the WSMP direct solver [35] for two of the matrices from our test suite, namely, *af_shell7* and *nastran-b*. The preconditioners used are levels of fill based incomplete Cholesky factorization [61] of PETSc [5], the sparse approximate inverse preconditioner [10, 15] of Hypre [26], the multilevel inverse-based incomplete Cholesky preconditioner [50] from ILUPACK [51], and the threshold based incomplete Cholesky preconditioner (described in detail in sections 2.2–2.6) from WSMP [36]. For each type of preconditioner, we generated preconditioners of varying sizes (by changing the *threshold* parameter of ParaSails, level of fill K for PETSc IC(K), and *drop-tolerance* for ILUPACK and WSMP ICT). We plot the size of the preconditioner relative to that of the complete factor on the x-axis and the preconditioner generation time relative to the direct factorization time on the y-axis.

Figure 2.1(a) clearly shows that preconditioner generation, across a diversity of preconditioners, is a much less efficient process than complete factorization. The preconditioner generation time relative to complete factorization increases rapidly with the size of the preconditioner and it often takes much longer to generate preconditioners that are a fraction of the size of the complete factor. Therefore, it remains practical to use only small preconditioners. Note that PETSc IC(K) is relatively faster to generate compared to other non-blocked preconditioners because incomplete factorization based on levels of fill benefits from the predetermined structure of the factors. There is often a correlation between the size and the quality of preconditioners, particularly for preconditioners based on incomplete factorization. A smaller preconditioner is likely to result in a slower rate of convergence and is more likely to fail compared to larger one (although, increasing the preconditioner size beyond a point may lead to slower overall solution because the reduction in the number of iterations may not be enough to offset the increased cost of preconditioner generation as well as that of each iteration). Therefore, the inefficiency of the preconditioner generation process has implications for not only the speed, but also the robustness of iterative methods because it renders preconditioners beyond a certain size impractically slow to use.

Figure 2.1(a) also shows that WSMP's block incomplete Cholesky method (described in sections 2.2–2.6) takes the least time to generate a preconditioner of a certain given size, thus providing empirical evidence of the benefits of blocking. The generation efficiency of the non-blocked version of WSMP's preconditioner is comparable to that of other non-blocked preconditioners. Figure 2.1(b) provides further empirical evidence of the benefits of blocking, this time in the iterative phase of the solution process. In Figure 2.1(b), we plot the speed of the conjugate gradient

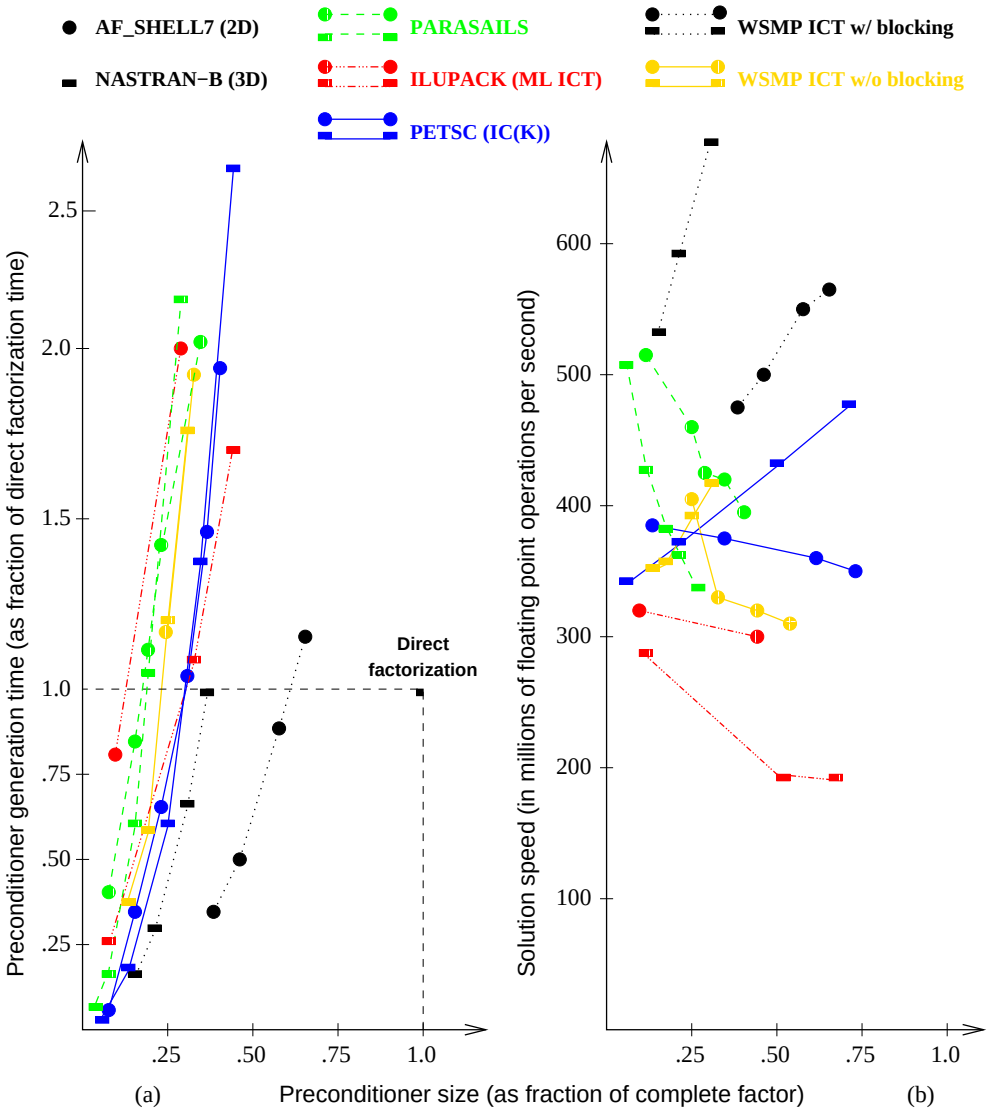

FIG. 2.1. *(a) Memory and time comparison between direct factorization and generation of ILUPACK ML ICT, PETSc IC(K), Hypre ParaSails, and WSMP ICT preconditioners. (b) Speed of the conjugate gradient algorithm with the preconditioner configurations in (a).*

(CG) [33] algorithm in Megaflops (million floating point operations per seconds) with each preconditioner-parameter combination used in Figure 2.1(a). While computing the Megaflops rate of preconditioned CG, we include the operations required for sparse matrix-vector multiplication and for solving a system of the form $y = M^{-1}b$ in each iteration, where M is the preconditioner. We ignore the cost of vector operations. While there does not seem to be a clear overall pattern in how the speed of preconditioned CG changes with increasing preconditioner size, WSMP's CG implementation with blocking appears to achieve the highest Megaflops rate. The Megaflops rate of CG with WSMP's non-blocked preconditioner are comparable to those of other non-blocked solvers.

During the preconditioner generation phase, blocking would help in two ways. First, a single step of indirect addressing would afford access to a whole block of nonzeros instead of a single element, thus reducing overhead. Secondly, it would permit the use of higher level BLAS [20, 21], thus improving the cache efficiency of the program. Note that when we refer to the use of higher level BLAS, we do not necessarily mean making calls to a BLAS library. Typically, the blocks in sparse incomplete factors are small and it may not be efficient to make a library call for BLAS operations on these blocks due to large constant overheads, such as those for error checking, associated with typical BLAS library calls. The key here is to use the blocks to improve spatial and temporal locality for better cache performance, which in our implementation, we achieve through light-weight BLAS-like kernels instead of making calls to an actual BLAS library.

During the solution phase, a block column of size m improves temporal locality because each element of the vector, once loaded from the main memory, can be reused m times. A block row of size m improves spatial locality because m consecutive elements of the vector are used. These benefits are in addition to the reduction in indirect addressing overhead for accessing the elements of the preconditioner.

In the remainder of this section, we describe how blocking is used in WSMP's iterative solver for sparse symmetric positive definite systems. Since we are working with symmetric matrices only, henceforth, we will only mention column blocks because a column block implies an equivalent row block. We shall also refer to these blocks as supernodes. Note that in direct sparse solver literature, the term supernode typically refers to block rows or columns of the complete factors whose structure is determined through a symbolic factorization step [30, 47]. We refer to the block columns of the incomplete factors as supernodes, whose structure is either based on the natural block structure, if any, of the coefficient matrix, or is determined dynamically during the incomplete factorization process.

2.2. Incomplete Factorization with Static Blocking. The coefficient matrices of many sparse systems have a naturally occurring block structure. Such matrices usually result from the association of multiple unknowns with each node of the grid used to discretize coupled differential equations. The columns of the coefficient matrix corresponding to the unknowns associated with the same grid point have identical nonzero patterns. These column blocks can be detected [2] easily and are treated as single columns for all symbolic purposes. In WSMP, we relax the criterion for a group of columns to belong to a single block by permitting all columns with a user selected percentage of overlap in their indices to belong to a single block. We do this by first detecting exact blocks using Ashcraft's algorithm [2] and then making a second pass that searches the neighbors of each block column for nearly identical patterns for inclusion in the block. This notion of relaxed row/column blocks (supernodes) is used fairly routinely in direct solvers [3]. The column labeled *BlkSz* in Table 1.1 shows the average size of (relaxed) blocks of columns detected by WSMP with 90% or more overlap. Note that the matrices *af_shell7* and *nastran-b* used in Figure 2.1, have average block sizes of 5.0 and 3.04, respectively.

The first step in our static block incomplete Cholesky algorithm is to construct a supernodal elimination tree. An elimination tree [44, 45] is a minimum task dependency graph for factoring a sparse matrix and is used frequently in sparse direct solvers. Each vertex in the tree corresponds to a column of the sparse matrix and a column can be factored after the columns corresponding to all its children in the elimination tree have been factored. In a supernodal elimination tree, each vertex

```
1.  begin function sup_inc_chol1(S_{k,m})
2.     for each child C of S in the elimination tree do
3.        sup_inc_chol1(C);
4.     end for
5.     Construct the potential row index set of S as the union of its original index
          set and that of all supernodes that have a row index j, k ≤ j < k + m;
6.     Allocate l × m space for S, where l is the number of potential row indices;
7.     Update columns k...k + m − 1 by supernodes that have a row index j,
          k ≤ j < k + m, using level 3 BLAS;
8.     Perform dense panel factorization on the l × m supernode S;
9.     Apply dropping strategy to reduce l to l_final;
10.    Compact and store l_final × m supernode S;
11. end function sup_inc_chol1.
```

FIG. 2.2. *Outline of incomplete Cholesky factorization with static supernodes.*

represents a block column or supernode instead of an individual column. We denote a supernode by $\mathcal{S}_{k,m}$, which is uniquely identified by the index k of its starting column and the size (number of columns) m. The factorization process can be viewed as a postorder depth-first traversal of the elimination tree, as shown in Figure 2.2.

Our block incomplete factorization is a left-looking algorithm [30]. While visiting a supernode $\mathcal{S}_{k,m}$, the corresponding block of columns $k...k+m-1$ is assembled and updated just before it is factored. Recall that these columns have a nearly identical structure in the coefficient matrix. If $l_{initial}$ is the number of row indices in this block column in the coefficient matrix, then it is stored in an $l_{initial} \times m$ array, with a single integer array of length $l_{initial}$ storing the row indices for all columns in the block. If an index is missing in any column, then an explicit zero is stored in the corresponding location of the block. We maintain this type of a data-structure for $\mathcal{S}_{k,m}$ at all stages of the incomplete factorization process. During the process of getting updated, the supernode $\mathcal{S}_{k,m}$ incurs fill-in. All incomplete factor block columns (which belong to the subtree rooted at $\mathcal{S}$) that have a nonzero in any row j, $k \leq j < k+m$, contribute to the block column $k...k+m-1$. Let us assume that the number of row indices in $\mathcal{S}_{k,m}$ after the update is l. Note that $l_{initial} \leq l$. Once updated, a dense panel factorization is performed on this $l \times m$ block. Finally, the dropping strategy (described in detail in section 2.3) is applied to the rows of this supernode.

2.3. Dropping Strategy. In this section, we describe the dropping strategy used in WSMP's incomplete block Cholesky factorization to reduce the length of an $l \times m$ supernode after it is factored. WSMP implements a dual dropping strategy of the form introduced by Saad [55]. A user defined threshold (τ,γ) suggests that (1) an entry $L(i,k)$ in the incomplete lower triangular factor L whose magnitude is smaller than $\tau \times L(k,k)$ can be dropped, and (2) we strive to restrict the final length l_{final} of column k of L to $\gamma \times l_{initial}$. Note that, for dropping an element from a factored column, we consider its magnitude relative to that of the corresponding diagonal entry. Other dropping strategies have been used for incomplete factorization. These include dropping $L(i,k)$ based on its magnitude relative to the 2-norm of column k of L [55, 56] and dropping $L(i,k)$ if $L(i,k) \times L(i,k)$ is smaller than $\tau \times L(k,k) \times L(i,i)$ [52]. We found dropping $L(i,k)$ based on its magnitude relative to that of L(k,k) to be more effective than than the other two on an average for the problems in our test suite.

Bollhöfer [49] proposed a dropping strategy in which $L(i,k)$ is dropped based on its magnitude relative to an approximate norm of row k of L^{-1}. We implemented Bollhöfer's dropping criterion and found that while it was more robust than our simpler criterion at avoiding the breakdown of the incomplete Cholesky process, it was slower in general. When combined with our technique discussed in section 3 to address Cholesky breakdown, our dropping method outperformed Bollhöfer's method on our suite of test problems. This is also evident from a comparison in section 5 of the robustness and performance of WSMP ICT preconditioner with that of ILUPACK, which implements Bollhöfer's technique.

In our blocked incomplete Cholesky factorization, we apply the dropping criterion described above to entire rows of a supernode $\mathcal{S}_{k,m}$. A row i that satisfies the dropping criterion is dropped in its entirety; i.e., all elements $L(i,k)...L(i,k+m-1)$ are dropped. Other rows are kept as part of the incomplete factor in their entirety. To implement row dropping, we compute a drop score $dscr_{\mathcal{S}}(i)$ for each row $i \geq k+m$ of the factored supernode $\mathcal{S}_{k,m}$ as follows:

$$dscr_{\mathcal{S}}(i) = \frac{1}{m} \sum_{j=k,k+m-1} \left| \frac{L(i,j)}{L(j,j)} \right|, i \geq k+m. \tag{2.1}$$

Row i is dropped if $dscr_{\mathcal{S}}(i) < \tau$, where $i \geq k+m$. No dropping takes place within the $m \times m$ diagonal block of $\mathcal{S}_{k,m}$.

After dropping the rows based on their drop scores, the number of remaining rows l_{remain} in $\mathcal{S}_{k,m}$ may still be greater than the target of $\gamma \times l_{initial}$. If this is the case, then we compute a secondary drop-tolerance $\tau_{\mathcal{S}}$ for $\mathcal{S}_{k,m}$ as follows:

$$\tau_{\mathcal{S}} = \frac{d_{max} \times d_{min}}{d_{min} + \frac{\gamma \times l_{initial}}{l_{remain}}(d_{max} - d_{min})}. \tag{2.2}$$

Here d_{max} is the maximum drop score less than or equal to 1.0 in $dscr_{\mathcal{S}}$ and d_{min} is the minimum drop score greater than or equal to τ. We now perform a second round of dropping and drop row i if $dscr_{\mathcal{S}}(i) < \tau_{\mathcal{S}}$. The rationale behind the choice of d_{max} is that we never consider any rows with drop scores greater than 1.0 for dropping. Equation 2.2 chooses $\tau_{\mathcal{S}}$ such that $\tau_{\mathcal{S}}^{-1}$ partitions the range between d_{max}^{-1} and d_{min}^{-1} in the ratio $\gamma \times l_{initial}$ to $l_{remain} - \gamma \times l_{initial}$.

There are multiple advantages to reinterpreting and applying the fill-factor γ in terms of a secondary drop-tolerance $\tau_{\mathcal{S}}$. First, it is much faster to apply the drop-tolerance $\tau_{\mathcal{S}}$ in a single pass rather than exactly determining $\gamma \times l_{initial}$ largest drop scores. Secondly, using a secondary drop-tolerance $\tau_{\mathcal{S}}$ offers some adaptability of the dropping criteria to the actual distribution of values in the supernodes; i.e., supernodes with a large number of high drop scores will retain more rows than supernodes with a large number of small drop scores.

As we shall see in section 2.4, our method of applying the fill-factor γ results in factors of sizes that are quite close to the desired size for the non-blocked version of our algorithm when γ is not too small. For the blocked version, it results in factors whose size is greater than γ times the size of the coefficient matrix. In section 2.4, we also discuss the impact of dropping or retaining entire rows of the supernodes on the quality of the preconditioner.

af_shell7 with blocking

	Thresholds	Fnnz (mill.)	Ftime (sec.)	#Iters	Stime (sec.)	Itime (sec.)	FStime (sec.)
1.	(3.5e-3,3.5)	43.6	7.4	106	44.4	0.42	51.8
2.	(1.0e-3,4.5)	55.1	12.9	62	28.8	0.46	41.7
3.	(5.0e-4,5.5)	61.2	16.9	46	22.8	0.50	39.7
4.	Direct	94.7	14.7	1	1.6	1.60	16.3

af_shell7 without blocking

	Thresholds	Fnnz (mill.)	Ftime (sec.)	#Iters	Stime (sec.)	Itime (sec.)	FStime (sec.)
5.	(3.0e-2,0.8)	12.5	5.9	292	40.8	0.14	46.7
6.	(1.5e-2,1.5)	16.7	9.5	198	36.9	0.19	46.4
7.	(7.5e-3,2.5)	23.3	17.2	145	46.3	0.32	63.5
8.	(3.5e-3,3.5)	30.3	27.2	103	49.2	0.46	76.4
9.	(1.0e-3,4.5)	41.9	49.4	64	41.0	0.64	90.4
10.	(5.0e-4,5.5)	50.7	69.5	55	42.7	0.78	112.2

nastran-b with blocking

	Thresholds	Fnnz (mill.)	Ftime (sec.)	#Iters	Stime (sec.)	Itime (sec.)	FStime (sec.)
11.	(1.5e-2,1.5)	119.5	55.4	390	572.7	1.47	628.1
12.	(7.5e-3,2.5)	164.5	106.5	287	477.6	1.66	584.1
13.	(3.5e-3,3.5)	217.4	195.6	226	419.3	1.86	614.9
14.	(1.0e-3,4.5)	309.2	434.3	193	412.3	2.14	846.6
15.	(5.0e-4,5.5)	369.2	650.8	124	287.8	2.32	938.6
16.	Direct	1021.3	653.1	1	8.7	8.73	661.8

nastran-b without blocking

	Thresholds	Fnnz (mill.)	Ftime (sec.)	#Iters	Stime (sec.)	Itime (sec.)	FStime (sec.)
17.	(3.0e-2,0.8)	71.2	111.7	470	544.1	1.16	655.8
18.	(1.5e-2,1.5)	97.0	169.2	307	478.5	1.56	647.7
19.	(7.5e-3,2.5)	129.7	238.6	235	491.9	2.09	730.5
20.	(3.5e-3,3.5)	169.7	386.2	185	464.6	2.51	850.8
21.	(1.0e-3,4.5)	248.2	776.5	172	533.9	3.10	1310.4
22.	(5.0e-4,5.5)	305.3	1147.5	123	428.5	3.48	1576.0

TABLE 2.1

Factor size (Fnnz), factorization time (Ftime), number of CG iterations (#Iters), solution time (Stime), per iteration time (Itime), and total factor and solve time (FStime) for af_shell7 *and* nastran-b *with and without blocking. A threshold* (τ,γ) *indicates that an entry* $L(i,k)$ *in the factor smaller than* $\tau \times |L(k,k)|$ *is dropped and we strive to restrict the length of column* k *of the incomplete factor* L *to* γ *times the length of the lower triangular part of column* k *of the coefficient matrix.*

2.4. Performance Evaluation of Blocked Incomplete Cholesky. In Figure 2.1, we compared the performance of both the preconditioner generation and the iterative solution phases of WSMP's static block incomplete Cholesky preconditioner with that of some non-blocked preconditioners for two of the matrices from our test suite. We now take a closer look at the impact of blocking by means of a detailed comparison with a non-blocked implementation of incomplete Cholesky factorization. Table 2.1 shows factor size (*Fnnz*), factorization time (*Ftime*), number of CG iterations (*#Iters*), solution time (*Stime*), per iteration time (*Itime*), and total factor and solve time (*FStime*) of both the blocked and the non-blocked versions for our incomplete Cholesky preconditioner for the same two matrices. A comparison of the performance of the solver with and without blocking clearly shows the improvement

that results from blocking. For the same dropping criteria, the blocked version of incomplete Cholesky factorization is roughly 2 to 4 times faster than its non-blocked counterpart. The time for each iteration is also generally smaller for the blocked version of the solver, although the difference is less stark. The best total time for the non-blocked version is achieved for much sparser factors than for the blocked version and remains higher than the best total time for the blocked version. This has implications for the robustness of the iterative solver. The two matrices considered in Table 2.1 are fairly well behaved and do not encounter a breakdown of the Cholesky process, even for very sparse factors. Incomplete Cholesky factorization is likely to breakdown for many matrices for thresholds that lead to very sparse factors, thus forcing the use of thresholds in a range in which the blocked algorithm is much more efficient than the non-blocked one.

Apart from the performance impact, there are other side effects of blocking that can be observed from the data in Table 2.1. An examination of the factor sizes (*Fnnz*) reveals that for the same thresholds, the blocked version of incomplete Cholesky may require substantially more memory than the non-blocked version. The sizes of the blocked factors are roughly 20-50% higher for *af_shell7* and roughly 20-30% higher for *nastran-b* compared to their non-blocked counterparts. For $\tau \geq 2.5$ in Table 2.1, the *Fnnz* entries for the non-blocked algorithm are remarkably close to $\gamma\times$ the number of nonzeros in the lower triangular part of the matrices, which is 9.05 million for *af_shell7* and 56.56 million for *nastran-b*. This shows that if γ is not too small, then Equation 2.2 translates a fill factor threshold into a drop tolerance threshold fairly accurately. Of course, this does not hold for the blocked algorithm, which retains a much larger number of nonzeros in the incomplete factor. The reason for the memory overhead is that the block columns may retain a substantial number of zero or small entries, which are discarded in the non-blocked version of incomplete factorization with the same dropping thresholds. Small or zero entries, for example, can be retained in a row of a supernode that has a drop score greater than the drop-tolerance due to a single large entry.

Larger factor size of the blocked algorithm may explain why the improvement in the per iteration time due to blocking is not as dramatic as the improvement in the incomplete factorization time. During blocked factorization, the use of level 3 BLAS offers a significant performance advantage over elementwise operations of a non-blocked factorization. The speed of level 3 BLAS completely overshadows the overhead due to computations performed on the extra zero or small entries stored in the supernodes. The solution phase with a blocked preconditioner, however, involves level 2 type computations, which offer less significant performance advantage over elementwise operations. In some cases, this performance advantage is not enough to overcome the overhead of the extra computation due to larger factors.

For *af_shell7*, the number of CG iterations for a given threshold is almost the same for the blocked and the non-blocked versions. However, for *nastran-b*, the blocked version of the solver seems to require more iterations than its non-blocked counterpart, particularly for thresholds that result in sparser factors. The reason for this is that the blocked algorithm can potentially drop an element from a supernode of width m even if that element is up to m times larger in magnitude than the dropping threshold. This would happen, for example, when only one $L(i, j)$ in Equation 2.1 is nonzero and the rest are all zeros.

To summarize our observations, the speedup resulting from the use of blocked incomplete Cholesky over its non-blocked counterpart comes at the expense of ad-

ditional memory requirement and some deterioration in the quality of the preconditioner. The reason is that drop-tolerances are applied inexactly in the blocked version of the algorithm. Within a supernode, it may retain some entries smaller than the drop-tolerance in order to maintain the block structure of the supernode and it may drop entries whose magnitude exceeds the drop-tolerance by a factor up to the width of the supernode. As a result, the block dropping approach may be particularly bad for certain problems that are inherently poorly scaled, such as semiconductor device simulation [27] or incompressible Navier-Stokes problems [17] with high Reynolds numbers. However, in such cases, a drop-tolerance based incomplete factorization itself is perhaps a poor choice of preconditioner, with or without blocking, and an incomplete factorization based on levels of fill would not only be a more suitable preconditioner, but would also not incur a quality penalty for using blocks. WSMP's iterative solver includes a user defined limit that can be used to restrict the maximum size of a supernode during incomplete factorization, should the memory or iteration count with the natural supernodes become excessive. The total factor and solve time numbers in the last column of Table 2.1 as well as the performance results in section 5, however, suggest that the performance advantage of blocking, in general, outweighs its drawbacks. This is particularly true for larger factor sizes, which may be required to solve harder problems and may be too costly to be practical for non-blocked solvers.

2.5. An Adaptive Blocking Approach. As we have seen in section 2.4, the use of block columns can give a substantial boost to the performance of an iterative solver based on incomplete factorization preconditioning. However all matrices do not have an inherent block structure that can be exploited. In our suite of test problems shown in Table 1.1, thirteen of the thirty matrices have an average block size of less than 2.0. Therefore, in WSMP, we have implemented two block incomplete factorization procedures: one that uses the statically determined block structure, and the other for those matrices that do not have a block structure in their original form. The software first attempts to determine a relaxed block structure, as discussed in section 2.2. If a suitable block structure with an average block size of at least 2.0 is found, then the static block algorithm is used. The results in Table 2.1 are from the static block algorithm because both *af_shell7* and *nastran-b* are suitable candidates for this algorithm. If the average block size in the coefficient matrix is less than 2.0, then a different algorithm is used, which dynamically determines a block structure as the incomplete factorization proceeds. We shall describe this algorithm in detail in section 2.6. The choice of the algorithms is transparent to the user and is made automatically by the software.

2.6. Incomplete Factorization with Dynamic Blocking. In this section, we describe a block incomplete Cholesky factorization algorithm for matrices that do not have a suitable initial block structure. Just like the algorithm described in section 2.2, this algorithm too is guided by an elimination tree. However, the elimination tree in this case is not a supernodal elimination tree. Instead, each vertex of the tree represents a single column of the coefficient matrix.

We first recall some basic properties of elimination trees [44, 45] in the context of complete Cholesky factorization. In our discussion of elimination trees, we assume that a postorder numbering has been applied to the tree nodes, and consequently to the rows and columns of the matrix. This ordering numbers the nodes in the order in which they are processed and has no impact on the fill-in during factorization. With this numbering, if k is the only child of its parent, then the parent of k is $k+1$. In an elimination tree, if k is a child of l, then the set of row indices in the subdiagonal

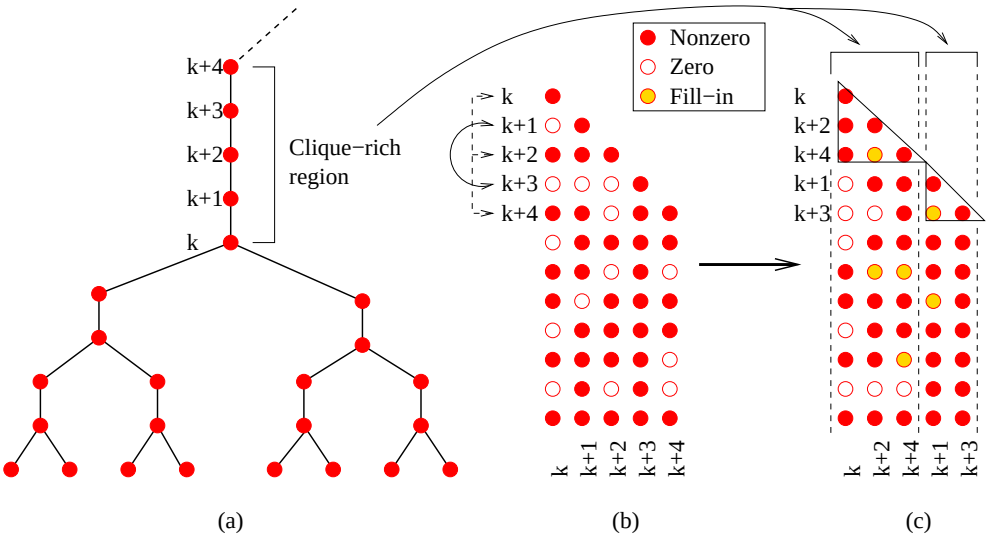

FIG. 2.3. *(a) Subtree of a hypothetical elimination tree. (b) Hypothetical structure of columns k ... $k+4$ just before these columns are factored. (c) Reordering of columns k ... $k+4$ to form two block columns of size 3 and 2, respectively.*

portion of column k of the lower triangular factor L is a subset of the set of row indices in column l. Moreover, if k is the only child of l, then the set of row indices in column l of L consists of (1) the set of row indices in the subdiagonal portion of column k in L, and (2) the set of rows that have a nonzero in the lower triangular part of column l of the original matrix, but not in column k of L. As a result, in complete Cholesky factorization, sets of columns of the factor L of the form $\{k, k+1, ..., k+m-1\}$ tend to have identical (when set (2) above is empty) or nearly identical structures if $k+i$ is the only child of $k+i+1$, $0 \leq i < m-1$.

With this background, we now describe our block incomplete factorization algorithm for matrices without a natural block structure. Once again, just like the algorithm in section 2.2, we proceed with a left-looking approach following the elimination tree. In that algorithm, just before factorization, we assemble and update columns $k...k+m-1$ if they belong to the same statically determined supernode. In the dynamic blocking algorithm, illustrated in Figure 2.3, we assemble and update columns $k...k+m-1$ if nodes $k+1...k+m-1$ have only one child in the elimination tree. We maintain a single row index set of length l for this block column, padding the block with zeros where necessary. We then scan the first m row indices of column k for nonzeros and perform a local symmetric permutation such that these rows are in adjacent locations starting at k. For example, in Figure 2.3(b), rows k, $k+2$, and $k+4$ have a nonzero in column k. So we permute the indices of the matrices to make k, $k+2$, and $k+4$ adjacent. In general, if r_1 nonzeros are found within rows k to $k+m-1$ of column k, then after permutation, we have an $l \times r_1$ column block that we treat as the first supernode of the section $k...k+m-1$ of the matrix. We perform dense panel Cholesky factorization on this block, drop rows based on the dropping strategy describe in section 2.3, update the remaining $l \times (m-r_1)$ portion of the block that we had assembled, and finally, store the $l_{final} \times r_1$ block corresponding to the newly discovered supernode $\mathcal{S}_{k,r_1}$. The next step is to scan column $k+r_1$ for nonzero row entries between $k+r_1$ and $k+m-1$ in order to discover the second supernode

of size r_2. The process is repeated until all m columns are factored.

A few practical implementation details of the above algorithm are worth mentioning. Since m can be very large, especially close to the root of the elimination tree, we place an upper limit of 32 on m in order to avoid making l too large and using an excessive number of zeros for padding. Thus, we break larger straight sections of the elimination tree into blocks with a maximum of 32 columns. In section 2.4 and Table 2.1 we observed that the effectiveness of blocking for reducing the total computation time increases with the density of the incomplete factors and the degradation in the quality of the preconditioner increases as the preconditioner gets sparser. Therefore, in addition to the search zone m, we must also restrict the size r_i $(0 < i \leq m)$ of the supernodes. In our software, we use an upper bound of $\max(1, \lfloor \gamma \rfloor)$ as the default upper bound on r_i, where (τ, γ) is the user defined defined threshold for dropping. This choice of upper bound on r_i lets the preconditioner adapt the blocking to its density, thus maximizing its potential performance benefit and minimizing the quality degradation.

We omit showing separate results for this algorithm, which, for matrices without a natural block structure, look similar to those in Table 1.1 without a significant degradation in the quality of the sparser preconditioners due to our choice of the upper bound on r_i.

2.7. Related Work. Use of blocking in incomplete factorization has been considered by many researchers. Recognizing the potential role of blocking in improving the performance of iterative solvers, Chow and Heroux [12] propose a general object-oriented framework to support preconditioning with sparse or dense blocks. Axelsson [4], Beauwens and Bouzid [6], Chow and Saad [13], and Saad and Zhang [58] etc. have considered the use of sparse blocks to improve locality in preconditioner generation and application. Naturally occurring dense blocks have been used by Kaporin et al. [42] to obtain a block SSOR preconditioner that uses the inverses of the entire dense diagonal blocks instead of just the diagonal elements as in the conventional SSOR method [62]. In BlockSolve95 [41], the naturally occurring dense blocks are used in the context of incomplete factorization based on levels of fill. Fan et al. [27] have used such blocks in a two-level preconditioning scheme for semiconductor device simulation and Benzi et al. [8] have used them for constructing a block sparse approximate inverse preconditioner for structural mechanics applications. Hénon et al. [37] propose using blocks determined by a symbolic factorization step to improve the performance of incomplete factorization based on levels of fill. In all these cases, the preconditioner has a static predefined structure. Unlike these methods, we apply blocking to a matrix structure that evolves during factorization with dropping based on values instead of position. Ng et al. [53] propose performing a symbolic factorization step, as used in direct solvers [47], to identify block columns for a sparse incomplete Cholesky factorization with threshold based dropping. In contrast, for matrices without natural dense blocks, we identify and even create block columns dynamically in regions of the matrix where there is a high probability of finding such blocks; i.e., those regions of the matrix where complete factorization would have created dense blocks.

3. Handling Breakdown of the Cholesky Process. While the Cholesky factorization LL^T of a symmetric positive definite matrix is guaranteed to exist, there is no such guarantee of the existence of an incomplete factorization of this form. The reason is that the errors introduced due to dropping entries from the factor may result in zero or negative values at the diagonal. Many remedies proposed in the literature [7] to handle the breakdown of the Cholesky process can either add cost [9, 59]

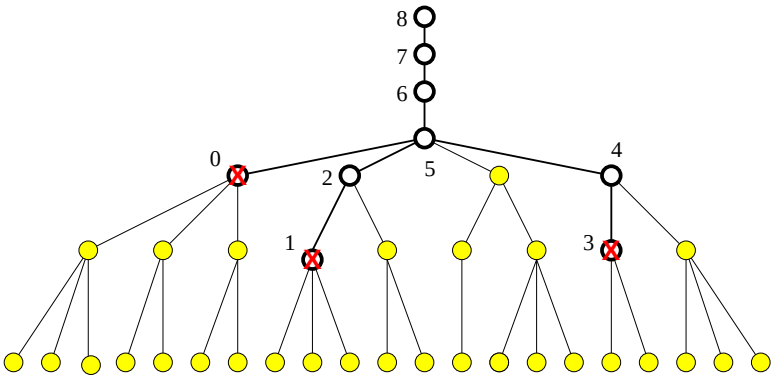

FIG. 3.1. *If the Cholesky process breaks down at the nodes marked 'X', then a direct LDL^T factorization is applied to an incomplete Schur complement consisting of the labeled nodes.*

or errors [1], or both for matrices that may not have suffered a breakdown without applying the remedy. Some methods [48] involve costly restarting of the incomplete factorization, possibly multiple times. Some others apply local perturbations in response to an unsuitable pivot [43], which is an inexpensive strategy, but is generally not very effective [7].

We handle breakdowns in the Cholesky process by simply dropping the onerous requirement that the preconditioner be positive definite. Note that the conjugate gradient algorithm requires that both the coefficient matrix and the preconditioner be positive definite. As described in section 2, our incomplete factorization algorithm is guided by the elimination tree. We stop the Cholesky process at any node in the tree where a small or negative pivot is encountered. This is illustrated in Figure 3.1. The incomplete Cholesky process can continue on any node that does not encounter breakdown or is not on the path between the root and a node with breakdown. For example, if the nodes with an 'X' in Figure 3.1 encounter an unsuitable pivot, then only the unlabeled nodes participate in incomplete Cholesky factorization.

The detailed incomplete factorization algorithm that can handle Cholesky breakdowns is given in Figure 3.2. The first pass of the algorithm, namely *sup_inc_chol2*, is a modification of the algorithm of Figure 2.2 to account for failures in the incomplete Cholesky process with predetermined supernodes. A version for dynamically determined supernodes discussed in section 2.6 can be formulated along similar lines. In the algorithm in Figure 3.2, a node is considered to have failed if it encounters an unsuitable pivot or if any of its children in the elimination tree have failed. If incomplete Cholesky cannot complete all the way to the root, then the second pass of the algorithm, namely *construct_schur*, is invoked. This function visits only those nodes that failed in *sup_inc_chol2* (for example, the nodes labeled 0 to 8 in Figure 3.1) and constructs an incomplete Schur complement matrix consisting of all the columns of the matrix that belong to the failed nodes. We call it an incomplete Schur complement because after generating a column of this matrix, we drop entries whose magnitude relative to that of the corresponding diagonal entry is smaller than $\tau^{1.5}$, where (τ, γ) is the dual threshold for dropping entries of L. The reason for using a smaller drop tolerance, $\tau^{1.5}$, is that in this case, we are dropping entries before factorization and there is a higher chance of dropping meaningful entries. We have experimentally confirmed that using $\tau^{1.5}$ instead of τ for dropping at this stage typically results in a

```
1.  begin function sup_inc_breakdownfree()
2.      /* Let R be the root supernode. */
3.      fail(R) = sup_inc_chol2(R);
4.      if fail(R) ≠ 0 then
5.          shr_size = construct_schur(R,0);
6.          Perform sparse direct LDL^T factorization of shr_size × shr_size Schur
                complement matrix;
7.      end if
8.  end function sup_inc_breakdownfree.
9.
10. begin function sup_inc_chol2(S_{k,m})
11.     fail(S) = 0;
12.     for each child C of S in the elimination tree do
12.         fail(S) += sup_inc_chol2(C);
13.     end for
14.     if fail(S) == 0 then
15.         Construct the potential row index set of S as the union of its original index
                set and that of all supernodes that have a row index j, k ≤ j < k + m;
16.         Allocate l × m space for S, where l is the number of potential row indices;
17.         Update columns k...k + m − 1 by supernodes that have a row index j,
                k ≤ j < k + m, using level 3 BLAS;
18.         Perform dense panel factorization on the l × m supernode S;
19.         if factorization of S completes without breakdown then
20.             Apply dropping strategy to reduce l to l_final;
21.             Compact and store l_final × m supernode S;
22.         else
23.             fail(S) = 1;
24.         end if
25.     end if
26.     return(fail(S));
27. end function sup_inc_chol2.
28.
29. begin function construct_schur(S_{k,m},insize)
30.     outsize = insize;
31.     if fail(S) ≠ 0 then
32.         for each child C of S in the elimination tree do
33.             outsize += construct_schur(C,outsize);
34.         end for
35.         Construct the potential row index set of S as the union of its original index
                set and that of all supernodes that have a row index j, k ≤ j < k + m;
36.         Allocate l × m space for S, where l is the number of potential row indices;
37.         Update columns k...k + m − 1 by supernodes that have a row index j,
                k ≤ j < k + m, using level 3 BLAS;
38.         for (j = k; j < k + m; j = j + 1) do
39.             Drop small elements from column j and add it as the outsize-th column of
                    the Schur complement matrix;
40.             outsize++;
41.         end for
42.     end if
43.     return(outsize);
44. end function construct_schur.
```

FIG. 3.2. *Outline of a supernodal incomplete factorization algorithm that can handle breakdown of the Cholesky process for SPD matrices.*

better quality preconditioner. Finally, we use direct LDL^T factorization on the Schur complement matrix.

During the solution phase, we chose the Krylov subspace method depending on whether or not any failures were encountered in incomplete Cholesky factorization. CG is used if the preconditioner is SPD, and GMRES or symmetric QMR is applied if the preconditioner is indefinite. The choice of the appropriate solver is made in the software without intervention from the user.

Note that this method is appropriate for inexpensively recovering from a moderate number of failures. An excessive number of failures may result in the loss of convergence even with a general solver like GMRES or QMR because of excessive error in the preconditioner, of which Cholesky breakdown is merely a symptom. Nevertheless, experiments on a test suite of a reasonable size in section 5 suggest that the algorithm in Figure 3.2 is generally robust. Nine of the 30 matrices in our test suite encountered a breakdown of the Cholesky process, and of these, only *bst-2* and *inline_1* failed to converge in a reasonable number of iterations. An additional benefit of this technique is that it can handle mildly indefinite symmetric matrices; i.e., matrices with a few negative eigenvalues relative to their dimension. The algorithm is particularly well suited for those matrices in which the negative eigenvalues correspond to existing zero or negative diagonal entries in the coefficient matrix. In such matrices, these diagonals can be permuted in advance to locations near the root of the elimination tree in order to limit the size of the Schur complement matrix subject to direct LDL^T factorization.

4. Selection and Tuning of Parameters. The performance of incomplete factorization preconditioners with threshold based dropping is extremely sensitive to the choice of the thresholds [31]. Therefore, a fair amount of experimentation may be required to find the suitable ordering, drop tolerance, and fill factor for a given application. Even if generally good parameters are found for a given application, it is impractical to manually tune the parameters to individual problems. To make matters worse, many applications require the solution of a series of systems with the coefficient matrices changing gradually. The set of parameters that are best for the first system may not be optimum for the later ones because successive matrices in the sequence may become progressively more or less ill-conditioned.

We have built features in our incomplete Cholesky factorization based preconditioner to make initial parameter selection, to minimize the impact of initial parameter selection, and to automatically monitor and tune the thresholds for dropping when multiple systems with changing coefficient matrices are solved. We start by using reverse Cuthill-McKee (RCM) [16, 23] ordering and (7.5e-3,2.5) as the dual threshold by default if the matrix is diagonally dominant; otherwise, we use nested dissection [30, 34] ordering and (1.0e-3,4.75) as the threshold. The reason is that sparser incomplete factors may be sufficient for diagonally dominant matrices, while others may require denser factors for robust preconditioning. The default thresholds have been experimentally determined to be generally good choices for the two classes of matrices. The choice of ordering is based on our and others' [24] observation that RCM generally performs better with low fill-in and nested dissection performs better with relatively higher fill-in.

For applications that need to solve a sequence of systems with gradually changing coefficients, the solver incorporates a relatively simple search strategy to change the thresholds in each iteration in an attempt to minimize the sum of preconditioner generation and iterative solution time. After solving the first system, the solver

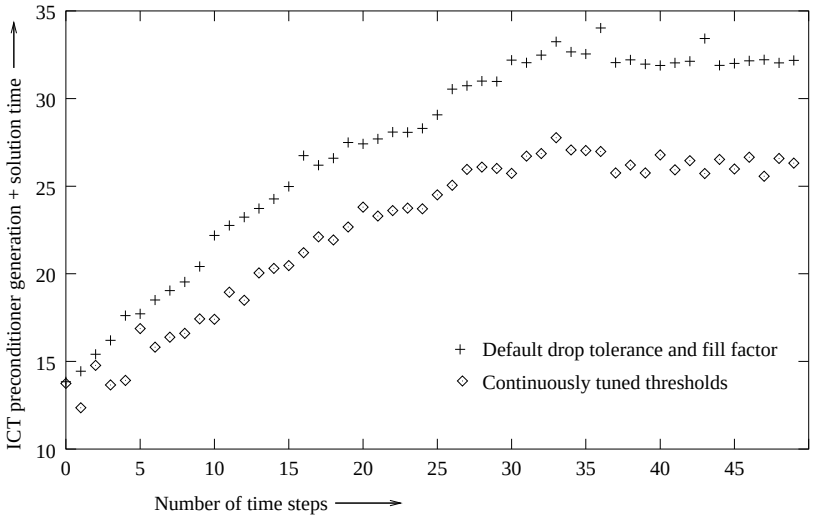

FIG. 4.1. *Comparison of the sum of preconditioner generation and iterative solution time with and without automatic tuning for 50 time steps of the solution of a diffusion equation.*

chooses an initial search direction (i.e., whether to increase or reduce fill) and a search step size (i.e., percentage change in the thresholds) based on the relative time spent in preconditioner generation and iterative solution. It determines the threshold for the next system based on the previous threshold, the search direction, and the step size. The search direction is changed if the new solution time is more than the previous one, else it remains the same. The step size is gradually increased in each step that the search direction stays the same (to reach the optimum point quickly) and is reduced each time the search direction reverses (to choose thresholds with values between the previous two choices).

Figure 4.1 demonstrates the benefits of continuous adjustment of dropping thresholds with an application involving a finite volume discretization of the diffusion equation as the test case. The figure shows the total solution time for linear systems in each of the 50 time steps of the application, both with and without automatic tuning tuned on. The '+'s plot the time for solving each system with the default thresholds and the diamonds plot the time, when the thresholds are adjusted according to the search method described above. The plot clearly shows the widening gap between the two choices of thresholds as the time steps proceed.

Note that there are two potential drawbacks to our threshold tuning method. First, the search could get trapped into a local minimum. This problem could be avoided by using more sophisticated search techniques, while still taking advantage of the key idea that solving multiple systems with gradually changing coefficient matrices presents an opportunity to automatically tune the parameters of the linear solver. The second problem is that if the search parameters are a function of the preconditioner generation and iterative solution times, then they could change from one run to another depending on slight variations in the measured time. Therefore, running the same application multiple times could yield different results. This may not be a concern in certain situations. For situations in which bitwise reproducible results are desired, WSMP permits the user to switch to tuning based on the approximate operation counts of preconditioner generation and iterative solution, instead of their times.

Tuning aimed at minimizing operation counts rather than times is likely to be less accurate though, because the relative Flops rate of preconditioner generation and the solution phases may change from one machine to another and from one application to another.

5. Experimental Comparison with Other Preconditioners. In this section, we compare the performance of the incomplete Cholesky factorization based preconditioner in WSMP (development version 7.12) with several preconditioners from some well-known iterative solver packages, such as PETSc [5] (release version 2.3.3-p0), Trilinos [39] (release version 8.0.3), Hypre [26] (release version 2.0.0), and ILUPACK [51] (development version 2.2). The overall set of preconditioners that we use to compare with WSMP's blocked ICT (threshold based incomplete Cholesky) includes IC(K) (levels of fill based incomplete Cholesky [61]) and ILUTP (threshold based ILU with pivoting [14, 56]) preconditioners from PETSc; IC(K) [40], ILUT (dual threshold based ILU [40]), ParaSails (sparse approximate inverse [11, 15]), and BoomerAMG (algebraic multigrid [38, 54]) preconditioners from Hypre; IC(K), ICT, ILUT, and ML (multilevel or algebraic multigrid [29, 60]) preconditioners from Trilinos; and ML ICT (multilevel ICT [50]) from ILUPACK. We used CG for symmetric preconditioners and tried GMRES with restart values of 30, 65, and 100 for the unsymmetric preconditioners. Most preconditioners have multiple tunable parameters. In addition, the solver packages often provide multiple preprocessing options, such as different orderings. For each preconditioner, we tried a fairly comprehensive set of preprocessing and parameter combinations [31]. A total of 806 solver, preconditioner, and parameter combinations were tried for each matrix.

We use performance profiles [19] for comparing different solvers. Performance profiles are a highly effective means of simultaneously comparing performance and robustness. We compare the memory requirement as well as the total preconditioner generation and solution time and present two sets of experimental results for each. For each preconditioner-package combination, through our comprehensive (but not exhaustive) trials, we picked the parameter combination that had the best overall performance on the entire test suite. Our first set of results compare the various preconditioners with their overall best configuration. For our second set of results, we pick a preconditioner's best parameter configuration from our trials individually for each test matrix.

The measure of quality that we use for chosing the overall best and problemwise best parameter configurations is the product of the memory used by the solver for this configuration and the total time required for preconditioner generation and the solution of the system. Henceforth, we shall refer to this measure as the memory-time product. The reason for choosing the memory-time product as a measure of quality for selecting the best set of parameters is that neither time, nor memory would be adequate for this purpose, when considered individually. For most preconditioners, there is a range of parameter choices in which there is a trade-off between solution time and memory consumption, although it is possible to make parameter choices that increase or decrease both time and memory simultaneously. The optimum operating point of a preconditioner for a given problem lies in a trade-off zone. Therefore, picking the parameters for these preconditioners based solely on either their time or memory requirements would simply yield winners that are extreme cases and are of little practical interest. For example, considering memory consumption only would favor a very sparse low-quality preconditioner that may require too many iterations to solve. On the other hand, the direct solver results in the overall fastest solution

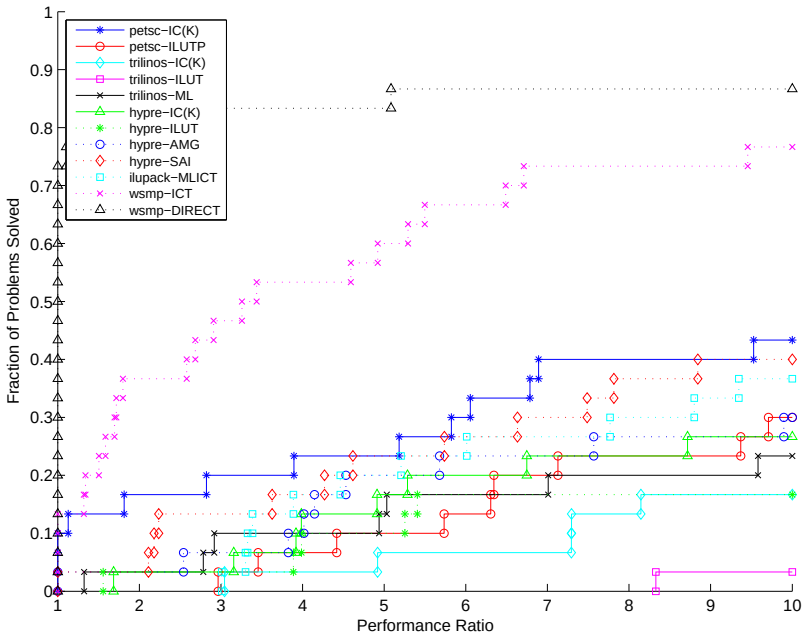

FIG. 5.1. *Time profiles of various preconditioners with configurations resulting in the smallest overall memory-time products.*

time for many problems in our test suite, albeit at the cost of a significantly high memory consumption. A certain parameter configuration could simply emulate the direct solver and emerge as the fastest configuration.

Figure 5.1 shows the time profiles of the overall best configurations of various preconditioners, along with the time profile of WSMP's direct solver. The x-axis in this figure plots the *performance ratio* and the y-axis plots the fraction of problems solved. In this case, the performance ratio of a preconditioner for a given problem is simply the ratio of total time (sum of preconditioner generation and iterative solution time) of that preconditioner to the least total time taken by any preconditioner to solve that problem. For example, the interpretation of the point (5.0,0.6) on the curve for WSMP's ICT preconditioner means that this preconditioner solved 60% of the problems in the test suite in time that was within a factor of 5 of the best time for these problems. Similarly, the point (1.0,0.7) on the curve for the direct solver indicates that it was faster than any of the other solvers for 70% of the problems. This is followed by PETSc IC(0), which is the fastest preconditioner for about 19% of the problems, presumably those with a good degree of diagonal dominance. However, understandably, PETSc IC(0) does not do as well for more difficult problems and its time profile curve is soon surpassed by that of WSMP ICT. Note that Figure 5.1 displays the time profile curve for PETSc IC(0) because the number of levels of fill K is an input parameter of the IC(K) preconditioner, and for PETSc IC(K), the fastest overall solution time was obtained for $K = 0$.

Figure 5.2 shows the memory profiles of the overall best configurations of various preconditioners, along with the memory profile of the direct solver. Hypre BoomerAMG and PETSc IC(0) appear to be the most memory efficient; however, Hypre BoomerAMG is more robust and is able to solve 80% of the problems as compared to

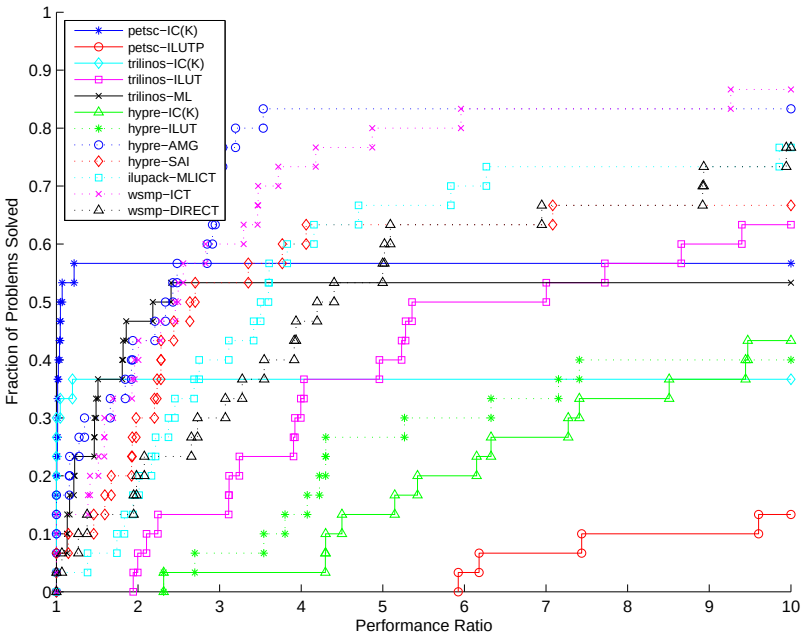

FIG. 5.2. *Memory profiles of various preconditioners with configurations resulting in the smallest overall memory-time products.*

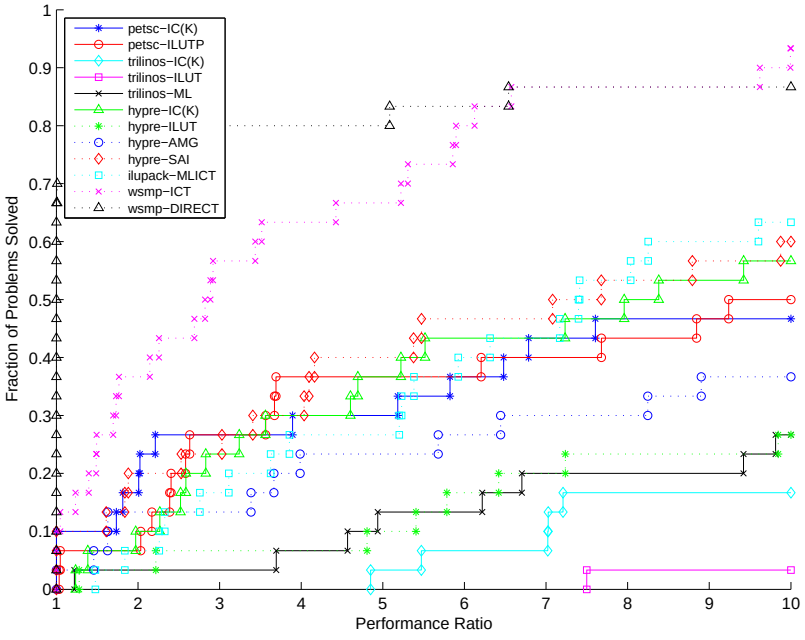

FIG. 5.3. *Time profiles of various preconditioners with parameter configurations for the least memory-time product for each problem.*

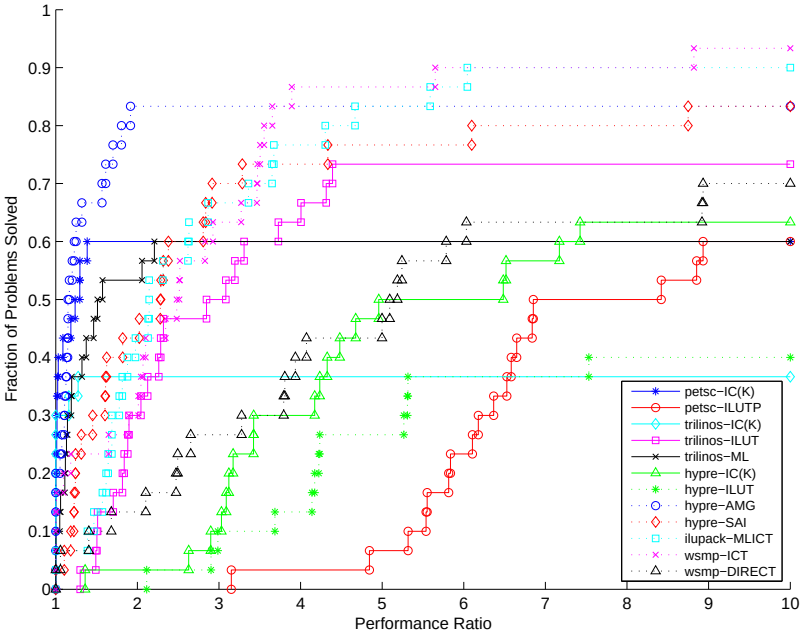

FIG. 5.4. *Memory profiles of various preconditioners with parameter configurations for the least memory-time product for each problem.*

60% for PETSc IC(0). The next best group of curves includes those for Trilinos ML, Hypre ParaSails, WSMP ICT, and ILUPACK. Note that ILUT preconditioner implementations appear to have the worst memory profiles. This is primarily because they need to store both triangular factors instead of just one, as in the case of symmetric preconditioners. There are at least six preconditioners that handily outperform the direct solver in terms of memory consumption.

Figure 5.3 shows the time profiles of all preconditioners when the parameter configuration for each preconditioner was chosen individually to minimize its memory-time product for each problem. While the best parameter configuration for the other solvers was chosen manually through experimentation, WSMP ICT was made to run through four steps of self tuning (section 4) and the time of the step with the least memory-time product was picked. A comparison of Figures 5.1 and 5.3 shows that the time of almost all preconditioners, including WSMP ICT, improves significantly when the parameters are tuned for individual problems. The degree of improvement, though, varies from one preconditioner to the other. ILUPACK appears to be quite sensitive to its input parameters and shows the most improvement with problemwise parameter selection.

Figure 5.4 shows the memory profiles of all preconditioners when, once again, the parameter configuration for each problem was chosen individually to minimize its memory-time product for each problem. Just like the time profiles, the memory profiles of the preconditioners improve significantly when compared to those for the overall best parameter configurations in Figure 5.2.

Among the iterative solvers compared in this section, WSMP ICT clearly has the smallest performance ratios for the most number of matrices in the time profiles with both the overall best and problemwise best parameters. In the memory profiles, it

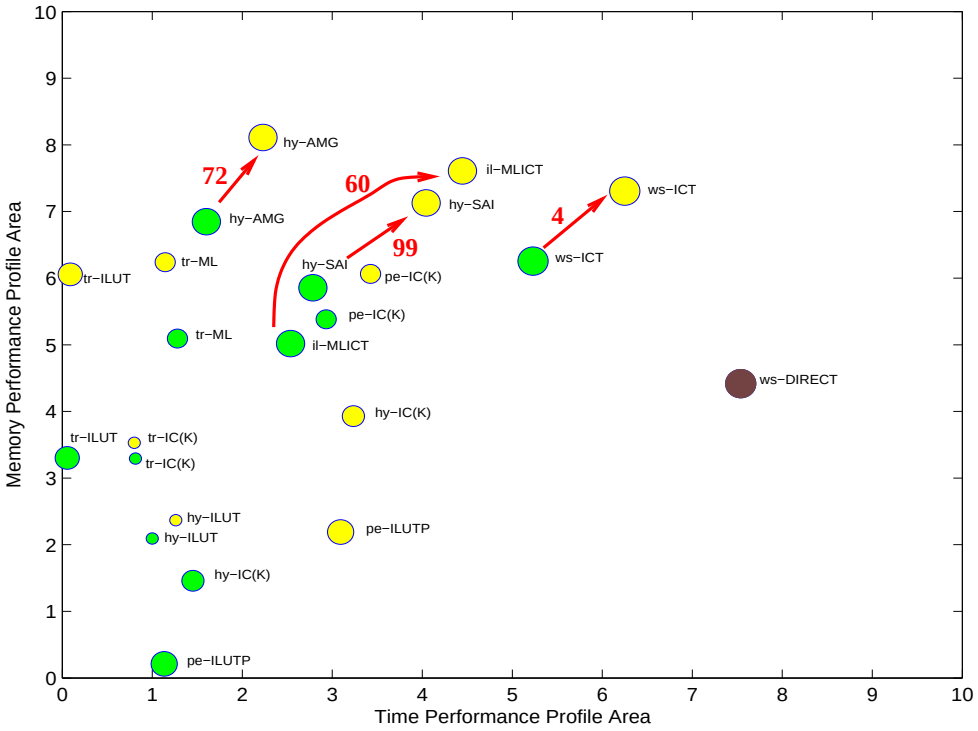

FIG. 5.5. *Plot of the time profile area versus the memory profile area for various preconditioners. Each circle represents a preconditioner whose name consists of the first two letters of the name of the package followed by the type of preconditioner. The size of a circle is proportional to the number of problems solved. The dark circles correspond to profile areas for the overall best parameter configuration and the light ones correspond to profile areas for problem-specific best parameters.*

is not the best, but is among the top performers. This is expected because, as we discussed in detail in section 2, our blocked version of incomplete Cholesky factorization does require more space than its non-blocked counterpart. In order to assess the tradeoff between memory and time for WSMP's iterative solver, we computed the memory-time product in our experimental results. We found that with problemwise best parameters, among the iterative solvers, WSMP's memory-time product was the smallest for 15 out of the 30 test cases. This was followed by six for PETSc IC(0), five for Hypre ParaSails, two for Hypre BoomerAMG, and two for ILUPACK.

A comparison of the memory and time performance of the iterative solvers relative to WSMP's direct solver confirms the conventional wisdom that direct solvers are generally fast and robust, but require more memory. Conventional wisdom also holds that the preconditioned iterative solvers should outperform the direct solver on larger problems. In addition, the performance crossover point between iterative and direct solvers would be observed for relatively larger matrices that result from two dimensional physical problems as compared to three dimensional ones. Our results indicate that, although the dimension of 40% of the matrices in our test suite is more than half a million and half the problems are three-dimensional (Table 1.1), the average problem size is still too small for most iterative solvers to outperform the direct solver in terms of solution time.

Different preconditioners and solvers have different strengths and weaknesses.

They have different degrees of robustness. Some are more memory efficient than others, while some are faster than others. They have different degrees of sensitivity to parameter tuning. Figure 5.5 displays all this relative information about the performance of various preconditioners by means of a single information-rich graphic. The performance profile curves shown in Figures 5.1–5.4 enable a visual comparison of the performance and robustness of the preconditioners. For Figure 5.5, we introduce a quantitative measure of performance, which is the area under a performance profile curve for a given upper limit on performance ratio (which we have chosen to be 10 in this paper). This figure has two sets of circles for each preconditioner. The dark circles correspond to the overall best parameter configurations and the light circles correspond to problem-specific best parameters. The x- and y-coordinates of each circle are the areas of the time and memory profiles of the corresponding preconditioner derived from Figures 5.1–5.4. The size of each circle is proportional to the number of problems solved. Thus, Figure 5.5 gives a multidimensional ranking of various preconditioners (including the direct solver). It shows that WSMP ICT has the highest ranking on the time axis among the iterative solvers. Its memory performance, while not the best, is a close third. The thick arrows in the figure show the number of trials that were performed for each matrix for some of the key preconditioners to select the best performing configuration. Note that WSMP ICT achieved a reasonably good performance improvement along both the time and memory axes within just four trials. This provides empirical evidence of the effectiveness of the parameter selection and tuning methods discussed in section 4.

6. Concluding Remarks. We have introduced techniques to improve the reliability and performance of incomplete Cholesky factorization based preconditioners for solving symmetric positive definite or mildly indefinite systems. The goal of this work was to produce an industrial strength iterative solver for SPD systems that can reliably replace a direct solver in many real applications, particularly those in which iterative solvers are not traditionally used due to robustness or performance concerns. The results presented in the paper confirm that our solver can approach the reliability of a well implemented direct solver, while consuming much less memory on an average. While the direct solver turned out to be faster, in general, than all the iterative solvers that we tried, our iterative solver was the second best performing solver on our randomly selected suite of test problems. In fact, the speed of the direct solver was a motivating factor behind our block incomplete factorization method, and suggests a need for incorporating some of the direct solver techniques into iterative solvers [22, 37, 53].

The adaptive nature of the choice of some of the parameters and algorithmic components based on the properties of the matrix contributes to the performance and robustness of our solver. The performance advantage of our incomplete Cholesky factorization based preconditioner stems from its blocking scheme that works well even on coefficient matrices that do not have a natural block structure and automatic selection of the more appropriate of the two blocking algorithms. The switch to LDL^T factorization upon encountering very small or negative pivots accompanied by a switch in the solver from CG to GMRES contributes to its robustness even in the face of failure of the Cholesky process. Continuous monitoring of the performance of the preconditioner generation and the iterative solution to tune dropping thresholds makes it even more suitable for real industrial applications that often involve solving a sequence of linear systems with different coefficient matrices. The block incomplete factorization scheme and threshold tuning introduced in this paper in the context

of SPD systems can be easily adapted for incomplete LU factorization for general systems.

Acknowledgements. We would like to thank Vanessa Lopez for interfacing her finite volume code with WSMP's symmetric iterative solver for the experiment reported in Figure 4.1.

REFERENCES

[1] M. A. Ajiz and A. Jennings. A robust incomplete Choleski-conjugate gradient algorithm. *International Journal on Numerical Methods in Engineering*, 20:949–966, 1984.

[2] Cleve Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM Journal on Scientific Computing*, 16:1404–1411, 1995.

[3] Cleve Ashcraft and Roger G. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software*, 15(4):291–309, 1989.

[4] Owe Axelsson. A general incomplete block matrix factorization method. *Linear Algebra and its Applications*, 74:179–190, 1986.

[5] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 2.3.3, Argonne National Laboratory, 2007.

[6] R. Beauwens and M. Ben Bouzid. On sparse block factorization iterative methods. *SIAM Journal on Numerical Analysis*, 24(5):1066–1076, 1987.

[7] Michele Benzi. Preconditioning Techniques for Large Linear Systems: A Survey. *Journal of Computational Physics*, 182(2):418–477, 2002.

[8] Michele Benzi, R. Kouhia, and Miroslav Tuma. Stabilized and block approximate inverse preconditioners for problems in solid and structural mechanics. *Computer Methods in Applied Mechanics and Engineering*, 190:6533–6554, 2001.

[9] Michele Benzi and Miroslav Tuma. A robust incomplete factorization preconditioner for positive definite matrices. *Numerical Linear Algebra with Applications*, 10:385–400, 2003.

[10] Edmond Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM Journal on Scientific Computing*, 21(5):1804–1822, 2000.

[11] Edmond Chow. Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. *International Journal of High Performance Computing Applications*, 15(1):56–74, 2001.

[12] Edmond Chow and Michael A. Heroux. An object-oriented framework for block preconditioning. *ACM Transactions on Mathematical Software*, 24:159–183, 1998.

[13] Edmond Chow and Yousef Saad. Approximate inverse techniques for block partitioned matrices. *SIAM Journal on Scientific Computing*, 18(6):1657–1675, 1997.

[14] Edmond Chow and Yousef Saad. Experimental study of ILU preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics*, 86(2):387–414, 1997.

[15] Edmond Chow and Yousef Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM Journal on Scientific Computing*, 19(3):995–1023, 1998.

[16] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference of the ACM*, pages 152–172, 1969.

[17] E. F. D'Avezedo, Peter A. Forsyth, and Wei-Pai Tang. Drop tolerance preconditioning for incompressible viscous flow. *International Journal of Computer Mathematics*, 44(1):301–312, 1992.

[18] Timothy A. Davis. The University of Florida Sparse Matrix Collection. Technical report, Department of Computer Science, University of Florida, Jan 2007.

[19] E. D. Dolan and J. J. Moré. Benchmarking Optimization Software with Performance Profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[20] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[21] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[22] Iain S. Duff. Sparse numerical linear algebra: Direct methods and preconditioning. Technical Report RAL-TR-96-047, Rutherford Appleton Laboratory, 1996.

[23] Iain S. Duff, A. M. Erisman, and John K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, UK, 1990.

[24] Iain S. Duff and G. A. Meurant. The effect of ordering on preconditioned conjugate gradient. *BIT*, 29:635–657, 1989.

[25] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[26] Robert D. Falgout and Ulrike Meier Yang. *hypre*, High Performance Preconditioners: Users manual. Technical report, Lawrence Livermore National Laboratory, 2006. Paper appears in International Conference on Computational Science (3), 2002.

[27] Qing Fan, Peter A. Forsyth, J. R. F. McMacken, and Wei-Pai Tang. Performance issues for iterative solvers in device simulation. *SIAM Journal on Scientific Computing*, 17(1):100–117, 1996.

[28] Roland W. Freund and Noel M. Nachtigal. QMRPACK: A package of QMR algorithms. *ACM Transactions on Mathematical Software*, 22(1):46–77, 1996.

[29] Michael W. Gee, Chris M. Siefert, Jonathan J. Hu, Raymond S. Tuminaro, and Marzio G. Sala. ML 5.0 Smoothed Aggregation User's Guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

[30] Alan George and Joseph W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, NJ, 1981.

[31] Thomas George, Anshul Gupta, and Vivek Sarin. An experimental evaluation of iterative solvers for large spd systems of linear equations. In *10th Copper Mountain Conference on Iterative Methods*, April, 2008. Also available as RC 24479, IBM T. J. Watson Research Center, Yorktown Heights, NY (*http://www.cs.umn.edu/~agupta/doc/copper08.pdf*).

[32] John R. Gilbert and Sivan Toledo. An assessment of incomplete-LU preconditioners for non-symmetric linear systems. *Informatica*, 24:409–425, 2000.

[33] Gene H. Golub and Charles Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 1996.

[34] Anshul Gupta. Fast and effective algorithms for graph partitioning and sparse matrix ordering. *IBM Journal of Research and Development*, 41(1/2):171–183, January/March, 1997.

[35] Anshul Gupta. WSMP: Watson sparse matrix package (Part-I: direct solution of symmetric sparse systems). Technical Report RC 21886 (98462), IBM T. J. Watson Research Center, Yorktown Heights, NY, November 16, 2000. *http://www.cs.umn.edu/~agupta/wsmp*.

[36] Anshul Gupta. WSMP: Watson sparse matrix package (Part-III: iterative solution of sparse systems). Technical Report RC 24398, IBM T. J. Watson Research Center, Yorktown Heights, NY, November 5, 2007. *http://www.cs.umn.edu/~agupta/wsmp*.

[37] Pascal Hénon, Francois Pellegrini, Pierre Ramet, and Jean Roman. Blocking issues for an efficient parallel block ilu preconditioner. In *International SIAM Conference On Preconditioning Techniques For Large Sparse Matrix Problems In Scientific And Industrial Applications, Atlanta, USA*, 2005.

[38] Van Emden Henson and Ulrike Meier Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics: Transactions of IMACS*, 41(1):155–177, 2002.

[39] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Raymond S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Transactions on Mathematical Software*, 31(3):397–423, 2005.

[40] David Hysom and Alex Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM Journal on Scientific Computing*, 22(6):2194–2215, 2000.

[41] Mark T. Jones and Paul E. Plassmann. Blocksolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL-95/48, Argonne National Laboratory, Argonne, IL, 1995.

[42] Igor E. Kaporin, L. Yu. Kolotilina, and A. Yu. Yeremin. Block SSOR preconditionings for high order 3D FE systems. II Incomplete BSSOR preconditionings. *Linear Algebra and its Applications*, 154-156:647–674, 1991.

[43] D. S. Kershaw. The incomplete cholesky-conjugate gradient method for interative solution of linear equations. *Journal of Computational Physics*, 26:43–65, 1978.

[44] Joseph W.-H. Liu. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Transactions on Mathematical Software*, 12(2):127–148, 1986.

[45] Joseph W.-H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[46] Joseph W.-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice.

SIAM Review, 34:82–109, 1992.

[47] Joseph W.-H. Liu, Esmond G.-Y. Ng, and Barry W. Peyton. On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applictations*, 14:242–252, 1993.

[48] T. A. Manteuffel. An Incomplete Factorization Technique for Positive Definite Linear Sys tems. *Mathematics of Computation*, 34(150):473–497, 1980.

[49] Matthias Bollhöfer. A robust and efficient ILU that incorporates the growth of the inverse triangular factors. *SIAM Journal on Scientific Computing*, 25(1):86–103, 2003.

[50] Matthias Bollhöfer and Yousef Saad. Multilevel preconditioners constructed from inverse-based ILUs. *SIAM Journal on Scientific Computing*, 27(5):1627–1650, 2006.

[51] Matthias Bollhöfer, Yousef Saad and Olaf Schenk. ILUPACK - preconditioning software package. Available online at *http://www.math.tu-berlin.de/ilupack*, January 2006.

[52] N. Munksgaard. Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients. *ACM Transactions on Mathematical Software*, 6(2):206–219, 1980.

[53] Esmond G.-Y. Ng, Barry W. Peyton, and Padma Raghavan. A blocked incomplete cholesky preconditioner for hierarchical-memory computers. In D. R. Kincaid and A. C. Elster, editors, *Iterative Methods in Scientific Computation IV, IMACS Series in Computational and Applied Mathematics*, pages 211–221. 1999.

[54] J. W. Ruge and K. Stüben. Multigrid methods. In Stephen F. McCormick, editor, *Frontiers in Applied Mathematics*, volume 3, pages 73–130. SIAM, Philadelphia, PA, 1987.

[55] Yousef Saad. ILUT: A dual threshold incomplete *LU* factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

[56] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, 1996.

[57] Yousef Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[58] Yousef Saad and Jun Zhang. BILUM: Block versions of multielimination and multilevel ILU preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, 20(6):2103–2121, 1999.

[59] M. Tismenetsky. A new preconditioning technique for solving large sparse linear systems. *Linear Algebra and its Applications*, 154/156:331–353, 1991.

[60] Raymond S. Tuminaro. Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, 2000.

[61] J. W. Watts-III. A conjugate gradient truncated direct method for the iterative solution of the reservoir pressure equation. *Society of Petroleum Engineers Journal*, 21:345–353, 1981.

[62] David Young. Iterative methods for solving partial difference equations of elliptic type. *Transactions of the American Mathematical Society*, 76(1):92–111, 1954.