Factorization-based sparse solvers and preconditioners

X S Li[1], M Shao[2], I Yamazaki[1] and E G Ng[1]

[1]One Cyclotron Road, Lawrence Berkeley National Laboratory, MS 50F-1650, Berkeley, CA 94720.
[2]School of Mathematical Sciences, Fudan University, Shanghai 200433, China.

E-mail: xsli@lbl.gov

Abstract. Efficient solution of large-scale, ill-conditioned and highly-indefinite algebraic equations often relies on high quality preconditioners together with iterative solvers. Because of their robustness, the factorization-based algorithms could play a significant role when they are combined with iterative methods, particularly in the development of scalable solvers. We present our recent work in using the direct solver SuperLU code base to develop a new supernode-based ILU preconditioner and a domain-decomposition hybrid solver. Our ILU preconditioner is a modification of the classic ILUTP approach, incorporating a number of techniques to improve robustness and performance, which include new dropping strategies that accommodate the use of supernodal structure in the factored matrix. Our hybrid solver is based on the Schur complement method. We use parallel graph partitioning to obtain hierarchical interface/domain decomposition, and multiple parallel direct solvers to solve the subdomain problems simultaneously, and parallel preconditioned iterative solvers to solve the interface problem. We will demonstrate the effectiveness of our new techniques by applying them to two SciDAC applications, modeling next-generation particle accelerators and fusion devices.

1. A supernodal approach to incomplete LU factorization

1.1. Introduction

A critical component of the iterative solution techniques is the construction of effective preconditioners. Physics-based preconditioners are quite effective for structured problems, such as those arising from discretized partial differential equations. On the other hand, a class of methods based on incomplete LU decomposition are still regarded as the generally applicable "black-box" preconditioners for unstructured systems arising from a wide range of applications areas. A variety of ILU techniques have been studied extensively in the past, including distinct strategies of dropping elements, such as the level-of-fill structure-based approach (i.e., ILU(k)) [1], numerical threshold-based approach [2], and more recently, numerical inverse-based multilevel approach [3].

The value-based threshold method is often more reliable than the level-based method, but it is harder to implement efficiently. One of the most sophisticated value-based methods is ILUTP proposed by Saad [2, 1], which combines a dual dropping strategy with numerical pivoting ("T" stands for threshold, and "P" stands for pivoting). The dual dropping rule in ILU(τ, p) first removes the elements that are smaller than τ from the current factored row or column. It then keeps only the largest p elements to control the memory requirement.

Our method can be considered to be a variant of the ILUTP approach, and we modified our high-performance direct solver SuperLU [4] to perform incomplete factorization. One key component in SuperLU is supernode, which gives significant performance advantages over a non-supernodal (e.g., column-wise) algorithm on modern cache-based architectures, and on GPU-type accelerators. Although the average size of the supernodes in an incomplete factor is expected to be smaller than in a complete factor because of dropping, we attempt to retain supernodes as much as possible. We have adapted the

dropping rules to incorporate the supernodal structures as they emerge during factorization. Therefore, our new algorithm has the combined benefits of retaining numerical robustness of ILUTP as well as achieving fast construction and application of the ILU preconditioner.

Our main contributions can be summarized as follows. We adapted the classic dropping strategies of ILUTP in order to incorporate supernode structures and to accommodate dynamic supernodes due to partial pivoting. For the secondary dropping strategy, we proposed an area-based fill control method, which is more flexible and numerically robust than the traditional column-based scheme. Furthermore, we incorporated several heuristics for adaptively modifying various threshold parameters as the factorization proceeds, which improves the robustness of the algorithm.

1.2. Sketch of the supernodal ILU algorithm

Our base algorithm framework is the left-looking, partial pivoting, supernodal sparse LU factorization algorithm implemented in SuperLU [4]. A key concept in SuperLU is to exploit dense blocks appearing in the L and U factors. In particular, we define a supernode in L to be a range $(r : t)$ of columns with the triangular block on the diagonal being full, and the identical nonzero structure elsewhere among the columns. Using the same supernode partition to the rows of U, the nonzero structure of each column in U consists of a number of dense segments. Thus, the compressed data structure for L consists of a collection of supernodes as dense submatrices, and that for U consists of a collection of dense subvectors.

The factorization algorithm is left-looking, with a supernode-panel update kernel. A panel is a set of consecutive columns, and the size of panel is an algorithmic blocking parameter used to enhance data reuse in the memory hierarchy; it enables use of Level 3 BLAS. At each step of panel factorization, we obtain a panel in the U factor and a panel in the L factor.

Our incomplete factorization algorithm retains most of the algorithmic ingredients from SuperLU, with added dropping rules that are applied to the L and U factors on-the-fly. The description of the high level algorithm is given in Algorithm 1. The steps marked as **bold** correspond to the new steps introduced to perform ILU. Since partial pivoting with row interchange is used, the resulting factorization is performed on the matrix $P_r P_0 D_r A D_c P_c^T$, where D_r and D_c are diagonal scaling matrices, P_0 is the row permutation matrix returned from MC64 [5] that is meant to permute large magnitude entries on the main diagonal (this step is optional), P_c is the column permutation matrix for sparsity preservation, and P_r is the row permutation matrix from partial pivoting. The matrices D_r, D_c, P_0 and P_c are obtained before factorization, and P_r is obtained during factorization. In the following sections, we describe our adaptation of the dropping rules to the situation when supernodes are present.

1.3. Value-based dropping criteria

Our primary dropping criteria are threshold-based and akin to the ILUTP variants [2, 6]. That is, while performing Gaussian elimination with partial pivoting, we set to zero the entries in L and U with modulus smaller than a prescribed threshold τ, where $\tau \in [0, 1]$.

Since our compressed storage is column oriented for both L and U, the dropping rule is also column oriented. The upper triangular matrix U is stored in a normal compressed column format, we can easily remove the small elements while storing the newly computed column into the compressed storage, using the first criterion given in Figure 1.

The lower triangular matrix L is stored as a collection of supernodes. Our goal is to retain the supernodal structure to the largest extent as in the complete factorization. In a naive implementation of ILU, we may apply the traditional dropping to each individual column. But after dropping, the nonzero structures among the columns in the original supernode will be different, then we will need to regroup the columns into smaller supernodes, resulting in a performance penalty. Instead, we adopt an alternative approach that retains the original supernode partition as much as possible. That is, we either keep or drop an entire row in a supernode when it is formed at the current step. This is similar to what was proposed by Gupta and George in the context of incomplete Cholesky factorization [7]. Our dropping criterion is the second rule shown in Figure 1. Since we use partial pivoting, the magnitude of the elements in L is bounded above by one, and so the absolute quantity is the same as the relative quantity. The use of

∞-norm for row i of a supernode implies that when row i is dropped, the magnitude of every element in this row is smaller than τ. Therefore, in a traditional column-wise algorithm, these elements should be dropped as well. We did an experiment to compare the supernodal ILU and the column-wise ILU (setting maximum supernode size to be one). For 54 matrices, GMRES with supernodal ILU converged for 47 cases, and the column-wise ILU succeeded with only 42 matrices. This shows that our supernodal version is numerically superior.

Algorithm 1. *Left-looking, supernode-panel ILU algorithm*

(i) Preprocessing

1.1) **(optional)** Use MC64 to find a row permutation P_0 and row and column scaling factors D_r and D_c such that $P_0 D_r A D_c$ is an I-matrix;
1.2) If step 1.1) is not performed, do a simple LAPACK-style row/column equilibration to obtain $D_r A D_c$;
1.3) Compute a column permutation P_c to preserve sparsity of the LU factorization of $P_0 D_r A D_c P_c^T$;

(ii) Factorization of $P_0 D_r A D_c P_c^T$

FOR each panel of columns DO
2.1) *Symbolic factorization:* determine which supernodes to the left will update the current panel and a topological order of updates;
2.2) *Panel factorization:*
FOR each updating supernode DO
Apply triangular solve to obtain the U part;
Apply matrix-matrix multiplication to obtain the L part;
END FOR
2.3) *Inner factorization:*
FOR each column j in the panel DO
Update the current column j;
Apply the dropping rule to the U part;
Find pivot in this column;
(optional) Modify the diagonal entry to handle zero-pivot breakdown;
Determine supernode boundary;
IF column j starts a new supernode THEN
Apply the dropping rule to the newly formed supernode $L(:, r : j-1)$;
END IF
END FOR
END FOR

Value-based dropping criteria for ILU(τ)

1) Dropping elements in U: If $|u_{ij}| < \tau \|A(:, j)\|_\infty$, we set u_{ij} to zero.

2) Dropping elements in L: In a supernode $L(:, r : t)$, if $\|L(i, r : t)\|_\infty < \tau$, we set the entire i-th row to zero.

Figure 1. The value-based dropping criteria.

1.4. Secondary dropping to control fill-in adaptively

ILU(τ) works well if there is sufficient memory, but it may still have too much fill. A secondary dropping can be used to alleviate the problem. In Saad's ILU(τ,p) approach [2], p is the largest number of nonzeros (not the level-of-fill) allowed in each row of F (in a row-wise algorithm). Gupta and George suggested using $p(j) = \gamma \cdot \text{nnz}(A(:, j))$ for the j-th column instead of a constant, where γ is an upper bound of the fill

ratio defined by a user [7]. They also proposed a method of computing a secondary dropping tolerance by an interpolation formula rather than sorting the largest p entries. But Gupta's heuristic depends largely on the distribution of the nonzero modulos in F.

We now present a new strategy for choosing p. Given a user-desired upper bound of the overall fill ratio γ, we define an upper bound function $f(j)$ for each column j, $f : [1, n] \to [1, \gamma]$, which satisfies $f(n) \le \gamma$. Then at the j-th column, if the current fill ratio

$$\frac{\text{nnz}(F(:, 1 : j))}{\text{nnz}(A(:, 1 : j))} \tag{1}$$

exceeds $f(j)$, we choose a maximum possible value p such that when we keep the largest p elements, the current fill ratio is bounded by $f(j)$. This criterion can be adapted to our supernodal algorithm as follows. For a supernode with k columns, p may be computed as

$$p = \max\left\{\frac{f(j) \cdot \text{nnz}(A(:, 1 : j)) - \text{nnz}(F(:, 1 : j - k))}{k},\ \ k\right\}. \tag{2}$$

In other words, if we keep the largest p rows of this supernode, the current fill ratio is guaranteed not to exceed $f(j)$. The second k term in $\max\{\ldots\}$ is to ensure that we do not drop any row in the diagonal block of the supernode.

This is also an ILU(τ,p) approach with adaptive p, similar to Gupta's scheme. However, our fill ratio definition (1) is *area-based* instead of column-based, because we count all the fill-ins from column 1 to column j. That is, we only monitor the overall memory growth instead of that of each individual column. This is more flexible than the column-based method in that it allows larger amount of fill for certain columns so long as the cumulative fill ratio in the previous columns is small. At the end of factorization, the total fill ratio is still bounded by γ because of the condition $f(n) \le \gamma$.

Since L and U are stored in different data structures, we may choose two functions, $f_L(j)$ for L and $f_U(j)$ for U, so long as $f_L(n) + f_U(n) \le \gamma$. A simple way is to assign $f_L(n)$ and $f_U(n)$ to be the areas of $L(:, j)$ and $U(:, j)$ relative to $F(:, 1 : j)$, as follows:

$$f_U(j) = \frac{j}{2n}\gamma, \quad f_L(j) = \left(1 - \frac{j}{2n}\right)\gamma. \tag{3}$$

Then we split the fill quota proportionally with $f_U(j) : f_L(j)$ ratio.

In conjunction with the dynamic, area-based strategy for choosing p, we devised an adaptive scheme for choosing τ as well. Specifically, let $\tau(1) = \tau_0$ be the user-input threshold, at column j, if the fill ratio given by Equation (1) is larger than $f(j)$, we set $\tau(j + 1) = \min\{1, 2\,\tau(j)\}$, otherwise, we set $\tau(j + 1) = \max\{\tau_0, \tau(j)/2\}$. That is, we maintain $\tau(j) \in [\tau_0, 1]$.

1.5. Numerical experiments

We tested our algorithms on an Opteron cluster running a Linux operating system at NERSC.[5] Each node contains dual Opteron 2.2 GHz processors, with 5 GBytes usable memory. We use only one processor of a node. The processor's theoretical peak floating-point performance is 4.4 Gflops/sec. We use PathScale `cc` compiler with the following optimization flags: `-O3 -OPT:IEEE_arithmetic=1 -OPT:IEEE_NaN_inf=ON`, which conforms to the IEEE-754 standard. We have chosen 54 test matrices: 5 are from the M3D-C^1 code for extended MHD modeling in fusion energy study [8], which is being developed in the CEMM SciDAC project; 49 are from Matrix Market [9] and the University of Florida Sparse Matrix collection [10]. These are all unsymmetric matrices of medium to large size. The iterative solver is restarted GMRES with our ILU as a right preconditioner (i.e. solving $PAM^{-1}y = Pb$). The stopping criterion is $\|r_k = b - Ax_k\|_2 \le \delta\,\|b\|_2$, here we use $\delta = 10^{-8}$ which is in the order of the square

[5] `http://www.nersc.gov/nusers/systems/jacquard`

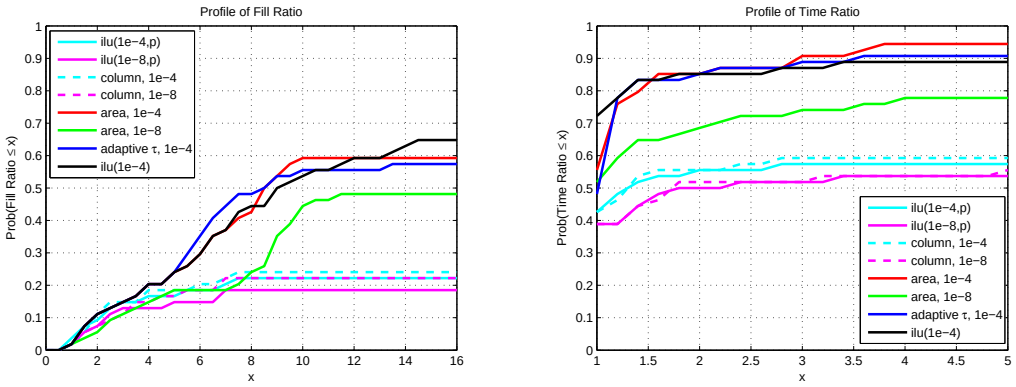
root of IEEE double precision machine epsilon. We set the dimension of the Krylov subspace to be 50 and maximum iteration count to be 1000.

We now present the results of the tests comparing various parameter settings. The ILU configurations include:

- ILU(τ), $\tau = 10^{-4}$;
- ILU(τ, p), $\tau = 10^{-4}$ or 10^{-8}, $p = \gamma \cdot \text{nnz}(A)/n$;
- column-based adaptive p, $\tau = 10^{-4}$ or 10^{-8};
- area-based adaptive p, $\tau = 10^{-4}$ or 10^{-8};
- area-based adaptive $\tau(j)$, $\tau_0 = 10^{-4}$, no secondary dropping

Figure 2 shows the performance profiles of the fill ratio and the time ratio for the 54 test matrices. The former shows the fraction of the problems that a solver could solve within the fill ratio x, and the latter shows the fraction of the problems that a solver could solve within a multiple of x of the best time among all the solvers. We can see that a small τ such as 10^{-8} is generally not good, that is, it is not efficient to use the secondary dropping rule only. The value-based dropping criterion in Figure 1 should play a significant role.

A key conclusion is that our new area-based scheme is much more robust than the column-based scheme; it is also better than ILU(τ) when the fill ratio does not exceed the user-desired γ. ILU(τ) becomes better only when the fill ratio is unbounded (i.e., allow it to exceed γ). This is consistent with the intuition that an ILU preconditioner tends to be more robust with more fill-ins.



(a) Profile of fill ratio (b) Profile of time ratio

Figure 2. Performance profiles after incorporating the secondary dropping rules; $\gamma = 10$.

Figure 2(b) shows the runtime comparison of the solvers. In this plot, the matrices with fill ratio larger than 10 are considered as failure. Thus, the comparison is made under the same memory constraint, and none of the solvers are allowed to consume much more memory than the others. The top three solvers are much better than the others. Our area-based adaptive-p or adaptive-τ schemes have similar performance, with the adaptive-p scheme having a slight edge over the other one.

Taking into account both memory and time, we can see that the secondary dropping helps achieve a good trade-off, with controlled fill-in and the solver not being much slower. Either our "red" scheme or "blue" scheme can be used as a default setting in the code.

Table 1 compares the new ILU-preconditioned GMRES solver to SuperLU for a sequence of matrices from the fusion M3D-C^1 code. The ILU parameters are: $\tau = 10^{-4}$, and $\gamma = 10$. The advantage of GMRES+ILU over SuperLU is remarkable, both in time and in memory: SuperLU fails for the two larest problems; for **matrix61**, the largest that both solvers succeed, ILU only incurs one-tenth the amount of fill of SuperLU, and GMRES+ILU is about 10x faster than SuperLU. For all problems, GMRES can succeed with very low fill ratio in ILU—at or below 3.0.

Problems	Order	Nonzeros (millions)	ILU		GMRES		SuperLU	
			time	fill ratio	time	iters	time	fill ratio
matrix31	17,298	2.7 m	8.2	2.7	0.6	9	33.3	13.1
matrix41	30,258	4.7 m	18.6	2.9	1.4	11	111.1	17.5
matrix61	66,978	10.6 m	54.3	3.0	7.3	20	612.5	26.3
matrix121	263,538	42.5 m	145.2	1.7	47.8	45	fail	–
matrix181	589,698	95.2 m	415.0	1.7	716.0	289	fail	–

Table 1. Results of the test matrices from the fusion simulation code M3D-C^1.

2. Hybrid linear solver

2.1. Introduction

Although significant progress has been made in the development of high performance direct solvers [11, 12, 13], the size of the systems that can be directly factored is limited due to the large memory requirement. Preconditioned iterative solvers [14, 1, 15] can reduce the memory requirement, but they often suffer from slow convergence due to the ill-conditioning and highly-indefinite nature of the systems. To address these challenges, a number of parallel hybrid solvers have been developed based on a domain decomposition idea called the Schur complement method [16, 17]. In this method, the unknowns in interior domains are first eliminated directly using techniques from the direct solvers, and the remaining Schur complement system is solved approximately using a preconditioned iterative solver. This method has the potential of balancing the robustness of the direct solver with the efficiency of the iterative solver since the unknowns in each interior domain can be eliminated efficiently, and in parallel, while the sparsity can be enforced for solving the Schur complement system, where most of fill occurs. Furthermore, for a symmetric positive definite system, it can be shown that the Schur complement has a smaller condition number than the original coefficient matrix [18]. Consequently, the preconditioned iterative solver often requires fewer iterations to solve the Schur complement system. Unfortunately, for a general linear system, this method can still suffer from slow convergence as the size of the Schur complement increases, especially with the existing parallel hybrid solvers which are designed primarily to achieve good scalability of time to compute the preconditioners.

To overcome these drawbacks, as part of our SciDAC project (TOPS) [19], we have been developing a new parallel implementation of the Schur complement method which provides the robustness and flexibility to solve large highly-indefinite linear systems on a large number of processors. In this section, we outline our implementation and present our preliminary results for solving linear systems of this type, which arise from two SciDAC applications: the next-generation particle accelerators modeling (ComPASS) [20] and the fusion devices simulation (CEMM) [21].

2.2. Schur complement method

The Schur complement method is a non-overlapping domain decomposition method, which is also referred to as iterative substructuring. Specifically, the original linear system is first reordered into a 2×2 block system of the following form,

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}, \tag{4}$$

where A_{11} is a block-diagonal matrix, each of whose diagonal blocks represents an *interior domain*, A_{22} represents *separators*, and A_{12} and A_{21} are the *interfaces* between A_{11} and A_{22}. After one step of the block Gaussian elimination, the 2×2 block system (4) becomes

$$\begin{pmatrix} A_{11} & A_{12} \\ 0 & S \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ \widehat{b}_2 \end{pmatrix}, \tag{5}$$

where S is the Schur complement defined as

$$S = A_{22} - A_{21}A_{11}^{-1}A_{12}, \tag{6}$$

and $\widehat{b_2} = b_2 - A_{21}A_{11}^{-1}b_2$. Hence, the solution of the linear system (4) can be computed by

(i) first solving the Schur complement system

$$S\,x_2 = \widehat{b_2}, \tag{7}$$

(ii) then solving the interior system

$$A_{11}x_1 = b_1 - A_{12}x_2. \tag{8}$$

Since most of the fill occurs in the Schur complement S, our hybrid solver follows the approach, in which the interior system is solved directly, while the Schur complement system is solved approximately.

2.3. Parallel implementation

Our high performance implementation takes full advantage of the state-of-the-art software. For example, we form the 2×2 block system (4) based on the hierarchical interface decomposition (HID), which is constructed by Hybrid Iterative Parallel Solver (HIPS) [22]. We then invoke either a serial direct solver SuperLU [12] or parallel direct solver SuperLU_DIST [13] to compute the LU factorization of each interior domain in parallel. To preserve sparsity of the LU factors, each interior domain is permuted with the nested dissection ordering obtained either by METIS [23] or by PT-SCOTCH [24], which is a serial or parallel algorithm, respectively. Next, the Schur complement system (7) is solved using a Krylov subspace method from PETSc [25] combined with an ILU preconditioner. Finally, the solution of the interior system (8) can be computed efficiently, and in parallel, since the LU factorizations of each interior domains have already been computed.

Since computing the preconditioner for solving the Schur complement system is often the computational and memory bottleneck, we give a brief description of how the preconditioner is computed. In our current implementation, the preconditioners are the exact LU factors of a sparsified Schur complement. Specifically, let us denote the ℓ-th interior domain and the corresponding interfaces by $A_{11}^{(\ell)}$, $A_{12}^{(\ell)}$, and $A_{21}^{(\ell)}$, respectively, such that the coefficient matrix in the 2×2 block system (4) can be written as

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{cccc|c} A_{11}^{(1)} & & & & A_{12}^{(1)} \\ & A_{11}^{(2)} & & & A_{12}^{(2)} \\ & & \ddots & & \vdots \\ & & & A_{11}^{(k)} & A_{12}^{(k)} \\ \hline A_{21}^{(1)} & A_{21}^{(2)} & \dots & A_{21}^{(k)} & A_{22} \end{array}\right). \tag{9}$$

If each interior domain $A_{11}^{(\ell)}$ is factored by a single processor, then the ℓ-th processor stores the nonzeros of $A_{11}^{(\ell)}$ and $A_{21}^{(\ell)}$ in a row-wise order, and the nonzeros of $A_{12}^{(\ell)}$ in a column-wise order. If multiple processors are used for each interior domain, then the rows of $A_{11}^{(\ell)}$ and $A_{12}^{(\ell)}$, and the columns of $A_{21}^{(\ell)}$ are evenly distributed among the processors. Furthermore, the rows of A_{22} are evenly distributed among the processors that solve the Schur complement system (7). To simplify the discussion, we assume a single processor is used to factor each interior domain. Hence, the ℓ-th processor computes the LU factorization of $A_{11}^{(\ell)}$, i.e., $A_{11}^{(\ell)} = L^{(\ell)}U^{(\ell)}$.

The following pseudocode shows how the ℓ-th processor computes the corresponding rows of the approximate Schur complement $\widetilde{S}$ when the processor owns the k_1-th to k_2-th rows of A_{22}.

1. $\widetilde{F}^{(\ell)} \approx (L^{(\ell)})^{-1}A_{12}^{(\ell)}$
2. $\widetilde{E}^{(\ell)} \approx (U^{(\ell)})^{-T}(A_{21}^{(\ell)})^T$

3. $T^{(\ell)} = (E^{(\ell)})^T F^{(\ell)}$
4. $S(k_1 : k_2, :) = A_{22}(k_1 : k_2, :)$
5. **for** $i = 1 \ldots k$
6. $\quad S(k_1 : k_2, :) = S(k_1 : k_2, :) - T^{(i)}(k_1 : k_2, :)$
7. **end for**
8. $\widetilde{S}(k_1 : k_2, :) \approx S(k_1 : k_2, :)$

On lines 1 and 2 of the above pseudocode, the lower-triangular systems are solved with regard to the sparsity of the interfaces $A_{12}^{(\ell)}$ and $A_{21}^{(\ell)}$. Then, the nonzeros with magnitude less than a user-specified threshold σ_1 are discarded to enforce sparsity of $\widetilde{E}^{(\ell)}$ and $\widetilde{F}^{(\ell)}$. On line 3, each processor computes the contribution $T^{(\ell)}$, and on lines 4 to 7, the contributions from other processors are gathered to compute the k_1-th to k_2-th rows of the Schur complement S. Finally, on line 8, after S is scaled and permuted to improve the numerics, nonzeros with the magnitude less than a user-specified threshold σ_2 are discarded to compute the approximate Schur complement $\widetilde{S}$.

After $\widetilde{S}$ is computed, SuperLU_DIST is invoked to compute the LU factor of $\widetilde{S}$, which is used as the preconditioner. We note that the matrix-vector product with the Schur complement S within the Krylov method is computed by applying the sequence of the sparse matrix operations (6) on the vector, and hence, the exact Schur complement S does not have to be stored explicitly for this phase of the solver.

2.4. Numerical experiments

We present preliminary results of our hybrid solver to solve large highly-indefinite linear systems of equations. For our numerical experiments, we used two test matrices from different SciDAC applications, the **tdr190k** matrix of dimension 1,100,242 from the next-generation particle accelerators modeling (ComPASS) [20], and the **matrix211** matrix of dimension $801,378$ from the fusion M3D-C^1 code (CEMM) [21]. All the experiments were conducted on the NERSC Cray XT4 machine.

To demonstrate the effectiveness of our hybrid solver, we compare its performance with that of a direct solver SuperLU_DIST [13], and that of another hybrid solver HIPS [16]. The primary difference between our hybrid solver and HIPS is the way the preconditioner is computed for solving the Schur complement system (7). HIPS computes the preconditioner based on the ILU factorization of S, but sparsity of the preconditioner is enforced based on both numerical values and locations of nonzeros. Specifically, the fill is allowed only between separators adjacent to the same domain. Because of this, even though the numerical drop tolerance in our numerical experiments was set to be zero, HIPS still enforces the sparsity of the preconditioner based on the locations of nonzeros. On the other hand, our hybrid solver currently computes the exact LU factorization of $\widetilde{S}$.

In Table 2, we compare the total numbers of nonzeros in the LU and ILU factors. The sparsity of the matrices $\widetilde{E}$ and $\widetilde{F}$ is enforced using the drop tolerances $\sigma_1 = 10^{-6}$ and 10^{-7} for the matrices **tdr190k** and **matrix211**, respectively. An additional drop tolerance $\sigma_2 = 10^{-5}$ was used in our hybrid solver to enforce the sparsity of $\widetilde{S}$. In these numerical experiments, a single processor is used to factor each interior domain. We first note that the total number of nonzeros with SuperLU_DIST increases as the number of processors increases. This is because the quality of the nested dissection ordering degrades with the increase in the number of processors.[6] For the **tdr190k** and **matrix211** matrices, HIPS failed to converge within $1,000$ unrestarted GMRES iterations using 32 and 128 processors, respectively. On the other hand, our hybrid solver (denoted by Hybrid in the table) is flexible enough to achieve convergence within 30 iterations for all the numerical experiments presented here.

Figure 3 shows the total solution times as a function of the number of processors in a strong scaling study. We see that SuperLU_DIST does not scale beyond 128 processors for these test matrices. On the other hand, our hybrid solver could still reduce the solution time using 512 processors. Furthermore, even when HIPS converged, our total solution time was less than theirs since our hybrid solver takes full advantage of the state-of-the art software. These numerical results demonstrate the potential of our

[6] PT-SCOTCH and ParMETIS are used for the **tdr190k** and **matrix211** matrices, respectively.

	tdr190k					**matrix211**			
# procs	2	8	32	128	512	8	32	128	512
Hybrid	750	673	545	621	735	491	461	504	593
HIPS	987	718	719	−−	−−	875	680	440	−−
SuperLU_DIST	768	787	937	1192	1466	1590	1751	1957	2010

Table 2. Total number of nonzeros in the LU and ILU factors.

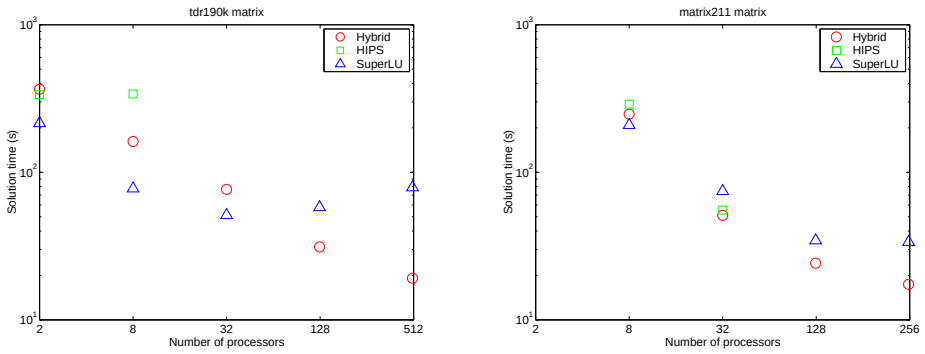


Figure 3. Scalability of time to solve the linear systems on a distributed memory computer.

hybrid solver to be more flexible and robust for solving large highly-indefinite systems on a large number of processors.

We note that HIPS only exploits single-level parallelism, that is, each subdomain solution is performed by only one processor. Then the following dilemma would be faced: when the problem size is large and we need to use large processor count, we will need to generate large number of subdomains, leading to an increase of the Schur complement size as well as the iteration count (sometimes divergence). Our remedy for this problem is to exploit two-level parallelism. That is, we keep the number of interior subdomains fixed and relatively small, and so is the size of the Schur complement, while allowing multiple processors to solve each subdomain problem. The number of processor groups is the same as the number of subdomains. We also use multiple processors to solve the Schur complement system. This ensures that the convergence rate is independent of the number of processors used.

Figure 4 shows the performance of our hybrid solver using multiple processors to factor each interior subdomain. Here, we use larger drop tolerance, $\sigma_1 = 10^{-5}$ and $\sigma_2 = 10^{-4}$, in order to reduce the memory requirement. In the "two-level" case, we fixed the number of subdomains to be 8. The "single-level" case also refers to our code, but keeps the number of subdomains the same as the number of processors. Our two-level parallel scheme also uses less memory than the single-level scheme; it needs less than one-third of the memory needed by SuperLU_DIST. These results show the great potential of this "two-level parallelization" to reduce the memory requirement while achieving good parallel scalability.

3. Conclusions

We are developing a high performance incomplete LU factorization preconditioner and a hybrid direct and iterative solver based on the Schur complement method. The experiments showed that our supernode-based ILUTP is superior to the classic ILUTP. For some fusion problems, it can reduce memory by nine fold, while solving the problem ten times faster. Our hybrid solver is more flexible in exploiting multiple levels of parallelism, and numerically more reliable than the state-of-the-art hybrid solver HIPS. For the highly indefinite accelerator and fusion SciDAC problems, our hybrid solver scales beyond 512 processors, much faster than the parallel direct solver. The memory reduction can be two to four fold.

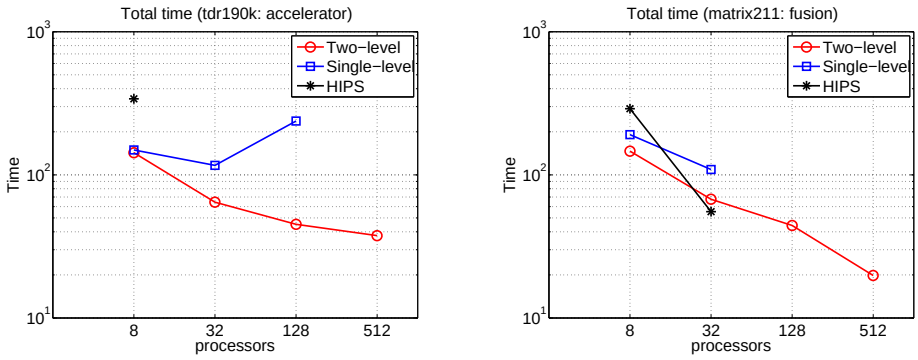


Figure 4. Comparison of single-level and two-level parallel hybrid solver.

Acknowledgments

This research was supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We used the resources at the National Energy Research Scientific Computing Center.

References

[1] Saad Y 2004 *Iterative methods for sparse linear systems* (Philadelphia: SIAM)
[2] Saad Y 1994 *Numerical Linear Algebra with Applications* **1** 387–402
[3] Bollhöfer M and Saad Y 2006 *SIAM J. Scientific Computing* **27** 1627–1650
[4] Demmel J W, Eisenstat S C, Gilbert J R, Li X S and Liu J W H 1999 *SIAM J. Matrix Analysis and Applications* **20** 720–755
[5] Duff I S and Koster J 1999 *SIAM J. Matrix Analysis and Applications* **20** 889–901
[6] Saad Y 1996 *Iterative methods for sparse linear systems* (Boston, MA: PWS Publishing Company)
[7] Gupta A and George T 2008 Adaptive techniques for improving the performance of incomplete factorization preconditioning Tech. Rep. RC 24598(W0807-036) IBM Research Yorktown Heights, NY
[8] Jardin S C, Breslau J and Ferraro N 2007 *Journal of Computational Physics* **226** 2146–2174
[9] Matrix Market `http://math.nist.gov/MatrixMarket/`
[10] Davis T A University of Florida Sparse Matrix Collection `http://www.cise.ufl.edu/research/sparse/matrices`
[11] Amestoy P, Duff I, Koster J and L'Excellent J Y 2001 *SIAM Journal on Matrix Analysis and Applications* **23** 15–41
[12] Demmel J, Eisenstat S, Gilbert J, Li X and Liu J 1999 *SIAM J. Matrix Analysis and Applications* **20** 720–755
[13] Li X and Demmel J 2003 *ACM Trans. Mathematical Software* **29** 110–140
[14] Axelsson O 1994 *Iterative solution methods* (New York: Cambridge University Press)
[15] van der Vorst H 2003 *Iterative Krylov methods for large linear systems* (New York: Cambridge University Press)
[16] Gaidamour J and Henon P 2008 HIPS: a parallel hybrid direct/iterative solver based on a schur complement *Proc. PMAA*
[17] Giraud L, Haidar A and Watson L T 2008 *Parallel Computing* **34** 363–379
[18] Smith B, Bjorstad P and Gropp W 1996 *Domain Decomposition* (New York: Cambridge University Press)
[19] Towards Optimal Petascale Simulation (TOPS) http://www.scalablesolvers.org/
[20] Community Petascale Project for Accelerator Science and Simulation (ComPASS) https://compass.fnal.gov
[21] Center for Extended MHD Modeling (CEMM) URL: http://w3.pppl.gov/cemm/
[22] Henon P and Saad Y 2006 *SIAM J. Sci. Comput* **28** 2266–2293
[23] Karypis Lab, Digital Technology Center, Department of Computer Science and Engineering, University of Minesota METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering http://glaros.dtc.umn.edu/gkhome/metis/metis
[24] Laboratoire Bordelais de Recherche en Informatique (LaBRI) SCOTCH - Software package and libraries for graph, mesh and hypergraph partitioning, static mapping, and parallel and sequential sparse matrix block ordering http://www.labri.fr/perso/pelegrin/scotch/
[25] Mathematics and Computer Science Division, Argonne National Laboratory The portable, extensible, toolkit for scientific computation (PETSc) www.mcs.anl.gov/petsc