Numer. Math. Theor. Meth. Appl.
doi: 10.4208/nmtma.2010.33.5

Vol. **3**, No. 3, pp. 352-366
August 2010

Preconditioning Schur Complement Systems of Highly-Indefinite Linear Systems for a Parallel Hybrid Solver[†]

I. Yamazaki[*], X. S. Li and E. G. Ng

Lawrence Berkeley National Laboratory, Berkeley, California, USA.

Received 30 September 2009; Accepted (in revised version) 27 January 2010

Available online 2 July 2010

Abstract. A parallel hybrid linear solver based on the Schur complement method has the potential to balance the robustness of direct solvers with the efficiency of preconditioned iterative solvers. However, when solving large-scale highly-indefinite linear systems, this hybrid solver often suffers from either slow convergence or large memory requirements to solve the Schur complement systems. To overcome this challenge, we in this paper discuss techniques to preprocess the Schur complement systems in parallel. Numerical results of solving large-scale highly-indefinite linear systems from various applications demonstrate that these techniques improve the reliability and performance of the hybrid solver and enable efficient solutions of these linear systems on hundreds of processors, which was previously infeasible using existing state-of-the-art solvers.

AMS subject classifications: 65F10, 15A12, 65N55

Key words: Schur complement method, preconditioning, matrix preprocessing.

1. Introduction

A number of parallel linear solvers have been implemented based on a domain decomposition idea called the Schur complement method [7, 8]. In this method, the unknowns in interior subdomains are first eliminated using a direct solver. Then, the remaining Schur complement system is solved using a preconditioned iterative method. This method often exhibits great parallel performance because interior subdomains can be solved in parallel. Furthermore, this *hybrid* approach has the potential to balance the robustness of direct solvers with the efficiency of iterative solvers because the unknowns in the relatively-small interior subdomains can be eliminated efficiently using a direct solver, while the sparsity can be enforced for solving the Schur complement system, where most of the fill occurs. In

[†]This paper is presented at the International Conference on Preconditioning Techniques for Scientific and Industrial Applications, August 24-26, 2009, Hong Kong.
[*]Corresponding author. *Email addresses:* ic.yamazaki@gmail.com (I. Yamazaki), XSLi@lbl.gov (X. S. Li), EGNg@lbl.gov (E. G. Ng)

addition, for a symmetric positive definite system, the Schur complement has a smaller condition number than the original coefficient matrix [20, Section 4.2], and fewer iterations are often required for solving the Schur complement system. Unfortunately, for a highly-indefinite linear system, the preconditioned iterative method often suffers from either slow convergence or large memory requirement to solve the Schur complement system.

To address this challenge, we discuss in this paper techniques to preprocess the Schur complement systems in parallel. Effective preprocessing techniques have been already developed to solve highly-indefinite linear systems of equations. For example, a matrix permutation to preserve the sparsity of a preconditioner significantly reduces the computational and memory requirements [11, 13]. Furthermore, unsymmetric scaling and permutations that place large entries on the diagonal often improve the reliability and performance of the preconditioned iterative solver [3,6]. However, the effectiveness of these preprocessing techniques on the performance of a *parallel* hybrid solver has not been well studied. There are software packages which compute matrix permutations in order to preserve the sparsity of the preconditioners on a distributed memory system [4,12]. However, these packages are designed primarily for sparse matrices, and their performance suffers on the Schur complements, which are relatively dense. Furthermore, a robust parallel implementation to place large entries on the diagonal has not yet been developed [5,9,18]. The primary purpose of this paper is to fill this gap.

The rest of this paper is organized as follows: In Section 2 we review the Schur complement method. In Section 3 we discuss the techniques to preprocess the Schur complements in parallel. Then, in Section 4 we present numerical results to demonstrate the effectiveness of the preprocessing techniques for solving the Schur complement systems of large-scale highly-indefinite linear systems of equations. We also present numerical results to demonstrate that our new parallel hybrid linear solver incorporates these preprocessing techniques and efficiently solves these linear systems on a large number of processors. Finally, in Section 5 we conclude with final remarks.

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:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}, \tag{2.1}$$

where A_{11} is a block-diagonal matrix, each of whose diagonal blocks represents an *interior subdomain*, 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 (2.1) becomes

$$\begin{pmatrix} A_{11} & A_{12} \\ 0 & S \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ \widehat{b}_2 \end{pmatrix}, \tag{2.2}$$

where S is the Schur complement defined as

$$S = A_{22} - A_{21}A_{11}^{-1}A_{12}, \tag{2.3a}$$

and

$$\widehat{b}_2 = b_2 - A_{21}A_{11}^{-1}b_1. \tag{2.3b}$$

Hence, the solution of the linear system (2.1) can be computed by

1. first forming the Schur complement S and the right-hand-side vector $\widehat{b}_2$;
2. then solving the Schur complement system

$$Sx_2 = \widehat{b}_2; \tag{2.4}$$

3. and finally solving the interior system

$$A_{11}x_1 = b_1 - A_{12}x_2. \tag{2.5}$$

Since most of the fill occurs in the Schur complement S, the Schur complement system is solved using an iterative method, while the interior system is typically solved using a direct method. See [20] and the references within for a detailed discussion on the Schur complement method.

As we will discuss in Section 3.1, each interior subdomains of A_{11} are solved in parallel. To utilize a large number of processors, many interior subdomains are often needed. This increases the size of the Schur complement S, and the solution of (2.4) often suffers from either slow convergence or large memory requirement. This is especially true when solving highly-indefinite linear systems, as we will demonstrate in Section 4.

3. Parallel matrix preprocessing

To efficiently solve large-scale highly-indefinite linear systems on a large number of processors, we have been developing a new parallel implementation of the Schur complement method. In this section, we discuss one feature of our implementation: parallel preprocessing techniques to improve the reliability and performance of the solver.

3.1. Schur complement construction

Before our discussion on the parallel preprocessing techniques, let us describe how the Schur complement S is constructed in our current implementation. For our discussion here, the ℓ-th interior subdomain and corresponding interfaces are denoted by $A_{11}^{(\ell)}$, and $A_{12}^{(\ell)}$ and $A_{21}^{(\ell)}$, respectively, such that the coefficient matrix in the 2×2 block system (2.1) with k interior subdomains can be written as

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{cccc|c} A_{11}^{(1)} & & & & A_{12}^{(1)} \\ & A_{11}^{(2)} & & & A_{12}^{(2)} \\ & & \ddots & & \vdots \\ & & & A_{11}^{(k)} & A_{12}^{(k)} \\ \hline A_{21}^{(1)} & A_{21}^{(2)} & \dots & A_{21}^{(k)} & A_{22} \end{array}\right). \tag{3.1}$$

If each interior subdomain $A_{11}^{(\ell)}$ 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 to factor each interior subdomain, then the rows of $A_{11}^{(\ell)}$ and $A_{21}^{(\ell)}$, and the columns of $A_{12}^{(\ell)}$ are evenly distributed among the processors. Furthermore, the rows of A_{22} are evenly distributed among the processors solving the Schur complement system (2.4).

With the block structure (3.1) and the LU factorization of the interior subdomain $A_{11}^{(\ell)}$, which is denoted by $A_{11}^{(\ell)} = L_{11}^{(\ell)} U_{11}^{(\ell)}$,[§] the Schur complement is computed as

$$\begin{aligned} S &= A_{22} - \sum_{\ell=1}^{k} A_{21}^{(\ell)} (A_{11}^{(\ell)})^{-1} A_{12}^{(\ell)} \\ &= A_{22} - \sum_{\ell=1}^{k} \left((U_{11}^{(\ell)})^{-T} (A_{21}^{(\ell)})^{T}\right)^{T} \left((L_{11}^{(\ell)})^{-1} A_{12}^{(\ell)}\right) \\ &= A_{22} - \sum_{p=1}^{p_A} E^{(\ell)}(:, j_1^{(p)} : j_2^{(p)}) F^{(\ell)}(j_1^{(p)} : j_2^{(p)}, :), \end{aligned} \tag{3.2}$$

where p_A is the total number of processors, $E^{(\ell)}(:, j_1^{(p)} : j_2^{(p)})$ and $F^{(\ell)}(j_1^{(p)} : j_2^{(p)}, :)$ are the $j_1^{(p)}$-th through the $j_2^{(p)}$-th columns and rows of the matrices

$$E^{(\ell)} = \left((U_{11}^{(\ell)})^{-T} (A_{21}^{(\ell)})^{T}\right)^{T} \quad \text{and} \quad F^{(\ell)} = (L_{11}^{(\ell)})^{-1} A_{12}^{(\ell)}, \tag{3.3}$$

respectively. Hence, the p-th processor computes the local outer-product updates of the Schur complement S, and sends the rows of the updates to the processor that owns the corresponding rows of A_{22}. Subsequently, if p_S denotes the number of processors used to solve the Schur complement system, and n_2 is the dimension of S, then the p-th processor computes the nonzeros in the $\left(1+(p-1)\frac{n_2}{p_S}\right)$-th through the $\left(p\frac{n_2}{p_S}\right)$-th rows of S, and stores them in a row-wise order, $1 \le p \le p_S$.[¶]

[§]The matrix $A_{11}^{(\ell)}$ is scaled and permuted to enhance the numerical stability and to preserve the sparsity of $L_{11}^{(\ell)}$ and $U_{11}^{(\ell)}$. For clarity, these scaling and permutation are not shown in the expression.

[¶]If n_2 cannot be devided by p_S, then the remaining rows are assigned to the last processor.

3.2. Parallel unsymmetric scaling and permutation

After the Schur complement S is computed as described in Section 3.1, nonzeros of S with magnitudes less than a user-specified drop tolerance are discarded to form a sparsified Schur complement $\widetilde{S}$. Then, an ILU factor of $\widetilde{S}$ is computed and used as the preconditioner for solving the Schur complement system (2.4). Unfortunately, as discussed in Section 1, the preconditioned iterative solver often suffers from slow convergence or large memory requirements as the number of interior subdomains increases.

To improve the reliablity and performance of the preconditioned iterative method, we study two techniques to preprocess the Schur complement S in parallel:

1. *Scaling with infinity-norm.* Each row of S is first scaled using its infinity-norm. Then after the row scaling is applied, each column is scaled by its infinity-norm. This is one of the matrix preprocessing techniques implemented in SuperLU_DIST [15]. The infinity-norms of the rows can be computed in parallel since S is distributed by rows as described in Section 3.1. To compute the norms of the columns, global communication is needed.

2. *Bipartite weighted matching.* One powerful preprocessing technique is unsymmetric scaling and permutation that place large entries on the diagonal based on bipartite weighted matching [17]. An efficient serial implementation such as MC64 [6] has been developed, and its effectiveness for solving highly-indefinite linear systems of equations has been demonstrated [3]. However, this serial implementation cannot be used for preprocessing the Schur complement due to the large memory required to form S on each processor. A robust parallel implementation has not been developed [5, 9, 18]. As a remedy, we compute scaling and permutations of *local matrices*. Specifically, if the p-th processor owns the $i_1^{(p)}$-th through the $i_2^{(p)}$-th rows of S, then the processor computes the scaling and permutation of the local diagonal block $S(i_1^{(p)} : i_2^{(p)}, i_1^{(p)} : i_2^{(p)})$, which is the diagonal block of S between the $i_1^{(p)}$-th and $i_2^{(p)}$-th rows and columns.[‖] These scaling and permutation can be computed using an existing serial code such as MC64.

After the application of the preprocessing technique, S is sparsified to form $\widetilde{S}$. The numerical results using the preprocessing techniques will be presented in Section 4.

3.3. Parallel supernodal nested dissection

Before computing an ILU preconditioner of the sparsified Schur complement $\widetilde{S}$, we permute $\widetilde{S}$ to preserve the sparsity of the preconditioner. There are a number of parallel software packages [4,12] that compute fill-reducing permutations of a sparse matrix based on a nested dissection algorithm. Unfortunately, $\widetilde{S}$ is relatively dense even after the sparsification, and an existing software package does not perform well, especially on a large number of processors (numerical results will be presented in Section 4).

[‖]If the diagonal block is singular, some diagonal entries can be zero. A postprocessing may be required to avoid zero pivots while computing preconditioners. Zero pivots were not encountered in our numerical experiments.

Table 1: Pesudocode for the row compression.

```
Algorithm 3.1:
1.  $i_0 = i_1^{(p)}$                          // first row of current supernode
2.  $n_0 = \sigma \cdot \text{nnz}(\widetilde{S}(i_0,:))$   // max. number of mismatched nonzeros
3.  $j = 1$                                    // number of supernodes
4.  $s_j = \{i_0\}$                            // set of rows in the j-th supernode
5.  for $i = i_1^{(p)}+1, i_1^{(p)}+2, \ldots, i_2^{(p)}$
6.    $miss = 0$
7.    for each nonzeros $\widetilde{s}_{ij}$ in $\widetilde{S}(i,:)$
8.      if $\widetilde{s}_{i_0 j}$ is zero then
9.        $miss = miss + 1$
10.     end if
11.   end if
12.   if $miss > n_0$ then                     // create a new supernode
13.     $i_0 = i$ and $n_0 = \sigma \cdot \text{nnz}(\widetilde{S}(i,:))$
14.     $j = j+1$ and $s_j = \{i_0\}$
15.   else                                     // assign to current supernode
16.     $s_j = s_j \cup \{i\}$
17.   end if
18. end for
```

To reduce the time spent computing the permutation, we compress the rows of $\widetilde{S}$ that have similar sparsity patterns into a single *supernode* before computing the permutation. This is done in parallel; specifically, the p-th processor compresses the local matrix by comparing the sparsity pattern of each row with that of the first row in the current supernode. The pseudocode in Table 1 describes this algorithm. On line 2 of the pseudocode, $\widetilde{S}(i_0,:)$ is the i_0-th row of $\widetilde{S}$, and σ is a user-specified threshold that specifies the allowable difference among the sparsity patterns of the rows in a supernode, $0 \le \sigma \le 1$; specifically, it specifies the maximum ratio of the number of nonzeros in the i-th row, which do not match the sparsity pattern of the first row in the same supernode, over the total number of nonzeros in the i-th row. Hence, if $\sigma = 0$, the sparsity pattern of the first row contains those of all the rows in the same supernode. A larger σ allows a greater difference in the sparsity pattern and a greater compression rate. This will reduce the time needed to compute the permutation, but may degrade the quality of the resulting permutation (numerical results will be presented in Section 4). On line 3, j counts the number of supernodes, and on line 4 the first supernode s_j is initialized to contain only the first row of the local matrix. On lines 6 through 11, we count the number of mismatched nonzeros in the i-th row using the counter *miss*. Then, on lines 13 and 14, if *miss* is greater than the user-specified threshold, the current row is assigned to a new supernode. Otherwise, the row is assigned to the current supernode on line 16.

Note that in Algorithm 3.1, we are comparing the sparsity pattern of the i-th row against that of the first row of a supernode, but not vice versa. Specifically, the first row may have many nonzeros which do not match with the sparsity patterns of the rest of the rows in the supernode. The quality of the supernode can be improved by comparing the sparsity pattern of the first row against those of the rest of the rows, but this will require some additional computations. In practice, we observed that $\widetilde{S}$ has dense block structures, and Algorithm 3.1 is effective enough, as will be shown in Section 4.

After the rows of $\widetilde{S}$ are compressed into supernodes, the corresponding columns are also compressed. Finally, we compute a nested dissection of this supernodal graph using an existing parallel software package. The supernodal graph has an edge between two supernodes if the adjacency graph of $\widetilde{S}$ has an edge between two vertices, each belonging to a different supernode of those two supernodes. After the nested dissection ordering of the supernodal graph is computed, the permutation of $\widetilde{S}$ is computed simply by expanding the permutation of the supernodes.

4. Numerical experiments

Table 2: The matrices used in the numerical experiments.

name	n	sym	k	n_2	nnz(S)
tdr190	1,100,242	yes	32	31,272	97,600,856
dds.qd	380,698	yes	32	24,878	71,335,044
dds.ln	834,575	yes	31	26,110	66,407,552
mat211	801,378	no	31	27,660	90,519,264
tmt.sy	726,713	yes	125	22,941	7,823,701

In this section, we present numerical results of the preprocessing techniques described in Section 3. In Table 2, we show some properties of the Schur complements S of the indefinite matrices used in the experiments; "n" is the dimension of the original matrix A, under "sym," we identify if the matrix is symmetric (the matrix elements are real), "k" is the number of interior subdomains**, and "n_2" and "nnz(S)" are the dimension of and the number of nonzeros in S, respectively. The matrices **tdr190**, **dds.qd**, and **dds.ln** arise from numerical simulations of particle accelerator cavity designs [2,14]: **tdr190** is for an international linear collider, and **dds.qd** and **dds.ln** are for dumped detuned structures with quadratic and linear elements, respectively. The simulation involves nonlinear eigenvalue problems for solving discretized Maxwell equations, where the solutions of the linear systems are needed for the shift-and-invert operations. When the shift is close to an actual eigenvalue, these linear systems are close to singular and extremely difficult to solve using a preconditioned iterative method. The matrix **mat211** is from a linear system of the discretized extended MHD equations for a numerical simulation of fusion device modeling [1]. The last matrix **tmt.sy** is from a symmetric electromagnetics problem, and is available from the University of Florida Sparse Matrix Collection††.

In Sections 4.1 and 4.2, we present numerical results of solving the Schur complement systems. The unrestarted GMRES of PETSc [16] was used as our preconditioned itera-

**The parameter k is chosen such that the dimensions of the Schur complements are about 25,000. The 2×2 block system (2.1) is computed using an existing software package, Hierarchical Interface Decomposition (HID) [10].

††`http://www.cise.ufl.edu/research/sparse/matrices/CEMW/tmt_sym.html`

tive method. The GMRES iteration was started with the zero vector, and the computed solution $\bar{x}_2$ was considered to be converged when the ℓ_2-norm of the initial residual was reduced by at least twelve order of magnitude, i.e., $\|\bar{b}_2 - \bar{S}\bar{x}_2\|_2 / \|\bar{b}_2\|_2 \leq 10^{-12}$, where $\bar{S}$ and $\bar{b}_2$ are the Schur complement and the right-hand-side vector, respectively, after the preprocessing is applied. Then, in Section 4.3, we compare the performance of our parallel hybrid solver using the preprocessing techniques to solve the original linear systems with those of existing state-of-the-art parallel solvers. Our hybrid solver was implemented in C.

4.1. Numerical behaviors

We first study the effects of the preprocessing techniques on the numerical behavior of GMRES using a single processor. The numerical experiments were conducted on a desktop machine with Intel 2.7GHz Quad-Core CPUs and 8GB of main memory. All the codes were compiled using the **gcc** compiler and **-O3** optimization flag.

Table 3 shows the performance of GMRES using three preprocessing techniques: no preprocessing, scaling with the infinity-norms, and ones generated using MC64 ("none", "∞-scale, and "MC64," respectively, in the table). The sparsity of $\widetilde{S}$ is enforced using a drop tolerance τ_1 such that the (i,j)-th element s_{ij} of S is discarded if $|s_{ij}| < \tau_1\sqrt{|s_{ii}s_{jj}|}$. ILU preconditioners were computed using the PETSc interface to the ILUTP subroutine of SPARSKIT [19], which uses a drop tolerance τ_2 to enforce the sparsity of the preconditioners. In the table, "fill" is the fill-ratio, "itrs" is the number of GMRES iterations needed for the solution convergence, and "time" is the total solution time required to solve the Schur complement system in seconds (i.e., the total time spent to preprocess and sparsify S, compute the preconditioner, and perform GMRES iterations). The fill-ratio in the table is the ratio of the total number of nonzeros in $\widetilde{S}$ and its ILU factors over the total number of nonzeros in the matrices A_{21}, A_{22}, and A_{12}, namely,

$$\frac{nnz(\widetilde{S}) + nnz(ILU(\widetilde{S}))}{nnz(A_{21}) + nnz(A_{22}) + nnz(A_{12})}.$$

This approximately measures the memory required for solving the Schur complement systems. Note that when the hybrid solver is used to solve a linear system, most of the fill occurs in the Schur complement S, and this fill-ratio is large (i.e., $nnz(\widetilde{S})$ is large). On the other hand, the overall fill-ratio, which approximates the total memory required for solving the entire system, can be expressed as the ratio of the total number of nonzeros in the LU factors of A_{11} and the ILU factor of $\widetilde{S}$ over the number of nonzeros in the matrix A, namely

$$\frac{nnz(LU(A_{11})) + nnz(ILU(\widetilde{S}))}{nnz(A)}.$$

Typically, this overall fill-ratio is much smaller than the fill-ratio shown in the table. In all the numerical experiments presented in the table, the overall fill-ratio was between 7.1 and 48.8. Under "itrs" in the table, "——" indicates that GMRES did not converge within 250 iterations.

Table 3: Effects of the matrix preprocessing.

			none			∞-scale			MC64		
matrix	τ_1	τ_2	fill	itr.	time	fill	itr.	time	fill	itr.	time
tdr190	10^{-8}	0.0	78	104	2949	87	4	3527	81	3	2753
	10^{-8}	10^{-8}	75	104	2567	82	4	2879	78	3	2389
	10^{-8}	10^{-6}	69	104	1968	77	5	2722	76	4	2135
	10^{-6}	0.0	56	−−	−−	78	8	3447	73	8	2724
	10^{-6}	10^{-6}	48	−−	−−	67	9	2086	63	9	1664
	10^{-6}	10^{-4}	37	−−	−−	56	86	1079	50	52	778
	10^{-5}	0.0	38	−−	−−	69	23	3282	64	23	2620
	10^{-5}	10^{-5}	24	−−	−−	54	24	1528	48	24	1137
	10^{-5}	10^{-4}	19	−−	−−	47	95	1047	41	56	745
dds.qd	2×10^{-7}	0.0	85	−−	−−	108	4	6357	108	3	6354
	2×10^{-7}	2×10^{-7}	84	−−	−−	107	4	5727	107	3	5750
	2×10^{-7}	2×10^{-5}	79	−−	−−	92	8	3664	92	8	3658
	10^{-4}	0.0	7	−−	−−	86	79	6145	86	80	6138
	10^{-4}	10^{-4}	6	−−	−−	62	86	2651	60	80	2479
	10^{-4}	10^{-3}	5	−−	−−	76	−−	−−	76	−−	−−
dds.ln	10^{-8}	0.0	137	18	1160	203	2	1524	201	2	1468
	10^{-8}	10^{-8}	135	18	1114	202	2	1479	200	2	1442
	10^{-8}	10^{-4}	102	20	525	165	12	666	160	12	675
	5×10^{-5}	0.0	5	−−	−−	129	21	1098	129	21	1109
	5×10^{-5}	5×10^{-5}	2	−−	−−	101	21	579	102	21	595
	5×10^{-5}	5×10^{-4}	1	−−	−−	79	31	298	79	32	307
mat211	10^{-6}	0.0	92	−−	−−	76	16	10970	52	16	4533
	10^{-6}	10^{-6}	72	−−	−−	43	16	3377	30	16	1472
	10^{-6}	10^{-4}	72	−−	−−	22	−−	−−	15	42	291
	10^{-5}	0.0	75	−−	−−	67	36	9737	43	35	3783
	10^{-5}	10^{-5}	68	−−	−−	25	45	1410	14	35	480
	10^{-5}	10^{-4}	67	−−	−−	17	49	634	9	−−	−−
tmt.sy	10^{-4}	10^{-4}	29	88	19	50	55	24	50	55	25
	10^{-4}	10^{-3}	18	98	15	28	87	15	28	88	17

The table clearly indicates that for fixed drop tolerances (τ_1, τ_2), preprocessing significantly improves the GMRES convergence rate. Even when the drop tolerances were chosen to achieve a similar fill-ratio, GMRES still converged within fewer iterations using preprocessing. For example, with $(\tau_1, \tau_2) = (10^{-8}, 10^{-6})$ and no preprocessing on **tdr190**, the fill-ratio was 69 and GMRES converged with 104 iterations. On the other hand, with ∞-scale and MC64, a similar fill-ratio was achieved using $(\tau_1, \tau_2) = (10^{-6}, 10^{-6})$, and GMRES needed only 9 iterations. Furthermore, MC64 often performed better than ∞-scale. For example, with $\tau_1 = 10^{-6}$ and $\tau_2 = 0.0$ or 10^{-6}, GMRES required the same numbers of iterations using ∞-scale and MC64. However, the fill-ratio was smaller with MC64, leading to faster total solution time. Note that ILUTP dynamically permutes columns of $\widetilde{S}$ to avoid small pivots, which may increase the fill-ratio. We found that ILUTP performs more permutations using ∞-scale, which may be the reason for the greater fill (e.g., with $(\tau_1, \tau_2) = (10^{-6}, 10^{-4})$, ILUTP performed 2,246 and 79 permutations with ∞-scale and MC64, respectively). Furthermore, with $(\tau_1, \tau_2) = (10^{-6}, 10^{-4})$, MC64 resulted in a smaller fill-ratio and fewer iterations, leading to significantly faster solution time. For most of the cases with **mat211**, when both τ_1 and τ_2 were fixed, GMRES required fewer iterations with MC64 than ∞-scale, leading to faster solution time. An exception was when $(\tau_1, \tau_2) = (10^{-5}, 10^{-4})$. Alternatively, when only τ_1 is fixed, MC64 required a smaller fill-ratio to achieve a similar number of GMRES iterations, hence leading to faster solution time (e.g., With ∞-scale and $(\tau_1, \tau_2) = (10^{-5}, 0.0)$, GMRES converged with 36 iterations with the fill-ratio of 67. With MC64 and $(\tau_1, \tau_2) = (10^{-5}, 10^{-5})$, GMRES converged with 35 iterations with the fill-ratio of only 14). For **dds.qd**, ∞-scale and MC64 lead to similar performance improvements over no preprocessing, while for **tmt.sy** and **dds.ln**, preprocessing only had a small effect. The effects of preprocessing are problem dependent, but for all the test cases, it did not degrade the convergence while the overhead was small. Therefore, MC64 is our default choice.

4.2. Parallel performance

We now present numerical results on a distributed memory system. These numerical results were collected on a Cray XT4 machine named Franklin at the National Energy Research Scientific Computing Center (NERSC). The **pgcc** compiler and **-fastsse** optimization flag were used to compile the codes.

Table 4: Effects of the supernodal nested dissection on **tdr190**, $(\tau_1, \tau_2) = (10^{-5}, 0.0)$.

					Time	
p_S	σ	s	nnz$_{LU}$	perm.	fact.	total
2	none	31,272	101M	21.98	62.80	100.68
	0.3	15,748	125M	8.56	78.87	103.82
	0.5	5,583	136M	2.38	77.29	97.14
8	none	31,272	116M	8.63	20.28	33.25
	0.3	15,752	119M	4.23	20.90	29.81
	0.5	5,591	132M	0.97	20.00	25.60
32	none	31,272	112M	5.13	5.34	11.80
	0.3	15,750	124M	2.02	5.89	9.20
	0.5	5,604	136M	0.50	5.95	7.87

Let us first examine the effectiveness of the supernodal nested dissection discussed in Section 3.3. For these experiments, we used PT-SCOTCH [4] and SuperLU_DIST [15] to compute the permutation and LU factorization of $\widetilde{S}$ (i.e., $\tau_2 = 0.0$), respectively. Table 4 shows the numerical results for **tdr190**, where "p_S" is the number of processors used to solve the Schur complement system, "s" is the number of supernodes found in $\widetilde{S}$, and under "Time," we show the times spent computing the permutation, which includes the compression of the rows, and the time spent for the numerical LU factorization of $\widetilde{S}$ ("perm" and "fact" in the table, respectively), and "total" is the total time spent for the LU factorization. These times are in seconds. We see that without the row compression, the time spent by

PT-SCOTCH does not scale as well as the time spent for the numerical factorization. As a result, a larger fraction of time is spent in PT-SCOTCH as a larger number of processors is used. Now, with the row compression, the time of PT-SCOTCH is significantly reduced. When more rows were compressed using a large σ, the quality of the permutation degraded slightly; specifically, both the number of nonzeros in the LU factors and the time required for the numerical LU factorization increased. However, the time spent to compute the permutation was reduced more significantly (with $\sigma = 0.5$, it was reduced by an order of magnitude), and the total factorization time was reduced by a factor of 1.5 using $\sigma = 0.5$ on 32 processors.

We now examine the effect of parallel processing techniques on the performance of GMRES. For these experiments, we used MC64 and $(\tau_1, \sigma) = (10^{-5}, 0.5)$ to preprocess and sparsify S, and used SuperLU_DIST to compute the preconditioners (i.e., $\tau_2 = 0.0$). In Table 5, "fill" is the fill ratio as defined for Table 3, "itrs" is the number of GMRES iterations, "Pt" is the time spent to preprocess and sparsify S, "Lt" is the time spent to compute preconditioners, "St" is the time for the GMRES iterations, and "Tt" is the total solution time. These times are in seconds. We first note that the time spent for preprocessing is only a small fraction of the total solution time. If ∞-scale was used, the preprocessing time would be less. We next note that as the number of processors (p_S) increases, MC64 computes the scaling and permutation of smaller local diagonal matrices. The table shows that both fill and itrs are similar to those in Table 3 for $(\tau_1, \tau_2) = (10^{-5}, 0.0)$, where the scaling and permutation were computed based on the global S. These results demonstrate that the scaling and permutation based on the diagonal blocks of S are as effective as those based on the global S.

Table 5: Performance to solve Schur complement system, $(\tau_1, \tau_2, \sigma) = (10^{-5}, 0.0, 0.5)$.

	tdr190						**mat211**					
p_S	fill	itrs	Pt	Lt	St	Tt	fill	itrs	Pt	Lt	St	Tt
4	69	23	1.6	47.5	8.6	57.7	32	35	1.1	38.8	12.9	52.8
8	67	23	0.8	24.6	7.2	32.6	30	35	0.5	21.4	12.0	33.9
16	69	23	0.4	14.5	6.2	21.1	30	34	0.3	12.6	9.8	22.7
32	69	23	0.2	7.9	5.8	13.9	29	34	0.1	8.3	8.2	16.6

4.3. Solver comparison

In this subsection, we compare the performance of our parallel hybrid solver using the parallel preprocessing techniques to solve the entire system (2.1) with those of state-of-the-art parallel solvers.

In Table 6, we first compare the performance of a parallel direct solver SuperLU_DIST with that of our hybrid solver, where SuperLU_DIST is used to compute the preconditioner (i.e., $\tau_2 = 0.0$). An additional drop tolerance τ_0 was used to discard small nonzeros from the matrices $E^{(\ell)}$ and $F^{(\ell)}$ of (3.2), and sparsity of $\widetilde{S}$ was enforced using $\tau_1 = 10^{-5}$. In the tables, "p_A" is the total number of processors, "nnz" is the number of nonzeros in the preconditioners, and the rest of the fields are the same as those used in Section 4.2. We

Table 6: Performance to solve entire system, $(\tau_0, \tau_1, \sigma) = (10^{-6}, 10^{-5}, 0.5)$ for **tdr190**, $(\tau_0, \tau_1, \sigma) = (10^{-7}, 10^{-6}, 0.5)$ for **mat211**.

		SuperLU_DIST				HYBRID+SuperLU_DIST				
matrix	p_A	nnz	Lt	St	Tt	nnz	itrs	Lt	St	Tt
tdr190	8	711M	71.1	2.7	73.8	667M	12	101.4	8.5	109.9
	32	721M	48.6	2.0	50.6	567M	25	59.6	6.1	65.6
	127	721M	56.8	1.4	58.1	608M	40	22.6	8.4	31.0
	504	747M	83.5	1.0	84.5	704M	32	10.4	8.2	18.6
mat211	8	1271M	163.0	1.0	164.1	491M	14	123.4	8.3	131.7
	32	1461M	87.0	0.6	87.6	463M	16	34.4	3.8	38.2
	128	1484M	54.6	0.4	55.0	507M	14	16.2	3.0	19.2

Table 7: Performance to solve entire system, $(\tau_1, \tau_2) = (10^{-6}, 0.0)$ for **tdr190**, $(10^{-7}, 0.0)$ for **mat211**.

		PHIDAL					HIPS				
matrix	p_A	nnz	itrs	Lt	St	Tt	nnz	itrs	Lt	St	Tt
tdr190	8	459M	24	322.6	21.5	345.0	624M	10	90.1	2.7	92.8
	16	368M	−−	107.7	−−	−−	574M	75	87.8	8.9	96.8
	32	−·−	−·−	−·−	−·−	−·−	491M	−−	92.1	−−	−−
mat211	8	294M	64	179.3	33.5	212.8	875M	26	277.9	6.7	284.6
	32	252M	−−	40.9	−−	−−	680M	54	51.8	3.6	55.4
	128	−·−	−·−	−·−	−·−	−·−	441M	−−	26.1	−−	−−

used one processor to factor each interior subdomain, and half of the total processors to solve the Schur complement systems (i.e., $p_S = p_A/2$). The table shows that for **tdr190**, SuperLU_DIST did not scale beyond 32 processors, while our hybrid solver could reduce the solution time using up to 504 processors. Similarly, for **mat211**, our hybrid solver scaled much better than SuperLU_DIST. As a result, the hybrid solver achieved a speedup of 4.5 using 504 processors for **tdr190** and a speedup of 2.9 using 128 processors for **mat211** over SuperLU_DIST. We also note that for **mat211**, the number of nonzeros in the preconditioner was reduced by a factor of up to 3.1 using the hybrid solver over that using the direct solver. The reduction was smaller for **tdr190**. This is because as discussed earlier, **tdr190** is extremely difficult to solve using a preconditioned iterative method and requires a high-quality preconditioner. Even though our current implementation may not reduce the total memory requirement in comparison to the direct solver, it provides the flexibility to efficiently solve the linear system on a large number of processors and the potential to reduce the memory requirement per processor.

In Table 7, we show the performance of PHIDAL [10], which is a parallel preconditioned iterative solver based on GMRES combined with an ILU preconditioner, and the performance of HIPS [7], which is another implementation of the Schur complement method. Due to the highly-indefinite nature of the problem, ILU preconditioners were not effective for solving the entire system. In the table, "– · –' indicates that we have not performed the experiments since GMRES is expected not to converge. The primary difference between our hybrid solver and HIPS is the way the preconditioner is computed for solving the Schur complement system (2.4). Similarly to our implementation, HIPS computes the preconditioner based on the ILU factorization of S, but the 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 subdomain. Hence, 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. This allows HIPS to achieve a good scalability as it can be seen when the number of processors increased from 8 to 32 for **mat211**. However, because of these strict sparsity constraints, the number of GMRES iterations increases quickly with the number of interior subdomains (p_A), and HIPS failed to converge with 32 interior subdomains for **tdr190** and with 128 interior subdomains for **mat211**. Even when HIPS converged, its solution time was about twice as long as that of HYBRID+SuperLU_DIST for **mat211** due to the slower time to construct the preconditioner. For **tdr190**, when HIPS converged, the solution times using HIPS and HYBRID+SuperLU_DIST were about the same. We also note that there were more nonzeros in the preconditioners computed by HIPS than those computed by HYBRID+SuperLU_DIST.

Finally, we note that our hybrid solver can use the preconditioners included in PETSc. For example, we tested additive Schwartz preconditioners to solve the Schur complement systems and to solve the entire linear systems. This is a popular approach used in domain decomposition, and there is a parallel hybrid solvers based on this type of preconditioner [8]. Unfortunately, our preliminary results indicated that these preconditioners were not effective for solving highly-indefinite linear systems; specifically, the number of GMRES iterations increased quickly with the number of processors. For example, when the Schur complement system for **tdr190** is generated with 32 subdomains, an additive Schwartz preconditioner with 16 subdomains required 174 GMRES iterations. GMRES failed to converge within 250 iterations for solving the Schur complement system with 128 subdomains using the additive Schwartz preconditioner with 64 subdomains. To solve the entire system using the additive Schwartz precondition with 8 subdomains, GMRES did not converge within 250 iterations.

5. Conclusion

In this paper, we studied matrix preprocessing techniques for solving the Schur complement systems of large-scale highly-indefinite linear systems of equations using a preconditioned iterative method. The numerical results were presented to demonstrate that the preprocessing techniques improve both the reliability and the performance of the solver. Furthermore, we showed the effectiveness of these preprocessing techniques on a distributed memory system; computing the scaling and permutation based on local matrices can achieve similar performance improvement as that based on the global matrix, and compressing the rows of the Schur complement can reduce the time to compute the permutation by an order of magnitude. Finally, we demonstrated that by using these preprocessing techniques, our new parallel hybrid solver achieves good parallel scalability to solve large-scale highly-indefinite linear systems on a large number of processors, which was previously infeasible using existing state-of-the-art parallel solvers.

Acknowledgments The authors gratefully thank Cedric Chevellar, Mahantesh Halappanavar, Lie-Quan Lee, and François Pellegrini for helpful discussions. 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] Center for Extended MHD Modeling (CEMM). `http://w3.pppl.gov/cemm/`.

[2] Community Petascale Project for Accelerator Science and Simulation (ComPASS). https://compass.fnal.gov.

[3] M. Benzi, J. Haws, and M. Tuma. Preconditioning highly indefinite and nonsymmetric matrices. *SIAM J. Sci. Comput.*, 22(4):1333–1353, 2000.

[4] C. Chevalier and F. Pellegrini. PT-Scotch. *Parallel Computing*, 34(6–8):318–331, 2008.

[5] I. Duff. Developments in matching and scaling algorithms. In *6th international congress on industrial and applied mathematics*, 2007.

[6] I. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, 1999.

[7] J. Gaidamour and P. Hènon. HIPS: a parallel hybrid direct/iterative solver based on a schur complement. In *Proc. PMAA*, 2008.

[8] L. Giraud, A. Haidar, and L. T. Watson. Parallel scalability study of hybrid preconditioners in three dimensions. *Parallel Computing*, 34:363–379, 2008.

[9] M. Halappanavar, F. Bobrian, and A. Pothen. A parallel half-approximation algorithm for the weighted matching problem. In *SIAM conference on Computational Science and Engineering*, 2009.

[10] P. Hènon and Y. Saad. A parallel multilevel ilu factorization based on a hierarchical graph decomposition. *SIAM J. Sci. Comput*, 28(6):2266–2293, 2006.

[11] 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`.

[12] Karypis Lab, Digital Technology Center, Department of Computer Science and Engineering, University of Minesota. ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering. `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.

[13] 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/`.

[14] L-Q. Lee, Z. Li, C.-K. Ng, and K. Ko. Omega3P: A parallel finite-element eigenmode analysis code for accelerator cavities. Technical Report SLAC-PUB-13529, Stanford Linear Accelerator Center, 2009.

[15] X. Li and J. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, 2003.

[16] Mathematics and Computer Science Division, Argonne National Laboratory. The portable, extensible, toolkit for scientific computation (PETSc). `www.mcs.anl.gov/petsc`.

[17] M. Olschowka and A. Neumaier. A new pivoting strategy for gaussian elimination. *Linear Algebra Appl.*, 240:131–151, 1996.

[18] J. Riedy. Auctions for distributed (and possibly parallel) matchings. In *CERFACS algorithm team meeting*, 2008.

[19] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report RIACS-90-20, Research Institute for Advanced Computer Science, 1990.

[20] B. Smith, P. Bjorstad, and W. Gropp. *Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, New York, 1996.