# On techniques to improve robustness and scalability of the Schur complement method

Ichitaro Yamazaki and Xiaoye S. Li

Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA

**Abstract.** A hybrid linear solver based on the Schur complement method has great potential to be a general purpose solver scalable on tens of thousands of processors. It is imperative to exploit two levels of parallelism; namely, solving independent subdomains in parallel and using multiple processors per subdomain. This hierarchical parallelism can lead to a scalable implementation which maintains numerical stability at the same time. In this framework, load imbalance and excessive communication, which can lead to performance bottlenecks, occur at two levels: in an intra-processor group assigned to the same subdomain and among inter-processor groups assigned to different subdomains. We developed several techniques to address these issues, such as taking advantage of the sparsity of right-hand-side columns during sparse triangular solutions with interfaces, load balancing sparse matrix-matrix multiplication to form update matrices, and designing an effective asynchronous point-to-point communication of the update matrices. We present numerical results to demonstrate that with the help of these techniques, our hybrid solver can efficiently solve large-scale highly-indefinite linear systems on thousands of processors.

## 1 The Schur complement method and parallelization

Modern numerical simulations give rise to large-scale sparse linear systems of equations that are difficult to solve using standard techniques. Matrices that can be directly factorized are limited in size due to large memory requirements. Preconditioned iterative solvers require less memory, but often suffer from slow convergence. To mitigate these problems, several parallel hybrid solvers have been developed based on a non-overlapping domain decomposition idea called the Schur complement method [5, 7].

In the Schur complement method, the original linear system is first reordered into a $2 \times 2$ block system of the following form:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}, \tag{1}$$

where $A_{11}$ and $A_{22}$ respectively represent *interior subdomains* and *separators*, and $A_{12}$ and $A_{21}$ are the *interfaces* between $A_{11}$ and $A_{22}$. By eliminating the unknowns associated with the interior subdomains $A_{11}$, we obtain

$$\begin{pmatrix} A_{11} & A_{12} \\ 0 & S \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ \widehat{b}_2 \end{pmatrix}, \tag{2}$$

where $S$ is the Schur complement defined as

$$S = A_{22} - A_{21}A_{11}^{-1}A_{12}, \tag{3}$$

and $\widehat{b}_2 = b_2 - A_{21}A_{11}^{-1}b_1$. Subsequently, the solution of the linear system (1) can be computed by first solving the Schur complement system

$$Sx_2 = \widehat{b}_2, \tag{4}$$

and then solving the interior system

$$A_{11}x_1 = b_1 - A_{12}x_2. \tag{5}$$

For a detailed discussion of the Schur complement method, see [13] and the references therein.

The inverse of the interior subdomains $A_{11}$ is needed to form the Schur complement $S$ of (3) and to compute the correspoinding parts of the solution vector by the backward substitution (5). The existing parallel hybrid solvers use a direct method to factorize these interior subdomains, while a preconditioned iterative method is used to solve the Schur complement system (4), where most of the fill occurs. These solvers often exhibit great parallel performance since the interior subdomains can be factorized independently from one other, and a direct method is effective for factorizing these relatively small subdomains. Furthermore, for a symmetric positive definite system, the Schur complement has a smaller condition number than the original matrix [13, Section 4.2], and fewer iterations are often needed to solve the Schur complement system. General purpose parallel preconditioners for the Schur complement system have been developed, and their effectiveness has been shown for some applications [5, 7]. Unfortunately, for highly-indefinite systems, we found that these solvers still suffer from slow convergence.

In this paper, we present some of the challenges encountered in the development of a robust and efficient general purpose hybrid solver targeted for thousands of processors and our approaches to resolving these issues. Our parallel implementation consists of the following three phases:

*1) Extracting and factorizing the interior subdomains.* We use a parallel nested disection algorithm implemented in PT-SCOTCH [8] to extract interior subdomains. For an unsymmetric matrix $A$, PT-SCOTCH is applied to the graph of $|A|+|A|^T$. Then, these interior subdomains are factorized using a direct method.

There are two approaches to assigning processors to factorize subdomains. One approach is to assign a single processor to factorize one or more interior subdomains, which we refer to as a *one-level* parallel approach. An advantage of this approach is that multiple subdomains can be assigned to a processor such that the workload is balanced among the processors. A serious drawback of this approach, however, is that many subdomains must be generated in order to use large numbers of processors. This increases the size of the Schur complement, and

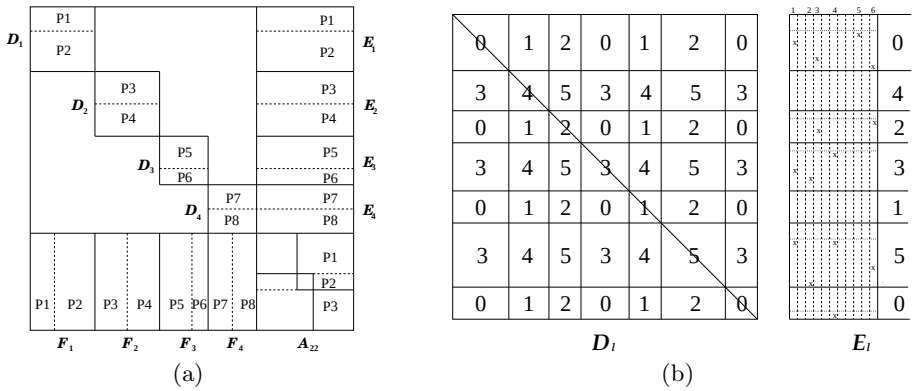



**Fig. 1.** (a): distribution of the coefficient matrix. Each group $g_\ell$ contains two processors per subdomain, and the group $g_S$ contains three processors. (b): subdomain $D_\ell$ stored in a 2D block-cyclic format using a $2 \times 3$ process grid, and its corresponding right-hand-side (RHS) vectors $E_\ell$ with a block size of six. An "x" in $E_\ell$ represents the first nonzero in an individual column of the supernodal block, and a horizontal dotted line represents the first nonzero in the entire block.

often leads to slow or even no convergence. An alternative is to assign multiple processors to each interior subdomain, which allows us to increase the processor count without increasing the number of subdomains or the size of the Schur complement. This approach is refered to as a *two-level* parallel approach and is the focus of our study in this paper.

Specifically, when $k$ interior subdomains are extracted, the coefficient matrix of Eq. (1) has the following block-structure:

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{ccc|c} D_1 & & & E_1 \\ & D_2 & & E_2 \\ & & \ddots & \vdots \\ & & D_k & E_k \\ \hline F_1 & F_2 & \ldots\ F_k & A_{22} \end{array}\right), \tag{6}$$

where $D_\ell$ is the $\ell$-th subdomain, and $E_\ell$ and $F_\ell$ are the interfaces between $D_\ell$ and $A_{22}$. In our implementation, each processor is assigned to a processor groups $g_\ell$ that factorizes the subdomain $D_\ell$. Then, the rows of $D_\ell$ and $E_\ell$, and the columns of $F_\ell$ are distributed among the processors in the processor group $g_\ell$. The nonzeros of $D_\ell$ and $F_\ell$ are stored in the Compressed Row Storage (CRS) format, while those of $E_\ell$ are stored in the Compressed Column Storage (CCS) format [3]. Another processor group $g_S$ is created, which consists of a subset of avaliable processors and will be used to solve the Schur complement system. The rows of $A_{22}$ are distributed among the processors in this processor group $g_S$. The selection of processors to be assigned to $g_S$ will be discussed in Section 2.3. Fig. 1(a) shows an example of a matrix distribution using our two-level approach. Finally, the parallel direct solver SuperLU_DIST [10] is used to factorize each subdomain.

*2) Computing an approximate Schur complement.* This is the most challenging phase of the parallel algorithm, especially in a two-level parallel framework. We need to deal with the load imbalance and communication not only within an *intra-processor group* assigned to the same subdomain, but also among the *inter-processor groups* assigned to different subdomains. We have developed a number of techniques to enhance performance of our hybrid solver to compute the approximate Schur complement $\widetilde{S}$. These techniques are the focus of our paper and will be discussed in Section 2.

*3) Computing the solution.* A preconditioned Krylov method of PETSc [11] is used to solve the Schur complement system (4), where the preconditioner is the exact LU factors of an approximate Schur complement $\widetilde{S}$. SuperLU_DIST is used to compute the preconditioner. At each iteration, the matrix-vector multiplication with $S$ is computed by applying a sequence of the sparse matrix operations (3) on the vector, and hence, $S$ is not stored explicitly. To improve the load balance of the matrix-vector multiplication, the matrices $D_\ell$, $E_\ell$, and $F_\ell$ are distributed among the processors in the processor group $g_\ell$ such that they each own a similar number of nonzeros. Recall that SuperLU_DIST uses a 2D block-cyclic format internally (see Fig. 1(b)). Hence, the performance of SuperLU_DIST to compute the LU factorization of $D_\ell$ and to solve the corresponding linear system is not affected by the initial distribution of the coefficient matrix $D_\ell$ and the RHS columns $E_\ell$ and $F_\ell$. The final solution is computed by solving the interior system (5) with the already-computed LU factors.

## 2 Efficient computation of an approximate Schur complement

In this section, we describe the techniques to enhance the performance of our hybrid solver to compute an approximate Schur complement $\widetilde{S}$. To demonstrate the effectiveness of these techniques, we use the numerical results of a highly-indefinite matrix from the numerical simulation of an accelerator cavity design [1, 9]; namely, **tdr455k** of dimension $2,738,556$ with $112,756,352$ nonzeros. For the numerical experiments, we extracted 16 subdomains $D_\ell$ using PT-SCOTCH. All the experiments were conducted on the Cray XT4 machine at NERSC.

Given an LU factorization $D_\ell = L_\ell U_\ell$,[1] the Schur complement $S$ of (3) is computed as follows:

[1] The matrix $D_\ell$ is scaled and permuted to enhance numerical stability and preseve the sparsity of $L_\ell$ and $U_\ell$. For clarity, the scaling and permutation are not shown in the expression.

$$S = A_{22} - \sum_{\ell=1}^{k} F_\ell D_\ell^{-1} E_\ell \tag{7}$$

$$= A_{22} - \sum_{\ell=1}^{k} (U_\ell^{-T} F_\ell^T)^T (L_\ell^{-1} E_\ell) \tag{8}$$

$$= A_{22} - \sum_{p=1}^{n_p} W^{(p)} G^{(p)}, \tag{9}$$

where $n_p$ is the number of processors used to solve the entire system, and the matrices $G^{(p)}$ and $W^{(p)}$ are given by

$$G^{(p)} = G(j_p : (j_{p+1} - 1),\ :), \quad W^{(p)} = W(:,\ j_p : (j_{p+1} - 1)), \tag{10}$$

such that the $p$-th processor owns the $j_p$-th through $(j_{p+1} - 1)$-th rows of $G = L_{11}^{-1} A_{12}$ and the corresponding columns of $W = (U_{11}^{-T} A_{21}^T)^T$, where the LU factorization $A_{11} = L_{11} U_{11}$ is given by $L_\ell$ and $U_\ell$. Once the matrices $G^{(p)}$ and $W^{(p)}$ are computed, the $p$-th processor computes its update matrix $T^{(p)} = W^{(p)} G^{(p)}$. To efficiently manage the required memory, the memory for storing each row of $W^{(p)}$ is freed as soon as the corresponding row of $T^{(p)}$ is computed. After $T^{(p)}$ is computed, the rows of $T^{(p)}$ are sent to the $q$-th processor which owns the corresponding rows of $A_{22}$, and the $q$-th processor computes the corresponding rows of the Schur complement $S$.

Large amounts of fill may occur in $G^{(p)}$ and $W^{(p)}$. To reduce the memory and computational costs, their approximations $\widetilde{G}^{(p)}$ and $\widetilde{W}^{(p)}$ are computed by discarding nonzeros with magnitudes less than a prescribed drop tolerance, and an approximate update matrix $\widetilde{T}^{(p)}$ is computed by $\widetilde{T}^{(p)} = \widetilde{W}^{(p)} \widetilde{G}^{(p)}$. Then, if the $p$-th processor belongs to the processor group $g_S$, to compute its local portion of an approximate Schur complement, it gathers the corresponding rows of $\widetilde{T}^{(q)}$ from all the processors and explicitly computes $\widehat{S}^{(p)} = A_{22}^{(p)} - \sum_q \widetilde{T}^{(q)}(i_p : (i_{p+1} - 1), :)$, where the $p$-th processor owns the $i_p$-th through $(i_{p+1} - 1)$-th row of $A_{22}$; i.e., $A_{22}^{(p)} = A_{22}(i_p : (i_{p+1} - 1), :)$. To further reduce the costs, small nonzeros are discarded from $\widehat{S}^{(p)}$ to form its approximation $\widetilde{S}^{(p)}$. Prior to discarding the small nonzeros, we preprocess $\widehat{S}^{(p)}$ to enhance numerical stability by permuting large nonzeros to the diagonal. This preprocessing is performed in a distributed fashion; namely, the $p$-th processor uses an existing serial code MC64 [4] and computes the permutation of its local matrix that corresponds to the $p$-th diagonal block of $\widetilde{S}$. The off-diagonal blocks are permuted accordingly. This distributed preprocessing technique enhances the numerical stability without forming the global approximate Schur complement $\widehat{S}$ on each processor. See [14] for more details on the preprocessing technique.

We now describe several techniques to enhance the performance of computing the approximate Schur complement $\widetilde{S}^{(p)}$.

### 2.1 Sparse triangular solution with sparse right-hand-side columns

In the current version of SuperLU_DIST, the LU factors $L_\ell$ and $U_\ell$ are stored in a 2D block-cyclic format based on the supernodal structure of $L_\ell$ (see Fig. 1(b)). The RHS columns are assumed to be dense, and are distributed by block rows conforming to the supernodal partition. Since the communication and computation patterns of the triangular solutions do not change between the RHS columns, a symbolic triangular solution subroutine is invoked once to compute static communication and computation schedules. Then, the triangular systems are solved by a series of scheduled block operations with the supernodal blocks.

Our first performance-enhancing technique is to exploit the sparsity of $E_\ell$ and $F_\ell^T$ when solving the triangular systems to form $G^{(p)}$ and $W^{(p)}$, respectively. For this, we modified the symbolic triangular solution subroutine of SuperLU_DIST so that only non-empty messages are sent and only computations with non-empty blocks are performed. Since the sparsity pattern of each column of $E_\ell$ or $F_\ell$ is different, this subroutine is invoked for each triangular solution with each column. This symbolic subroutine sets up the communication and computation schedules with respect to the supernodal blocks, which are typically not dense. Because of the supernodal structure of LU factors, when the fill occurs in the solution vector, it occurs all the way to the boundary of the supernodes. Hence, during numerical solution, we keep track of the first nonzero in each supernodal block of the RHS column. Then, the block operations are performed only for the elements below the first nonzero location so that the operations with explicit zeros are eliminated. We have observed that exploiting the sparsity of the RHS columns leads to an order-of-magnitude speedup in computing $W^{(p)}$ and $G^{(p)}$.

There are typically tens to hundreds of thousands of columns in $E_\ell$. Hence, it could be costly to perform the triangular solution one column at a time. Further optimization can be achieved by grouping $E_\ell$ into blocks of multiple columns and solving one block at a time. There are several advantages with blocking: 1) the symbolic solution only needs to be computed per block, 2) fewer messages need to be sent to compute $W^{(p)}$, and 3) the data locality to access the LU factors may be improved. During numerical solution with the multiple RHS columns, we keep track of the first nonzero within each supernodal block of the multiple columns (see Fig. 1(b)). The disadvantage is that we need to pad explicit zeros so that these columns have the same nonzero pattern. The padded zeros occur between the first nonzero position of the multiple columns and that of the individual columns. Hence, blocking introduces a trade-off between the data locality and the number of unwanted padded zeros. Specifically, data locality may be improved by increasing the block size; however, this increases the number of padded zeros. In the special case in which the block size is set to be 1, there are no operations with explicit zero operands, but only a small amount of locality is available. For our test matrix **tdr455k**, the advantages outweighted the disadvantages, and the average and maximum speedups of 5.7 and 7.4, respectively, were achieved by blocking with our defalut block size of 50 and using one preocessor per domain.

To reduce the number of padded zeros introduced by blocking, we employ the following technique: We first permute the rows of $E_\ell$ according to a postorder of the elimination tree of $D_\ell$. Then, the columns of $E_\ell$ are permuted in the descending order of the row indices of their first nonzeros. The columns of $F_\ell^T$ are similarly permuted. One reason why this ordering reduces the number of padded zero is as follows: When a column has the first nonzero at the location corresponding to the $i$-th node of the elimination tree, then according to the Gilbert's path theorem [6], this first nonzero will generate the fill in the solution vector at the positions corresponding to the nodes on the path from the $i$-th node to the root of the elimination tree. After the RHS columns are sorted based on the postorder of their first nonzero row indices, the paths of the adjacent columns are likely to have their starting nodes close together, and the large parts of the paths overlap in the elimination tree. Hence, the solution vectors in the same column block are likely to have fill at similar locations, reducing the number of padded zeros. Furthermore, during the triangular solution, only the columns of the $L$-factor corresponding to the nodes on the paths are accessed. Hence, postordering the RHS columns also improves the data locality to access the $L$-factor. For our test matrix **tdr455k**, average and maximum speedups of about 1.3 and 1.6 were achieved using this postordering technique and one processor per domain. Similar topological orderings have been used for a sparse triangular solution with multiple sparse RHS columns [12] and for computing elements of the inverse of a sparse matrix [2]. We have also introduced another ordering technique using a hypergraph model to maximize the similarity of the sparsity patterns among the solution vectors in a column block [15].

### 2.2 Intra-processor load balance

We have developed a technique to improve the intra-processor load balance to compute the sparse matrix-matrix multiplication $\widetilde{T}^{(p)} = \widetilde{W}^{(p)}\widetilde{G}^{(p)}$. This is done by distributing the rows of $\widetilde{W}^{(p)}$ and $\widetilde{G}^{(p)}$ so that each processor in the same processor group $g_\ell$ owns a similar number of nonzeros. Specifically, Fig. 2 shows the pseudocode to compute $\widetilde{G}^{(p)}$, where the $p$-th processor belongs to the processor group $g_\ell$, $\widehat{E}_\ell$ contains the non-empty columns of $E_\ell$, $\widehat{E}^{(p)}$ is the rows of $\widehat{E}_\ell$ stored by the $p$-th processor, $n_c$ is the number of columns of $\widehat{E}_\ell$, $\beta$ is the column block size, and $n_b$ is the number of blocks (i.e., $n_b = \lfloor \frac{n_c}{\beta} \rfloor$). At Step 1.b of the pseudocode, the consecutive rows of $\widehat{E}^{(p)}$ are distributed among all the processors (see Fig. 1(a)), but the solution vector $X^{(p)}$ is distributed into block rows conforming to the supernodal partition and only among the diagonal processors (see Fig. 1(b)). After each triangular solution with a block of RHS columns, the diagonal processor compresses each column of $X^{(p)}$ into $\widetilde{X}^{(p)}$, excluding the explicitly padded zeros and discarding small nonzeros (Step 1.c). Then, $\widetilde{X}^{(p)}$ is incrementaly stored in $Y^{(p)}$ using the CCS format (Step 1.d). Once all the solution blocks are computed, $Y^{(p)}$ is redistributed among all the processors in $g_\ell$ so that each processor owns consecutive rows of the solution vectors and roughly the same number of nonzeros (Step 2). Note that the remaining columns of the solution vectors are computed separately (Step 3). This is because the data

1. Compute the solution vectors for the 1-st through $(n_b\beta)$-th columns
$Y^{(p)} := [\,]$
`for` $k := 1, \ldots, n_b$ `do`
  a. Extract the next right-hand-side block,
    $B^{(p)} := \widehat{E}^{(p)}(:,\ ((k-1)\beta+1) : (k\beta))$
  b. Compute the sparse triangular solution,
    $X^{(p)} \leftarrow L_\ell^{-1} B$
  c. Sparsify the solution vectors,
    $\widetilde{X}^{(p)} \leftarrow X^{(p)}$
  d. Store the solution vectors,
    $Y^{(p)} \leftarrow [Y^{(p)} \widetilde{X}^{(p)}]$
`end for`
2. Distribute $Y^{(p)}$ from the diagonal processors to all the processors
    $\widetilde{G}^{(p)} \leftarrow Y^{(p)}$
3. Compute the remaining solution vectors
    a. $B^{(p)} := \widehat{E}^{(p)}(:,\ (n_b\beta+1) : n_c)$
    b. $X^{(p)} \leftarrow L_\ell^{-1} B$
    c. $\widetilde{G}^{(p)} \leftarrow [\widetilde{G}^{(p)} \widetilde{X}^{(p)}]$

**Fig. 2.** Pseudocode to compute $G^{(p)}$.

structure for the triangular solution inside SuperLU_DIST must be reinitialized when the block size changes, and the redistribution of $Y^{(p)}$ into $\widetilde{G}^{(p)}$ needs to be peformed before the block size changes. These remaining colums of $\widetilde{G}^{(p)}$ and the solution vectors $\widetilde{W}^{(p)}$ of the upper triangular system are distributed into the format that has been set up to load balance the first $(n_b\beta)$ columns of $\widetilde{G}^{(p)}$. In our numerical experiments using four processors for each of the 16 interior subdomains of **tdr455k**, without this load balancing technique, some processors had only a negligible amount of work to compute the matrix-matrix multiplication, and the load imbalance as measured by the computation time was an up to five order of magnitude difference. With the technique described here, the load imbalance became less than a factor of two. As a result, this technique reduced the time to compute the Schur complement by a factor of 2.6 and the total solution time by a factor of 1.7.

### 2.3 Inter-processor load balance

For the computation of the approximate Schur complement, all the processors first compute their local update matrices $\widetilde{T}^{(p)} = \widetilde{W}^{(p)}\widetilde{G}^{(p)}$. Then, the corresponding rows of $\widetilde{T}^{(p)}$ are sent to the processors in the processor group $g_S$, which solve the Schur complement system. In comparison to the original system, the Schur complement system is typically much smaller in dimension. Hence, only a subset of processors is used to solve the Schur complement system. In this section, we study two techniques to improve the inter-processor load balance: a strategy to accommodate this *all-to-subset* communication of $\widetilde{T}^{(p)}$ and one to select processors to solve the Schur complement system.

To accomodate the all-to-subset communication of $\widetilde{T}^{(p)}$, an MPI all-to-all communication subroutine can be used. Even though this simpifies the implementation, there are two shortcomings with this approach. First, there are often large variations in the sizes of the subdomains $D_\ell$ and in the sparsity of the interfaces $E_\ell$ and $F_\ell$. Even though the intra-processor load balance is improved by the technique described in Section 2.2, this leads to poor load balance among the inter-processor groups to compute $\widetilde{T}^{(p)}$. Since the global all-to-all communication imposes synchronization among all the processors, this load imbalance forces some processors to be idle while waiting for the other processors to complete the computation of $\widetilde{T}^{(q)}$. Second, the all-to-all communication requires a large buffer to recieve the corresponding rows of $\widetilde{T}^{(q)}$ from all the processors.

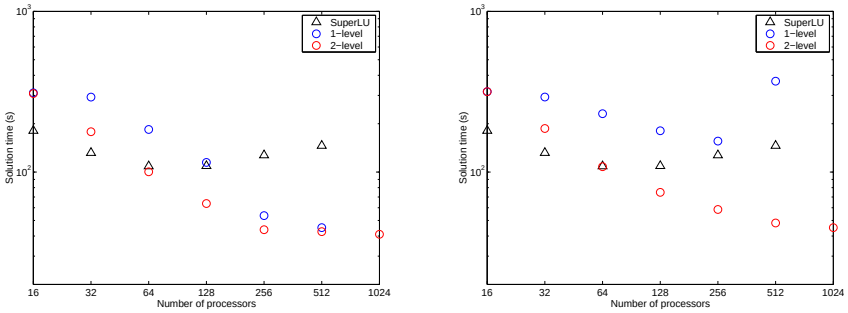
To mitigate these problems, we designed an asynchronous point-to-point communication protocol to transfer $\widetilde{T}^{(p)}$. Fig. 3 shows the pseudocode of this communication protocol, where $p$ is the ID of this processor, $m_s$ is the number of processors to which $\widetilde{T}^{(p)}$ needs to be sent, $m_r$ is the number of processors from which nonempty $T^{(q)}(p) = \widetilde{T}^{(q)}(i_p : (i_{p+1} - 1),\ :)$ will be received, ISend($*$, $q$, $t$) and IRecv($*$, $q$, $t$) indicate nonblocking send operation to and receive operation from the $q$-th processor with a tag $t$, Wait($*$, $t$) blocks until the nonblocking receive operation with a tag $t$ is processed, and Recv($*$, $q$, $t$) is a blocking receive operation from the processor $q$, where ANY_SOURCE can be used in the place of $q$ to indicate a receive operation from any source, and SENDER_SOURCE indicates the ID of the sender processor. Furthermore, Allocate($U(k)$, size($k$)) allocates the buffer $U(k)$ to receive $T^{(q)}(p)$ in the CSR format, where size($k$) is the maximum number of nonzeros, and Free($U(k)$) frees the buffer $U(k)$. In our implementation, all the matrix operations are performed by taking advantage of their sparsity. For efficient memory management, rows of $\widetilde{T}^{(p)}$ are freed, once they are received, and we alternatly reuse two receiving buffers, $U(0)$ and $U(1)$, whose sizes are stored in size(0) and size(1), respectively.

Our point-to-point communication is desinged to overlap the computation of $\widetilde{T}^{(q)}$ with the communication and summation of $T^{(q)}(p)$. Hence, there is a greater chance of overlap when the processors with less work to compute $\widetilde{T}^{(q)}$ are assigned to the Schur complement. To achieve this, we assign processors from relatively small interior subdomains to the Schur complement. This not only increases the potential for the overlap, but also improves the overall load balance of memory requirement. Furthermore, on Line 1 of the pseudocode, we set $q_{\pi_{m_r}} = p$ so that communication of $T^{(q)}(p)$ from other processors can be overlapped with the computation and summation of the local $T^{(p)}(p)$. For our test matrix **tdr455k**, the size of the Schur complement was only about 0.5% of the total dimension, and the summation of the update matrices required only a neglibile amout of time (i.e., less than one second out of 120 seconds spent to compute $\widetilde{S}^{(p)}$ on 16 processors). As a result, this point-to-point communication did not reduce the computation time significantly. However, for other problems with large interfaces, the solution time may be reduced more significantly using this point-to-point communication.

```
    /* All the processors perform Lines 1 through 10. */
1.  for q = q_{π_1}, q_{π_2}, ..., q_{π_{m_s}} do
2.      T(q) := W^{(p)}(i_q : (i_{q+1} − 1), :) * G^{(p)} /* compute the rows of T to be sent to the q-th processor */
3.      if(q == p) then /* sending to itself */
4.          size(0) = nnz(T(q))
5.          U(0) := T(q)
6.      else
7.          ISend(nnz(T(q), q, t_0)
8.          if( nnz(T(q)) > 0 ) then Isend(T(q), q, t_1)
9.      end if
10. end if

    /* The processor subset responsible for solving the Schur complement perform Lines 11 through 75. */
11. if( p ∈ g_S ) then
12.     p_0 := p
13.     n_m := 1 /* number of received messages */
14.     while( size(0) == 0 and n_m < m_r ) do /* find the size of the first nonempty message */
15.         Recv(size(0), ANY_SOURCE, t_0)
16.         n_m := n_m + 1
17.         p_0 := SENDER_SOURCE
18.     end do
19.     if( size(0) > 0 ) then
20.         n_k := 1 /* number of received nonempty messages */
21.         if( p_0 ≠ p ) then
22.             Allocate( U(0), size(0) )
23.             IRecv(U(0), p_0, t_1)
24.         end if
25.     end if
26.     if( n_m < m_r ) then
27.         IRecv(size(1), ANY_SOURCE, t_0) /* request the size of the next message */
28.     end if
29.     S := A_{22}^{(p)} /* initialize the Schur complement */
30.     if( n_m < m_r ) then /* find the size of the second nonempty message */
31.         Wait(size(1), t_0)
32.         n_m := n_m + 1
33.         while( size(1) == 0 and n_m < m_r ) do
34.             Recv(size(1), ANY_SOURCE, t_0)
35.             n_m := n_m + 1
36.         end do
37.     end if
38.     p_1 := SENDER_SOURCE
39.     if( size(1) > 0 ) then
40.         Allocate(U(1), size(1)) /* allocate the buffer for the second message */
41.         IRecv(U(1), p_1, t_1)
42.         if( n_m < m_r ) then /* request the size of the next message */
43.             IRecv(size(0), ANY_SOURCE, t_0)
44.         end if
45.     end if
46.     if( p_0 ≠ p ) then
47.         Wait(U(0), t_1)
48.     end if
49.     S := S − U(0) /* sparse update of the local Shur complement */
50.     while( n_m ≤ m_r ) do /* while there is more nonempty messages */
51.         if( n_m < m_r ) then
52.             Wait(nnz, t_0)
53.             n_m := n_m + 1
54.             while( nnz == 0 and n_m < m_r ) do
55.                 Recv(nnz, ANY_SOURCE, t_0)
56.                 n_m := n_m + 1
57.             end do
58.             p_1 := SENDER_SOURCE
59.             if( nnz > 0 ) then
60.                 if( nnz > size(mod(n_k − 1, 2))) then
61.                   Free(U(mod(n_k − 1, 2)))
62.                   Allocate( U(mod(n_k − 1, 2)), nnz )
63.                   size(mod(n_k − 1, 2)) = nnz
64.                 end if
65.                 IRecv(U(mod(n_k − 1, 2)), p_1, t_1)
66.             end if
67.             if( n_m < m_r ) then
68.                 IRecv(nnz, ANY_SOURCE, t_0)
69.             end if
70.         end if
71.         Wait(U(mod(n_k, 2)), t_1)
72.         S := S − U(mod(n_k, 2))
73.         n_k := n_k + 1
74.     end do
75. end if
```

**Fig. 3.** Pseudocode of the point-to-point communication to form $S$.




(a) $(\sigma_1, \sigma_2) = (10^{-6}, 10^{-5})$ (b) $(\sigma_1, \sigma_2) = (10^{-5}, 10^{-4})$

**Fig. 4.** Solution times required by SuperLU_DIST and our hybrid solver on **tdr455k**.

More importantly, though, in comparison to the all-to-all communication, this point-to-point communiation can significantly reduce memory requirements. This is because the point-to-point communication alternately uses only two receiving buffers, instead of a large buffer to hold all the receiving updates in the all-to-all communication. When four processors were used on each of the 16 interior subdomains of **tdr455k**, the point-to-point communication reduced the total number of nonzero elements in the receiving buffers by a factor of about 3.1 on average, and up to 6.3. Notice that the size of the all-to-all communication buffer may increase with the number of processors. As a result when 64 processors were used on each of the 16 interior subdomains of **tdr455k**, the buffer size was reduced by a factor of 55.9 on average, and up to 97.0 using the point-to-point communication.

## 3 Parallel performance

We now present parallel performance of our hybrid solver. For these numerical experiments, the Schur complement systems were solved using a preconditioned Krylov method of PETSc. The initial aproximation to the solution is the zero vector, and the computed solution was considered to have converged when the $\ell_2$-norm of the initial residual was reduced by at least twelve orders of magnitude. This is the solution accuracy required in the actual simulations.

Fig. 4(a) compares the total solution times required by SuperLU_DIST and our hybrid solver to solve the **tdr455k** linear system. The hybrid solver used a drop tolerance $\sigma_1$ to enforce the sparsity of $\widetilde{E}$ and $\widetilde{F}$, $\sigma_2$ to enforce the sparsity of $\widetilde{S}$, and unrestarted GMRES for solving the Schur complement system. With the one-level parallel approach of our hybrid solver, the number of interior subdomains was set to be equal to the total number of processors. With the two-level approach, the number of interior subdomains was fixed to be 16, and the processors were evenly distributed among the interior subdomains. The figure shows that the solution time with our hybrid solver scaled better than that with Su-

perLU_DIST. The numerical results have also shown that the required memory scales better with our hybrid solver. For example, with 16 subdomains and the small drop tolerances $(\sigma_1, \sigma_2) = (10^{-6}, 10^{-5})$, the total number of nonzeros in the preconditioner was reduced only by 10% using our hybrid solver. However, on 256 processors, the maximum memory required by a processor was about 2.3GB using our two-level approach, while SuperLU_DIST still required about 3.2GB.

Fig. 4(a) also shows that with the small drop tolerances, the scaling of the one-level and two-level approaches were similar. This is because the number of GMRES iterations was nearly independent of the number of interior subdomains, and GMRES converged within 20 iterations even when more interior subdomains were needed for the one-level approach to use more processors. For comparison, we have tested a state-of-the-art hybrid solver HIPS [5], which implements the one-level parallelization. HIPS computes the preconditioner for solving the Schur complement system based on an ILU factorization of $\widetilde{S}$, where the sparsity of the preconditioner is enforced based on both the numerical values and locations of nonzeros. Specifically, fill is allowed only between separators adjacent to the same subdomain. As a result, the computation of the preconditioner scales to a large number of processors. Unfortunately, this preconditioner was not effective for **tdr455k**; specifically, it required 151 iterations on 16 processors, and it failed to converge within $1,000$ iterations on 32 processors even though the drop tolerances were set to be zero. Moreover, even when HIPS converged, our solver solved the linear system faster.

Larger drop tolerances reduce the memory required by our hybrid solver. For example, in Fig. 4(b), less memory was needed since the drop tolerances were increased by an order of magnitude from those in Fig. 4(a). Specifically, in Fig. 4(a), about 15% of the nonozeros were discarded from the matrices $E$ and $F$, and about 50% of the nonzeros were discarded from the Schur complement $\widehat{S}$. The respective percentages of the discarded nonzeros for Fig. 4(b) were about 30% and 75%. Unfortunately, with large drop tolerances, the number of GMRES iterations may increase as more interior subdomains are generated. For example, in Fig. 4(b), the number of iterations increased from 32 to 290 when the number of interior subdomains increased from 16 to 256. As a result, with the one-level approach, the solution time did not scale. On the other hand, the two-level approach demonstrated more robust performance since the processor count can be increased while fixing the size of the Schur complement. These results illustrate the advantage of the two-level approach. For our target of solving larger problems, a more strict sparsity constraint may be needed. Therefore, Fig. 4(b) represents the more practical behavior of our hybrid solver.

Finally, in Table 1, we show the timing results of our hybrid solver to solve another linear system **tdr8cavity** of dimension $17,799,228$ with $727,163,784$ nonzeros. For these experiments, we fixed the number of subdomains to be 64, which resulted in the average subdomain dimension of $277,220$ and the Schur complement dimension of $57,150$. In the table, $n_p$, $n_{g\ell}$, and $n_{gS}$ are the numbers of processors used to solve the entire system, to factorize a subdomain, and to solve the Schur complement system, respectively; itrs is the number of

| $n_p$ | $n_{g_\ell}$ | $n_{g_S}$ | itrs | Time LU($D_\ell$) | Comp($\widetilde{S}$) | LU($\widetilde{S}$) | Solve | Total | Speedup |
|---|---|---|---|---|---|---|---|---|---|
| 64 | $1 \times 1$ | $8 \times 8$ | 9 | 334.0 | 305.5 | 4.4 | 37.7 | 681.7 | |
| 128 | $1 \times 2$ | $8 \times 8$ | 9 | 176.0 | 165.0 | 4.3 | 31.8 | 377.4 | 1.81 |
| 256 | $2 \times 2$ | $8 \times 8$ | 9 | 91.3 | 80.1 | 4.8 | 24.2 | 200.6 | 1.89 |
| 512 | $2 \times 4$ | $8 \times 8$ | 8 | 52.5 | 47.9 | 4.6 | 19.3 | 125.5 | 1.59 |
| 1024 | $4 \times 4$ | $8 \times 8$ | 8 | 32.7 | 28.1 | 4.7 | 20.7 | 86.4 | 1.47 |
| 2048 | $4 \times 8$ | $8 \times 8$ | 10 | 22.9 | 20.9 | 4.9 | 22.6 | 71.6 | 1.19 |
| 4096 | $8 \times 8$ | $8 \times 8$ | 8 | 20.3 | 13.1 | 4.5 | 25.2 | 63.5 | 1.13 |

**Table 1.** Solution times required by our hybrid solver on **tdr8cavity**.

BiCGSTAB iterations required for the solution convergence; LU($D_\ell$), Comp($\widetilde{S}$), LU($\widetilde{S}$), Solve, and Total are the times in seconds for factorizing the $\ell$-th subdomain, computing $\widetilde{S}$, factorizing $\widetilde{S}$, computing the solution vector, and solving the entire system, respectively; and Speedup is the speedup gained by increasing the processor count by a factor of two. The table shows that the first two phases of the hybrid solver LU($D_\ell$) and Comp($\widetilde{S}$) scaled to thousands of processors. Since $n_{g_S}$ is fixed to be 64, LU($\widetilde{S}$) stayed the same. Even though the last phase Solve did not scale, the total solution time scaled to thousands of processors. We were not able to use SuperLU_DIST to solve this linear system due to the excessive communication required for the triangular solution.

## 4 Conclusion

We presented several techniques to improve the robustness and scalability of our parallel hybrid solver based on the Schur complement method. Numerical results have shown that our solver is numerically more robust than another hybrid solver, HIPS, while its solution time scales better than that of the direct solver SuperLU_DIST. We are studying other techniques to further improve the performance of our hybrid solver such as improving initial partition, assigning different numbers of processors to subdomains, distributing the Schur complement based on the separator boundaries, and other parallel preconditioning techniques for the Schur complement system.

## Acknowledgements

We gratefully thank Bora Uçar at CNRS, François-Henry Rouet at ENSEEIHT-IRIT, and Esmond Ng at LBNL for helpful discussions. We also thank Lie-Quan Lee at SLAC for providing the test matrix. This research was supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. DOE under Contract No. DE-AC02-05CH11231.

## References


1. Community Petascale Project for Accelerator Science and Simulation (ComPASS). https://compass.fnal.gov.
2. P. Amestoy, F.-H. Rouet, and B. Uçar. On computing arbitary entries of the inverse of a matrix. In *SIAM Workshop on Combinatorial Scientific Computing (CSC09), Monterey, 2009*, 2009.
3. R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems: Building blocks for the iterative methods*. SIAM, Philadelphia, PA, 1994.
4. 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.
5. J. Gaidamour and P. Henon. HIPS: a parallel hybrid direct/iterative solver based on a schur complement. In *Proc. PMAA*, 2008.
6. J. Gilbert. Predicting structure in sparse matrix computations. *SIAM J. Matrix Analysis and Applications*, 15:62–79, 1994.
7. L. Giraud, A. Haidar, and L. T. Watson. Parallel scalability study of hybrid preconditioners in three dimensions. *Parallel Computing*, 34:363–379, 2008.
8. 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/.
9. 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.
10. 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.
11. Mathematics and Computer Science Division, Argonne National Laboratory. The portable, extensible, toolkit for scientific computation (PETSc). www.mcs.anl.gov/petsc.
12. Tz. Slavova. *Parallel triangular solution in an out-of-core multifrontal approach for solving large sparse linear systems*. PhD thesis, Institut National Polytechnique de Toulouse, Toulouse, France, 2009.
13. B. Smith, P. Bjorstad, and W. Gropp. *Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, New York, 1996.
14. I. Yamazaki, X. Li, and E. Ng. Preconditioning schur complement systems of highly-indefinite linear systems for a parallel hybrid solver. In *the proceedings of the international conference on preconditioning techniques for scientific and industrial applications*, 2009.
15. I. Yamazaki, X. Li, E. Ng, F.-H. Rouet, and B. Uçar. Combinatorial issues in a paralle hybrid linear solver. In *SIAM Workshop on Parallel Processing (PP10), Seatle*, 2010.