2012 IEEE 26th International Parallel and Distributed Processing Symposium

ShyLU: A Hybrid-Hybrid Solver for Multicore Platforms

Sivasankaran Rajamanickam[1], Erik G. Boman[1], and Michael A. Heroux[1],
E-mail:{srajama@sandia.gov, egboman@sandia.gov and maherou@sandia.gov}
[1] Sandia National Laboratories.

***Abstract*—With the ubiquity of multicore processors, it is crucial that solvers adapt to the hierarchical structure of modern architectures. We present ShyLU, a "hybrid-hybrid" solver for general sparse linear systems that is hybrid in two ways: First, it combines direct and iterative methods. The iterative part is based on approximate Schur complements where we compute the approximate Schur complement using a value-based dropping strategy or structure-based probing strategy.**

Second, the solver uses two levels of parallelism via hybrid programming (MPI+threads). ShyLU is useful both in shared-memory environments and on large parallel computers with distributed memory. In the latter case, it should be used as a *subdomain solver*. We argue that with the increasing complexity of compute nodes, it is important to exploit multiple levels of parallelism even within a single compute node.

We show the robustness of ShyLU against other algebraic preconditioners. ShyLU scales well up to 384 cores for a given problem size. We also study the MPI-only performance of ShyLU against a hybrid implementation and conclude that on present multicore nodes MPI-only implementation is better. However, for future multicore machines (96 or more cores) hybrid/ hierarchical algorithms and implementations are important for sustained performance.

I. Introduction

The general trend in computer architectures is towards hierarchical designs with increasing node level parallelism. In order to scale well in these architectures, applications need hybrid/hierarchical algorithms for the performance critical components. The solution of sparse linear systems is an important kernel in scientific computing. A diverse set of algorithms is used to solve linear systems, from direct solvers to iterative solvers. A common strategy for solving large linear systems on large parallel computers, is to first employ domain decomposition (e.g., additive Schwarz) on the matrix to break it into subproblems that can then be solved in parallel on each core or on each compute node. Typically, applications run one MPI process per core, and one subdomain per MPI process. A drawback of domain decomposition solvers or preconditioners is that the number of iterations to solve the linear system will increase with the number of subdomains. With the rapid increase in the number of cores one subdomain per core is no longer a viable approach. However, one subdomain per node is reasonable since the recent and future increases in parallelism are and will be primarily on the node. Thus, an increasingly important problem is to solve linear systems in parallel on the compute node. Our hybrid-hybrid method is "hybrid" in two ways: the solver combines direct and iterative algorithms, and uses MPI and threads in a hybrid programming approach.

In order to be scalable and robust it is important for solvers and preconditioners to use the hybrid approach in both meanings of the word. The hybrid programming model ensures good scalability within the node and the hybrid algorithm ensures robustness of the solver. A sparse direct solver is very robust and the BLAS based implementations are capable of performing near the peak performance of desktop systems for specific problems. However, they have high memory requirements and poor scalability in distributed memory systems. An iterative solver, while highly scalable and customizable for problem specific parameters, is not as robust as a direct solver. A hybrid preconditioner can be conceptually viewed as a middle ground between an incomplete factorization and a direct solver.

A. ShyLU Scope and Our Contributions

Current iterative solvers and preconditioners (such as ML [1] or Hypre [2]) need a node level strategy in order to scale well in large petascale systems where the degree of parallelism is extremely high. The natural way to overcome this limitation is to introduce a new level of parallelism in the solver. We envision three levels of parallelism: At the top, there is the inter-node parallelism (typically implemented with MPI) and two levels of parallelism (MPI+threads) on the node. Section V describes the various options to use a parallel node level preconditioner with three levels of parallelism.

The node level is where the degree of parallelism is rapidly increasing and this is the scope of ShyLU. Previous Schur/hybrid solvers all solve the global problem, competing with multigrid. This approach instead complements the multigrid methods and focuses on the scalability on the node. In this paper, we concentrate on the node level parallelism and leave the integration into the third (inter-node) level as future work.

Our first contribution is a new scalable hybrid sparse solver, ShyLU (Scalable Hybrid LU, pronounced Shy-Loo), based on the Schur complement framework. ShyLU is based on Trilinos [3] and also intended to become a Trilinos package. It is designed to be a "black box" algebraic solver that can be used on a wide range of problems. Furthermore,

1530-2075 2012

DOI 10.1109/IPDPS.2012.64

it is suitable both as a solver on a single-node multicore workstation and as a *subdomain solver* on a compute node of a petaflop system. Our target is computers with many CPU-like cores, not GPUs.

Second, we revisit every step of the Schur complement framework to exploit node level parallelism and to improve the robustness of ShyLU as a preconditioner. ShyLU uses a new probing technique that exploits recent improvements in parallel coloring algorithms to get a better approximation of the Schur complement.

Third, we try to answer the question: "When will the hybrid implementation of a complex algorithm be better than a pure MPI-based implementation?". We use ShyLU as our target "application" as a complex algorithm like linear solver, with a pure MPI-based implementation and a hybrid MPI+Threads implementation should be able to provide a reasonable answer to this question. The answer is dependent on algorithms, future changes in architectures, problem sizes and various other factors. We address this question for our specific algorithm and our target applications.

B. Previous work

Many good parallel solver libraries have been developed over the last decades; for example, PETSc [4], Hypre [2], and Trilinos [3]. These were mainly designed for solving large distributed systems over many processors. ShyLU's focus is on solving medium-sized systems on a single compute node. This may be a subproblem within a larger parallel context. Some parallel sparse direct solvers (e.g., SuperLU-MT [5], [6] or Pardiso [7]) have shown good performance in shared-memory environments, while distributed-memory solvers (for example MUMPS [8], [9]) have limited scalability. Pastix [10] is an interesting sparse direct solver because it uses hybrid parallel programming with both MPI and threads. However, any direct solver will require lots of memory due to fill-in and they are not ready to handle the O(100) to O(1000) expected increase in the node concurrency (in their present form at least). To reduce memory requirements, incomplete factorizations is a natural choice. There are only few parallel codes available for incomplete factorizations in modern architectures. (e.g., [11], [12])

Recently, there has been much interest in *hybrid* solvers that combine features of both direct and iterative methods. Typically, they partially factor a matrix using direct methods and use iterative methods on the remaining Schur complement. Parallel codes of this type include HIPS [13], MaPhys [14], and PDSLin [15]. ShyLU is similar to these solvers in a conceptual way that all these solvers fall into the broad Schur complement framework described in section II. This framework is not new, and similar methods were already described in Saad *et al.* [16]. However, each of these solvers, including ShyLU, is different in the choices made at different steps within the Schur complement framework. Furthermore, we are not aware of any code that is hybrid in both the mathematical and in the parallel programming sense. In contrast to the other hybrid solvers our target is a multicore node. See section IV for how these solvers differ from ShyLU in the different steps of the Schur complement framework.

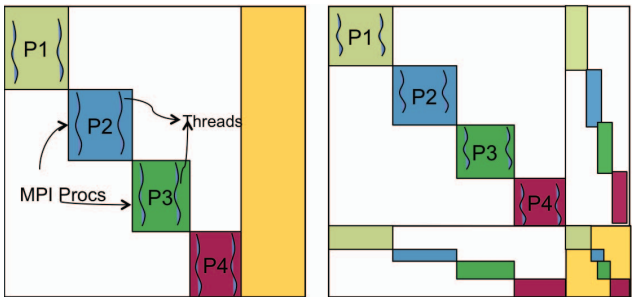


Figure 1. Partitioning and reordering of a (a) nonsymmetric matrix and (b) symmetric matrix.

II. Schur Complement Framework

This section describes the framework to solve linear systems based on the Schur complement approach. There has been lot of work done in this area; see for example, Saad [17, Ch.14] and the references therein.

A. Schur complement formulation

Let $Ax = b$ be the system of interest. Suppose A has the form

$$A = \begin{pmatrix} D & C \\ R & G \end{pmatrix}, \tag{1}$$

where D and G are square and D is non-singular. The Schur complement after elimination of the top row is $S = G - R * D^{-1}C$. Solving $Ax = b$ then consists of solving

$$\begin{pmatrix} D & C \\ R & G \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{2}$$

by solving

1) $Dz = b_1$.
2) $Sx_2 = b_2 - Rz$.
3) $Dx_1 = b_1 - Cx_2$.

The algorithms that use this formulation to solve the linear system in an iterative method or a hybrid method essentially use three basic steps. We like to call this the Schur complement framework:

Partitioning: The key idea is to permute A to get a D that is easy to factor. Typically, D is diagonal, banded or block diagonal and can be solved quickly using direct methods. As the focus is on parallel computing, we choose D to be block diagonal in our implementation. Then R corresponds to a set of coupling rows and C is a set of coupling columns. See Figure 1 for two such partitioning. The symmetric case in Figure 1(b) is identical to the Schur complement formulation. The nonsymmetric case in Figure 1(a) can be

solved using the same Schur complement formulation even though it appears different.

Sparse Approximation of S: Once D is factored (either exactly or inexactly), the crux of the Schur complement approach is to solve for S iteratively. There are several advantages to this approach. First, S is typically much smaller than A. Second, S is generally better conditioned than A. However, S is typically dense making it expensive to compute and store. All algorithms compute a sparse approximation of S ($\bar{S}$) either to be used as a preconditioner for an implicit S or for an inexact solve.

Fast inexact solution with S: Once $\bar{S}$ is known there are multiple options to solve S and then to solve for A. For example, the algorithms can choose to solve D exactly and just iterate on the Schur complement system (S) using $\bar{S}$ as a preconditioner and solve exactly for the full linear system, or use an incomplete factorization for D and then use iterative methods for solving both S and A, using an inner-outer iteration. The options for preconditioners to S vary as well.

Different hybrid solvers choose different options in the above three steps, but they follow this framework.

B. Hybrid Solver vs. Preconditioner

Hybrid solvers typically solve for D exactly using a sparse direct solver. This also provides an exact operator for S. Note that S does not need to be formed explicitly but the action of S on a vector can be computed by using the identity $S = G - R * D^{-1}C$. This can save significant memory, since S can be fairly dense.

We take a slightly different perspective: We design an *inexact* solver that may be used as a preconditioner for A where A corresponds to a subdomain problem within a larger domain decomposition framework. As a preconditioner, we no longer need to solve for D exactly. Also, we don't need to form S exactly. If we solve for S using an iterative method, we get an inner-outer iteration. The inner iteration is internal to ShyLU, while the outer iteration is done by the user. When the inner iteration runs for a variable number of iterations, it is best to use a flexible Krylov method (e.g., FGMRES) in the outer iteration.

C. Preconditioner Design

As is usual with preconditioners (see e.g., IFPACK [18]), we split the preconditioner into three phases: (i) Initialize, (ii) Compute, and (iii) Solve. Initialize (Algorithm 1) only depends on the sparsity pattern of A, so may be reused for a sequence of matrices. Compute (Algorithm 2) recomputes the numeric factorization and $\bar{S}$ if any matrix entry has changed in value. Solve (Algorithm 3) approximately solves $Ax = b$ for a right-hand side b.

Algorithm 1 Initialize

Require: A is a square matrix
Require: k is the desired number of parts (blocks)
  Partition A into k parts.
Ensure: Let D be block diagonal with k blocks.
Ensure: Let R be the row border and C the column border.

Algorithm 2 Compute

Require: Initialize has been called.
  Factor D.
  Compute $\bar{S} \approx G - R * D^{-1}C$.

III. Narrow Separators vs Wide Separators

The framework in Section II depends on finding separators to partition the matrix into the bordered form. The traditional way to find this separator is to represent the matrix as graph or hypergraph and find a partitioning of the graph or hypergraph. Let (V_1, V_2, P) be a partition of the vertices V in a graph $G(V, E)$. P is a *separator* if there is no edge (v, w) such that $v \in V_1$ and $w \in V_2$. Separator P is called a *wide separator* if any path from V_1 to V_2 contains at least two vertices in P. A separator that is not wide is called a *narrow separator*. Note that the edge separator as computed by many of the partitioning packages corresponds to a wide vertex separator.

Wide separators were originally used as part of ordering techniques for sparse Gaussian elimination [19]. The intended application at that time was sparse direct factorization [20]. We revisit this comparison with respect to hybrid solvers here.

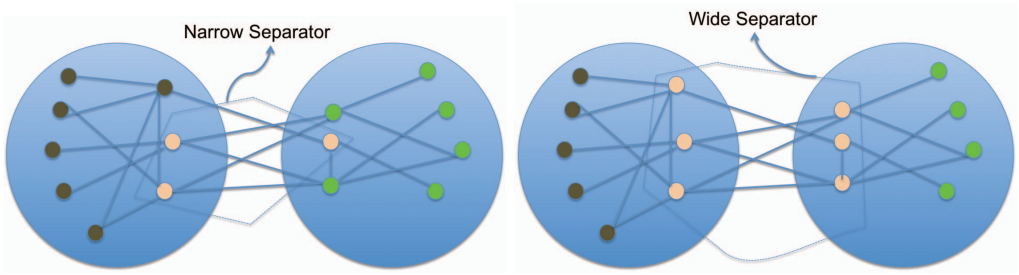
From the perspective of the graph of the matrix, the narrow separator is shown in Figure 2(a). The corresponding wide separator is shown in Figure 2(b). The doubly bordered block diagonal form of a matrix A when we use a narrow separator is shown below (for two parts).

$$A_{narrow} = \begin{pmatrix} \hat{D}_{11} & 0 & \hat{C}_{11} & \hat{C}_{12} \\ 0 & \hat{D}_{22} & \hat{C}_{21} & \hat{C}_{22} \\ \hat{R}_{11} & \hat{R}_{12} & \hat{G}_{11} & \hat{G}_{12} \\ \hat{R}_{21} & \hat{R}_{22} & \hat{G}_{21} & \hat{G}_{22} \end{pmatrix} \quad (3)$$

All the $\hat{R}_{ij}$ blocks and $\hat{C}_{ij}$ blocks can have nonzeros in them. As a result, every block in the Schur complement might require communication when we compute it. For example, while using the matrix from the narrow separator A_{narrow} to compute the $\hat{S}_{11}$ block of the Schur complement we do

Algorithm 3 Solve

Require: Compute has been called.
  Solve $Dz = b_1$.
  Solve either $Sx_2 = b_2 - Rz$ or $\bar{S}x_2 = b_2 - Rz$.
  Solve $Dx_1 = b_1 - Cx_2$.



(a) Narrow Separator. (b) Wide Separator.

Figure 2. Wide Separator and Narrow Separator of a graph G.

$$\hat{S}_{11} = \hat{G}_{11} - \hat{R}_{11} * \hat{D}_1^{-1} * \hat{C}_{11} + \hat{R}_{12} * \hat{D}_2^{-1} * \hat{C}_{21} \quad (4)$$

Computing the Schur complement in the above form is expensive due to the communication involved. However, the doubly bordered block diagonal form for two parts when we use a wide separator has more structure to it as shown below.

$$A_{wide} = \begin{pmatrix} D_{11} & 0 & C_{11} & 0 \\ 0 & D_{22} & 0 & C_{22} \\ R_{11} & 0 & G_{11} & G_{12} \\ 0 & R_{22} & G_{21} & G_{22} \end{pmatrix} \quad (5)$$

Although this block partition is similar to Equation (3), the matrix blocks will in general have different sizes since a wide separator is larger than the corresponding narrow separator. Consider that rows of D_{ii} are the interior vertices in part i and the rows in R_{ij} are boundary vertices in part i then we observe that all blocks R_{ij} and C_{ij} will be equal to zero when $i \neq j$. This follows from the definition of the wide separator.

As R and C are block diagonal matrices, we can compute the Schur complement without any communication. For example, to compute the S_{11} block of the Schur complement of A_{wide} we do

$$S_{11} = G_{11} - R_{11} * D_1^{-1} * C_{11} \quad (6)$$

Thus computing S in the wide separator case is fully parallel. The off-diagonal blocks of the Schur complement are equal to the off-diagonal blocks of G. However, the wide separator can be as much as two times the size of the narrow separator. This results in a larger Schur complement system to be solved when using the wide separator. When the separator was considered as a serial bottleneck (when they were originally designed for direct solvers) there was a good argument to use the narrow separators. However, in hybrid solvers, we solve the Schur complement system in parallel as well. As a result, while the bigger Schur complement system leads to increased solve time, the much faster setup due to increased parallelism offsets the small increase in solve time. All the experiments in the rest of this work use wide separators for increased parallelism. Note that the Schur complement using the wide separator is similar to the local Schur complement [17].

The edge separator from graph and hypergraph partitioning gives a wide (vertex) separator by simply taking the boundary vertices. Although this is a good approach for most problems, we observed that on problems with a few dense rows or columns, the narrow separator approach works better. Therefore, ShyLU also has the option to use narrow separators. While some partitioners can compute narrow vertex separators directly, we implemented a simple heuristic to compute a (narrow) vertex separator from the edge separator so we can use any partitioner.

IV. Implementation

This section describes the implementation details of ShyLU for each step of the Schur complement framework. ShyLU uses an MPI and threads hybrid programming model even within the node. Notice that in the Schur complement framework the partitioning and reordering is purely algebraic. This reordering exposes one level of data parallelism. ShyLU uses MPI tasks to solve for each D_i and the Schur complement. A further opportunity for parallelism, is within the diagonal blocks D_i. where a threaded direct solver, for example, Pardiso [7] or SuperLU-MT [5], [6], is used to factor each block D_i. The assumption here is multithreaded direct solvers (or potentially incomplete factorizations in the future) can scale well within a uniform memory access (UMA) region, where all cores have equal (fast) access to a shared memory region. Using MPI between UMA regions mitigates the problems with data placement and non-uniform memory accesses and also allows us to run across nodes, if desired.

ShyLU uses the Epetra package in Trilinos with MPI for the matrix A. When combined with a multithreaded solver for the subproblems, we have a hybrid MPI-threads solver. This is a very flexible design that allows us to experiment with hybrid programming and the trade-offs of MPI vs. threads. In the one extreme case, the solver could partition and use MPI for all the cores and use no threads. The other extreme case is to only use the multithreaded direct

solver. We expect the best performance to lie somewhere in between. A reasonable choice is to partition for the number of sockets or UMA regions. We will study this in Section VI.

The framework consists of *partitioning*, *sparse approximation of the Schur complement*, and *fast, inexact (or exact) solution of the Schur complement*. The first two steps only have to be done once in the setup phase.

A. Partitioning

ShyLU uses graph or hypergraph partitioning to find a D that has a block structure and is suitable for parallel solution. To exploit locality (on the node), we partition A into k parts, where $k > 1$ may be chosen to correspond to number of cores, sockets, or UMA regions. The partitioning induces the following block structure:

$$A = \begin{pmatrix} D & C \\ R & G \end{pmatrix}, \tag{7}$$

where D again has a block structure. As shown in Figure 1 there are two cases. In the symmetric case, ShyLU uses a symmetric permutation PAP^T to get a doubly bordered block form. In this case, $D = diag(D_1, \ldots, D_k)$ is a block diagonal matrix, R is a row border, and C is a column border. In the nonsymmetric case, there is no symmetry to preserve so we allow nonsymmetric permutations. Therefore, instead we find PAQ with a singly bordered block diagonal form (Figure 1(b)). A difficulty here is that the "diagonal" blocks are rectangular, but we can factor square submatrices of full rank and form R, the row border after the factorization. ShyLU can use a direct factorization that can factor square subblocks of rectangular matrices. There are no multithreaded direct solvers that can handle this case now. We focus on the structurally symmetric case here. In our experiments for unsymmetric matrices, we apply the permutation in a symmetric manner to form the DBBD form.

Several variations of graph partitioning can be used to obtain block bordered structure. Traditional graph partitioning attempts to keep the parts of equal size while minimizing the edge cut. We will consider the edge separator as our separator. The other hybrid solvers we know (see Section I) use some form of graph partitioning. Hypergraph partitioning is a generalization of graph partitioning that is also well suited for our problem because it can minimize the border size directly. Also, it naturally handles nonsymmetric problems, while graph partitioning requires symmetry. ShyLU uses hypergraph partitioning in both the symmetric and nonsymmetric cases. ShyLU uses the Zoltan/PHG partitioner [21] and computes the wide separators and narrow separators from a distributed matrix.

B. Diagonal block solver

The blocks D_i are relatively small and will typically be solved on a small number of cores, say in one UMA region. Either exact or incomplete factorization can be used. We choose to use a sparse direct solver. All the results in Section VI use Pardiso [7] from Intel MKL, which is a multithreaded solver. Since the direct solver typically will run within a single UMA region, it does not need to be NUMA-aware. ShyLU uses the Amesos package [22] in Trilinos which is a common interface to multiple direct solvers. This enables ShyLU to switch between any direct solver supported by the Amesos package. The other hybrid solvers mentioned in Section I all use a serial direct solver in this step.

C. Approximations to the Schur Complement

The exact Schur complement is $S = G - R * D^{-1}C$. In general, S can be quite dense and is too expensive to store. There are two ways around this: First, we can use S implicitly as an operator without ever forming S. Second, we can form and store a sparse approximation $\bar{S} \approx S$. As we will see, both approaches are useful.

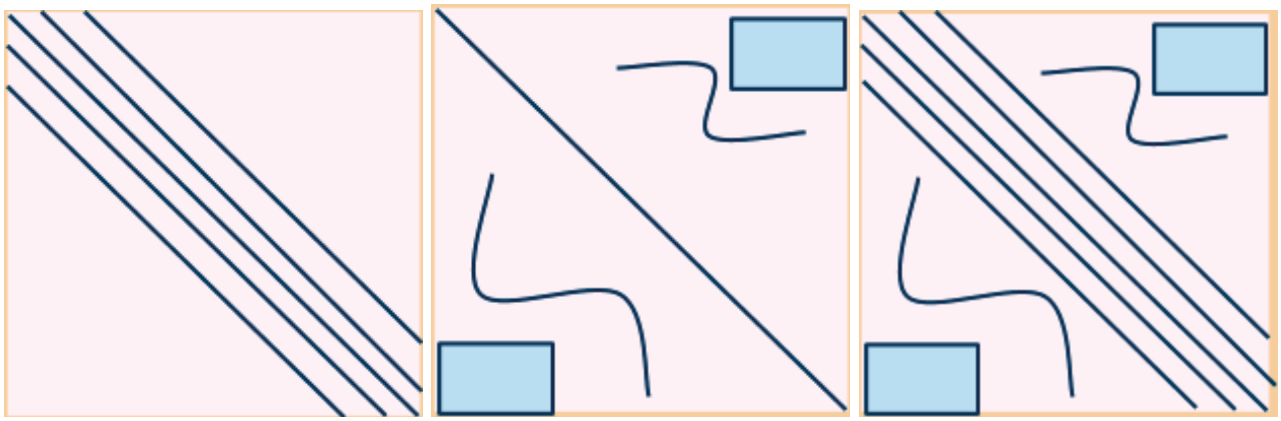
The Schur complement itself has a block structure

$$S = \begin{pmatrix} S_{11} & S_{12} & \ldots & S_{1k} \\ S_{21} & S_{22} & \ldots & S_{2k} \\ \vdots & \vdots & & \vdots \\ S_{k1} & S_{k2} & \ldots & S_{kk} \end{pmatrix} \tag{8}$$

where it is known that the diagonal blocks S_{ii} are usually quite dense but the off-diagonal blocks are mostly sparse [17]. Note that the local Schur complements S_{ii} can be computed locally by $S_{ii} = G_{ii} - R_i * D_{ii}^{-1} C_i$. A popular choice is therefore to use the local Schur complements as a block diagonal approximation. As we use wide separators as discussed above all the fill is in our local Schur complement and all the offdiagonal blocks have the same sparsity pattern as the corresponding G_{ij}. To save storage, the local Schur complements themselves need to be sparsified [23], [15].

We investigate two different ways to form $\bar{S} \approx S$: *Dropping* and *Probing*. Both methods attempt to form a sparser version of S while preserving the main properties of S.

1) Dropping (value-based): With *dropping* we only keep the largest (in magnitude) entries of S. This is a common strategy and was also used in HIPS and PDSLin. Symmetric dropping is used in [24]. When forming $S = G - R * D^{-1}C$, we simply drop entries less than a given threshold. We use a relative threshold, dropping entries that are smaller relative to the large entries. Since S can be quite dense, we only form a few columns at a time and immediately sparsify. Note we do not drop entries based on $U^{-1}C$ or $R * U^{-1}$ where L and U are the LU factors of D, as in HIPS or PDSLin. Since our dropping is based on the actual entries in S, we believe our approximation $\bar{S}$ is more robust. However, even with the parallelism at the MPI level, computing local S_{ii} to drop the entries is itself expensive. Instead of trying to parallelize the sparse triangular solve to compute the $R_{ii} * D_{ii}^{-1} * C_{ii}$, we compute the columns of S in chunks and exploit the



(a) Structure of typical banded probing for $\bar{S}$, B. (b) Structure of G submatrix. (c) Structure of $\bar{S} = B \cup G$ for ShyLU's probing.

Figure 3. A sketch of the pattern used for probing in ShyLU.

parallelism available from using the multiple right hand sides in a sparse triangular solve.

2) Probing (structure-based): Since dropping may be expensive in some cases, ShyLU can also use *probing*. Probing was developed to approximate interfaces in domain decomposition [25], which is also a Schur complement. In probing, we select the sparsity pattern of $\bar{S} \approx S$ first. Given a sparsity pattern we probe the Schur complement operator S for the entries given in the sparsity pattern instead of computing the entire Schur complement and drop the entries. It is possible to probe the operator efficiently by coloring the sparsity pattern of $\bar{S}$ and computing a set of probing vectors, V, based on the coloring of $\bar{S}$. V is a n-by-k matrix where k is the number of colors and n is the dimension of S. The number of colors in the coloring problem corresponds to the number of probing vectors needed. The coloring of the pattern computes the orthogonal columns in $\bar{S}$, so we apply the operator to only the few vectors that are needed.

Finally, we apply $S = G - RD^{-1}C$ as an operator to the probing vectors V to obtain SV, which then gives us the numerical values for $\bar{S}$ in a packed format. We need to unpack the entries to compute $\bar{S}$. We refer to Chan et al. [25] for the probing algorithm.

Generally, the sparser $\bar{S}$, the fewer the number of probing vectors needed. Choosing the sparsity pattern of $\bar{S}$ can be tricky. For PDE problems where the values in S decay away from the diagonal, a band matrix is often used [25]. We show how probing works by using a tridiagonal approximation of the Schur complement as an example. Coloring the pattern of a tridiagonal matrix results in three colors. Then the three probing vectors corresponding to the three colors are

$$V = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ \vdots & \vdots & \vdots \end{pmatrix} \tag{9}$$

with $V_{ij} = 1$ if column i had color j. $S * V$ gives us the entries corresponding to the tridiagonal of the Schur complement in a packed format. However, a purely banded approximation will lose any entries in S (and G) that are outside the bandwidth. To strengthen our preconditioner, we include the pattern of G in the probing pattern, which is simple to do as G is known a priori. To summarize, the pattern of $\bar{S}$ in ShyLU's probing is pattern of $B \cup G$, where B is a banded matrix.

Figure 3 shows a sketch of how ShyLU's probing technique compares to traditional probing. Figure 3(a) shows the structure of a typical banded probing assuming we are looking for a few diagonals. To this structure, our algorithm also includes the structure of G from the reordered matrix (Figure 3(b)), for probing. As result the structure for the probing is as shown in Figure 3(c). The idea behind adding G to the structure of $\bar{S}$ is that any entry that is originally part of G is important in $\bar{S}$ as well. Experimental results showed a bandwidth of 5% seems to work well for most problems.

Probing for a band structure is straight-forward since the probing vectors are trivial to compute. In our approach, we need to use graph coloring on the structure of $\bar{S}$ (which in our case is $B \cup G$) to find the probing vectors. We use the Prober in Isorropia package of Trilinos which in turn uses the distributed graph coloring algorithm [26] in Zoltan. Note that all the steps in probing - the coloring, applying the probing vector and extracting $\bar{S}$, are done in parallel. Probing

for a complex structure is computationally expensive, but we save quite a lot in memory as the storage required for the Schur complement is the size of G with a few diagonals. However, for problems where the above discussed structure of $\bar{S}$ is not sufficient, the more expensive dropping strategy can be used.

D. Solving for the Schur Complement

As in the steps before, there are several options for solving for the Schur complement as well. Recall that we have formed $\bar{S}$, a sparse approximation to S. A popular approach in hybrid methods is to solve the Schur complement system iteratively using $\bar{S}$ as a preconditioner. In each iteration, we have to apply S, which can be done implicitly without ever forming S explicitly. Note that implicit S requires sparse triangular solves for D in every iteration. We call this the *exact Schur complement* solver.

As we only need an inexact solve as a preconditioner, it is also possible to solve $\bar{S}$ instead of solving S. Now, even $\bar{S}$ is large enough that it should be solved in parallel. We solve for $\bar{S}$ iteratively using yet another approximation $\tilde{S} \approx \bar{S}$ as a preconditioner for $\bar{S}$. It should be easy to solve for $\tilde{S}$ in parallel. In practice, $\tilde{S}$ can be quite simple, for example, diagonal (Jacobi) or block diagonal (block Jacobi). The main difference from the exact method is that we do not use the Schur complement operator even for matrix vector multiplies in the inner iteration. Instead we use $\bar{S}$. We call this approach the *inexact Schur complement* solver. ShyLU can do both the exact and inexact solve for the Schur complement. We compare the robustness of both these approaches in Section VI.

Once the preconditioner ($\bar{S}$ or $\tilde{S}$) and the operator for our solve (either an implicit S or $\bar{S}$) is decided there are two options for the solver. If D is solved exactly and an implicit S is the operator it is sufficient to iterate over S (as in [13]) and not on A. Instead any scheme that uses an inexact solve for D or an iterative solve on $\bar{S}$ (instead of S) or both implies an inner-outer iterative method for the overall system. as it is required to iterate on A. It is because of this reason ShyLU uses an inner-outer iteration, where the inner iteration is only on the Schur complement part. The inner iteration (over S or $\bar{S}$) is internal in the solver and invisible to the user, while the outer iteration (over A) is controlled by the user. We expect a trade-off between the inner and outer iterations. That is, if we iterate over S we need few outer iterations while if we iterate on $\bar{S}$ we may need more outer iterations but fewer inner iterations.

By default, we do 30 inner iterations or to an accuracy of 10^{-10} whichever comes first.

E. Parallelism

Our implementation of the Schur complement framework is parallel in all three steps. We use Zoltan's parallel hypergraph partitioning to partition and reorder the problem. The block diagonal solvers are multithreaded in addition to the parallelism from the MPI level. We use parallel coloring from Zoltan to find orthogonal columns in the structure of $\bar{S}$ and sparse matrix vector multiplication to do the probing. The Schur complement solve uses our parallel iterative solvers for solving for S or $\bar{S}$ which use a multithreaded matrix vector multiplication.

V. Parallel Node Level Preconditioning

ShyLU is a hybrid solver designed for the multicore node and uses MPI and threads even within the node. This is different from other approaches where MPI + threads model spans across the entire system, not just the node, and there is only one MPI processes per node. We see two problems with one MPI process per node approach:

1) Parts of the applications other than the solver have fewer MPI processes limiting their scalability.
2) Scaling the multithreaded solvers on the compute nodes with NUMA accesses is a harder problem.

Instead, we believe one MPI process per socket or UMA region is a more practical approach for scalability at least in the near term. ShyLU also decouples the idea that one subdomain corresponds to one MPI process. An MPI based subdomain solver like ShyLU allows the subdomain, in a domain decomposition method, to span several MPI processes.

Nothing prevents us from using ShyLU across the entire system as it is based on MPI, however the separator size (and thus the Schur complement) will grow with the number of parts. While using a multithreaded solver for the D blocks limit the size of the Schur complement to a certain extent (by partitioning for fewer MPI processes) we recommend using a domain decomposition method with little communication (e.g., additive Schwarz) at the global level, and ShyLU on the subdomains. Such a scheme will exploit three levels of parallelism, where the top level requires little communication while the lower levels require more and more communication. In essence, we adapt the solver algorithm to the machine architecture. We believe this is a good design for future exascale computers that will be hierarchical in structure.

The Schur complement framework and the MPI+threads programming model also allow ShyLU be fully flexible in terms of how applications use it. We envision ShyLU to be used by the applications in three different modes:

1) When applications start one MPI process per UMA region in the near future, a simple *MPI_Comm_Split()* can map all the MPI processes *in a node* to ShyLU's MPI processes. A subdomain will be defined as one per node.
2) When applications start one MPI process per node, additive Schwarz will use a threads-only ShyLU.
3) Applications that now run one MPI process per core remain that way, the additive Schwarz preconditioner

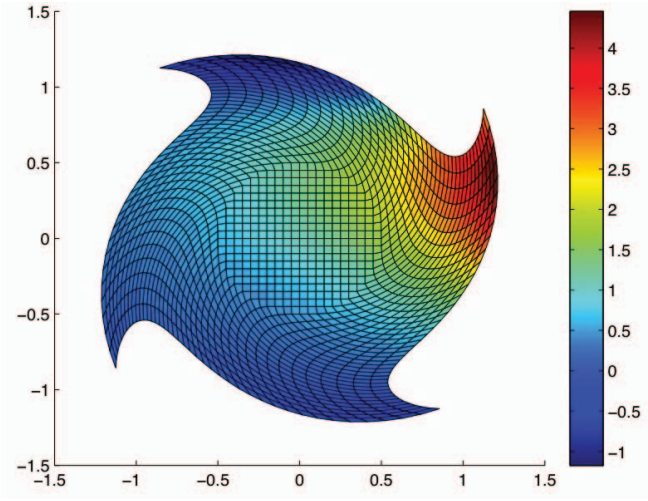


Figure 4. Cross-section of 3D unstructured mesh on an irregular domain.

(which will use ShyLU on subdomains) can define the subdomains as one per node and transform the matrix for ShyLU. ShyLU will not be able to use additional threads in this case.

Thus the MPI+threads programming model in ShyLU's design helps make the application migration to the multicore systems smooth depending on how the applications want to migrate.

VI. Results

We perform three different set of experiments. First, we wish to test robustness of ShyLU compared to other common algebraic preconditioners. Second, we study ShyLU performance on multicore platforms, and in particular the trade-off between MPI-only vs. hybrid models. This study will also look at performance of ShyLU while doing strong scaling. Third, we study weak scaling of ShyLU on both 2D and 3D problems.

A. Experimental setup

We have implemented ShyLU in C++ within the Trilinos [3] framework. We leverage several Trilinos packages, in particular:

1) *Epetra* for matrix and vector data structures and kernels.
2) *Isorropia* and *Zoltan* for matrix partitioning and probing.
3) *AztecOO* and *Belos* for iterative solves (GMRES).

In addition to the Trilinos packages we also use PARDISO as our multithreaded direct solver. We use two test platforms. The first is Hopper, a Cray XE6 at NERSC. Hopper has 6392 nodes, each with two twelve-core AMD MagnyCours processors running at 2.1 GHz. Thus, each node has 24 cores and is a reasonable prototype for future multicore nodes. Furthermore, the Hopper system is attractive to us because of its NUMA properties. The 24 cores in a node are in fact four six-core UMA sets. We use Hopper for all our strong scaling and weak scaling studies. Our other test platform is an eight-core (dual-socket quad-core) Linux workstation that represents current multicore systems. We use this workstation for our robustness experiments.

All experimental results show the product of inner and outer iterations that will be seen by the user of ShyLU. When there are many tunable parameters there are two ways to do experiments. Either choose the best parameters for each problem, or always use the same parameters for a given solver on the entire test set. All the experiments in this section use solver specific parameters and there is no tuning for a particular problem, since this is how users typically use software. For probing we add 5% of diagonals to the structure of G. For dropping, our relative dropping threshold is 10^{-3}. We use 30 inner iterations or 10^{-7} relative residual whichever comes first and 500 outer iterations or 10^{-7} relative residual whichever comes first. This is fully utilized when we use the inexact Schur complement.

B. Robustness

We validate the different methods in ShyLU by comparing it to incomplete factorizations and the HIPS [13] hybrid solver. We use three different variations of ShyLU, approximations based on dropping and probing with the exact Schur complement solver and approximations based on dropping and the inexact Schur complement solver. All three approaches have tunable parameters that can be difficult to choose. We used a fixed dropping/probing tolerance in all our tests. The relative threshold for dropping is 10^{-3}. Similarly, we tested HIPS preconditioner with fixed settings same as ShyLU. Our goal is to demonstrate the robustness of ShyLU compared to one other hybrid solver that is commonly used today. The tests also include ILU with one level of fill. The number of iterations of the three methods should not be compared directly, since the fill and work differ in the various cases. The methods can be made comparable by tuning the knobs. However, we have used the parameters as they are used in our various applications.

We chose nine sparse matrices from a variety of application areas, taken from the University of Florida sparse matrix collection [27]. We added one test matrix from a Sandia application, TC_N_360K. The results are shown in Table I. We see that the dropping approximation with the exact Schur complement is the most robust approach among all the approaches, in the sense it has fewer failures. This has been observed in the past by others as well. The dropping with exact Schur complement is better or very close to HIPS in terms of the number of iterations. Generally, the drop-tolerance version requires fewer iterations (though not necessarily less run time) than the probing version.

A dash indicates that GMRES failed to converge to the desired tolerance within 500 iterations. Note that the circuit

Matrix Name	N	Symmetry	ShyLU Dropping Exact Schur	ShyLU Probing Exact Schur	ShyLU Dropping Inexact Schur	HIPS	ILU
venkat50	62.4K	Unsymmetric	12	76	-	8	374
TC_N_360K	360K	Symmetric	32	82	17	19	203
Pres_Poisson	14.8K	Symmetric	14	26	14	11	-
FEM_3D_thermal2	147K	Unsymmetric	3	6	3	3	20
bodyy5	18K	Symmetric	3	5	3	3	120
Lourakis_bundle1	10K	Symmetric	7	18	7	10	26
af_shell3	504K	Symmetric	50	-	39	29	-
Hamm/bcircuit	68.9K	Unsymmetric	6	6	4	42	-
Freescale/transient	178.8K	Unsymmetric	88	-	-	440	-
Sandia/ASIC_680ks	682K	Unsymmetric	4	34	2	2	57

Table I
COMPARISON OF NUMBER OF ITERATIONS OF SHYLU DROPPING AND PROBING WITH EXACT SOLVE, SHYLU DROPPING WITH INEXACT SOLVE, HIPS AND ILU(1) FOR MATRICES FROM UF COLLECTION. A DASH INDICATES NO CONVERGENCE.

matrices *bcircuit* and *transient* are difficult for HIPS, but ShyLU does well in these matrices. The matrix *af_shell3* has been called horror matrix in the past for posing difficulty to preconditioners. Both ShyLU and HIPS could solve this problem easily. ILU(1) does poorly on all the problems when compared with the two hybrid preconditioners. However, that is expected given the fact that ILU(1) uses considerably less memory and has little information in the preconditioner itself.

We further observe that the dropping version is more robust than the probing version, as it solved all 10 test problems while the probing version failed in 2 out of 10 cases. The inexact approach, as one would expect, is not as robust as the exact approach with dropping. However, it converged faster when it worked. We have also verified that ShyLU takes less memory than a direct solver UMFPACK [28]. The amount of memory used by ShyLU depends on the size of the Schur complement, dropping criterion, and the solver used for the block diagonals.

C. MPI+threads vs MPI performance

We implemented ShyLU with MPI at the top level. Each MPI process corresponds to a diagonal block D_i. We used multi-threaded MKL-Pardiso as the solver for the D_i blocks. We wish to study the trade-off between MPI-only and hybrid models. Our design allows us to run any combination of MPI processes and threads. Note that when we vary the number of MPI processes, we also change the number of D_i blocks so the preconditioner changes as well. Thus, what we observe in the performance is a combined effect of changes in the solver algorithm and in the programming model (MPI+threads).

Initially, we ran on one node of Hopper (24 cores). However, the number of cores on a node is increasing rapidly. We want to predict performance on future multicore platforms with hundreds of cores. We simulate this by running ShyLU on several compute nodes. Since we use MPI even within the node, ShyLU also works across nodes. We expect future multicore platforms to be hierarchical with highly non-uniform memory access and running across the nodes will reasonably simulate future systems. We expect

	MPI Processes x Number of Threads in each node			
Nodes (Cores)	4x6	6x4	12x2	24x1
1 (24)	19.6 (79)	17.9 (91)	11.8 (122)	8.3 (144)
2 (48)	14.6 (115)	12.3 (122)	7.0 (144)	6.9 (196)
4 (96)	8.3 (122)	7.2 (144)	5.3 (196)	6.0 (227)
8 (192)	6.4(176)	5.2(196)	**3.9**(227)	6.9 (332)

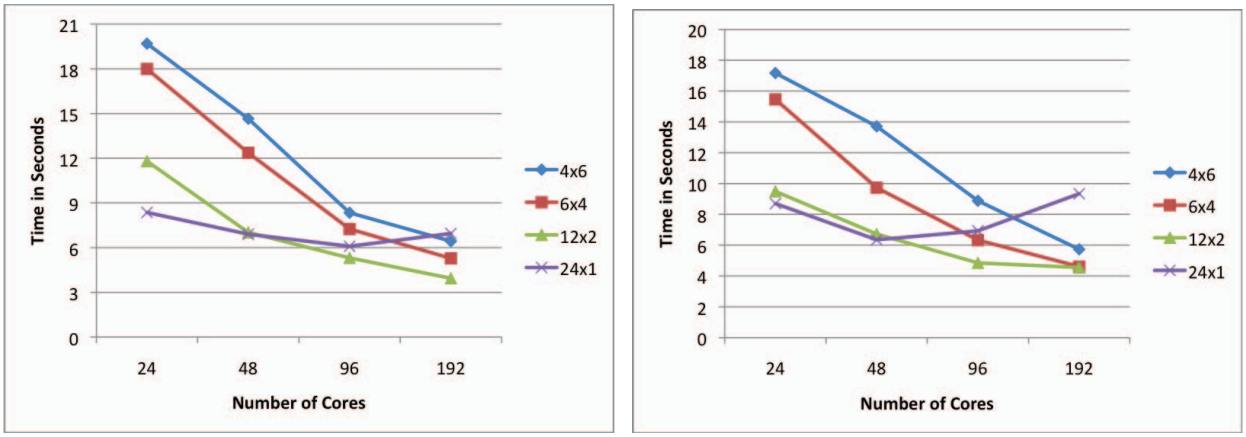
Table II
STRONG SCALING AND HYBRID VS MPI-ONLY PERFORMANCE: SOLVE TIME IN SECONDS (#ITERATIONS) FOR SHYLU DROPPING METHOD TO SOLVE A LINEAR SYSTEM OF SIZE 360K.

the performance figures for more than 24 cores to get better. However, we do not know how much MPI and threads performance are going to get better. Assuming they improve at the same rate, we compare the performance of the MPI-only code with hybrid code to understand the possible differences in future systems.

For this experiment we used a 3D finite element discretization of Poisson's equation on an irregular domain, shown in Figure 4. The matrix dimension was $360K$. We use the drop-tolerance version of ShyLU for our first set of tests. For each node with 24 cores, we tested the following configurations of MPI processes $\times$ threads: 4×6, 6×4, 12×2, and 24×1. The results for run-time and iterations are shown in Table II. More than 6 threads per node is not a recommended configuration for Hopper so those results are not shown in Table II. The solve time is also shown in Figure 5(a).

There are several interesting observations. First, we see that although the number of iterations increase with the number of MPI processes (going across the rows in Table II), the run times may actually decrease. On a single node, we see that the all-MPI version (24x1) is fastest, even though it uses more iterations.

Second, we see that, as we add more nodes, the run times decrease much more rapidly for the hybrid configurations. For example, with four nodes, the 12x2 configuration gives the fastest solve time. This is good news for hybrid methods



(a) Dropping (b) Probing

Figure 5. **Strong Scaling**: ShyLU's dropping and probing methods for a matrix of size 360K. Solve Time shown for MPI tasks x Threads.

	MPI Processes x Number of Threads in each node			
Nodes (Cores)	4x6	6x4	12x2	24x1
1 (24)	17.1 (64)	15.4 (70)	9.4 (83)	8.7 (97)
2 (48)	13.7 (76)	9.7 (83)	6.7 (97)	6.3 (114)
4 (96)	8.8 (98)	6.3 (97)	4.8 (114)	6.9 (148)
8 (192)	5.7 (111)	4.6 (114)	**4.5**(148)	9.3 (218)

Table III
STRONG SCALING AND HYBRID VS MPI-ONLY PERFORMANCE: SOLVE TIME IN SECONDS (#ITERATIONS) FOR SHYLU PROBING METHOD TO SOLVE A LINEAR SYSTEM OF SIZE 360K.

	MPI Processes x Number of Threads in each node		
Nodes (Cores)	6x4	12x2	24x1
2 (48)	25.1(90)	15.0(104)	11.5(115)
4 (96)	13.8(104)	9.2(115)	6.2(130)
8 (192)	9.5(115)	5.7(130)	5.1(139)
16 (384)	5.1(130)	**3.2**(139)	4.8(177)

Table IV
STRONG SCALING AND HYBRID VS MPI-ONLY PERFORMANCE: SOLVE TIME IN SECONDS (#ITERATIONS) FOR SHYLU DROPPING METHOD TO SOLVE A LINEAR SYSTEM OF SIZE 720K.

as they can take advantage of the node level concurrency. We believe that this is mainly due to the subproblems getting smaller. We conjecture that using more threads would be helpful on smaller problem sizes per core.

To understand how the algorithmic choices affect our strong scaling results we also repeated the experiment with the same 360Kx360K problem with probing. The time for the solve is shown in Figure 5(b). The results are almost identical to the dropping method. The MPI only version started performing poorly at 96 cores. At 192 cores any MPI+thread combination beats MPI-only implementation. However, MPI only is still the best choice at 24 cores. The number of iterations for this experiment is shown in Table III. We can see that the number of iterations for the probing method is better than the dropping method.

To verify our conjecture, that the size of the problem in each subdomain is important for hybrid performance, we repeated the experiment, this time with a larger problem 720Kx720K. We did not use the 4x6 configuration as it was the slowest in our previous experiment. The results are shown in Table IV. Note that ShyLU scales well up to 384 cores. Furthermore, we see that the crossover point where MPI+threads beats MPI-only implementation is different for this larger problem (384 cores). The result can be seen clearly in Figure 6 where we compare the 12x2 case against 24x1 for both the problems (360K and 720K). When the problem size per subdomain is about 3500 unknowns the performance is almost the same for all four cases. As the problem size per subdomain gets smaller the hybrid programming model gets better.

A consistent trend in our results is that as the number of cores increase, and the size of the problems get smaller, the hybrid (MPI+threads) solver outperforms the MPI-only based solver.

D. Strong scaling

We can also get strong scaling results by looking at a column at a time at the Tables II – IV. In the 360K problem's dropping case, the 4×6 configuration gives a speedup of 2.3 going from one to four nodes, while the 24×1 only gave a speedup of 1.4. Although the first is quite decent when one takes the communication across nodes into account, one should keep in mind that ShyLU was primarily intended to be a fast solver on a single node. The results are similar when we go to eight nodes (192 cores). The best speedup the hybrid model achieved is 3.4 while MPI-only is able to get a speedup of 1.2 for the dropping method. The 6x4 and 12x2 configurations in the 720K problem size case

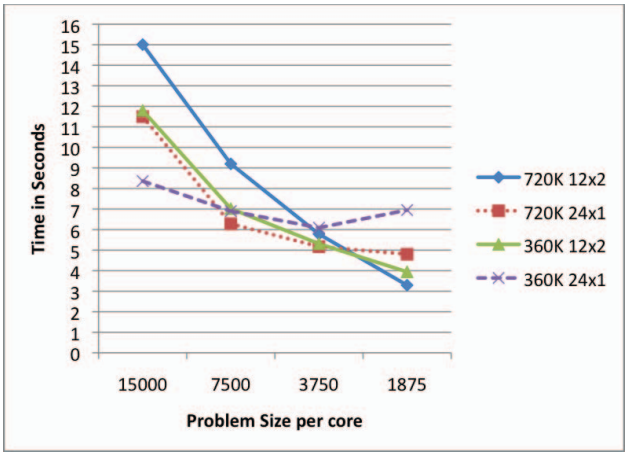


Figure 6. Solve time for MPI-only and MPI+threads implementations for different problem sizes per subdomain.

	MPI Processes x Number of Threads in each node			
Nodes (Problem Size)	4x6	6x4	12x2	24x1
1 (60K)	0.25(10)	0.19(10)	0.35(26)	0.21(11)
2 (120K)	0.31(11)	0.22(10)	0.40(26)	0.61(26)
4 (240K)	0.33(11)	0.67(26)	0.20(12)	0.74(26)
8 (480K)	0.41(11)	0.29(11)	0.60(26)	0.82(26)

Table V
SHYLU (PROBING) WEAK SCALING RESULTS: TIMING IN SECONDS (#ITERATIONS) FOR 2D FINITE ELEMENT PROBLEM.

(Table IV) achieve a speed up of 4.92 and 4.68 going from 48 to 384 cores. MPI-only implementation gained a speedup of 2.39 for this case. Overall, ShyLU is able to scale up to 384 cores reasonably well.

One should also note that the MPI+threads approach has allowed us to reduce the iteration creep that we would expect to see in many precondtioners as the problem size and number of processes increase. For example, in Table II, we can see that the configuration of 8 nodes with 12 MPI processes and 2 threads in each node and the configuration of 4 nodes with 24 MPI processes (1 thread in each process) gives us the same number of iterations – 227. However, the former is using the two threads for better scalability and takes only 65% of the time.

E. Weak scaling

We perform weak scaling experiments on both 2D and 3D problems where we keep the number of degrees of freedom (matrix rows) per core constant. This is not the intended use case for ShyLU (as a subdomain solver, strong scaling is more relevant) but we wish to show that ShyLU also does reasonably well in this setting.

Our 2D test problem is a finite element discretization of an elliptic PDE on a structured grid but with random

	MPI Processes x Number of Threads in each node			
Nodes (Problem Size)	4x6	6x4	12x2	24x1
1 (60K)	0.37(17)	0.31(18)	0.20(22)	0.39(27)
2 (120K)	0.48(20)	0.51(26)	0.30(27)	0.50(30)
4 (240K)	0.82(29)	0.49(25)	0.38(28)	0.44(31)
8 (480K)	0.83(29)	0.66(30)	0.44(30)	0.55(32)

Table VI
SHYLU (DROPPING) WEAK SCALING RESULTS: TIMING IN SECONDS (#ITERATIONS) FOR 2D FINITE ELEMENT PROBLEM.

	MPI Processes x Number of Threads in each node			
Nodes (Problem) (Size)	4x6	6x4	12x2	24x1
1 (90K)	3.0(47)	2.53(54)	1.76(67)	1.37(73)
2 (180K)	4.55(71)	3.93(80)	2.78(95)	2.41(110)
4 (360K)	8.34(122)	7.25(144)	5.31(196)	6.09(227)
8 (720K)	10.30(103)	9.59(115)	5.78(130)	5.17(139)

Table VII
SHYLU (DROPPING) WEAK SCALING RESULTS: TIMING IN SECONDS (#ITERATIONS) FOR THE 3D PROBLEM.

coefficients, generated in Matlab by the command `A = gallery('wathen',nx,ny)`. We vary the number of nodes from one to eight. Again, we designed ShyLU to be run within a node but we want to demonstrate scaling beyond 24 cores, so we run our experiments across multiple nodes.

We see in Tables V– VI that both run time and number of iterations increase slowly with the number of cores (as we go down the columns). The dropping version demonstrates a smooth and predictable behavior, while the probing version has sudden jumps in number of iterations and time. We conjecture that this is because the preconditioner is sensitive to the probing pattern (which is difficult to choose). For the dropping version, the 12x2 configuration with 12 MPI processes and 2 threads each per node is consistently the best.

Our 3D test problem is a finite element discretization of an elliptic PDE on the unstructured grid show in Figure 4. The weak scaling results for this problem are shown in Table VII. We observe that going from 1 to 8 nodes, the number of iterations roughly doubles while the run time roughly triples. Although worse than the optimal $O(n)$ scaling that multigrid methods may be able to achieve, this is much better than the typical $O(n^2)$ operations scaling by general sparse direct solvers. ShyLU's intended usage as a subdomain solver also places more emphasis on *strong scaling than weak scaling*, as the problem size per node is not growing as fast as the node concurrency. We conclude that ShyLU is a good subdomain solver for problems of moderate size and scales quite well up to 384 cores. Thus, it can also be used as a solver/preconditioner in itself on such problems and

platforms.

VII. Future Work

We plan several improvements in ShyLU. Some of these deal with combinatorial issues in the solver algorithm, others are numerical.

First, we wish to further study the partitioning and ordering strategy. In concurrent work [29] we explored the trade-off between load imbalance in the diagonal blocks and the size of the Schur complement. By allowing more imbalance in the diagonal blocks, the partitioner can usually find a smaller block border. We have also observed that the load balance in the system for the inner solve (S) may be poor even though the load balance for the outer problem (A) is good. With current partitioning tools one can balance the interior vertices but not the work in the sparse factorization or solve. Furthermore, it is not sufficient to balance the interior vertices (or factorization work) because ShyLU would require the boundary vertices to be balanced as well as that corresponds to the number of triangular solves and matrix vector multiplies while constructing the Schur complement. We believe this issue poses a partitioning problem with multiple constraints and objectives, and cannot be adequately handled using standard partitioning models.

Second, we intend to extend the code to handle structurally nonsymmetric problems with nonsymmetric permutations. Our current implementation uses symmetric ordering and partitioning, even for nonsymmetric problems. We can use the hypergraph partitioning and permutation to singly bordered block form as shown in Figure 1. However, this requires a multithreaded direct solver that can handle rectangular blocks.

Third, one could study the effect of inexact solves (e.g., with incomplete factorizations) on the diagonal blocks (D_i). This will require the iterative solver to iterate on the entire system, not just the Schur complement. The number of iterations will likely increase, but both the setup and each solve on the diagonal blocks will be faster. This variation would also need less memory.

Fourth, we should test ShyLU on highly ill-conditioned problems, such as indefinite problems and systems from vector PDEs. Although ShyLU is robust on the range of problems we tested here, harder test problems may reveal the need for some algorithmic adjustments.

Finally, we plan to integrate ShyLU as a subdomain solver within a parallel domain decomposition framework. This would comprise a truely hierarchical solver with three different layers of parallelism in the solver.

We remark that none of these issues are specific to ShyLU and many also apply to other hybrid solvers. Discussions with developers of other such solvers have confirmed that they face similar issues. In particular, we believe research on the combinatorial problems above may help advance a whole class of solvers.

VIII. Conclusions

We have introduced a new hybrid-hybrid solver, ShyLU. ShyLU is hybrid both in the mathematical sense (direct and iterative) and in the parallel computing sense (MPI + threads). ShyLU is both a robust linear solver and a flexible framework that allows researchers to experiment with algorithmic options. We introduced and explored several such options: a new probing based Schur complement approximation vs. the traditional dropping strategy, wide vs. narrow separators, and exact vs. inexact solves for the Schur complement system. Performance results show ShyLU can scale well for up to 384 cores in the hybrid mode.

We also studied the question, that given a complex algorithm, with a MPI-only implementation and hybrid (MPI +Threads) implementation, for a fixed set of parameters: Can the hybrid implementation beat the MPI-only implementation? Empirical results on a 24-core MagnyCours node show that it is advantageous to run MPI on the node. This is not surprising since MPI gives good locality and memory affinity. However, we project that for applications and algorithms with smaller problem size per domain, MPI-only works well up to about 48 cores, but for 96 or more cores hybrid methods are faster. The crossover point where the hybrid model beats MPI depends on the problem size per subdomain. We conclude that MPI-only solvers is a good choice for today's multicore architectures. However, considering the fact that the number of cores per node is increasing steadily and memory architectures are changing to favor core-to-core data sharing, hybrid (hierarchical) algorithms and implementations are important for future multicore architectures. We predict multiple levels of parallelism will be essential on future exascale computers.

Acknowledgment

Sandia is a multiprogram laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

The authors thank the Department of Energy's Office of Science and the Advanced Scientific Computing Research (ASCR) office for financial support. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the DOE under Contract No. DE-AC02-05CH11231.

References

[1] M. Gee, C. Siefert, J. Hu, R. Tuminaro, and M. Sala, "ML 5.0 smoothed aggregation user's guide," Sandia National Laboratories, Tech. Rep. SAND2006-2649, 2006.

[2] R. D. Falgout and U. M. Yang, "Hypre: A library of high performance preconditioners," *Lecture Notes in Computer Science*, vol. 2331, pp. 632–??, 2002.

[3] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the Trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

[4] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object-oriented numerical software libraries*. Birkhauser Boston Inc., 1997, pp. 163–202.

[5] J. W. Demmel, J. R. Gilbert, and X. S. Li, "An asynchronous parallel supernodal algorithm for sparse gaussian elimination," *SIAM J. Matrix Anal. Appl.*, vol. 20, pp. 915–952, July 1999.

[6] X. S. Li, "An overview of SuperLU: Algorithms, implementation, and user interface," *ACM Trans. Math. Softw.*, vol. 31, pp. 302–325, September 2005.

[7] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," *Journal of Future Generation Computer Systems*, vol. 20, no. 3, pp. 475–487, 2004.

[8] P. Amestoy, I. Duff, J.-Y. L'Excellent, and J. Koster, *MUltifrontal Massively Parallel Solver (MUMPS Versions 4.3.1) Users' Guide*, 2003.

[9] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, "MUMPS home page," 2003, http://www.enseeiht.fr/lima/apo/MUMPS.

[10] P. Henon, P. Ramet, and J. Roaman, "PaStiX: A parallel sparse direct solver based on a static scheduling for mixed 1d/2d block distributions," in *Proceedings of Irregular'2000*, ser. Lecture Notes in Comput. Sci., S. Verlag, Ed., vol. 1800, 2000, pp. 519–525.

[11] D. Hysom and A. Pothen, "A scalable parallel algorithm for incomplete factorization," *SIAM J. on Sci. Comp.*, vol. 22, no. 6, pp. 2194–2215, 2001.

[12] J. I. Aliaga, M. Bollhöfer, A. F. Martín, and E. S. Quintana-Ortí, "Exploiting thread-level parallelism in the iterative solution of sparse linear systems," *Parallel Comput.*, vol. 37, pp. 183–202, March 2011.

[13] J. Gaidamour and P. Henon, "A parallel direct/iterative solver based on a schur complement approach," *Computational Science and Engineering, IEEE International Conference on*, vol. 0, pp. 98–105, 2008.

[14] L. Giraud and A. Haidar, "Parallel algebraic hybrid solvers for large 3d convection-diffusion problems," *Numerical Algorithms*, vol. 51, pp. 151–177, 2009.

[15] I. Yamazaki and X. S. Li, "On techniques to improve robustness and scalability of a parallel hybrid linear solver," in *Proceedings of the 9th international conference on High performance computing for computational science*, ser. VECPAR'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 421–434.

[16] Y. Saad and M. Sosonkina, "Distributed Schur complement techniques for general sparse linear systems," *SIAM J. Sci. Comput*, vol. 21, pp. 1337–1356, 1997.

[17] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. SIAM, 2003.

[18] M. Sala and M. Heroux, "Robust algebraic preconditioners with IFPACK 3.0," Sandia National Laboratories, Tech. Rep. SAND-0662, February 2005.

[19] J. R. Gilbert and E. Zmijewski, "A parallel graph partitioning algorithm for a message-passing multiprocessor," *International Journal of Parallel Programming*, vol. 16, pp. 427–449, 1987.

[20] A. George, M. T. Heath, J. Liu, and E. Ng, "Sparse Cholesky factorization on a local-memory multiprocessor," *SIAM J. Sci. Stat. Comput.*, vol. 9, pp. 327–340, March 1988.

[21] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and U. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

[22] M. Sala, K. S. Stanley, and M. A. Heroux, "On the design of interfaces to sparse direct solvers," *ACM Trans. Math. Softw.*, vol. 34, pp. 9:1–9:22, March 2008.

[23] L. Giraud, A. Haidar, and Y. Saad, "Sparse approximations of the schur complement for parallel algebraic hybrid linear solvers in 3d," *Numerical Mathematics: Theory, Methods and Applications*, vol. 3, no. 3, pp. 276–294, 2010.

[24] E. Agullo, L. Giraud, A. Guermouche, and J. Roman, "Parallel hierarchical hybrid linear solvers for emerging computing platforms," *Comptes Rendus Mecanique*, vol. 339, pp. 96–103, 2011.

[25] T. F. C. Chan and T. P. Mathew, "The interface probing technique in domain decomposition," *SIAM J. Matrix Anal. Appl.*, vol. 13, pp. 212–238, January 1992.

[26] D. Bozdağ, U. V. Çatalyürek, A. H. Gebremedhin, F. Manne, E. G. Boman, and F. Özgüner, "Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation," *SIAM J. Sci. Comput.*, vol. 32, pp. 2418–2446, August 2010.

[27] T. A. Davis and Y. Hu, "The Univerity of Florida collection," *ACM Trans. Math. Software*, vol. 38, no. 1, 2011.

[28] T. A. Davis, "Algorithm 832: UMFPACK v4.3—an unsymmetric-pattern multifrontal method," *ACM Trans. Math. Softw.*, vol. 30, pp. 196–199, June 2004.

[29] E. G. Boman and S. Rajamanickam, "A study of combinatorial issues in a sparse hybrid solver," in *Proc. of SciDAC'11*, 2011.