SAND2011-2596 C

SAND2011-2596C

A Hybrid-Hybrid Solver for Manycore Platforms

Siva Rajamanickam
Sandia National Laboratories
Albuquerque, NM 87185
srajama@sandia.gov

Erik G. Boman
Sandia National Laboratories
Albuquerque, NM 87185
egboman@sandia.gov

Michael A. Heroux *
Sandia National Laboratories
Albuquerque, NM 87185
maherou@sandia.gov

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 method is based on approximate Schur complements using dropping or probing. Second, the solver uses two levels of parallelism via hybrid programming (MPI+threads). Our solver is useful both in shared-memory environments and on large parallel computers with distributed memory. In the latter case, our solver may be used as a subdomain solver. We argue that with the increasing complexity of compute nodes, it is helpful 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 compare flat MPI performance of ShyLU against a hybrid implementation. We conclude that on present multicore nodes flat MPI is better. However, for future manycore machines (96 or more cores) hybrid/ hierarchical algorithms and implementations are important for sustained performance.

Keywords

Parallel Computing, High Performance Computing, Sparse Solvers, Parallel Preconditioners, Hybrid Programming

1. INTRODUCTION

The general trend in computer architectures is towards hierarchical designs with increasing node level parallelism. As a result, 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 system (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 leading to one subdomain per core as well. A drawback of domain decomposition solvers (preconditioners) is that the number of iterations will increase with the number of subdomains. With the rapid increase in the number of cores, we argue that 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. In this paper we suggest a two-level approach on the node. Our hybrid-hybrid method is "hybrid" in two ways: first, the solver combines direct and iterative algorithms, and second, we use MPI and threads in a hybrid programming approach.

We argue that to be scalable and robust it is important for solvers and preconditioners to use the hybrid approach in both meanings of the word. A direct solver [24, 20] is very robust and the BLAS based implementations are capable of performing near the peak performance of desktop systems for specific problems. However, they suffer from 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.

Our major contribution in this paper 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[17, 16] 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, it is suitable both as a solver on a single-node multicore (manycore) workstation and as a subdomain solver on a compute node of a petaflop (exaflop) system. Our target is computers with many CPU-like cores, not GPUs.

*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.

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

1.1 Previous work

Many good parallel solver libraries have been developed over the last decades; for example, Trilinos, PETSc [6, 5, 4], and Hypre [12]. These were mainly designed for solving large distributed systems over many processors. Our 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 [9, 19] or Pardiso[24]) have shown good performance in shared-memory environments, while distributed-memory solvers (for example SuperLU-dist and MUMPS [2, 3]) have limited scalability. Pastix [15] 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 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. Although a few parallel codes exist (e.g., Euclid [18]), we are not aware of any implementation that uses threads or exploits shared memory.

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 iterate on the remaining Schur complement. Parallel codes of this type include HIPS [13], MaPhys [1], and PDSLin [25]. ShyLU is similar to these solvers in a conceptual way that all these solvers fall into the broad Schur complement framework described in section 2. 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 solvers our target is a many-core node. See section 4 for how these solvers differ from ShyLU in the different steps.

2. SCHUR COMPLEMENT FRAMEWORK

Our framework is a general way to solve linear systems based on the Schur complement approach. Much work has been done in this area; see for example, [21, Ch.14] and the references therein.

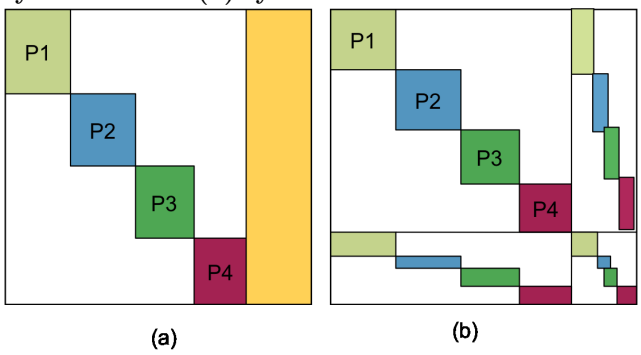
2.1 Schur complement formulation

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 the three steps:

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



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

where the vector subscripts correspond to the matrix block rows.

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 the framework. 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 either to be used as a preconditioner for an implicit S or for an inexact solve.

Fast inexact solution with S: Once there is an approximate S there are multiple options to solve using S and then solve for the entire system. For example, the algorithms can choose to just iterate on the Schur complement system and solve exactly for the full linear system, or use an iterative method for both, 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 tend to follow this framework.

2.2 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. We do not attempt to scale our "solver" to tens of thousands of cores, rather we envision it used as a subdomain solver. As a preconditioner, we no longer need to solve for D exactly. Also, we don't need to form (apply) 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.

2.3 Preconditioner Design

As is usual with preconditioners (see e.g., IFPACK [22]), we split the preconditioner into three phases: (i) Initialize, (ii) Compute, and (iii) Solve. Initialize() only depends on the sparsity pattern of A, so may be reused for a sequence of matrices. Compute() is called if any matrix entry has changed in value. Solve() 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 $\bar{S} \approx G - R * D^{-1}C$.

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 $Dx_1 = b_1 - Cx_2$.

3. HYBRID PROGRAMMING MODEL

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. We propose to use a threaded direct solver, for example, Pardiso [24] or SuperLU-MT [9, 19], to factor each block D_i. The assumption here is we can use multithreaded direct solvers (or potentially incomplete factorizations in the future) effectively within a uniform memory access (UMA) region, where all cores have equal (fast) access to a shared memory region. Using MPI between UMA regions help us mitigate the problems with data placement and non-uniform memory accesses.

In our implementation, we have chosen to use the Epetra package in Trilinos with MPI for the matrix A. Using MPI at this level helps us obtain data locality on a non-uniform access memory access (NUMA) node, and also allows us to run across nodes, if desired. 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, we could partition and use MPI for all the cores and use no threads. The opposite 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 5.

For massively parallel computing, we expect our solver to be used only for the subdomain since the matrix border size (and thus the Schur complement) will grow with the number of parts. Thus, 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. Applications that now run one MPI process per core remain that way, the additive Schwarz preconditioner (which will use our ShyLU on subdomains) can fuse the subdomains from all the cores in a node for ShyLU.
2. 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.
3. When applications start one MPI process per node in the medium term, additive Schwarz will use a threads-only ShyLU.

Thus the MPI+X programming model in ShyLU's design helps make the application migration to the manycore systems smooth depending on how the applications want to migrate. This is dependent on how much performance increase an application can gain in other portions of the application.

4. IMPLEMENTATION

Our framework consists of *partitioning*, *sparse approximation of the Schur complement*, and *fast, inexact solution of the Schur complement*. The first two steps only have to be done once in the setup phase. We now discuss our implementation for each of these steps in more detail.

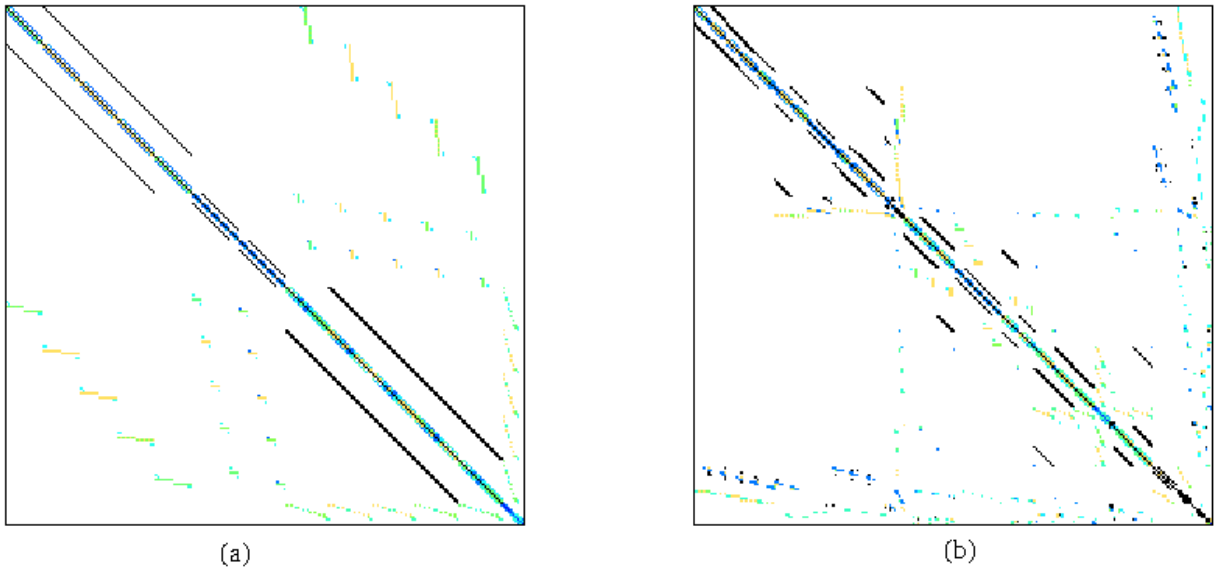


Figure 2: Sparse matrix before (left) and after partitioning and reordering (right). Here $k = 4$. The matrix is HB/orsirr_2

4.1 Partitioning

We use partitioning (domain decomposition) 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:

$$A = \begin{pmatrix} D & C \\ R & G \end{pmatrix}, \tag{2}$$

where D again has a block structure. As shown in Figure 1 we have two cases. In the symmetric case, we use 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. Figure 2 shows the matrix HB/orsirr_2 from the UF sparse matrix collection and the reordered matrix that will be used by ShyLU. 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). 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. The approach used by MA48 (direct factorization) [11] to make "diagonal" blocks square by including some columns from the border is an option too. We omit the details here, and focus on the structurally symmetric case.

Several variations of graph partitioning can be used to obtain block bordered structure. Traditional graph partitioning attempts to keep the parts (submatrices) of equal size while minimizing the edge cut (number of nonzeros in the border). Computing a small vertex separator would be better for our problem since the separator corresponds exactly to the border size. The other hybrid solvers we know (see Section 1) 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 (or separators) requires symmetry. Thus, we decided to use hypergraph partitioning in both the symmetric and nonsymmetric cases. We used the Zoltan/PHG partitioner [10] via the Isorropia package.

4.2 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 may be used. We choose to use a sparse direct solver. Our implementation uses Pardiso [24] 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[23] in Trilinos to interface with the direct solvers. This enables us to switch between any direct solver supported by the Amesos package. The other codes we have mentioned in Section 1 all use a serial direct solver in this step.

4.3 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

$$S = \begin{pmatrix} S_{11} & S_{12} & \ldots & S_{1k} \\ S_{21} & S_{22} & \ldots & S_{2k} \\ \vdots & \vdots & & \vdots \\ S_{k1} & S_{k2} & \ldots & S_{kk} \end{pmatrix} \tag{3}$$

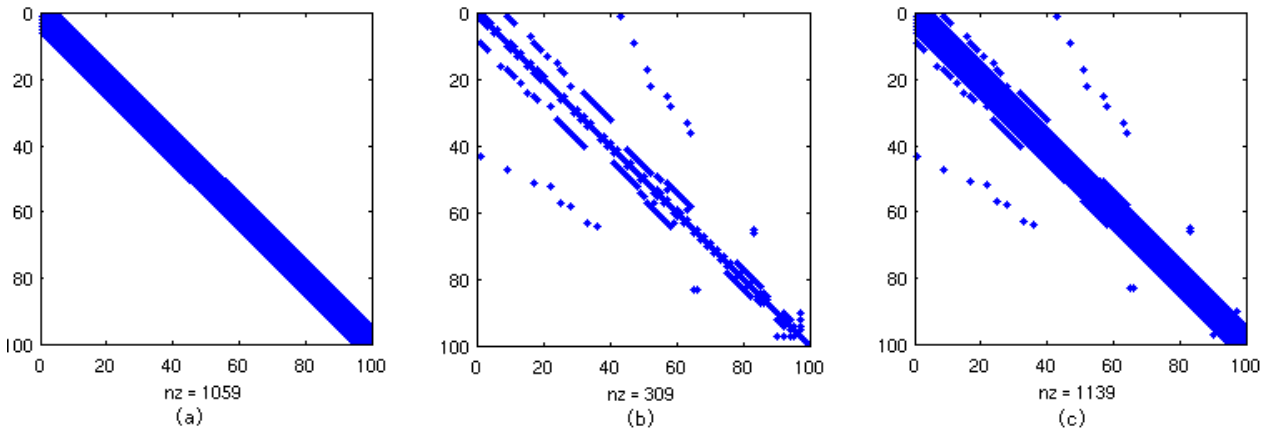


Figure 3: Structure to probe. (a) Structure of typical banded probing for $\bar{S}$. (b) Structure of G submatrix (c) Structure of $\bar{S} \cup G$ for ShyLU's probing

where it is known that the diagonal blocks S_{ii} are usually quite dense but the off-diagonal blocks are mostly sparse [21]. 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. This ignores any couplings between the local Schur complements. To improve robustness, we propose to keep the off-diagonal entries of S that correspond to entries in G. To save storage, the local Schur complements themselves need to be sparsified [14, 25].

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.

4.3.1 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 [1]. 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 diagonal entries. Since S can be quite dense, we only form a few columns at a time and immediately sparsify (drop). Note we do not drop based on $U^{-1}C$ or $R * U^{-1}$ where L and U are the LU factors, 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, this can be very expensive when there is not much to drop in S.

4.3.2 Probing (structure-based)

Since dropping may be expensive in some cases, we also implemented *probing*. Probing was developed to approximate interfaces in domain decomposition[7], which is also a Schur complement. In probing, we prescribe the sparsity pattern of $\bar{S} \approx S$. Then we compute a set of probing vectors, V, based on $\bar{S}$. This gives rise to a coloring problem, where the number of colors corresponds to the number of probing vectors 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 $\bar{S}$. Note we never need to form S explicitly. 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 [7]. 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 $\bar{S}$ is pattern of $B \cup G$, where B is a banded matrix.

Figure 3 uses just the G block of the reordered matrix from Figure 2(b) as an example. Figure 3(a) shows the structure of a typical banded probing assuming we are looking for 5% of the diagonals. To this structure, our algorithms 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 very important in $\bar{S}$ as well. Preliminary tests 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 again uses the parallel graph coloring algorithm in Zoltan. Probing for a complex structure is 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.

4.4 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.

As we only need an inexact solve as a preconditioner, we advocate a different approach. we can simply solve for $\bar{S}$ instead of for S. Now, even $\bar{S}$ is large enough that it should be solved in parallel. A simple approach is to apply a parallel direct solver to $\bar{S}$, but this may lead to too much fill. Instead, we solve for $\bar{S}$ iteratively. To solve for $\bar{S}$ iteratively, we use 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).

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 of the iterations. 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 $\bar{S}$ or both implies an inner-outer iterative method for the overall system. It is not sufficient to iterate on S but it is required to iterate on A. This is usually fine for a subdomain solver and might be required when we may do inexact factorization (instead of a direct factorization) in D as well, in the future. 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.

4.5 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 $\bar{S}$ in parallel.

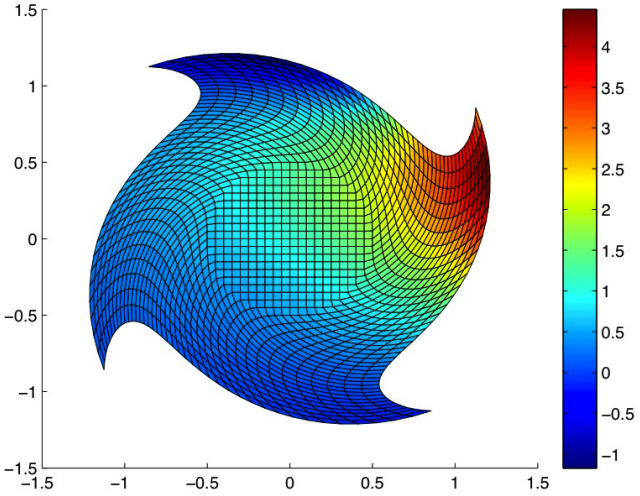
5. RESULTS

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 manycore platforms, and in particular the trade-off between flat MPI 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.

5.1 Experimental setup

We have implemented ShyLU in C++ within the Trilinos [17] framework. We leverage several Trilinos packages, in particular:

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



Epetra for matrix and vector data structures and kernels.

Isorropia/Zoltan for matrix partitioning and probing.

AztecOO/Belos for iterative solves (GMRES).

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 manycore 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 shows the number of outer iterations that will be seen by the user of ShyLU. When there are many tunable parameters there are two ways to do experiments. Always choose the best parameters for each problem, always use the best parameters for the entire solver. All the experiments in this section uses solver specific parameters and there is no tuning for a particular problem. 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^{-10} relative residual whichever comes first and 500 outer iterations or 10^{-9} relative residual whichever comes first. We use the native preconditioners in AztecOO package for solving for $\bar{S}$.

5.2 Robustness

We validate the robustness of ShyLU by comparing it to incomplete factorizations. We use two different variations of ShyLU, based on dropping and probing in the Schur complement. Both approaches have a tunable parameter that can be difficult to choose. We decided to use a fixed (default) dropping/probing tolerance in all our tests. We believe this

Table 1: Comparison of number of iterations of ShyLU probing and dropping with ILU(1) and ILUT(2, 1e-8). A dash indicates no convergence

Matrix Name	N	Symmetry	Dropping	Probing	ILU	ILUT
Pres_Poisson	14.8K	Symmetric	74	53	-	-
bodyy5	18K	Symmetric	76	76	173	109
Lourakis_bundle1	10K	Symmetric	33	29	38	31
FIDAP_ex35	19K	Unsymmetric	5	8	-	-
igbt3	10.9K	Unsymmetric	29	18	-	-
FEM_3D_thermal2	147K	Unsymmetric	12	8	24	23
venkat50	62.4K	Unsymmetric	40	33	-	-
airfoil2d	14.2K	Unsymmetric	25	15	153	97
nmos3	18.5K	Unsymmetric	30	-	-	-
FEMLAB_waveguide3D	21K	Unsymmetric	130	-	-	-
TC_N_360K	360K	Symmetric	58	48	342	201
Tramanto1	6K	Unsymmetric	114	-	-	-

reflects how a typical user would use the code. Similarly, we tested ILU(k) and ILUT preconditioners with fixed settings. Our goal is simply demonstrate the robustness of ShyLU compared to "black-box" methods that are commonly used today. The number of iterations 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, the typical use case in our applications ILU(1) and ILUT(2, 1e-8) and that is what we will compare against.

We chose ten sparse matrices from a variety of application areas, taken from the Univ. of Florida sparse matrix collection [8]. We added two test matrices from Sandia applications, Tramanto1 and TC_N_360K. The results are shown in Table 1. We see that both versions of ShyLU are more robust than ILU and ILUT, in the sense they have fewer failures. Generally, the drop-tolerance version requires fewer iterations (though not necessarily less run time) than the probing version.

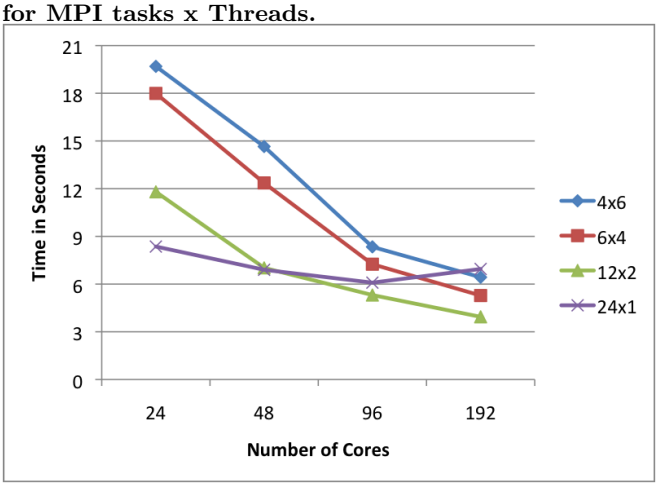
A dash indicates that GMRES failed to converge to the desired tolerance within 500 iterations. We observe that ShyLU is much more robust than ILU and ILUT. We expect ILU and ILUT could solve more problems by changing the level of fill or drop tolerance, but we used reasonable values.

We further observe that the dropping version is more robust than the probing version, as it solved all 12 test problems while the probing version failed in 3 out of 12 cases. However, the probing version converged faster when it worked. We will use the dropping version for our strong scaling tests and compare the two methods again for weak scaling.

5.3 Hybrid model and strong scaling

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 our the subdomain solver. We wish to study the trade-off between flat MPI 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 subdomains so the preconditioner changes as well. Thus, what we observe is a combined effect of changes in the solver algorithm and in the programming model (MPI+threads).

Figure 5: Strong Scaling of ShyLU's dropping method for a matrix of size 360K. Solve Time shown for MPI tasks x 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 manycore platforms with hundreds of cores. We simulate this by running ShyLU on several nodes. Since we use MPI even within the node, ShyLU also works across nodes. We expect future manycore platforms to be hierarchical with highly non-uniform memory access and running across the nodes will reasonably simulate future systems. We expect 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 $360K \times 360K$

For these experiments, we use the drop-tolerance version of

Figure 6: Strong Scaling of ShyLU's probing method for a matrix of size 360K. Solve Time shown for MPI tasks x Threads.

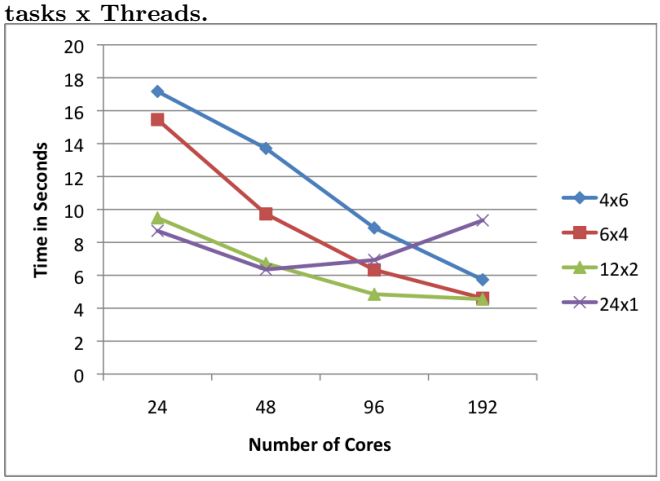


Table 2: Strong Scaling and Hybrid vs Flat MPI performance: Solve time in seconds (#iterations) for ShyLU dropping method to solve a linear system of size 360k

	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)

ShyLU. 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 2. More than 6 threads per node is not a recommended configuration for hopper so those results are not shown in Table 2. The solve time is also shown in Figure 5

There are several interesting observations. First, we see that although the number of iterations increase with the number of subdomains (MPI processes), 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.

Table 3: Strong scaling and Hybrid vs Flat MPI 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)	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 4: Strong scaling and Hybrid vs Flat MPI performance: Solve time in seconds (#iterations) for ShyLU dropping method to solve a linear system of size 720k

	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)

However, as we add more nodes (and cores), we see that the run times decrease much more rapidly for the hybrid configurations. For four nodes, the 12x2 configuration gives the fastest solve time. 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 6. The results are almost identical to the dropping method. The MPI only version became worse at 192 cores. At 192 cores any MPI+thread combination beats flat MPI. However, MPI only is still the best choice at 24 cores. The number of iterations for this experiment is shown in Table 3. 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 4. Note that ShyLU scales well up to 384 cores. Furthermore, we see that the crossover point where MPI+threads beats flat MPI is different for this larger problem (384 cores). The result can be seen clearly in Figure 7 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.

As shown above, the results can vary even for our solver algorithm, based on the problem, and other parameters. However, in our problems we see, as the available cores increase, and the size of the problems get smaller, hybrid solver beats flat MPI based solver.

We can also get strong scaling results by looking at a column at a time at the Tables 2 – 4. 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

Figure 7: Comparing flat MPI vs MPI+threads vs Problem size per subdomain.

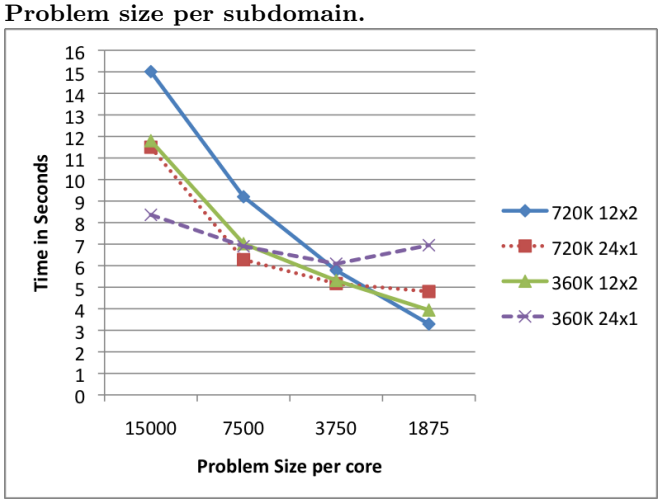


Table 5: ShyLU (probing) weak scaling results: Timing in seconds (#iterations) for 2D finite element problem.

	MPI Processes x Number of Threads in each node			
Nodes (Problem Size)	4x6	6x4	12x2	24x1
1 (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)

be a fast solver on a single node. The results are similar when we go to eight nodes (192 cores) too. The best scaling the hybrid model can achieve is 3.4 while flat MPI is able to get a speedup of 1.2 for the dropping method. The 6x4 and 12x2 configurations in the 720K problem size case (Table 4) achieve a speed up of 4.92 and 4.68 going from 48 to 384 cores. Flat MPI gained a speed up of 2.39 for this case. Overall, ShyLU is able to scale well upto 384 cores reasonably well.

5.4 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. Since ShyLU is a two-level hybrid solver, we expect the performance to be somewhere between a direct and a typical iterative solver.

Our 2D test problem is a finite element discretization of an elliptic PDE on a structured grid but with random 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 5–6 that both run time and number of iterations increase slowly with the number of nodes (cores). The dropping version demonstrates a smooth and predictable behavior, while the probing version has sudden jumps in number of iterations and time. We speculate that this is because the preconditioner is sensitive to the probing pattern. For the dropping version, the 12x2 configuration with 12 MPI threads per node is consistently the best.

Table 6: 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 (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 7: ShyLU (dropping) weak scaling results: Timing in seconds (#iterations) for the 3D 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)

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 7. 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 perfect scaling that multigrid methods may be able to achieve, this is much better than the $O(n^2)$ operations scaling by general sparse direct solvers. ShyLU's typical usage as a subdomain solver also places more emphasis on strong scaling, as the problem size per node is not growing as fast as the node concurrency, than weak scaling. We conclude that ShyLU is a good solver for problems of moderate size and scales quite well up to 192 cores.

6. FUTURE WORK

We plan several improvements in ShyLU. Many of these deal with combinatorial issues in the solver algorithm. First, we intend to extend the code to handle structurally nonsymmetric problems. A simple solution is to partition and reorder based on the symmetrized matrix $A+A^T$. This should work well for almost-symmetric problems but is not a good solution for highly nonsymmetric systems. Instead, we plan to hypergraph partitioning and permutation to singly bordered block form as shown in Figure 1. This requires us to find an efficient way to deal with rectangular blocks, for example, by using sparse LU with partial pivoting.

Second, we wish to study 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. Smaller border implies both less communication and a smaller Schur complement to solve.

Third, we have 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. We believe this issue poses a partitioning problem with multiple constraints and objectives, and cannot be adequately handled using standard partitioning models.

Fourth, we want to investigate better approximations to the Schur complement. We have shown that dropping and probing both have some advantages, so perhaps there is some combination that works better than either approach alone.

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 most also apply to other hybrid solvers. Discussions with the PDSLin developers have confirmed that they face similar issues. Thus, research into these combinatorial problems may help advance a whole class of solvers.

7. 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. Performance results show ShyLU can scale well for up to 384 cores in the hybrid mode for a given problem size.

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 MPI+Threads implementation beat the flat MPI 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 MPI+threads is 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 are increasing steadily and memory architectures are changing to favor core-to-core data sharing, hybrid/hierarchical algorithms and implementations are important for future manycore architectures.

8. ACKNOWLEDGMENTS

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.

9. REFERENCES

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

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

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

[4] S. Balay, W. Gropp, L. McInnes, and B. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[5] S. Balay, W. Gropp, L. McInnes, and B. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.22, Argonne National Laboratory, 1998.

[6] S. Balay, W. Gropp, L. McInnes, and B. Smith. PETSc home page. `http://www.mcs.anl.gov/petsc`, 1998.

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

[8] T. A. Davis and Y. Hu. The Univerity of Florida collection. Submitted. Web site at http://www.cise.ufl.edu/research/sparse/matrices.

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

[10] 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.

[11] I. S. Duff and J. A. Scott. A parallel direct solver for large sparse highly unsymmetric linear systems. *ACM Trans. Math. Software*, 30(2):95–117, 2004.

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

[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*, 0:98–105, 2008.

[14] 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*, 3(3):276–294, 2010.

[15] P. HENON, P. RAMET, and J. ROMAN. PaStiX: A parallel sparse direct solver based on a static scheduling for mixed 1d/2d block distributions. volume 1800 of *Lecture Notes in Comput. Sci.*, pages

519–525, 2000.

[16] M. A. Heroux. Trilinos home page., 2011. `http://trilinos.sandia.gov`.

[17] 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.*, 31(3):397–423, 2005.

[18] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factorization. *SIAM J. on Sci. Comp.*, 22(6):2194–2215, 2001.

[19] X. S. Li. An overview of superlu: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31:302–325, September 2005.

[20] X. S. Li and J. W. Demmel. Superlu-dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2):110–140, June 2003.

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

[22] M. Sala and M. Heroux. Robust algebraic preconditioners with IFPACK 3.0. Technical Report SAND-0662, Sandia National Laboratories, February 2005.

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

[24] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20(3):475–487, 2004.

[25] 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*, VECPAR'10, pages 421–434, Berlin, Heidelberg, 2011. Springer-Verlag.