Comparative Performance Modeling of Parallel Preconditioned Krylov Methods

Kanika Sood and Boyana Norris
Computer and Information Science
University of Oregon
Eugene, OR 97403, USA
Email: [kanikas,norris]@cs.uoregon.edu

Elizabeth Jessup
Computer Science
University of Colorado
Boulder, CO 80309, USA
Email: jessup@colorado.edu

***Abstract*—Many scientific and engineering computations rely on the scalable solution of large sparse linear systems. Preconditioned Krylov methods are widely used and offer many algorithmic choices whose performance varies depending on the characteristics of the linear system. In previous work, we have shown that the performance of different Krylov methods at small scales can be modeled using a small number of features based on structural and numerical properties of the input linear system. In this paper, we focus on comparing the scalability of parallel Krylov methods given different input properties without requiring extensive empirical measurements. We consider the PETSc implementations of Newton-Krylov methods to produce scalability rankings based on our new comparative modeling approach. The model-based ranking is validated by comparison with empirical results on a numerical simulation of driven fluid flow in a cavity.**

I. Introduction

The solution of large sparse linear systems of the form $Ax = b$, where $A = [a_{ij}]$ is an $n \times n$ matrix and b is a given right-hand-side vector, is central to many numerical simulations in science and engineering and is frequently the most time-consuming part of the computation. Advancements in these fields rely heavily on the efficient solution of these systems. On massively parallel architectures, iterative solvers [1] have gained popularity for approximating the solution x of sparse linear systems because they typically scale better than direct methods [2], which are based on the factorization of the coefficient matrix A into easily invertible matrices [3]. There is no clear boundary between classes of iterative and direct methods, however, as most iterative approaches exploit ideas and techniques from sparse direct solvers in the form of preconditioners, which increase the robustness of the iterative schemes without sacrificing scalability. Because of the reliability and good performance of iterative approaches, a very large number of methods and implementations are provided in such numerical software packages as PETSc [4], [5] and Trilinos [6].

With the growth in the number of solution methods, application writers face an increasingly difficult task in choosing a solution method that will perform best for their specific problem. Unlike other classes of algorithms, where complexity analysis is sufficient to inform such choices, preconditioned Krylov methods that have the same complexity may perform very differently, depending on the characteristics of the input matrix A. In general, the selection of the best-performing numerical method is extremely challenging. Determining it necessitates the comparative performance of various numerical methods on small and large scales.

In our previous work [7], we show the performance of different Krylov methods at small scale using machine learning techniques [8]. Our approach involves constructing a training data set by solving linear systems with multiple solver-preconditioner combinations and then classifying the combinations based on the time taken to solve a linear system with the given solver-preconditioner combination. The combination that solves the system in the least time is chosen as the base case for that system. Using the base case timing, other combinations are labeled as "good" and "bad".

When given a sufficiently varied set of input linear systems, the machine learning (ML) approach successfully captures the convergence behavior of solver configurations. However, the model was built based on the data collected on a small number of processors (72), and therefore does not capture differences in solvers' performance as larger problems are solved on larger processor counts. At the same time, collecting separate training data sets for different processor counts is prohibitively expensive and, for very large numbers of processors, simply infeasible. In this work we introduce an analytical approach to scalability-based ranking for different Krylov methods. This approach can be used in conjunction with the ML-based method to enable solver recommendations at different scales of parallelism.

In this paper, we present the details of our new communication-based model and describe how we can combine this ranking with the previously developed ML-based model to produce parallel preconditioned Krylov method recommendations. We validate the effectiveness of this model by applying it to the sparse linear system solution in a nonlinear PDE-based application for simulating driven cavity fluid flow.

This paper is organized as follows. Section II discusses the related work on the analytical and empirical modeling of Krylov methods. Section III focuses on the methodology of our modeling approach and shows the scalability ranking obtained with the model. Section IV covers our empirical evaluation of the model results. Section V outlines the conclusions and

future work.

II. Related Work

Both analytical and empirical modeling approaches are used to evaluate parallel linear algebra algorithms. A sequential algorithm can be described by its predicted run time, expressed as a function of its input size. Analytical evaluation of a parallel algorithm involves not only the size of the input but also machine characteristics such as the number of processors and communication parameters including overheads. In both the sequential and parallel cases, empirical modeling is based on the observations made from experiments with the algorithm. In this section, we present some examples of research into both types of models.

1) Analytical modeling of distributed-memory linear algebra algorithms: The authors of [9] focus on the inter-processor communication costs of computers and the communication costs inherent in parallel algorithms. They explain the gap between the theoretical measurements and analytical measurements of a parallel algorithm and encourage developers to create algorithms that are efficient not just theoretically but also practically. Communication occurs when a processor needs the data that is not available in its memory. Once a processor gets the required data from other processors, it does not need communication unless the data has been modified by the processors that own the data. One common technique for avoiding such communication is handled by replicating the data as local copies and re-accessing these copies. This is referred as the replication-communication relationship. The authors focus on the need for a better description of the relationship between communication and replication and an effective use of simulation and profiling tools.

LogP [10] is a computation model for parallel computation designed to enable the development of efficient and portable parallel algorithms. The authors convey that such algorithms typically align well with certain machine parameters. In particular, bottlenecks are connected with how algorithms interact with the computing bandwidth,the communication bandwidth, and the number of processors within a common hardware organization.

Another work particularly relevant to this category of modeling is [11] which predicts the performance of the constituent operations of Krylov methods on extreme scale computers via a model that includes topology and network acceleration. They motivate the development of pipelined implementations by observing the serious bottlenecks that result from the latency costs of reduction operations.

2) Empirical modeling of distributed-memory linear algebra: Fettig et al. have performed recent work [12] in analyzing the scaling behavior of parallel Krylov methods and multigrid methods from PETSc and hypre [13]. The solvers considered are Conjugate Gradient, GMRES and BiCGStab along with preconditioners Block Jacobi, ASM and multigrid methods. The results were collected on a Pentium III cluster and Itanium 1 cluster. Symmetric and nonsymmetric problems with the biggest problem involving 64 million unknowns were used. The results indicate that good scaling can be achieved on large scales for these solvers. They present scalability results for up to 256 processors on the Linux clusters. In this paper, we present scalability results with up to 12,288 processors for more solver methods than the ones covered by Fettig et al. In addition, we combine the communication-wise ranking with scalability ranking to decide the overall scalability of the Krylov methods.

III. Methodology

This section describes our approach to modeling two aspects of the performance of parallel Krylov methods: (1) convergence behavior and (2) parallel overhead. In this work, we consider a subset of the solvers and preconditioners available in PETSc: Conjugate Gradient (CG), BiConjugate Gradient (BiCG), BiConjugate Gradient Stabilized (BCGS), improved BCGS (iBCGS), Generalized Minimal Residual (GMRES), flexible GMRES (FGMRES), and TFQMR. We use the following preconditioning methods: Additive Schwarz (ASM), Jacobi and Block Jacobi along with no preconditioning.

For modeling convergence behavior, we use a supervised machine learning approach described in Sec III-A, which relies on a training dataset consisting of small-scale parallel timing results for a variety of linear systems and preconditioned Krylov methods. Even at a fixed number of processors (in our case, 72), creating the training dataset required us to solve tens of thousands of linear systems. Applying this approach to modeling the scalability of solvers for larger processor counts is infeasible because of the prohibitive computational time. Furthermore, no comprehensive matrix collections of large problems are available. The popular University of Florida matrix collection [14], which we used to collect our training data, consists of mostly small systems and therefore attempting to solve them on a large number of processors would not be useful for determining the best solution method for much larger problems. Hence, we introduce an analytical communication-based ranking of solution methods (described in Sec III-B), which enables solver predictions at large processor counts.

A. Solver classification using machine learning

In this work, we combine our previously developed machine learning-based solver classification with an analytical communication-based ranking. We use machine learning to classify solvers based on their convergence behavior. The training set contains features of over 1,800 matrices from the University of Florida Sparse Matrix collection and the solution times of 49 PETSc solver-preconditioner combinations (see Sec. III-C for the complete list) on 72 processors. We use the supervised learning approach from our prior work [7] for identifying the "good" solver-preconditioner combinations. Here, we briefly summarize our classification approach.

Initially we collected 34 linear system properties, which were reduced to the following set of six features used for the training and solver prediction: upper bandwidth, lower bandwidth, numeric value symmetry, diagonal sign pattern (5

possible values), diagonal number of nonzeros, and row variance (see [7] for more detailed descriptions of the features). Next, a binary label ("good", "bad") is assigned to each solver-preconditioner combination based on the time it takes to solve the linear system. The threshold for "good" solvers is chosen by varying the threshold parameter b, where $b \in [0, 1]$. We use a threshold of 0.45 for our experiments. The accuracy of the ML algorithm can be determined by the algorithm's correctness in identifying the "good" solvers as "good" as we are concerned only about the solvers that perform well. This accuracy is measured as $TPR = TP/P = TP/(TP + FN)$. A random forest classifier yielded overall accuracy of 98.8% and TPR accuracy of 98.4% (TPR is the most relevant performance metric because our goal is to suggest solvers that will likely perform well). With a train-test split of 66-34%, the overall accuracy obtained is 98.6% and the TPR accuracy is 98.1%.

B. Characterizing communication of Krylov methods

To address the challenge of modeling the scalability of algorithms, we introduce an analytical ranking of methods based on comparing the communication performed by the different preconditioned Krylov methods. While this ranking does not provide a complete performance model, it improves upon machine learning models that rely on matrix features and are necessarily trained on small-scale inputs. Most importantly, it enables more accurate selection of preconditioned Krylov methods at different parallelism scales

From the user's point of view, the process for selecting a solution method for a new linear system consists of two steps. First, the ML model is applied to generate a list of "good" solvers (e.g., those likely to converge and perform well). Second, if the system is sufficiently large (i.e., high levels of parallelism are required), we compute the intersection of the ML-generated list and the analytical solver rankings and select the top-ranked solver that is present in both.

1) Comparing Krylov method implementations: In this work, we analyze the solver and preconditioner algorithms provided by PETSc to measure the inter-process communication and identify the operations that perform communication for each iteration. Figure 1 shows the communication-performing matrix and vector operations in the aforementioned algorithms in PETSc. We analyzed these operations for each solver and preconditioner algorithm. The result is a communication-wise ranking for the preconditioned and non-preconditioned cases. In this stage, we analyze communication alone since solution time is not required. We count the number of times the operations that perform communication with other processors are called for each algorithm. The operations are discussed in detail later in this section. We verify these results by comparing them with the empirical measurements obtained by running real applications on the Edison Cray XC30 machine at NERSC. The result is a ranking based on the linear system solution time. For the empirical measurements, we solve five regular structured grids, resulting in square linear systems with up to 4,000,000 rows and columns and approximately 80,000,000 nonzeros. We solved the linear system in parallel with all the combinations of Krylov solvers and preconditioners and also without preconditioning. We present the communication analysis in the next section and discuss its validation detail in Section IV. Apart from the variation of solver and preconditioner, we use the default settings for other parameters in solving the system and analyzing the algorithms. We used the current version of PETSc (3.8) at the time of this writing.

There are two types of costs associated with a solver scheme: communication and computation. Communication in Krylov methods includes global reduction operations (for computing vector norms and dot products), matrix-vector products, and nearest-neighbor scatter/gather operations. To rank methods based on their scalability alone, we focus on the differences in the amount and type of communications between methods when solving the same problem on the same number of processors. We do not consider computation cost differences here because they are captured by our machine learning model (Sec. III-A).

Table I shows the operations that perform communication for these solvers and preconditioners. The table also presents their cost variables that will be used throughout the rest of the paper. The list in the table excludes operations that are common for all solvers and preconditioners. We compare communication cost by computing the difference in the number of calls made to operations that perform communication. We focus on the Krylov solver iteration and do not include initial setup, I/O functions or other functions that are called once. Because the number of iterations varies for each solver, we consider the normalized calls per iteration value instead of the raw counts.

In the following paragraphs, we describe how we determine the values for the communication cost variables for the different communication-containing functions listed in Table I. Here we focus only on *communication*, not on the computation in these kernels.

a) Reduction operations: Computing the norm of a distributed vector (`VecNorm`, the dot product of two vectors (`VecDot`, `VecTDot`) or a combination norm/dot product (`VecDotNorm2`) requires one or more global reduction operations (reducing an array of values to a single scalar). We designate the communication cost of a reduction operation by $q = \log p$, where p is the number of processors. The `VecMDot` function computes multiple dot products. The parameter k in `VecMDot_MPI` (the parallel implementation of `VecMDot` in PETSc) stands for the restart parameter which exists only in GMRES and FGMRES. `VecMDot_MPI` reduces x data items, where x ranges from $\{1, 2, \ldots, k\}$, where k is the restart parameter. The reduction cost of a single value is q and the sum of the series $1{+}2{+}3{+}\ldots{+}k$ is $k(k{+}1)/2$, resulting in the total cost of $q{*}k(k{+}1)/2$. The `MPIU_Allreduce` function is a replacement for the `MPI_Allreduce`. The latter combines values from all the processes and sends back the result to all the processes. The operation involves a constant amount of data ($\leq$ 96 bytes or 12 double precision values). This value

TABLE I: Matrix-vector operations with communication

Operation	Description	Cost Variable
MatMult	Computes matrix-vector product: $y = Ax$	m
MatMultTranspose	Computes matrix transpose times a vector $y = A'x$	m
VecNorm	Computes norm of the vector: $r = \|\|x\|\|$	q
VecDot	Computes the dot product of the vectors x and y	q
VecMDot	Computes one or more vector dot products.	q
VecMDot_MPI	Computes vector multiple dot products and performs reductions	$q * k(k+1)/2$
VecTDot	Computes indefinite vector dot product: $y^H x$, where y^H denotes the conjugate transpose of vector y	q
VecDotNorm2	Computes the inner product of two vectors and the 2-norm squared of the second vector	q
PCApply	Performs the preconditioning on the vector	c_{pc}
PCApplyTranspose	Applies the transpose of preconditioner to a vector	c_{pc}
VecScatterBegin	Performs a scatter from one vector to another	v
MPIU_Allreduce	Determines if the call from all the MPI processes occur from the same location in the code.	w

Function Name	Operation
MatAXPY(Mat Y, PetscScalar a,Mat X,MatStructure);	$Y = Y + a * X$
MatMult(Mat A,Vec x, Vec y);	$y = A * x$
MatMultAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A * x$
MatMultTranspose(Mat A,Vec x, Vec y);	$y = A^T * x$
MatMultTransposeAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A^T * x$
MatNorm(Mat A,NormType type, double *r);	$r = \|\|A\|\|_{type}$
MatDiagonalScale(Mat A,Vec l,Vec r);	$A = \text{diag}(l) * A * \text{diag}(r)$
MatScale(Mat A,PetscScalar a);	$A = a * A$
MatConvert(Mat A,MatType type,Mat *B);	$B = A$
MatCopy(Mat A,Mat B,MatStructure);	$B = A$
MatGetDiagonal(Mat A,Vec x);	$x = \text{diag}(A)$
MatTranspose(Mat A,MatReuse,Mat* B);	$B = A^T$
MatZeroEntries(Mat A);	$A = 0$
MatShift(Mat Y,PetscScalar a);	$Y = Y + a * I$

Function Name	Operation
VecAXPY(Vec y,PetscScalar a,Vec x);	$y = y + a * x$
VecAYPX(Vec y,PetscScalar a,Vec x);	$y = x + a * y$
VecWAXPY(Vec w,PetscScalar a,Vec x,Vec y);	$w = a * x + y$
VecAXPBY(Vec y,PetscScalar a,PetscScalar b,Vec x);	$y = a * x + b * y$
VecScale(Vec x, PetscScalar a);	$x = a * x$
VecDot(Vec x, Vec y, PetscScalar *r);	$r = \bar{x}' * y$
VecTDot(Vec x, Vec y, PetscScalar *r);	$r = x' * y$
VecNorm(Vec x,NormType type, PetscReal *r);	$r = \|\|x\|\|_{type}$
VecSum(Vec x, PetscScalar *r);	$r = \sum x_i$
VecCopy(Vec x, Vec y);	$y = x$
VecSwap(Vec x, Vec y);	$y = x$ while $x = y$
VecPointwiseMult(Vec w,Vec x,Vec y);	$w_i = x_i * y_i$
VecPointwiseDivide(Vec w,Vec x,Vec y);	$w_i = x_i / y_i$
VecMDot(Vec x,int n,Vec y[],PetscScalar *r);	$r[i] = \bar{x}' * y[i]$
VecMTDot(Vec x,int n,Vec y[],PetscScalar *r);	$r[i] = x' * y[i]$
VecMAXPY(Vec y,int n, PetscScalar *a, Vec x[]);	$y = y + \sum_i a_i * x[i]$
VecMax(Vec x, int *idx, PetscReal *r);	$r = \max x_i$
VecMin(Vec x, int *idx, PetscReal *r);	$r = \min x_i$
VecAbs(Vec x);	$x_i = \|x_i\|$
VecReciprocal(Vec x);	$x_i = 1/x_i$
VecShift(Vec x,PetscScalar s);	$x_i = s + x_i$
VecSet(Vec x,PetscScalar alpha);	$x_i = \alpha$

Fig. 1: Matrix and Vector operations in PETSc.

is constant because it is not dependent on the processor count or problem size.

b) Matrix-vector product: The communication cost of matrix-vector multiplication operations (`MatMult` and `MatMultTranspose`) can be estimated as follows. Given p processors, each processor sends its local nonzero values per row to all other processors and receives partial sum contributions to the local vector elements. Let us refer to a processor's average number of nonzero values per row as n. Therefore the total communication per processor here includes sending n values to every other processor and receiving the contributions for these n values from all the other processors. The communication cost of matrix-vector multiplication operations (given by m) can be written as $2 * n * (p-1)$.

c) Scatter-gather vector operations: The cost v of the nearest-neighbor scatter-gather operation, involving small amounts of data ($\leq$32 bytes or 4 double-precision scalar values) is constant, i.e., not dependent on the processor count or problem size.

d) Preconditioner application: The communication costs of the `PCApply` and `PCApplyTranspose` operations are designated by c_{pc}, where the subscript refers to the different preconditioners: c_{asm}, c_{jacobi} and $c_{bjacobi}$.

We compare the communication-related kernels for the following PETSc solvers and preconditioners:

- Performance of the solvers, without any preconditioning techniques: This category includes seven cases for the seven solvers, namely Conjugate Gradient, GMRES, FGMRES, TFQMR, BiCG, iBCGS, and BCGS.
- Performance of the solvers with each of the preconditioners: ASM(0), ASM(1), ASM(2), ASM(3), Jacobi and Block Jacobi, as well as no preconditioning, producing a total of 49 cases.

C. Communication-based Ranking

To compare the scalability of different linear solvers, we consider the difference between their communication costs in a single iteration. With this model, we are mainly targeting prediction at large numbers of processors $p >> 100$. Note that we do not consider the differences in computation costs – this aspect of performance is captured (in part) by our machine learning-based model described in Section III-A and [7].

First, we express the communication cost for each solver in terms of the number of calls to underlying communication-containing linear algebra kernels. Next, for each pair of solvers S_i and S_j, we compute the difference between these expressions, as shown later in greater detail. If the result is positive, then S_i has more communication; if the result is 0, both solvers have the same amount of communication, and if the result is negative, S_j has more communication. Instead

of focusing on low-level communication primitives (e.g., MPI functions), we consider each solver's use of matrix and vector operations that involve communication. Because these solvers contain different combinations of the operations in Table I, in most cases, we can determine whether the result of the subtraction is positive or not without considering the actual amount of communication for each operation. In a few cases, we do have to compare the specific communication costs (MPI level) of the basic matrix-vector operations. In the remainder of this section, we apply this approach to each pair of solvers, first without considering preconditioning and then adding a preconditioner that involves some communication.

D. Solvers with no preconditioning

First we consider Krylov methods without any preconditioning. Table II shows the number of calls to the functions performing communication for each Krylov method. After analyzing the code for obtaining the number of operations per iteration, we compute the communication cost for each case in comparison with the others. This step gives the total cost of communication for that solver. We show the case by case comparisons of communication costs of operations in the following subsections.

GMRES and Conjugate Gradient: Conjugate Gradient (CG) has the following operations that have communication: two matrix-vector products ($2m$), two vector norm computations ($2q$), and two vector dot products ($2q$), resulting in a total communication cost of $C_{CG} = 2m + 4q$. GMRES(k) where k is the restart parameter, incurs the following communication costs: $2m$, $2q$, and one `VecMDot_MPI`, whose communication cost is $qk(k+1)/2$, resulting in a total communication cost of $C_{GMRES} = 2m + 2q + qk(k+1)/2$. The difference $C_{GMRES} - C_{CG} = 2m + 2q + qk(k+1)/2 - 2m - 4q = (k(k+1)/2-4)q > 0$ for all $k > 2$. Hence, the communication cost of GMRES(k) is higher than that of CG for all $k > 2$.

Flexible GMRES and Conjugate Gradient: Similarly, we can compare Flexible GMRES (FGMRES) and CG. FGMRES has the following operations: two vector norm computations ($2q$), two matrix-vector products ($2m$) and one VecMDot_MPI operation. As shown previously, $C_{CG} = 4q + 2m$. The difference $C_{FGMRES} - C_{CG} = 2m + 2q + qk(k+1)/2 - 2m - 4q = (k(k+1)/2 - 4)q > 0$ for all $k > 2$. This shows that the communication cost for FGMRES is more than CG for all $k > 2$. The communication cost of FGMRES shown above and the cost for GMRES (shown in the previous case) are the same. Thus the comparison of CG with FGMRES method is the same as that of GMRES. Therefore the communication cost of FGMRES is more than that for CG and same as for the GMRES method.

TQFMR and Conjugate Gradient: TFQMR has two vector norm computations ($2q$), three vector dot products ($3q$), four matrix-vector products ($4m$). The difference $C_{TFQMR} - C_{CG} = 5q + 4m - (4q + 2m) = q + 2m > 0$ for all $q, m > 0$. Hence, TFQMR always has more communication than CG.

BiCG and Conjugate Gradient: BiCG has two vector norm computations ($2q$), two vector dot products ($2q$) and three matrix-vector products ($3m$). Comparing the communication costs of BiCG and CG we get: $C_{BiCG} - C_{CG} = 4q + 3m - (4q + 2m) = m > 0$ for all $m > 0$. This proves that BiCG always has higher communication cost than CG.

TABLE II: Number of calls of communication-relevant functions per iteration of several Krylov methods.

Operations	Parameter	Count
Conjugate Gradient		
VecTDot	q	2
VecNorm	q	2
PCApply	c_{pc}	2
MatMult	m	2
GMRES		
VecMDot_MPI	$q * r(r+1)/2$	1
VecNorm	q	2
PCApply	c_{pc}	2
MatMult	m	2
Flexible GMRES		
VecMDot_MPI	$q * r(r+1)/2$	1
VecNorm	q	2
PCApply	c_{pc}	1
MatMult	m	2
BCGS		
VecDot	q	2
VecNorm	q	2
VecDotNorm2	q	1
PCApply	c_{pc}	3
MatMult	m	3
iBCGS		
VecDot	q	2
VecNorm	q	1
MatMultTranspose	m	1
PCApply	c_{pc}	5
MatMult	m	3
MPIU_AllReduce	w	2
TFQMR		
VecDot	q	3
VecNorm	q	2
PCApply	c_{pc}	4
MatMult	m	4
BiCG		
VecDot	q	2
VecNorm	q	2
PCApply	c_{pc}	3
PCApplyTranspose	c_{pc}	2
MatMult	m	2
MatMultTranspose	m	1

BCGS and Conjugate Gradient, BiCG: BCGS has the following operations: two vector norm computations ($2q$), two vector dot products ($2q$), one VecDOTNorm operation q and three matrix-vector products ($3m$) totaling up to $5q+3m$. The difference $C_{BCGS} - C_{CG} = 5q + 3m - (4q + 2m) = q + m > 0$ for all $q, m > 0$. This shows that the communication cost for BCGS is more than that of Conjugate Gradient. Also, because BiCG's communication cost is more than that of CG, BCGS has more communication than BiCG as well.

BCGS and TFQMR: The difference $C_{TFQMR} - C_{BCGS} = 5q + 4m - (5q + 3m) = m > 0$ for all $m > 0$. Hence, TFQMR always has more communication than BCGS.

BCGS and GMRES: $C_{BCGS} - C_{GMRES} = (5q + 3m) - (2m + 2q + qk(k+1)/2)$. As described in section III, $m = 2n(p-1)$ and $q = \log p$. Substituting these values and considering the default value of k, the restart parameter in

GMRES, $k = 30$ and the average number of nonzeros per row (n) for the problem under consideration is 5. Further substituting the k and n values in the above equation we get: $C_{BCGS} - C_{GMRES} = -462 * \log p + 10(p-1) > 0$ for all $p > 257$. Hence, the communication cost is more for BCGS than GMRES for all $p > 257$, where p is the processor count.

BCGS and iBCGS: The comparison of BCGS and iBCGS can be done as follows: $C_{BCGS} = 5q + 3m$ and iBCGS consists of 3 MatMult ($3q$), 2 VecDot($2q$), 1 VecNorm(q) and 1 MatMultTranspose(m) operations resulting in a total of $3q + 4m$. The difference $C_{iBCGS} - C_{BCGS} = 3q + 4m - (5q + 3m) > 0$ for all $q, m > 0$. On substituting the values of m and q, we get the below equation: $10 * (p - 1) - \log p > 0$ for all $p > 1$. This shows that the cost of iBCGS is more than BCGS for all $p > 1$.

iBCGS and GMRES: iBCGS can be compared with GMRES as follows: $C_{iBCGS} - C_{GMRES} = (3q+4m) - (2m+2q+qk(k+1)/2)$. Substituting the values, $k = 30$, $m = 2n(p-1)$, $n = 5$ and $q = \log p$ we get: $C_{iBCGS} - C_{GMRES} = \log p + 20(p - 1) - 465 \log p > 0$ for all $p > 110$. Hence, iBCGS has more communication than GMRES for $p > 110$.

BiCG and GMRES: The comparison for BiCG cost with GMRES can be made as follows: $C_{BiCG} - C_{GMRES} = (4q + 3m) - (2q+2m+qk(k+1)/2)$. Substituting the values $k = 30$, $m = 2n(p - 1)$, $n = 5$ and $q = \log p$ we get: $\log p * k(k + 1)/2 - (2 \log p + 2n(p-1)) = 2n(p-1) - 463 \log p$, which is negative for $p \in [1, 258]$ and positive otherwise. Hence, BiCG has more communication than GMRES for $p > 258$.

GMRES and TFQMR: The last case compares GMRES with TFQMR as follows: $C_{TFQMR} - C_{GMRES} = 5q + 4m - (2q + 2m + qk(k + 1)/2)$. On substituting the values for m, q and n, we get the following: $20(p - 1) - 462 \log p > 0$ for all $p > 109$. Hence, the communication for TFQMR is more than that for GMRES for cases where $p > 109$.

Combining the comparison of BCGS with iBCGS and BCGS with CG, we can derive that iBCGS has more communication than CG. This is because BCGS has a higher cost of communication than CG and iBCGS have higher communication cost than BCGS. Therefore we can conclude that iBCGS has more communication cost than CG.

We compare two solvers at a time based on their communication cost in each of the above cases. With the comparison, these solvers can be arranged in a sequence of increasing communication costs. When all the solvers are placed in the sequence of increasing communication cost, we can compare all the solvers with respect to the others.

1) Solvers with Preconditioning: Similarly, we perform the communication-cost computation for all the solvers with the preconditioner cases as well. In this work, we analyze three parallel preconditioners offered by PETSc: the ASM, Jacobi and Block Jacobi methods. The Jacobi method applies the matrix diagonal as the preconditioner. Block Jacobi is similar to the Jacobi method except that Block Jacobi method divides the matrix into blocks and solves each block with the block diagonal as the preconditioner. The ASM preconditioner solves a problem by splitting the problem and solving it on sub-domains, which are smaller than the actual domain. There can be an overlap, which refers to the data that is common in two sub-domains. This is known as the overlap parameter of the ASM method. We consider four variations of this parameter with overlap values as 0, 1, 2 and 3. These are referred to as ASM(0), ASM(1), ASM(2) and ASM(3). The ASM preconditioner has additional communication in its PCApply operation which gets added to the overall cost for every solver preconditioned with this preconditioner. ASM communicates the overlapping data to one of the neighboring processors. The total cost of transferring the overlap data, *c_pc*, can be given as the product of the number of blocks to/from which the data is transferred and the cost of data communication given by q. This gives $c\_pc = number\ of\ blocks * transfer\ cost$. Thus the cost of the PCApply operation for ASM(0) is 0, ASM(1) is q, ASM(2) is $2q$ and ASM(3) is $3q$, where q refers to the cost variable shown in Table I. Jacobi and Block Jacobi do not have any additional communication in their PCApply operations, so the cost variable *c_pc* for these two methods is 0.

2) Preconditioning Methods: Similar to analyzing the communication with no preconditioning, we collect the operations with communication for all solvers preconditioned with ASM. The operations are shown in Table IV Conjugate Gradient has the least communication, and Flexible GMRES has the most communication.

Unlike ASM, Jacobi and Block Jacobi preconditioners do not perform communication. Hence, solvers preconditioned with Jacobi and Block Jacobi have the same communication cost as solvers with no preconditioning (but may converge in fewer iterations, which is captured by our ML-based model).

Using the results of the pair-wise comparisons, we rank solver configurations from 1 to 49 in order of increasing communication costs as shown in Table III. Solvers with the same communication cost share the same rank.

IV. Empirical Evaluation

We evaluated our approach by using an application for the simulation of driven cavity flow [15] that uses fully implicit Newton-Krylov methods to solve the resulting system of nonlinear PDEs. We selected this model problem because it has properties that are representative of many large-scale nonlinear PDE-based applications in domains such as computational aerodynamics [16], astrophysics [17], and fusion [18]. The most time-consuming portion of the simulation is the solution of large, sparse linear systems of equations.

The driven cavity model is a combination of lid-driven flow and buoyancy-driven flow in a two-dimensional rectangular cavity. The lid moves with a steady and spatially uniform velocity and sets a principal vortex by viscous forces. The differentially heated lateral walls of the cavity invoke a buoyant vortex flow, opposing the principal lid-driven vortex. The nonlinear system can be expressed in the form $f(u) = 0$, where $f : R^n \rightarrow R^n$. We discretize this system using finite differences with the usual five-point stencil on a uniform Cartesian mesh, resulting in four unknowns per mesh point (two-dimensional velocity, vorticity, and temperature). Further

TABLE III: Communication-based ranking of solvers for $p > 258$.

Ranking	NoPC	ASM(0)	ASM(1)	ASM(2)	ASM(3)	Jacobi	BJacobi
CG	1	4	5	6	7	1	1
GMRES	8	14	17	19	21	8	8
FGMRES	8	14	16	18	19	8	8
BCGS	26	29	31	34	35	26	26
TFQMR	39	42	47	48	49	39	39
BiCG	22	25	30	31	33	22	22
iBCGS	36	43	44	45	46	36	36

TABLE IV: Operations with communication in ASM.

Krylov method	Operations
Conjugate Gradient	7q + 2m + 4v
GMRES	5q + 2m + qr(r+1)/2 + 4v
Flexible GMRES	4q + 2m + qr(r+1)/2 + 4v
BCGS	10q + 3m + 4v
TFQMR	9q + 3m + 4v
BiCG	7q + 3m + 4v
iBCGS	9q + 4m + 4v

details are provided in [15]. The results discussed in this section employ a 1000×1000 mesh and the following nonlinearity parameters: default lid velocity (0.1), Prandtl number 1, and two different Grashof numbers: 100 and 1,000 (different Grashof numbers result in different numerical properties of the resulting linear system).

Table V shows the Krylov methods labeled "good" by the random forest classifier along with their ranking from our analytical communication model for driven cavity simulation on a 1000×1000 grid for Grashof values 100 and 1,000.

For small inputs that don't require many processors (e.g., < 200), the ML model alone can be used to choose a good solution by randomly selecting a method among the ones classified as "good". Larger problems require more parallelism, so, in addition to applying the ML-generated model to produce "good" solver recommendations, we also take into account the solver scalability as expressed by our communication-based ranking. Specifically, we find that the intersection of the ML-generated solver list and the analytical solver ranking can be used to select the top-ranked Krylov method.

TABLE V: Combining ML-based predictions with the analytical model ranking for systems arising in the driven cavity application (1000×1000 grid).

Grashof=100		Grashof=1,000	
Rank	Krylov method	Rank	Krylov method
14	FGMRES/ASM(0)	14	FGMRES/ASM(0)
21	GMRES/ASM(3)	36	iBCGS/Block Jacobi
30	BiCG/ASM(1)	43	iBCGS/ASM(0)
31	BiCG/ASM(2)	49	iBCGS/ASM(0)
34	BCGS/ASM(2)	-	
44	iBCGS/ASM(1)	-	
47	TFQMR/ASM(1)	-	
49	TFQMR/ASM(3)	-	

We validate the solver selection through empirical measurements obtained on the NERSC Edison supercomputer for problems of up to 80 million nonzeros solved on up to 12,288 processors. The empirical results in Tables VI and VII are sorted based on the measured average linear system solution time. (Additional results are available in http://tiny.cc/hpcc17ex). Each row shows the performance of a Krylov method on different numbers of MPI tasks compared with the solver with the best execution time and the speedup with respect to the default solver/preconditioner combination (GMRES/Block Jacobi). The predicted solver configuration (in bold font) is the highest-ranked (based on communication) Krylov method that was suggested by the ML model.

As expected, the selection based on communication overhead ranking is more effective at larger processor counts, where communication plays a larger role. While several of the ML-suggested solvers achieve significant speedups over the default, selecting a method among them based on the communication overhead did not always improve performance for smaller processor counts. To improve the quality of the ML predictions, in our future work, we plan to investigate other ML methods that allow ranking based on convergence instead of simple two-label classification.

TABLE VI: Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 100$.

Comm. Rank	Krylov Method	Avg. Time per Solve	Ratio w.r.t. Best	Speedup w.r.t. Def	# Its
1,536 MPI tasks:					
35	BCGS/ASM(3)	0.70	1.00	2.90	3
34	BCGS/ASM(2)	0.73	1.05	2.77	3
29	BCGS/ASM(0)	0.73	1.05	2.77	3
44	iBCGS/ASM(1)	0.73	1.05	2.77	3
42	TFQMR/ASM(0)	0.77	1.10	2.65	3
$\cdots$ 20 other solvers omitted $\cdots$					
14	**FGMRES/ASM(0)**	**3.20**	**4.57**	**0.64**	**2**
$\cdots$					
6,144 MPI tasks:					
19	FGMRES/ASM(3)	0.33	1.00	3.90	2
39	TFQMR/Block Jacobi	0.55	1.65	2.36	4
35	BCGS/ASM(3)	0.60	1.80	2.17	3
26	BCGS/Block Jacobi	0.73	2.20	1.77	3
31	BCGS/ASM(1)	0.83	2.50	1.56	3
42	TFQMR/ASM(0)	0.93	2.78	1.41	4
$\cdots$ 14 other solvers omitted $\cdots$					
14	**FGMRES/ASM(0)**	**2.47**	**7.40**	**0.53**	**2**
$\cdots$					
12,288 MPI tasks:					
14	**FGMRES/ASM(0)**	**0.33**	**1.00**	**7.40**	**2**
19	GMRES/ASM(2)	0.33	1.00	7.40	3
14	GMRES/ASM(0)	0.43	1.30	5.69	2
44	iBCGS/ASM(1)	0.60	1.80	4.11	3
45	iBCGS/ASM(2)	0.70	2.10	3.52	3
46	iBCGS/ASM(3)	0.73	2.20	3.36	3
$\cdots$					

TABLE VII: Solver prediction for the driven cavity problem on a 1000×1000 grid with $Grashof = 1,000$.

Comm. Rank	Krylov Method	Avg. Time per Solve	Ratio w.r.t. Best	Speedup w.r.t. Def	# Its
1,536 MPI tasks:					
31	BCGS/ASM(1)	0.50	1.00	4.30	4
34	BCGS/ASM(2)	0.73	1.45	2.97	4
36	iBCGS/Block Jacobi	0.85	1.70	2.53	3
45	iBCGS/ASM(2)	0.88	1.75	2.46	4
$\cdots$ 7 other solvers omitted $\cdots$					
8	GMRES/Block Jacobi	2.15	4.30	1.00	1
14	**FGMRES/ASM(0)**	**2.20**	**4.40**	**0.98**	**1**
$\cdots$					
6,144 MPI tasks:					
26	BCGS/Block Jacobi	0.55	1.00	2.91	1
29	BCGS/ASM(0)	0.65	1.18	2.46	1
34	BCGS/ASM(2)	0.70	1.27	2.29	4
35	BCGS/ASM(3)	0.70	1.27	2.29	4
26	BCGS/None	0.90	1.64	1.78	3
31	BCGS/ASM(1)	0.98	1.77	1.64	4
$\cdots$ 11 other solvers omitted $\cdots$					
14	**FGMRES/ASM(0)**	**3.45**	**6.27**	**0.46**	**1**
$\cdots$					
12,288 MPI tasks:					
26	BCGS/Jacobi	0.25	1.00	6.00	4
45	iBCGS/ASM(2)	0.65	2.60	2.31	4
44	iBCGS/ASM(1)	0.68	2.70	2.22	4
$\cdots$ 7 other solvers omitted $\cdots$					
14	**FGMRES/ASM(0)**	**1.45**	**5.80**	**1.03**	**1**
8	GMRES/Block Jacobi	1.50	6.00	1.00	1
$\cdots$					

The problem configuration was chosen to be both feasible and require a nontrivial amount of time on larger processor counts; the initial configuration was not changed in any way during the validation.

V. Conclusions and Future Work

We describe a new method for modeling the performance of parallel preconditioned Krylov methods that combines a machine learning model of convergence with an analytical parallel scaling model based on high-level communication estimates. Our new communication-based analytical model gives a scalability ranking of the solvers. We illustrated this approach on the Conjugate Gradient, GMRES, FGMRES, TFQMR, BiCG, iBCGS and BCGS solvers with preconditioners including ASM, Jacobi, Block Jacobi, and without preconditioning. We give a scalability ranking to all the solvers based on the amount of communication involved per KSP linear iteration. We evaluated our approach on a numerical simulation of driven fluid flow in a cavity, resulting in speedups of up to 7.4 over the default solver configuration on 12,288 processors.

Future work will include more solvers and preconditioners, including multigrid approaches. While the machine learning model generation is largely automated, the analytical ranking still involves manual effort; hence we plan to automate the solver ranking given specific input problem features.

Acknowledgment

This work is supported by the U.S. Department of Energy Office of Science (Contract No. DE-AC02-06CH11357) and by the National Science Foundation (NSF) awards CCF-1219089 and CNS-0821794. We used resources of the National Energy Research Scientific Computing Center (DOE Contract No. DE-AC02-05CH11231).

References

[1] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[2] T. A. Davis, *Direct methods for sparse linear systems*. SIAM, 2006.

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object-oriented numerical software libraries," in *Modern software tools for scientific computing*. Springer, 1997, pp. 163–202.

[4] S. Balay et. al, "PETSc Web page," http://www.mcs.anl.gov/petsc, 2017. [Online]. Available: http://www.mcs.anl.gov/petsc

[5] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[6] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger *et al.*, "An overview of trilinos," Citeseer, Tech. Rep., 2003.

[7] E. Jessup, P. Motter, B. Norris, and K. Sood, "Performance-based numerical solver selection in the Lighthouse framework," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S750–S771, 2016.

[8] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.

[9] J. P. Singh, E. Rothberg, and A. Gupta, "Modeling communication in parallel algorithms: A fruitful interaction between theory and systems?" in *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '94. New York, NY, USA: ACM, 1994, pp. 189–199. [Online]. Available: http://doi.acm.org/10.1145/181014.181329

[10] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken, "LogP: A practical model of parallel computation," *Commun. ACM*, vol. 39, no. 11, pp. 78–85, Nov. 1996. [Online]. Available: http://doi.acm.org/10.1145/240455.240477

[11] T. J. Ashby, P. Ghysels, W. Heirman, and W. Vanroose, "The impact of global communication latency at extreme scales on Krylov methods," in *Proceedings of Algorithms and Architectures for Parallel Processing. ICA3PP 2012*, 2012.

[12] J. Fettig, W.-Y. Kwok, and F. Saied, "Scaling behavior of linear solvers on large linux clusters," *National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign, IL*, 2002.

[13] R. D. Falgout and U. M. Yang, "hypre: A library of high performance preconditioners," in *International Conference on Computational Science*. Springer, 2002, pp. 632–641.

[14] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[15] T. S. Coffey, C. T. Kelley, and D. E. Keyes, "Pseudo-transient continuation and differential-algebraic equations," *SIAM J. Sci. Comput.*, vol. 25, no. 2, 2003.

[16] W. K. Anderson, W. D. Gropp, D. K. Kaushik, and et al, "Achieving high sustained performance in an unstructured mesh CFD application," in *SC99*, 1999.

[17] B. Fryxell, K. Olson, P. Ricker, and et al, "FLASH: An adaptive-mesh hydrodynamics code for modeling astrophysical thermonuclear flashes," *Astrophys. J. Suppl.*, 2000.

[18] X. Z. Tang, G. Y. Fu, S. C. Jardin, and et al, "Resistive magnetohydrodynamics simulation of fusion plasmas," Princeton Plasma Physics Laboratory, Tech. Rep. PPPL-3532, 2001.