Domain Decomposition Preconditioners for Communication-Avoiding Krylov Methods on a Hybrid CPU/GPU Cluster

Ichitaro Yamazaki†, Sivasankaran Rajamanickam*, Erik G. Boman*, Mark Hoemmen*,
Michael A. Heroux*, and Stanimire Tomov†
iyamazak@eecs.utk.edu, srajama@sandia.gov, egboman@sandia.gov, mhoemme@sandia.gov,
maherou@sandia.gov, and tomov@eecs.utk.edu
†University of Tennessee, Knoxville, Tennessee, U.S.A.
*Sandia National Laboratories, Albuquerque, New Mexico, U.S.A.

***Abstract*—Krylov subspace projection methods are widely used iterative methods for solving large-scale linear systems of equations. Researchers have demonstrated that communication-avoiding (CA) techniques can improve Krylov methods' performance on modern computers, where communication is becoming increasingly expensive compared to arithmetic operations. In this paper, we extend these studies by two major contributions. First, we present our implementation of a CA variant of the Generalized Minimum Residual (GMRES) method, called CA-GMRES, for solving nonsymmetric linear systems of equations on a hybrid CPU/GPU cluster. Our performance results on up to 120 GPUs show that CA-GMRES gives a speedup of up to 2.5x in total solution time over standard GMRES on a hybrid cluster with twelve Intel Xeon CPUs and three Nvidia Fermi GPUs on each node. We then outline a domain decomposition framework to introduce a family of preconditioners that are suitable for CA Krylov methods. Our preconditioners do not incur any additional communication and allow the easy reuse of existing algorithms and software for the subdomain solves. Experimental results on the hybrid CPU/GPU cluster demonstrate that CA-GMRES with preconditioning achieve a speedup of up to 7.4x over CA-GMRES without preconditioning, and speedup of up to 1.7x over GMRES with preconditioning in total solution time. These results confirm the potential of our framework to develop a practical and effective preconditioned CA Krylov method.**

I. Introduction

On modern computers, communication is becoming increasingly expensive compared to arithmetic operations in terms of throughput and energy consumption. "Communication" includes data movement or synchronization between parallel execution units, as well as data movement between levels of the local memory hierarchy. To address this hardware trend, researchers have been studying techniques to avoid communication in *Krylov subspace projection methods*, a popular class of iterative methods for solving large-scale sparse linear systems of equations and eigenvalue problems. Effectiveness of such *communication-avoiding* (CA) techniques [9], [3] to improve the performance of Krylov methods has been demonstrated on shared-memory multicore CPUs [10], on distributed-memory CPUs [19], and on multiple graphics processing units (GPUs) on a single compute node [17]. In this paper, we extend these studies and show the performance of a CA variant of the Generalized Minimum Residual (GMRES) method [13] for solving a nonsymmetric linear system of equations on a *hybrid CPU/GPU cluster*. Our performance results on up to 120 GPUs demonstrate speedups of up to 2.5x.

The CA Krylov methods were originally proposed as s-step methods over thirty years ago [16], but they have not been widely adopted in practice. One reason for this is that, in practice, Krylov methods depend on preconditioning to accelerate their convergence rate, but no effective preconditioners have been shown to work seamlessly with a CA method. This is partially because the existing CA preconditioning techniques revoke one of the most attractive features of Krylov methods, namely that they require no knowledge of the internal representation of the coefficient matrix A or of any preconditioners. In fact, most of the existing implementations of the Krylov methods only require two independent "black-box" routines, one for sparse-matrix vector product (*SpMV*) and the other for preconditioning (*Preco*). This black-box feature makes it easy to use any exisiting algorithm or software for *Preco* in the Krylov methods, and also to test a new preconditioning technique. The CA Krylov methods introduce a new computational kernel, *matrix powers kernel (MPK)*, that replaces *SpMV* and generates a set of Krylov subspace basis vectors all at once [9]. Then, all the existing techniques for preconditioning a CA method require close integration of *Preco* with *SpMV* within *MPK*, violating this handy black-box feature and making it difficult to test and develop an effective preconditioner.

To fill this crucial gap, in this paper, we propose a domain decomposition framework to develop an effective CA preconditioning technique. Our preconditioner may be considered as a variant of an additive Schwarz preconditioner, modified to ensure consistent interfaces between the subdomains without additional communication beyond what *MPK* already needs. However, more importantly, since we rely on domain decomposition, our preconditioner is not tightly coupled with *SpMV*. Hence, beside adding the local solver to apply the preconditioner to a local vector, our framework does not require any changes to *MPK*, and it can use any existing

978-1-4799-5500-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SC.2014.81

solver or preconditioner software package as the local solver on each subdomain. In other words, our CA preconditioner does not force any change to the sparse matrix's data structure or constrain the sparsity pattern of the subdomain solver. It thus defines a new category of CA preconditioning techniques. To study the proposed framework's performance, we combine this with CA-GMRES on a hybrid CPU/GPU cluster. Though this is merely our initial implementation, our experimental results on up to 30 GPUs show that CA-GMRES with preconditioning can obtain speedups of 7.4x over CA-GMRES without preconditioning, and speedups of 1.7x over GMRES with preconditioning.

In addition to demonstrating the potential of our framework, our current studies illustrate the importance and the challenge of developing such preconditioning techniques. Moreover, in this paper, we focus on CA-GMRES because we know from past experience [17] how to implement it efficiently and in a numerically stable way. However, CA variants of short-recurrence iterations like CG and BiCGSTAB spend much less time in vector operations than CA-GMRES, and thus, our work on an effective CA preconditioner may benefit CA versions of other Krylov methods even more than CA-GMRES.

The rest of the paper is organized as follows. After surveying related work in Section II, we first present, in Section III, our implementation of CA-GMRES and its performance on a hybrid CPU/GPU cluster. Then, in Section IV, we describe our CA preconditioning framework, outline our initial implementation, and give numerical and performance results on a hybrid cluster. Section V shows final remarks and future work.

Throughout this paper, we denote the i-th row and the j-th column of a matrix A by $\mathbf{a}_{i,:}$ and $\mathbf{a}_{:,j}$, respectively, while $A_{j:k}$ is the submatrix consisting of the j-th through the k-th columns of A, inclusive, and $A(\mathbf{i}, \mathbf{j})$ is the submatrix consisting of the rows and columns of A that are given by the row and column index sets $\mathbf{i}$ and $\mathbf{j}$, respectively. We use $nnz(A)$ and $|A|$ to denote the number of nonzeros in the matrix A and the matrix dimension, respectively. All the experiments were conducted on the Keeneland system[1] at the Georgia Institute of Technology. Each of its compute nodes consists of two six-core Intel Xeon CPUs and three NVIDIA M2090 GPUs, with 24GB of main CPU memory per node and 6GB of memory per GPU. We used the GNU `gcc` 4.4.6 compiler and CUDA `nvcc` 5.0 compiler with the optimization flag `-O3`, and linked with Intel's Math Kernel Library (MKL) version 2011_sp1.8.273 and OpenMPI 1.6.1. The test matrices used for our experiments are listed in Figure 9.

II. Related work

Most existing CA preconditioning techniques fall into one of two categories [9]. The first category naturally fits how CA methods compute sparse matrix-vector products. In contrast, the second class of exisiting CA preconditioners require radical changes to the representation of both the sparse matrix A and its preconditioner, such that their off-diagonal blocks are stored using a low rank representation. To the best of our knowledge, there is no implementation or empirical evaluation of such CA preconditioners. Furthermore, most users of Krylov solvers may not wish to make such radical changes to their data structures. Hence, for the rest of this section, we focus on the preconditioners in the first category.

The CA preconditioners in the first category include preconditioners like sparse approximate inverses with the same sparsity pattern as the matrix A, or block Jacobi and polynomial preconditioners [5], [15], [16]. For instance, in block Jacobi preconditioning, each processor (or GPU) independently solves its local problem. For a conventional Krylov method, this block Jacobi without overlap does not require any additional communication. However, even this is difficult to integrate into a CA method since after *Preco*, each local *SpMV* requires from its neighbors the preconditioned input vector elements on its interface, introducing extra communication. Depending on the sparsity structure of the matrix, a CA method may require significantly greater communication in order to use block Jacobi preconditioner. See [7, Chapter 7] for an illustration of this increasing communication requirement for a tridiagonal matrix that would result from a finite difference discretization of Poisson's equation with Dirichlet boundary conditions on a finite 1D domain. Previous authors proposed block Jacobi preconditioner, apparently without realizing the challenge of implementating it for a CA method. This was a surprising result that stirred us to develop the preconditioner framework presented in this paper. In Section IV, we discuss more details of these challenges.

For some types of problems, these preconditioners of the first category are not only difficult to integrate in a CA method, but it also may be only moderately effective in improving the convergence rate or in exploiting parallelism, or may introduce a significant overhead in computation or communication, depending on the sparsity structure of A. For example, the effectiveness of polynomial preconditioning, like that proposed in [12], to reduce the iteration count tends to decrease with the degree of the polynomial, while its computational cost increases. A recently proposed CA preconditioning technique based on an incomplete LU factorization, CA-ILU(0) [8], can be considered as an advanced member of this first category. Though CA-ILU(0) uses special global nested dissection ordering to limit the amount of required communication, the authors' experiments focus on structured grids, while leaving the extension to unstructured meshes as future work. Since no implementation of CA-ILU(0) is currently available, we postpone the empirical comparisons as our future study. We expect CA-ILU(0) would be more effective for a numerically difficult problem where the ILU(0) factors of the coefficient matrix can be partitioned well. However, CA-ILU(0) requires the ghosting of twice as many boundary layers, and our method may be more practical for a matrix with large separators.

Another critical aspect of these preconditioners in the first category is that they are still closely integrated with *MPK*. Hence, these preconditioners often require significant changes in how *MPK* interacts with the input vectors, and are designed

[1] http://keeneland.gatech.edu/KDS

GMRES(A, M, $\mathbf{b}$, m):
repeat (restart-loop)
1. Generate Krylov Basis: $O(m \cdot nnz(|A| + |M|) + m^2 n)$ flops on the GPUs.
   $\mathbf{q}_{:,1} = \mathbf{q}_{:,1}/\|\mathbf{q}_{:,1}\|_2$ (with $\widehat{\mathbf{x}} = \mathbf{0}$ and $\mathbf{q}_{:,1} = \mathbf{b}$, initially)
   for $j = 1, 2, \ldots, m$ **do**
   1.1. Preconditioner (*Preco*) Application:
   $\mathbf{z}_{:,j} := M^{-1}\mathbf{q}_{:,j}$
   1.2. Sparse Matrix-Vector (*SpMV*) Product:
   $\mathbf{q}_{:,j+1} := A\mathbf{z}_{:,j}$
   1.3. Orthonormalization (*Orth*):
   $\mathbf{q}_{:,j+1} := (\mathbf{q}_{:,j+1} - Q_{1:j}\mathbf{h}_{1:j,j})/h_{j+1,j}$,
   where $\mathbf{h}_{1:j,j} = Q_{1:j}^T\mathbf{q}_{:,j+1}$ and
   $h_{j+1,j} = \|\mathbf{q}_{:,j+1} - Q_{1:j}\mathbf{h}_{1:j,j}\|_2$.
   end for

2. Solve Projected Subsystem: $O(m^2)$ flops on the CPUs and $O(nm)$ flops on the GPUs.
   2.1. solve the least-squares problem $\mathbf{g} = \min_{\mathbf{t}} \|H\mathbf{t} - Q_{1:m+1}^T\mathbf{r}\|_2$
   where $\mathbf{r} = \mathbf{b} - A\widehat{\mathbf{x}}$
   2.2. to update solution $\widehat{\mathbf{x}} = \widehat{\mathbf{x}} + Z_{1:m}\mathbf{g}$
   2.3. restart with $\mathbf{q}_{:,1} = \mathbf{b} - A\widehat{\mathbf{x}}$
until solution convergence

Fig. 1. Pseudocode of GMRES on CPU/GPU. m is the restart length.

CA-GMRES(A, M, $\mathbf{b}$, s, m):
repeat (restart-loop)
1. Generate Krylov Basis: $O(m \cdot nnz(|A| + |M_j|) + m^2 n)$ flops on the GPUs.
   $\mathbf{q}_{:,1} = \mathbf{q}_1/\|\mathbf{q}_{:,1}\|_2$ (with $\widehat{\mathbf{x}} = \mathbf{0}$ and $\mathbf{q}_{:,1} = \mathbf{b}$, initially)
   for $j = 1, 1+s, \ldots, m$ **do**
   1.1. Matrix Powers Kernel (*MPK*):
   for $k = j+1, j+2, \ldots, j+s$ **do**
   $\mathbf{z}_{:,k} := M^{-1}\mathbf{q}_{:,k}$ (*Preco*)
   $\mathbf{q}_{:,k+1} := A\mathbf{z}_{:,k}$ (*SpMV*)
   end for
   1.2. Block Orthonormalization (*BOrth*):
   orthogonalize $Q_{j+1:j+s}$ against $Q_{1:j}$
   1.3. Tall-Skinny QR (*TSQR*) factorization:
   orthonormalizing $Q_{j+1:j+s}$ against each other
   end for

2. Solve Projected Subsystem: $O(m^2)$ flops on the CPUs and $O(nm)$ flops on the GPUs.
   2.1. solve the least-squares problem $\mathbf{g} = \min_{\mathbf{t}} \|H\mathbf{t} - Q_{1:m+1}^T\mathbf{r}\|_2$,
   where $\mathbf{r} = \mathbf{b} - A\widehat{\mathbf{x}}$
   2.2. update solution $\widehat{\mathbf{x}} = \widehat{\mathbf{x}} + Z_{1:m}\mathbf{g}$
   2.3. restart with $\mathbf{q}_{:,1} = \mathbf{b} - A\widehat{\mathbf{x}}$
until solution convergence

Fig. 2. Pseudocode of CA-GMRES on CPU/GPU. s is the *MPK* basis length and m is the restart length.

for specific types of preconditioners (e.g., approximate inverse or ILU(0)). On contrary, in this paper, we provide a framework that decouples *Preco* from *SpMV* and allows us to use any exisiting preconditioning software as a black-box routine for preconditioning CA methods.

III. CA-GMRES ON A HYBRID CPU/GPU CLUSTER

A. CA-GMRES Algorithm

The Generalized Minimum Residual (GMRES) method [13] is a Krylov subspace method for solving a nonsymmetric linear system. Its solution minimizes the residual norm over the generated projection subspace at each iteration. GMRES' j-th iteration first generates a new basis vector $\mathbf{q}_{:,j+1}$ by applying the preconditioner (*Preco*) to the previously orthonormalized basis vector $\mathbf{q}_{:,j}$, followed by the sparse-matrix vector product (*SpMV*) (i.e., $\mathbf{z}_{:,j} := M^{-1}\mathbf{q}_{:,j}$ and $\mathbf{q}_{:,j+1} := A\mathbf{z}_{:,j}$). Then, the new orthonormal basis vector is computed by orthonormalizing (*Orth*) the resulting vector $\mathbf{q}_{:,j+1}$ against the previously orthonormalized basis vectors $\mathbf{q}_{:,1}, \mathbf{q}_{:,2}, \ldots, \mathbf{q}_{:,j}$.

To reduce both the computational and storage requirements of computing a large projection subspace, users often restart GMRES after computing a fixed number $m+1$ of basis vectors. Before restart, GMRES updates the approximate solution $\widehat{\mathbf{x}}$ by solving a least-squares problem $\mathbf{g} := \arg\min_{\mathbf{t}} \|\mathbf{c} - H\mathbf{t}\|$, where $\mathbf{c} := Q_{1:m+1}^T(\mathbf{b} - A\widehat{\mathbf{x}})$, $H := Q_{1:m+1}^T A Z_{1:m}$, and $\widehat{\mathbf{x}} := \widehat{\mathbf{x}} + Z_{1:m}\mathbf{g}$. The matrix H, a by-product of the orthogonalization procedure, has upper Hessenberg form. Hence, the least-squares problem can be efficiently solved, requiring only about $3(m+1)^2$ flops. For an n-by-n matrix A with $nnz(A)$ nonzeros, and a preconditioner M whose application requires $nnz(M)$ flops, *SpMV*, *Preco*, and *Orth* require a total of about $2m \cdot nnz(|A|)$, $2m \cdot nnz(|M|)$, and $2m^3 n$ flops over the m iterations, respectively (i.e., $n, nnz(A), nnz(M) \gg m$). Figure 1 shows pseudocode for restarted GMRES.

Both *SpMV* and *Orth* require communication. This includes point-to-point messages or neighborhood collectives for *SpMV*, and global all-reduces in *Orth*, as well as data movement between levels of the local memory hierarchy (for reading the sparse matrix and for reading and writing vectors, assuming they do not fit in cache). Communication-Avoiding GMRES (CA-GMRES) aims to reduce this communication by redesigning the algorithm to replace *SpMV* and *Orth* with three new kernels – *MPK*, *BOrth*, and *TSQR* – that generate and orthogonalize a set of s basis vectors all at once. In theory, CA-GMRES communicates no more than a single GMRES iteration (plus a lower-order term), but does the work of s iterations. In Section IV, we propose a preconditioning technique that can be integrated into *MPK* without incurring any additional communication phases. Figure 2 shows pseudocode for restarted CA-GMRES.

B. CA-GMRES Implementation

Our CA-GMRES implementation on a hybrid CPU/GPU cluster extends our previous implementation on a multicore CPU with multiple GPUs on one node [17]. Namely, to utilize the multiple GPUs, we distribute the matrix A over the GPUs in a 1D block row format, using a matrix reordering or graph partitioning algorithm (see Section IV-B). The basis vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{s+1}$ are then distributed in the same format. Since CA-GMRES' computational cost is typically dominated by the first step of generating the basis vectors, we accelerate this step using GPUs, while the second step of solving the least squares problem is redundantly performed by each MPI process on CPU. On a hybrid CPU/GPU cluster with multiple GPUs on each node, a single MPI process can manage multiple GPUs on the node in order to combine or avoid the MPI communication to the GPUs on the same node. For the remaining of the paper, we use $A^{(d)}$ and $Q^{(d)}$ to denote the local matrices on the d-th GPU, while n_g is the number of available GPUs.

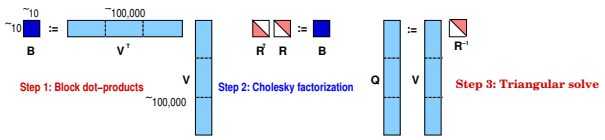
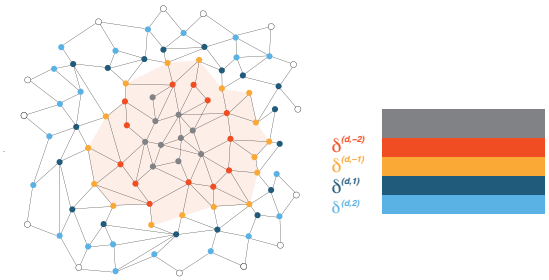


Fig. 3. Illustration of CholeskyQR Orthogonalization Process.



(a) Adjacency Graph of Local Submatrix. (b) Local Submatrix.

Fig. 4. Illustration of Extending and Sorting a Local Matrix for *MPK* ($s = 2$). The local subdomain is colored in light orange, while the vertices in the same color belong to either the underlap or overlap of the same level.

C. Orthogonalization Kernels

In our previous study [17], we investigated the performance of several orthogonalization procedures on multiple GPUs on a single compute node. In most cases in that study, CA-GMRES obtained the best performance using classical Gram-Schmidt (CGS) [1] and Cholesky QR (CholQR) [14] for *BOrth* and *TSQR*, respectively. Hence, in this paper, we use CGS-based *BOrth* and CholQR-based *TSQR* on the hybrid CPU/GPU cluster as well.

In CholQR, each GPU first computes the block dot products of its local vectors (i.e., $B^{(d)} := Q^{(d)T}_{1:s+1} Q^{(d)}_{1:s+1}$), using the optimized GPU kernels developed in [17], [18], and asynchronously copies the result to its MPI process. Second, the MPI process accumulates the results of its local GPUs, and computes the Gram matrix via a global MPI all-reduce (i.e., $B = \sum_{d=1}^{n_g} B^{(d)}$). Third, each MPI process redundantly computes the Cholesky factorization of the Gram matrix on the CPUs (i.e., $RR^T := B$), and broadcasts the Cholesky factor R to its local GPUs. Finally, each GPU orthogonalizes the local part of the basis vectors through a triangular solve (i.e., $Q^{(d)}_{1:s+1} := Q^{(d)}_{1:s+1} R^{-1}$). Figure 3 illustrates this process.

CGS is implemented similarly, and is based on optimized matrix-matrix multiplies on a GPU and orthogonalizes the next set of columns against all the previous columns at once [17].

D. Matrix Powers Kernel

We denote the *local* submatrix assigned to the d-th GPU by $\bar{A}^{(d)}$, and reorder the rows of $\bar{A}^{(d)}$ in the descending order of their edge distances from the subdomain boundary in the adjacency graph of A. We then expand the local submatrix to include the *nonlocal* entries that are at most ℓ edges away from the local matrix for $\ell = 1, 2, \ldots, s$. More specifically, for $\ell > 0$, let $\boldsymbol{\delta}^{(d,\ell)}$ be the set of the nonlocal vertices whose shortest path from a local vertex is of length ℓ, while for $\ell < 0$, $\boldsymbol{\delta}^{(d,\ell)}$ is the set of the local vertices whose shortest path from a nonlocal vertex is of length ℓ (see Figure 4 for an illustration). Then, the d-th GPU owns its extended local submatrix $\bar{A}^{(d,s)} = A(\mathbf{i}^{(d,s)}, :)$, where $\mathbf{i}^{(d,s)} = \bigcup_{\ell \le s} \boldsymbol{\delta}^{(d,\ell)}$ and the row index set $\mathbf{i}^{(d,s)}$ is sorted such that those row indexes in $\boldsymbol{\delta}^{(d,\ell)}$ with a smaller ℓ come first. For example, in Figure 4, we store the block rows of the local submatrix, that are colored in gray, red, orange, dark blue, and then light blue in that order. For the rest of the paper, we let $\ell > 0$, and refer to $\boldsymbol{\delta}^{(d,-\ell)}$ and $\boldsymbol{\delta}(d,\ell)$ as the ℓ-level *underlap* and *overlap*, respectively, of the d-th subdomain.

When applying *MPK*, each GPU first exchanges all the required vector elements to compute s matrix powers with its neighboring GPUs distributed over different MPI processes. To this end, each GPU first packs its local vector elements that are needed by the neighboring GPUs into a buffer, which is then asynchronously copied to the CPU. Once the MPI process receives the message from its GPU, it expands it into a full-length vector. After expanding the messages from all the local GPUs, the MPI process packs the vector elements required by another GPU into a single message, and asynchronously sends it to the corresponding MPI process. Finally, when the MPI process receives a message from another process, it expands the messages into a full-length vector, and after expanding all the messages, it packs and copies the required elements to its GPUs, which then expand the packed elements into their full-length vectors (see Figure 5 for the pseudocode).[2] After this inter-GPU communication, each GPU computes the k-th matrix power by independently invoking *Preco* and *SpMV* with the k-th extended local matrix $\bar{A}^{(d,\ell)}$ without further inter-GPU communication, where $\ell = s - k + 1$ for $k = 1, 2, \ldots, s$ (see Step 1.1 in Figure 2).

A larger step size s increases the computational and storage overheads of *MPK*, and may increase its communication volume. As a result, the optimal value of s for *MPK* may be smaller than the optimal s for *BOrth* or *TSQR*. This has been also observed in [10], where in some cases, *MPK*'s optimal step size was one. In addition, as we will discuss in Section IV, the quality of our CA preconditioner can degrade with a larger value of s. It thus becomes critical to use a relatively small s for *Preco*, especially with a large number of subdomains, or equivalently on a large number of GPUs (see Section IV-B for our choice of s). To improve the performance of CA-GMRES, our implementation can use different values of s for *MPK*, and for *BOrth* and *TSQR* [18]. Hence, our implementation can take three input parameters, CA-GMRES$(s, \hat{s}, m)$, where s and $\hat{s}$ are the step sizes used for *MPK* and for *BOrth* and *TSQR*, respectively, and m is the restart length. The special case of $s = 1$ means that CA-GMRES does not need a specialized *MPK* implementation; it merely uses the standard *SpMV*, and relies on CA-GMRES' orthogonalization kernels

[2]When we have more CPU cores than GPUs on a node, we assign multiple cores to each MPI process which can then process the messages from multiple GPUs in parallel using Pthreads.

`Setup`: exchange elements of local $\mathbf{q}_{:,1}^{(d)}$ to form global $\mathbf{q}_{:,1}^{(d,s)}$
// GPU-to-CPU communication, using CUDA
for each local d-th GPU **do**
  compress elements of $\mathbf{q}_{:,1}^{(d)}$ needed by other GPUs into $\mathbf{w}^{(d)}$
  asynch-send of $\mathbf{w}^{(d)}$ to this MPI process
end for
for each local d-th GPU **do**
  wait and expand $\mathbf{w}^{(d)}$ into a full vector $\mathbf{w}$ on CPU
end for
// CPU-to-CPU communication, using MPI
for each non-local d-th GPU **do**
  if any of local elements is needed by d-th GPU **then**
    compress elements of $\mathbf{w}$ required by d-th GPU into $\mathbf{w}^{(d)}$
    asynch-send of $\mathbf{w}^{(d)}$ to the MPI process owning d-th GPU
  end if
  if any local elements of d-th GPU is needed by local GPUs **then**
    asynch-receive from the MPI owning d-th GPU into $\mathbf{z}^{(d)}$
  end if
end for
for each non-local d-th GPU **do**
  wait and expand $\mathbf{z}^{(d)}$ into a full vector $\mathbf{z}$ on CPU
end for
// CPU-to-GPU communication, using CUDA
for each local d-th GPU **do**
  compress elements of $\mathbf{z}$ required by d-th GPU into $\mathbf{z}^{(d)}$
  asynch-send $\mathbf{z}^{(d)}$ to d-th GPU
  copy the local vector $\mathbf{q}_{:,1}^{(d)}$ into $\mathbf{q}_{\mathbf{i}^{(d,0)},1}^{(d,s)}$
  expand $\mathbf{z}^{(d)}$ into a full vector $\mathbf{q}_{:,1}^{(d,s)}$
end for

`Matrix Powers`: generate local $\mathbf{q}_{:,2}^{(d)}, \mathbf{q}_{:,3}^{(d)}, \ldots, \mathbf{q}_{:,s+1}^{(d)}$
for $k = 1, 2, \ldots, s$ **do**
  $\ell := s - k + 1$
  for $d = 1, 2, \ldots, n_g$ **do**
    Preco: compute $\mathbf{z}_{:,k}^{(d,\ell)} := (R^{(d,\ell)})^T (M^{(d,\ell)})^{-1} (R^{(d,\ell)}) \mathbf{q}_{:,k}^{(d,\ell)}$
    SpMV: compute $\mathbf{q}_{:,k+1}^{(d,\ell-1)} := (R^{(d,\ell-1)})^T \bar{A}^{(d,\ell)} (R^{(d,\ell)}) \mathbf{z}_{:,k}^{(d,\ell)}$
    save to the local vector, $\mathbf{q}_{:,k+1}^{(d)} = \mathbf{q}_{\mathbf{i}^{(d,0)},k+1}^{(d,\ell-1)}$
  end for
end for

Notation used for *MPK*:

$\boldsymbol{\delta}^{(d,-\ell)}$	: local vertices that are ℓ edges away from boundary
$\boldsymbol{\delta}^{(d,\ell)}$	: nonlocal vertices that are ℓ edges away from boundary
$\boldsymbol{\delta}^{(d,1:s)}$	: s-level overlap, i.e., $\bigcup_{\ell=1,2,\ldots,s} \boldsymbol{\delta}^{(d,\ell)}$
$\mathbf{i}^{(d,s)}$	: s-level row index set, i.e., $\bigcup_{\ell \le s} \boldsymbol{\delta}^{(d,\ell)}$
$\bar{A}^{(d,s)}$	: s-level extended local submatrix, i.e., $\bar{A}^{(d,s)} = A(\mathbf{i}^{(d,s)}, :)$
$\bar{A}^{(d)}$,	: local submatrix on d-th GPU, i.e., $\bar{A}^{(d,0)}$
$A^{(d,s)}$	: s-level diagonal block, i.e., $A^{(d,s)} = A(\mathbf{i}^{(d,s)}, \mathbf{i}^{(d,s)})$
$R^{(d,s)}$	: restriction from global domain to s-level local subdomain
$M^{(d,s)}$	: a local preconditioner on square portion of the s-level extended local submatrix

Fig. 5. Pseudocode of Matrix Powers Kernel, *MPK*(s, $\mathbf{q}_{:,1}$), where s is the number of basis vectors that *MPK* generates.

to improve its performance over GMRES (no communication is still needed between *Preco* and *SpMV*). This is a reasonable strategy for long-recurrence Krylov solvers like GMRES, but CA variants of short-recurrence methods like CG spend much less time in inner products, and may require a larger step size for the performance improvement.

E. Performance Studies

Figure 6 compares the parallel strong scaling performance of GMRES and CA-GMRES on up to 120 GPUs by showing

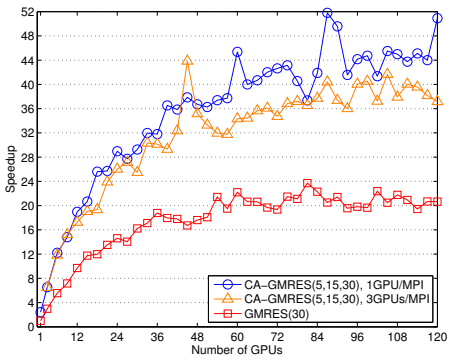


Fig. 6. Parallel Strong Scaling of CA-GMRES and GMRES on 120 distributed GPUs (over GMRES on one GPU), for the `G3_Circuit` matrix.

Method	No. of GPUs	6	12	24	48
GMRES(30)	No. of Restarts	39	35	130	79
	Time (s)	3.35	3.13	10.09	7.97
CA-GMRES$(1, 30)$	No. of Restarts	39	35	131	79
	Times (s)	1.75	1.64	7.00	4.55
	Speedup	1.91	1.91	1.44	1.75

Fig. 7. Parallel Weak Scaling Performance Studies for the `brick` matrices, starting with $n =$ 1M on 6 GPUs to $n =$ 8M on 48 GPUs using 3GPUs/MPI.

their total solution time speedups over the time required by GMRES on one GPU for the `G3_Circuit` matrix (see Figure 9 for the properties of our test matrices). The matrix A is distributed among the GPUs such that each GPU has about an equal number of rows after the reverse Cuthill-McKee (RCM) ordering is applied [6] (see Section IV-C for more detailed experimental setups, except the greater stopping criteria of 10^{-8} is used here). Assigning one GPU per MPI process (the blue markers in the figure), CA-GMRES obtained the average and maximum speedups of 2.06 and 2.53 over GMRES on the same number of GPUs, respectively. The speedup leveled off around 60 GPUs because the local submatrix became too small for the GPU to obtain any strong scaling speedup. On a larger number of compute nodes, the MPI communication becomes more dominant, and the speedups obtained by avoiding the communication may increase [19]. In addition, the orange markers in the figure show the performance of CA-GMRES that launches one MPI process on each node and lets each process manage the three local GPUs on the node. At least in our experiments, the overhead of each MPI to manage multiple GPUs (e.g., sequentially launching GPU kernels on multiple GPUs) outweighed the benefit of avoiding the intra-node communication, for which many MPI implementations are optimized. Hence, for the rest of the paper, unless otherwise specified, we use one MPI process to manage a single GPU.

Figure 7 shows the parallel weak scaling performance of GMRES and CA-GMRES, where the matrix dimension is increased linearly with the number of GPUs, starting from $1,035,351$ on 6 GPUs. To accomodate the large CPU memory usage during setup on 48 GPUs, we lauched one MPI process per node and let the process manage three GPUs on the node. The `brick` matrices come from the discretization of

the Poisson's equation with Dirichlet boundary conditions on a 3-D brick-shaped mesh, in which the number of elements in two dimensions are fixed. The third dimension has four different types of material blocks, two of which have the element sizes graded. When the problem size increases, the difficulty of the problem varies depending on the dimension that was scaled up. This results in the different number of restarts as we increase the problem size. Again, CA-GMRES provided better weak scaling than GMRES on this hybrid CPU/GPU cluster.

IV. CA Domain Decomposition Preconditioners

As discussed in Section I, preconditioning CA Krylov methods is difficult. Instead of forming the basis vectors for the Krylov subspace $\mathcal{K}_k(A, \mathbf{q}_{:,1}) = \text{span}\{\mathbf{q}_{:,1}, A\mathbf{q}_{:,1}, \ldots, A^{k-1}\mathbf{q}_{:,1}\}$, we must generate the basis vectors for the preconditioned subspace $\mathcal{K}_k(M^{-1}A, \mathbf{q}_{:,1})$ or $\mathcal{K}_k(AM^{-1}, \mathbf{q}_{:,1})$ (left or right preconditioning, respectively). The challenge is that to compute the local product with the k-th matrix power, each processor (or GPU) needs to know not only its local elements of the input vector at $\mathbf{i}^{(d,0)}$ but also those at the ℓ-level overlap $\boldsymbol{\delta}^{(d,1:\ell)}$, where $\ell = s - k + 1$ (for $k = 1, 2, \ldots, s$). Therefore, when applying the preconditioner (i.e., $\mathbf{z}_{:,k} := M^{-1}\mathbf{q}_{:,k}$ and $\mathbf{i}^{(d,\ell)} = \mathbf{i}^{(d,0)} \bigcup \boldsymbol{\delta}^{(d,1:\ell)}$), if the action of M^{-1} to generate these vector elements of $\mathbf{z}_{:,k}$ at $\mathbf{i}^{(d,\ell)}$ requires any element of the input vector $\mathbf{q}_{:,k}$ aside from those at the row index set $\mathbf{i}^{(d,\ell)}$, then additional communication is needed.

A simple preconditioner that works for CA methods is a diagonal preconditioner. For this, the d-th processor (or GPU) only needs to know the diagonals of the local submatrix and the s-level overlap, i.e., $\text{diag}(A^{(d,s)})$. This requires only small computational and storage overheads, but may only reduce the iteration count moderately. Real applications often prefer other types of preconditioners that have higher overheads but are more effective in reducing the iteration count. These include preconditioners based on domain decomposition or multigrid. We focus on domain decomposition preconditioners which are local in nature, useful in many applications and well suited for parallel computing.

A. CA-Preconditioning Framework

We now describe our communication avoiding domain decomposition preconditioners. As explained in Section II, it is difficult to integrate even a block Jacobi preconditioner into a CA method, where each processor (or GPU) independently applies *Preco* by solving its local problem associated with the local diagonal block $A^{(d)}$. This is because after the preconditioner is applied, each local *SpMV* requires from its neighbors the preconditioned input vector elements on the overlaps, introducing extra communication. To avoid this additional communication, we "shrink" the diagonal blocks to make them disjoint from the s-level overlaps. To define our preconditioners, recall our notations in Section III-D; in the view of the d-th subdomain, the distance-s neighbors of the vertices in the graph of A is the s-level overlap $\boldsymbol{\delta}^{(d,s)}$, while the set of

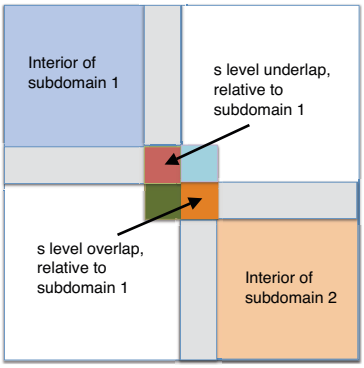


Fig. 8. Matrix Partitioning for the CA Preconditioner for two subdomains. The underlap and the overlap relative to subdomain 1 are shown.

local vertices distance-s away from a non-local vertex is the s-level underlap $\boldsymbol{\delta}^{(d,-s)}$. To simplify our notation, when it is clear from the context, we use $\mathbf{i}^{(-\ell_1)}$ and $\boldsymbol{\delta}^{(\ell_1:\ell_2)}$ to represent $\mathbf{i}^{(d,-\ell_1)}$ and $\boldsymbol{\delta}^{(d,\ell_1:\ell_2)}$, where $\mathbf{i}^{(d,-\ell_1)} = \bigcup_{\ell \le -\ell_1} \boldsymbol{\delta}^{(d,\ell)}$ and $\boldsymbol{\delta}^{(d,\ell_1:\ell_2)} = \bigcup_{\ell_1 \le \ell \le \ell_2} \boldsymbol{\delta}^{(d,\ell)}$. Then, the square s-level extended local submatrix $A^{(d,s)}$ has the following block structure:

$$\begin{pmatrix} A(\mathbf{i}^{(-s-1)}, \mathbf{i}^{(-s-1)}) & A(\mathbf{i}^{(-s-1)}, \boldsymbol{\delta}^{(-s:-1)}) & \\ A(\boldsymbol{\delta}^{(-s:-1)}, \mathbf{i}^{(-s-1)}) & A(\boldsymbol{\delta}^{(-s:-1)}, \boldsymbol{\delta}^{(-s:-1)}) & A(\boldsymbol{\delta}^{(-s:-1)}, \boldsymbol{\delta}^{(1:s)}) \\ & A(\boldsymbol{\delta}^{(1:s)}, \boldsymbol{\delta}^{(-s:-1)}) & A(\boldsymbol{\delta}^{(1:s)}, \boldsymbol{\delta}^{(1:s)}) \end{pmatrix}.$$

The global view of $A^{(d,s)}$ for two subdomains is shown in Figure 8. For example, in the figure, $A^{(1,s)}$ consists of the interior of subdomain 1, its s-level underlap and the edges to the interior of subdomain 1, its s-level overlap and the edges to the s-level underlap. Given $|A| = n$ and $\left|A^{(d,\ell)}\right| = n^{(d,\ell)}$, we define the standard rectangular n-by-$n^{(d,0)}$ extension matrix $(R^{(d,0)})^T$ with zeros and ones, which extends by zero the local vector associated with vertices of the local submatrix $A^{(d,0)}$ to form a global vector. The corresponding s-level variant $(R^{(d,s)})^T$ is defined in the same fashion. Correspondingly, $R^{(d,s)}$ is the restriction matrix that restricts a vector from the global domain to the s-level local subdomain. With that notation, the restricted additive Schwarz preconditioner [4] with s-level overlap becomes:

$$M_{RAS}^{-1} = \sum_{d=1}^{n_g} ((R^{(d,0)})^T)(A^{(d,s)})^{-1}(R^{(d,s)}),$$

where n_g is the number of the non-overlapping subsets of the row index set of A (i.e., the number of subdomains or GPUs). By varying the amount of overlap at each iteration, we obtain a sequence of s different preconditioners that we refer to as the s-step preconditioner; at the k-th iteration, we have

$$(M_{RAS}^{(k)})^{-1} = \sum_{d=1}^{n_g} ((R^{(d,0)})^T)(A^{(d,\ell)})^{-1}(R^{(d,\ell)}),$$

where $\ell = s - k + 1$ for $k = 1, 2, \ldots, s$. This is similar to the restricted additive Schwartz preconditioner. However, the s-step preconditioner changes at each iteration, shrinking both its underlap and overlap to match with the extended

Name	Source	n	nnz/n
G3_Circuit	UF Collection	$1,585,478$	4.8
PDE_1M(α)	Trilinos	$1,030,301$	26.5
PDE_10M(α)	Trilinos	$10,218,313$	26.8
brick_n	Trilinos	$\sim n$	~ 25

Fig. 9. Test Matrices.

local submatrix $A^{(d,\ell)}$ of *MPK*. While the restriction operator changes accordingly, the extension operator uses the same non-overlapping extension operator allowing unique updates from the sub-domains. It is easy to see that in the special case of $s = 0$, the s-step preconditioner reduces to the block Jacobi preconditioner.

In order to use the above framework as a stationary preconditioner for CA-GMRES, we need to provide a consistent view of the preconditioner both across subdomains (for correctness) and across iterations (for being stationary). To be consistent across subdomains, our implementation considers only the diagonal blocks of $A^{(d,s)}$ and uses diagonal preconditioning for the two diagonal blocks $A(\boldsymbol{\delta}^{(-s:-1)}, \boldsymbol{\delta}^{(-s:-1)})$ on the underlap and $A(\boldsymbol{\delta}^{(1:s)}, \boldsymbol{\delta}^{(1:s)})$ on the overlap. Then, to be consistent across iterations, we use a constant underlap s. Hence, at the k-th iteration, our *underlap* preconditioner is defined as

$$(M_{UN}^{(k)})^{-1} = \sum_{d=1}^{n_g} ((R^{(d,0)})^T)(\widehat{A}^{(d,\ell)})^{-1}(R^{(d,\ell)}),$$

where $\ell = s - k + 1$ for $k = 1, 2, \ldots, s$, and

$$\widehat{A}^{(d,\ell)} = \begin{pmatrix} A(\mathbf{i}^{(-s-1)}, \mathbf{i}^{(-s-1)}) & \\ & \text{diag}(A(\boldsymbol{\delta}^{(-s:\ell)}, \boldsymbol{\delta}^{(-s:\ell)})) \end{pmatrix}.$$

The restriction operator still shrinks the overlap not to incur CA-GMRES any additional communication cost.

In summary, our underlap preconditioner is a special case of the s-step domain decomposition preconditioner outlined above. Since we use diagonal preconditioners for both the underlap and overlap regions, an equivalent formulation of the local preconditioner $\mathscr{M}_{UN}^{(d,s)}$ corresponding to $\widehat{A}^{(d,s)}$ is

$$\mathscr{M}_{UN}^{(d,s)} = \begin{pmatrix} A(\mathbf{i}^{(-s-1)}, \mathbf{i}^{(-s-1)}) & \\ & \text{diag}(A(\boldsymbol{\delta}^{(-s:-1)}, \boldsymbol{\delta}^{(-s:-1)})) \end{pmatrix}.$$

One may use any traditional local subdomain preconditioner for an inexact solution of $A(\mathbf{i}^{(-s-1)}, \mathbf{i}^{(-s-1)})$, including incomplete factorizations or a fixed number of iterations of a stationary method such as Jacobi or Gauss-Seidel. This formula defines the preconditioner's action on each local subdomain. In addition, at the k-th iteration, each processor redundantly computes the diagonal preconditioner's action on the ℓ-level overlap $\boldsymbol{\delta}^{(1:\ell)}$ in a shrinking fashion. Mathematically, the preconditioner is fixed; it does not change across iterations.

B. Experimental Setup

We tested two matrix reordering algorithms to distribute the matrix A among the GPUs: reverse Cuthill-McKee (RCM) [6] from HSL[3], and k-way graph partitioning (KWY) from METIS[4]. With RCM, after reordering, we distribute the matrix so that each GPU has about an equal number of rows. As the local solver for our underlapping preconditioner, we investigated stationary iterative methods (Jacobi or Gauss-Seidel), the level or drop-tolerance based incomplete LU factorization, ILU(k) or ILU(τ), respectively, of ITSOL[5], and the sparse approximate inverse (SAI) of ParaSails[6]. Each MPI process independently computes its local preconditioner on the CPU and copies it to the GPU. Then, at each iteration, the MPI process applies the preconditioner (*Preco*) using cuSPARSE's sparse-matrix vector multiplies or sparse triangular solves in the Compressed Sparse Row (CSR) matrix storage format. It performs sparse matrix-vector multiplies (*SpMV*) with the matrix A using our own GPU kernel in the ELLPACKT format [17]. To maintain orthogonality, we always orthogonalize the basis vectors twice. We consider the computed solution to have converged when the residual ℓ_2-norm is reduced by at least twelve orders of magnitude.

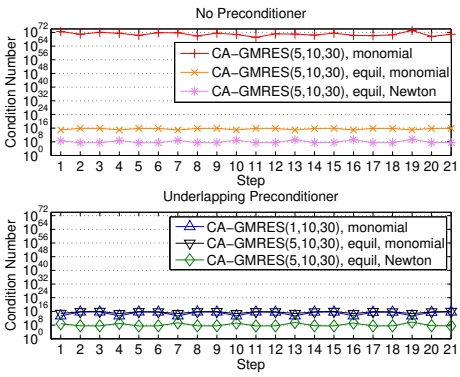
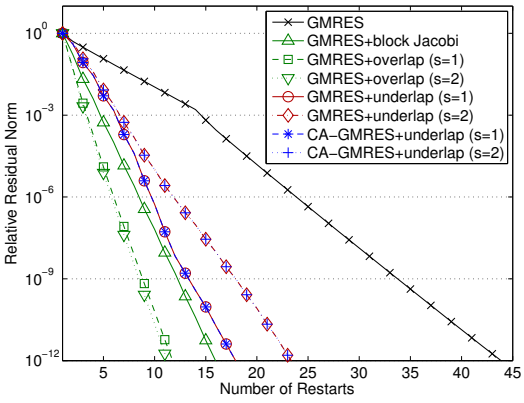


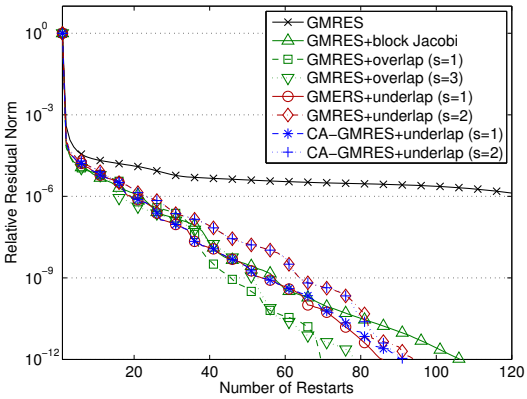
Fig. 10. Condition Number of Gram Matrix, `G3_Circuit` matrix, 6 GPUs.

Figure 9 shows the properties of the test matrices used for our experiments. The `G3_Circuit` matrix comes from a circuit simulation problem. Such matrices are difficult to precondition; doing so effectively is current research. The "PDE" problem comes from a scaling example in the TrilinosCouplings package of the Trilinos library. It arises from discretizing Poisson's equation with Dirichlet boundaries on a cube Ω, using a regular hexahedral mesh. The PDE is $\text{div}(T\nabla u) = f$ in Ω, $u = g$ on $\partial\Omega$, where T is a 3-by-3 material tensor, and f and g are given functions. Discretizing results in a linear system $Ax = b$, which is symmetric as long as T is. We set $T = \begin{pmatrix} 1 & 0 & \alpha \\ 0 & 1 & 0 \\ \alpha & 0 & 1 \end{pmatrix}$ and control the iteration count by varying α. When $\alpha = 0$, the problem takes few iterations. As α approaches 1, the problem takes more. For $\alpha > 1$, the

[3]http://www.hsl.rl.ac.uk/catalogue/mc60.xml
[4]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
[5]http://www-users.cs.umn.edu/~saad/software/ITSOL/index.html
[6]http://computation.llnl.gov/casc/parasails/parasails.html

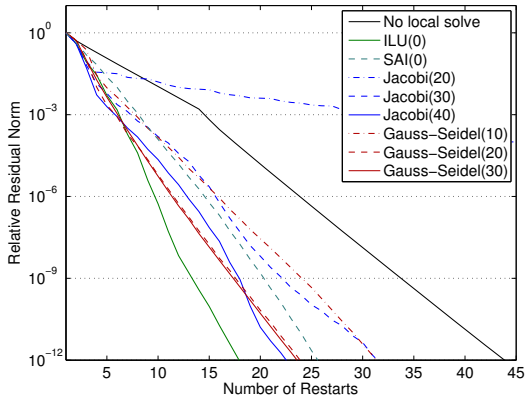


(a) PDE_1M(0.0) matrix, with restart = 20.

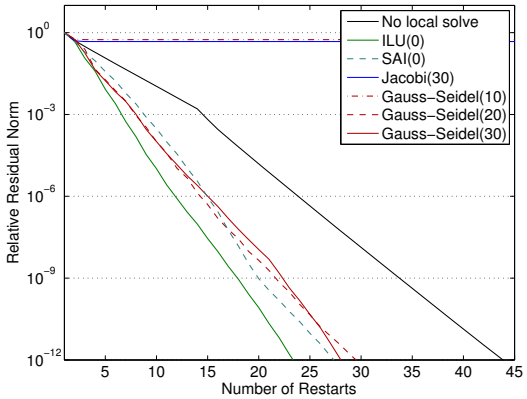


(b) G3_Circuit matrix, with restart = 30.

Fig. 11. Solution Convergence, using Different Domain Decomposition Preconditioners with Local ILU(0)'s on 6 GPUs.



(a) CA-GMRES(1, 20), for the PDE_1M(0.0) matrix.



(b) CA-GMRES(2, 20), for the PDE_1M(0.0) matrix.

Fig. 12. Solution Convergence, using Different Local Solvers for an Underlapping Preconditioner on 6 GPUs.

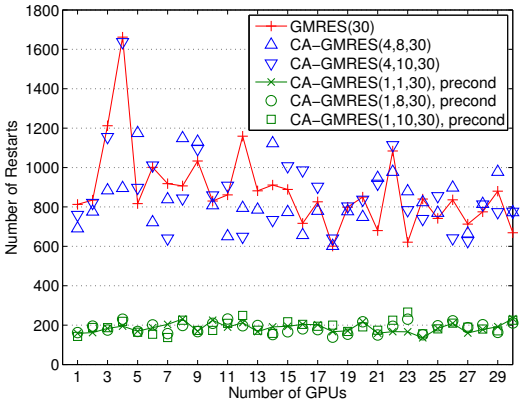
matrix A is no longer positive definite. We present results for different values of α.

In our experiments, we used the parameters which gave good performance of CA-GMRES without preconditioning in the previous experiments ($m = 20 \sim 60$ and $s = 1 \sim 5$) [17]. In many cases, this step size s gave good performance even with our preconditioner. However, for some problems, the iteration count of preconditioned CA-GMRES increased quickly as the number of subdomains increased. Hence, we reduced the step size to obtain more consistent reduction in the iteration count over a large number of subdomains.

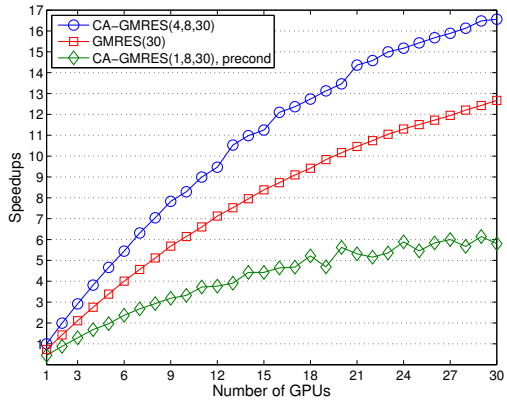
To enhance CA-GMRES' numerical stability, before solving, we equilibrate A and b [9]. That is, we first scale its rows, and then its columns, by their ∞-norms. We also use a Newton basis $\mathbf{q}_{:,k+1} = (AM^{-1} - \theta_k I)\mathbf{q}_{:,k}$, where the shifts θ_k are the eigenvalues of the first restart's Hessenberg matrix H, in a Leja order [2]. Figure 10 shows the condition number of the Gram matrices generated during CholQR for the G3_Circuit matrix using CA-GMRES($\widehat{s}, s, m$). These condition numbers are the square of the condition numbers of the basis vectors generated by *MPK* (see Section III-C). The figure shows that equilibration, using the Newton basis, and preconditioning all improve the condition numbers. While in our experiments, we used the same step size and orthogonalization parameters for testing CA-GMRES with and without preconditioning, we could potentially use a larger s or omit reorthogonalization when preconditioning. This may significantly increase the benefit of preconditioning.

C. Convergence Studies with a Fixed GPU Count

Figure 11 compares convergence history of the relative residual norm $\|\mathbf{b} - A\widehat{\mathbf{x}}\|_2/\|\mathbf{b}\|_2$, using different preconditioners on six GPUs, where "overlap" preconditioners refer to restricted additive Schwarz preconditioners, and the local solver is level-based ILU(0). For instance, for the PDE matrix in Figure 11(a), as expected, a larger overlap reduced the number of iterations required for the solution convergence, while a larger underlap increased it. However, either an overlap



(a) Number of Restarts.



(b) Average Restart Time (over CA-GMRES on One GPU).

Fig. 13. Parallel Strong Scaling Performance on Distributed GPUs of CA-GMRES using an Underlapping Preconditioner with Local ILU(0)'s, for the `G3_Circuit` matrix.

or underlap preconditioner significantly reduced the iteration count. In addition, CA-GMRES' convergence matches that of GMRES. For the `G3_Circuit` matrix, Figure 11(b) shows similar results, but for this more ill-conditioned system, the reduction in the iteration count was much greater.

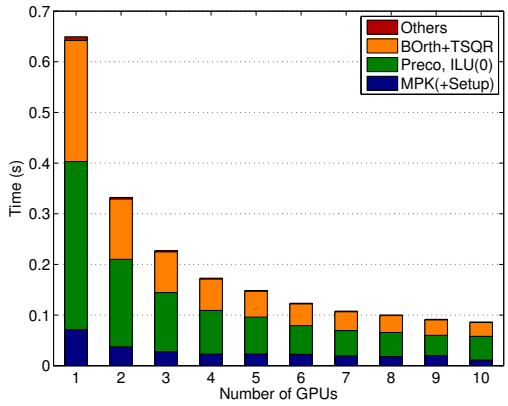
Figure 12 shows convergence, when we used different local solvers in combination with our underlapping preconditioner on six GPUs. For instance, we considered Jacobi iteration as a local solver, because on both CPUs and (esp.) GPUs, the sparse triangular solve that ILU requires is often much slower than the *SpMV* that *MPK* or Jacobi uses (see Section IV-D). Unfortunately, even for this relatively well-conditioned system, in order to match ILU(0)'s convergence, the Jacobi method required many iterations when $s = 1$, and it did not converge when $s = 2$. Even though Gauss-Seidel needed fewer iterations, the iteration count was still large, especially considering that each Gauss-Seidel iteration performs a (forward) triangular solve. Only the sparse approximate inverse (SAI) was competitive with ILU(0) on this small number of GPUs. However, SAI may not be effective on a larger number of GPUs, or for an ill-conditioned system such as the `G3_Circuit` matrix.

D. Parallel Scaling Studies

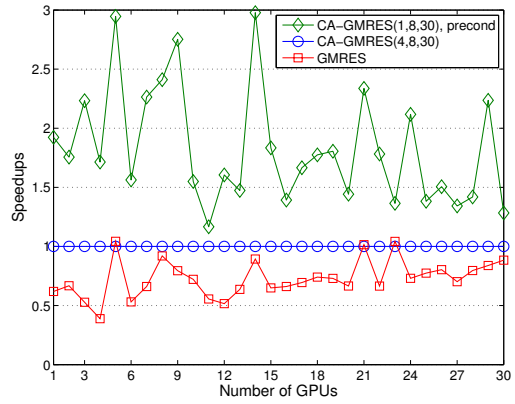
We first examine the performance of our underlapping preconditioner for the `G3_Circuit` matrix. This is a relatively ill-conditioned system and a sparse approximate inverse is not an effective preconditioner. Hence, we used ILU(0) as our local solver. The performance of cuSPARSE's sparse triangular solver depends strongly on the sparsity pattern of the triangular factor [11]. For our experiments, we first used a k-way graph partitioning to distribute the matrix among the GPUs, and each local submatrix is then reordered using a nested dissection algorithm. We have observed that the performance of the triangular solver can be significantly improved using the nested dissection ordering (e.g., a speedup of 1.56). Figure 14(a) shows the breakdown of the average time spent in one restart loop. Even with the nested dissection ordering, though *Preco* performs fewer floating-point operations than *MPK*, *Preco* required significantly longer time since the triangular solution is inherently serial (e.g., by a factor of 4.67, where *MPK* includes the setup time). As a result, Figure 13(b) shows that the time per iteration was significantly longer using the underlap preconditioner (e.g., by a factor of 3.41). However, Figure 13(a) shows that the preconditioner significantly reduced the iteration count, and Figure 14(b) illustrates that the total solution time was also greatly reduced using the preconditioner (e.g., by a factor of 2.95). Figure 16 shows the detailed performance results including the time and iteration count for the `G3_Circuit` matrix on up to 30 GPUs.

Compared to the `G3_Circuit` matrix, our `PDE` matrices are relatively well-conditioned, and SAI(0) is often effective as the local solver. Figure 15(a) shows the average breakdown of the restart loop time for the PDE problem. The time to apply the SAI(0) preconditioner was about the same as that of *SpMV* with the local submatrix and was significantly shorter than that of the sparse triangular solve required by ILU(0) (see Figure 14(a)). This can be seen from the relatively similar preconditioner apply times, compared to the *SpMV* time, in Figure 15(a). As a result, compared with Figure 14(b) for the `G3_Circuit` matrix, Figure 15(b) shows greater speedups obtained using the underlap preconditioner for the `PDE` matrices (e.g., by a factor 7.4).

Figure 17 shows the detailed performance profiles with different `PDE` matrices both in terms of the size and in terms of difficulty to solve, on up to 27 GPUs. The `PDE_1M(0.0)` matrix is positive definite, while the `PDE_1M(1.0275)` matrix is indefinite. In order to accelerate the converge for this indefinite problem, compared to $m = 20$ used for $\alpha = 0.0$, a larger restart length of $m = 60$ was used when $\alpha = 1.0275$. To accomodate the large CPU memory required to set up the `PDE_10M` matrices, we launched one MPI per node and let each process manage three GPUs on the node. The figure also shows that CA-GMRES with underlap

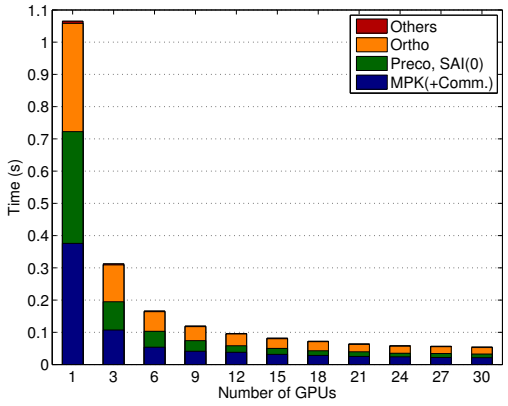


(a) Breakdown of Average Restart Time, CA-GMRES$(1, 8, 30)$.

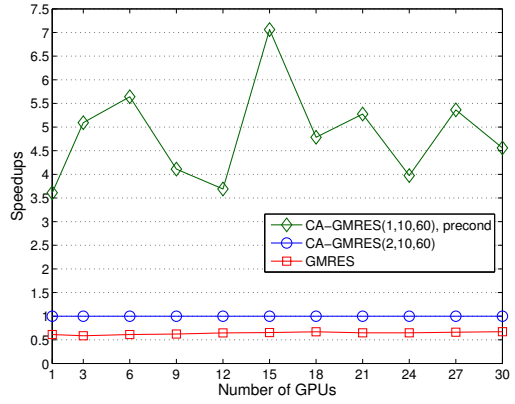


(b) Total Solution Time Speedup (over CA-GMRES).

Fig. 14. Parallel Strong Scaling Performance on Distributed GPUs of GMRES, CA-GMRES, and CA-GMRES using an Underlapping Preconditioner with Local ILU(0)'s, for the `G3_Circuit` matrix.



(a) Breakdown of Average Restart Time, CA-GMRES$(1, 10, 60)$.



(b) Total Solution Time Speedup (over CA-GMRES).

Fig. 15. Parallel Strong Scaling Performance on Distributed GPUs of GMRES, CA-GMRES, and CA-GMRES using an Underlapping Preconditioner with Local SAI(0)'s, for the `PDE_1M(1.0275)` matrix.

preconditioner improves the performance over GMRES with the same preconditioner (e.g., by a factor of 1.7). We expect that this improvement will increase on a computer where the communication between the parallel processes or threads is more dominant and for larger problem sizes.

We emphasize that the matrix A's distribution over the GPUs not only affects CA-GMRES' performance (e.g., storage, computation, and communication overheads associated with s-step overlap), but it also determines the preconditioner's effectiveness (e.g., the size of the s-step underlap). For the `PDE` matrices presented here, we distributed the matrix over GPUs using k-way graph partitioning. In contrast, when we used RCM ordering, the sizes of both overlap and underlap increased quickly with the number of subdomains. As a result, even on few (e.g., 21) GPUs, some underlaps extended all the way inside the local subdomains, and the local preconditioners became diagonal scaling. Hence, on a larger number of subdomains, not only did the computational and communication overheads of CA-GMRES increase, but also the number of iterations increased quickly. This led to a quick reduction in speedups gained using the underlap preconditioner on a larger number of GPUs.

V. Conclusion

This paper presents our implementation of CA-GMRES on a hybrid CPU/GPU cluster and demonstrates speedups of up to 2.5x over standard GMRES on up to 120 GPUs. We also proposed a novel framework based on domain decomposition to precondition CA Krylov methods, which allowed us to use existing software libraries for local preconditioning. Even when our implementation of this framework used simple ILU(0) and SAI(0) preconditioners to solve on each GPU's domain, the number of GMRES iterations was greatly reduced. As a result our preconditioned CA-GMRES was able to achieve speedups of up to 7.4x in total solution time over CA-GMRES and up to a 1.7x speedup in total solution time over

Number of GPUs	1	3	6	9	12	15	18	21	24	27	30
GMRES(30)	326 (813)	168 (1212)	73 (1001)	53 (1033)	48 (1159)	31 (889)	19 (601)	19 (680)	22 (840)	17 (713)	15 (669)
CA-GMRES(6, 30)	347 (1026)	116 (995)	142 (2257)	44 (1001)	26 (712)	29 (970)	23 (860)	25 (1027)	19 (823)	25 (1154)	19 (939)
CA-GMRES(8, 30)	202 (691)	89 (884)	39 (722)	42 (1131)	25 (795)	20 (774)	14 (602)	19 (947)	16 (824)	12 (665)	14 (773)
CA-GMRES(10, 30)	183 (760)	96 (1155)	45 (1011)	34 (1094)	17 (649)	22 (1007)	13 (640)	16 (918)	12 (739)	10 (627)	12 (776)
CA-GMRES(1, 1, 30)+ILU(0)	316 (158)	126 (183)	70 (187)	44 (169)	46 (212)	35 (194)	26 (167)	23 (159)	18 (136)	21 (160)	30 (233)
CA-GMRES(1, 8, 30)+ILU(0)	105 (162)	40 (175)	25 (201)	15 (167)	15 (197)	11 (166)	8 (139)	8 (150)	8 (151)	9 (187)	11 (211)
CA-GMERS(1, 10, 30)+ILU(0)	85 (143)	39 (187)	18 (154)	15 (173)	18 (249)	13 (217)	10 (200)	9 (174)	7 (156)	9 (190)	11 (225)

Fig. 16. Total Solution Time in Seconds (Number of Restarts), using an Underlapping Preconditioner with Local ILU(0)'s, and a Global k-way Partition and a Local Nested Dissection Ordering, for the `G3_Circuit` matrix. The seconds are rounded to the nearest integer.

Number of GPUs	1	3	9	15	21	27	1	3	9	15	21	27
GMRES(20)	11 (41)	3.22 (41)	1.37 (41)	1.04 (41)	0.98 (41)	0.92 (41)	2911 (2558)	847 (2449)	364 (2726)	228 (2492)	195 (2676)	164 (2566)
GMRES(20)+SAI(0), $s = 1$	7 (17)	1.84 (16)	0.82 (18)	0.67 (20)	0.63 (23)	0.55 (22)	686 (463)	130 (292)	66 (400)	40 (353)	28 (318)	21 (278)
CA-GMRES(2, 10, 20)	8 (41)	2.29 (41)	1.00 (41)	0.81 (41)	0.61 (41)	0.59 (41)	1780 (2558)	496 (2449)	227 (2726)	149 (2492)	127 (2676)	109 (2566)
CA-GMRES(1, 10, 20)+SAI(0)	6 (17)	1.56 (16)	0.68 (18)	0.55 (20)	0.51 (23)	0.43 (22)	493 (463)	97 (312)	55 (462)	21 (259)	24 (377)	20 (359)
CA-GMRES(2, 10, 20)+SAI(0)	6 (17)	1.92 (20)	0.76 (20)	0.66 (25)	0.55 (25)	0.53 (26)	492 (463)	109 (351)	45 (387)	37 (454)	28 (450)	38 (672)

(a) `PDE_1M(0.0)` (left, $m = 20$) and `PDE_1M(1.0275)` (right, $m = 60$) matrices.

Number of GPUs	3	9	15	21	27	3	9	15	21	27
GMRES(40)	81 (44)	29 (44)	19 (44)	16 (44)	13 (44)	126 (68)	45 (68)	28 (68)	22 (68)	18 (68)
GMRES(40)+SAI(0), $s = 1$	49 (19)	17 (19)	11 (20)	9 (21)	7 (22)	82 (32)	29 (33)	15 (28)	14 (35)	11 (35)
CA-GMRES(2, 10, 40)	55 (44)	21 (44)	14 (44)	11 (44)	10 (44)	86 (68)	31 (68)	20 (68)	15 (68)	14 (68)
CA-GMRES(1, 10, 40)+SAI(0)	39 (19)	13 (19)	9 (20)	7 (21)	6 (22)	65 (32)	23 (33)	12 (28)	11 (35)	9 (35)

(b) `PDE_10M(0.0)` (left) and `PDE_10M(1.0)` (right) matrices.

Fig. 17. Total Solution Time in Seconds (Number of Restarts), using an Underlapping Preconditioner with Local SAI(0)'s and a Global k-way Graph Partitioning, for the `PDE` matrices. Seconds are rounded to the nearest integer for clarity where appropriate.

preconditioned GMRES on a GPU cluster. This showed the proposed framework's potential for effectively preconditioning CA Krylov methods. We continue to explore ways to improve our framework's performance. For instance, since the same parameter s is used for the basis size in *MPK* and the subdomain underlap in *Preco*, there is a trade-off between reducing communication and increasing the iteration count as s varies. To address this issue, we are investigating other techniques to precondition the underlap or overlap regions, and a more flexible way of preconditioning the CA methods. We are also working to improve the performance of our CA-GMRES on a hybrid CPU/GPU cluster by exploiting both CPUs and GPUs, and by adaptively adjusting parameters such as the step size. We are also working on a production version of the communication avoiding solver that will be made available in the Belos package [**?**] in Trilinos.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR). This research was supported in part by DOE Grant #DE-SC0010042: "Extreme-scale Algorithms & Solver Resilience (EASIR)," NSF SI2-SSI-1339822 "Sustained Innovation for Linear Algebra Software (SILAS)," and DOE MAGMA - Office of Science Grant #DE-SC0004983, "Matrix Algebra for GPU & Multicore Architectures (MAGMA) for Large Petascale Systems." The NSF supports the Keeneland Computing Facility at the GIT under Contract OCI-0910735. Sandia National Laboratories is a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the DOE's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] N. Abdelmalek. Round off error analysis for Gram-Schmidt method and solution of linear least squares problems. *BIT Numerical Mathematics*, 11:345–368, 1971.

[2] Z. Bai, D. Hu, and L. Reichel. A Newton basis GMRES implementation. *IMA Journal of Numerical Analysis*, 14:563–581, 1994.

[3] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwarz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:1–155, 2014.

[4] X.-C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 21(2):792–797, Sept. 1999.

[5] A. T. Chronopoulos and C. W. Gear. Implementation of preconditioned s-step conjugate gradient methods on a multiprocessor system with memory hierarchy. *Parallel Comput.*, 11:37–53, 1989.

[6] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *24th National Conference*, pages 157–172, 1969.

[7] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in computing Krylov subspaces. Technical Report UCB/EECS-2007-123, University of California Berkeley EECS Department, October 2007.

[8] L. Grigori and S. Moufawad. Communication avoiding ILU0 preconditioner. Technical Report RR-8266, INRIA, 2013.

[9] M. Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, EECS Department, University of California, Berkeley, 2010.

[10] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *the proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 36:1–36:12, 2009.

[11] M. Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. Technical Report NVR-2011-001, Nvidia, 2011.

[12] Y. Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM J. Sci. Stat. Comput.*, 6:865–881, 1985.

[13] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[14] A. Stathopoulos and K. Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM J. Sci. Comput.*, 23:2165–2182, 2002.

[15] S. A. Toledo. *Quantitative performance modeling of scientific computations and creating locality in numerical algorithms*. PhD thesis, Massachusetts Institute of Technology, 1995.

[16] J. van Rosendale. Minimizing inner product data dependence in conjugate gradient iteration. In *Proc. IEEE Internat. Confer. Parallel Processing*, 1983.

[17] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra. Improving the performance of CA-GMRES on multicores with multiple GPUs. Technical Report UT-EECS-14-722, University of Tennessee, Knoxville, 2014. To appear in the proceedings of the 2014 IEEE International Parallel and Distributed Symposium (IPDPS).

[18] I. Yamazaki, S. Tomov, T. Dong, and J. Dongarra. Mixed-precision orthogonalization scheme and adaptive step size for CA-GMRES on GPUs, 2014. To appear in the proceedings of the 2014 International Meeting on High-Performance Computing for Computational Science (VECPAR).

[19] I. Yamazaki and K. Wu. A communication-avoiding thick-restart Lanczos method on a distributed-memory system. In *Workshop on Algorithms and Programming Tools for next-generation high-performance scientific and software (HPCC)*, 2011.