SAND2014-3284C

Domain Decomposition Preconditioners for Communication-Avoiding Krylov Methods on Distributed GPUs

Erik G. Boman, Michael A. Heroux, Mark Hoemmen, and Sivasankaran Rajamanickam
{egboman, maherou, mhoemme, srajama}@sandia.gov
Sandia National Laboratories, Albuquerque, New Mexico, U.S.A.

Stanimire Tomov and Ichitaro Yamazaki
{tomov, iyamazak}@eecs.utk.edu
University of Tennessee, Knoxville, Tennessee, U.S.A.

***Abstract*—Krylov subspace projection methods are the most 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 can give a speedup of 2.5x 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 can reduce the total solution time by a factor of 7.4x from CA-GMRES without preconditioning, and by a factor of 1.7x from GMRES with preconditioning. 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. This holds both 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. Effectiveness of such *communication-avoiding* (CA) techniques to improve the performance of Krylov methods has been demonstrated on shared-memory multicore CPUs [9], on distributed-memory CPUs [18], and on multiple graphics processing units (GPUs) on a single compute node [16]. In this paper, we extend these studies and show the performance of a CA variant of the Generalized Minimum Residual (GMRES) method [12] 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 [15], but they have not been widely adopted in practice. One reason for this is that though in practice, Krylov methods depend on preconditioning to accelerate their convergence rate, no effective preconditioners have been shown to work seamlessly with a CA method. This is 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 assumption makes it easy to use any exisiting algorithm or software for *Preco* in the Krylov methods. The CA 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 [8]. Then, all the existing techniques for preconditioning a CA method require close integration of *Preco* with *SpMV* within *MPK*, violating this handy black-box assumption and making it difficult to develop an effective preconditioner.

More importantly, these existing preconditioners of the first category are still closely integrated with *MPK*. Hence, these techniques often require significant changes in how *MPK* interacts with the input vectors, and are designed for specific types of preconditioners (e.g., approximate inverse or ILU(0)). Hence, it may be difficult to use an exisiting preconditioning software as a black-box routine. The second class of CA preconditioners requires radical changes to the representation of both the sparse matrix A and its preconditioner, such that off-diagonal blocks of both are stored using a low rank representation. To the best of our knowledge, there is no imple-

mentation or empirical evaluation of such CA preconditioners. Furthermore, most users of Krylov solvers do not wish to make such radical changes to their data structures.

To fill this crucial gap, in this paper, we propose a framework for effective CA preconditioning based on domain decomposition. 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* needs. However, more importantly, since we rely on domain decomposition, our preconditioner is not tightly coupled with *SpMV*, and it can use any existing solver or preconditioner software package as the local solver on each subdomain. Beside adding the local solver to apply the preconditioner, our preconditioner does not require any changes to *MPK* and does not increase its communication volume. Hence, 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 third 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 suggest speedups of 7.4x or 1.7x on up to 30 GPUs in comparison to CA-GMRES without preconditioning or GMRES with preconditioning, respectively, demonstrating the potential of our framework.

In addition to demonstrating the potential of our framework, our current studies clearly 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 [16] 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. In Section III, we first present our implementation of CA-GMRES and its performance on distributed GPUs. We then, in Section IV, describe our CA preconditioning framework, outline our sample implementation, and give numerical and performance results on distributed GPUs. 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 of our 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 built our code using 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.

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

II. Related work

Most existing CA preconditioning techniques fall into one of two categories [8]. The first category naturally fits how CA methods compute sparse matrix-vector products. These include preconditioners such as sparse approximate inverses with the same sparsity structure as the matrix A, or block Jacobi and polynomial preconditioners [4], [14], [15]. In block Jacobi preconditioning, each processor (or GPU) solves its local problem. For a conventional Krylov method, this block Jacobi without overlap does not require any additional communication. Unfortunately, even this is difficult to integrate into a CA method since after *Preco*, each local *SpMV* requires the preconditioned input vector elements from its neighbors, introducing extra communication. Depending on the sparsity structure of the matrix, a CA method may require significantly greater communication in order to use the block Jacobi preconditioner. See [6, Chapter 7] for an illustration 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, apparently without realizing its implementations for a preconditioned *MPK*. This was a surprising result that stirred us to develop the preconditioner framework presented in this paper.

A recently proposed CA preconditioning technique based on an incomplete LU factorization, CA-ILU(0) [7], can be considered as an advanced member of this category. Unfortunately, for some types of problems, these preconditioners of this first category 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 [11], to reduce the iteration count tends to decrease with the degree of the polynomial, while its computational cost increases. 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.

However, a potentially more 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 for specific types of preconditioners (e.g., approximate inverse or ILU(0)). Hence, it may be difficult to use an exisiting preconditioning software as a black-box routine. The second class of CA preconditioners requires radical changes to the representation of both the

repeat (restart-loop)
1. Generate Krylov Basis on GPUs: $\sim O(m \cdot nnz(|A| + |M|) + m^2 n)$ flops.
   $\mathbf{q}_{:,1} = \mathbf{q}_{:,1} / \|\mathbf{q}_{:,1}\|_2$ (with $\widehat{\mathbf{x}} = \mathbf{0}$ and $\mathbf{q}_{:,1} = \mathbf{b}$, initially)
   for $j = 1, 2, \ldots, m$ **do**
      Preconditioner (*Preco*) Application:
         $\mathbf{z}_{:,j} := M^{-1}\mathbf{q}_{:,j}$
      Sparse Matrix-Vector (*SpMV*) Product:
         $\mathbf{q}_{:,j+1} := A\mathbf{z}_{:,j}$
      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 on CPUs: $\sim O(m^2)$ flops.
   solve the least-squares problem $\mathbf{g} = \min_{\mathbf{t}} \|H\mathbf{t} - Q^T\mathbf{b}\|_2$
   to update solution $\widehat{\mathbf{x}} = \widehat{\mathbf{x}} + Z_{1:m}\mathbf{g}$ and
   restart with $\mathbf{q}_{:,1} = \mathbf{b} - A\widehat{\mathbf{x}}$

until solution convergence

Fig. 1. Pseudocode of GMRES(A, M, $\mathbf{b}$, m). m is the restart length.

repeat (restart-loop)
1. Generate Krylov Basis on GPUs: $\sim O(m \cdot nnz(|A| + |M_j|) + m^2 n)$ flops.
   $\mathbf{q}_{:,1} = \mathbf{q}_1 / \|\mathbf{q}_{:,1}\|_2$ (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 on CPUs: $\sim O(m^2)$ flops.
   solve the least-squares problem $\mathbf{g} = \min_{\mathbf{t}} \|H\mathbf{t} - Q^T\mathbf{b}\|_2$
   to update solution $\widehat{\mathbf{x}} = \widehat{\mathbf{x}} + X_{1:m}\mathbf{g}$ and
   restart with $\mathbf{q}_{:,1} = \mathbf{b} - A\widehat{\mathbf{x}}$

until solution convergence

Fig. 2. Pseudocode of CA-GMRES(A, M, $\mathbf{b}$, s, m). s is the *MPK* basis length and m is the restart length.

sparse matrix A and its preconditioner, such that off-diagonal blocks of both 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 do not wish to make such radical changes to their data structures.

III. CA-GMRES on Distributed GPUs

A. CA-GMRES Algorithm

The generalized minimum residual (GMRES) method [12] is a Krylov subspace projection method for iteratively computing an approximate solution to a nonsymmetric linear system of equations. It computes a solution with the minimum residual norm in the generated projection subspace. At each iteration, the approximate solution converges with monotonically nonincreasing residual norm. To this end, GMRES' j-th iteration first generates a new basis vector by applying the preconditioner (*Preco*) to the previously orthonormalized basis vector $\mathbf{q}_{:,j}$, followed by the sparse-matrix vector product (*SpMV*) with the resulting vector (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 $\mathbf{q}_{:,j+1}$ 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, the iteration is restarted after computing a fixed number $m+1$ of basis vectors. Before restart, the approximate solution $\widehat{\mathbf{x}}$ is updated 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, obtained as a by-product of the orthogonalization procedure, is in an upper Hessenberg form. Hence, the least-squares problem can be efficiently solved, requiring only about $3(m+1)^2$ flops. On the other hand, 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 that they are not small enough to 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 accomplishes 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. Cholesky QR Orthogonalization Kernel

In our previous study [16], we investigated the performance of several orthogonalization procedures for CA-GMRES on multiple GPUs on a single compute node. In that study, in most of the cases, CA-GMRES obtained the best performance using the classical Gram-Schmidt (CGS) procedure [1] and the Cholesky QR (CholQR) factorization [13] for *BOrth* and *TSQR*, respectively. Hence, in this paper, we focus on CGS-based *BOrth* and CholQR-based *TSQR* for the distributed GPUs as well. To maintain their orthogonality, in our experiments, we always orthogonalize the basis vectors twice.

Our CA-GMRES implementation on the distributed GPUs extends that of our previous implementation on the multiple GPUs of one node [16], where 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. Since the computational cost of CA-GMRES is typically dominated by the first step of generating the basis vectors, we

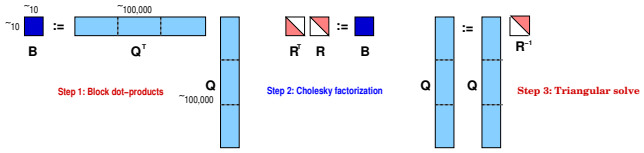
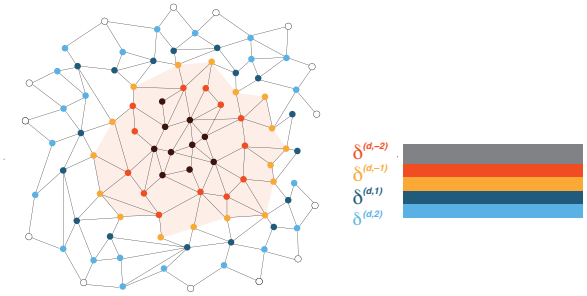


Fig. 3. Illustration of CholeskyQR Orthogonalization Process.



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

Fig. 4. Illustration of Extending and Sorting a Local Matrix for *MPK*.

use distributed GPUs to accelerate this step. To this end, 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. In our CholQR implementation, each GPU first computes the block dot products of its local vectors (i.e., $B^{(d)} := Q_{1:s+1}^{(d)T} Q_{1:s+1}^{(d)}$), using the optimized GPU kernels developed in [16], [17], 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., $B = \sum_{d=1,\ldots,n_g} B^{(d)}$). 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., $Q_{1:s+1}^{(d)} := Q_{1:s+1}^{(d)} R^{-1}$). Figure 3 illustrates this orthogonalization process. CGS is implemented in a similar fashion.

C. Matrix Powers Kernel

Our implementation reorders the rows of the *local* matrix $\bar{A}^{(d)}$ on the d-th GPU in descending order of their edge distances from the subdomain boundary in the adjacency graph of A. We then expand the local matrix to include the *nonlocal* entries that are ℓ edges away from the local matrix for $\ell = 1, 2, \ldots, s$. More specifically, let $\boldsymbol{\delta}^{(d,-\ell)}$ be the set of the local vertices whose shortest path from a nonlocal vertex is of length ℓ, while $\boldsymbol{\delta}^{(d,\ell)}$ is the set of the nonlocal vertices whose shortest path from a local vertex is of length ℓ (see Figure 4 for an illustration). We refer to $\boldsymbol{\delta}^{(d,-\ell)}$ and $\boldsymbol{\delta}(d,\ell)$ as the ℓ-level *underlap* and *overlap*, respectively, of the d-th subdomain. 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

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

Matrix Powers: generate q_{:,2}^{(d)}, q_{:,3}^{(d)}, ..., q_{:,s+1}^{(d)}
for k = 1, 2, ..., s do
    ℓ := s − k + 1
    for d = 1, 2, ..., n_g do
        Preco: compute z_{:,k}^{(d,ℓ)} := (R^{(d,ℓ)})^T (M^{(d)})^{-1} (R^{(d,ℓ)}) q_{:,k}^{(d,ℓ)}
        SpMV: compute q_{:,k+1}^{(d,ℓ-1)} := (R^{(d,ℓ-1)})^T Ā^{(d,ℓ)} (R^{(d,ℓ)}) z_{:,k}^{(d,ℓ)}
    end for
end for
```

Notation used for *MPK*:

$\boldsymbol{\delta}^{(d,s)}$	: nonlocal vertices that are s edges away
$\boldsymbol{\delta}^{(d,1:s)}$	: 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 matrix/vector on d-th GPU
$A^{(d,s)}$	: 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

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

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 black, red, green, blue, and then orange in that order.

When applying *MPK*, each GPU first exchanges all the required vector elements to compute s matrix powers with its neighboring GPUs. Then, for $k = 1, 2, \ldots, s$, each GPU computes the k-th matrix power by independently invoking *Preco* and *SpMV* with the k-th extended local matrix $\bar{A}^{(d,\ell)}$, where $\ell = s - k + 1$, without further inter-GPU communication (see Step 1.1 in Figure 2). Before the iteration to compute the matrix powers, the required vector elements must be commu-

nicated among 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]

Our implementation of CA-GMRES can use different values of the step size s for *MPK*, and for the orthogonalization kernels *BOrth* and *TSQR* [17]. This often improves the performance of CA-GMRES, because with a larger value of s, *MPK* adds computational and storage overheads, and may potentially increase communication volume. As a result, the optimal value of s for *MPK* may be smaller than the optimal s for *BOrth* or *TSQR*. (Mohiyuddin et al. [9] observed this already for the special case that *MPK*'s step size is 1.) 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. Hence, our implementation of CA-GMRES$(s, \hat{s}, m)$ can take three input parameters, 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 case $s = 1$ means that CA-GMRES does not need a specialized *MPK* implementation; it merely uses matrix-vector multiplies and preconditioner applications, and relies on CA-GMRES' orthogonalization kernels for performance improvements over GMRES. This is a reasonable strategy for long-recurrence Krylov solvers like GMRES, but CA variants of short-recurrence methods like CG will likely need an optimized *MPK*, since they spend much less time in inner products.

D. Performance Studies

Figure 6 compares the parallel strong scaling performance of GMRES and CA-GMRES by showing their total solution time speedups over the time required by GMRES on one GPU for the `G3_Circuit` matrix. 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 [5] (see Section IV-C for more detailed experimental setups, except the greater stopping criteria of 10^{-8} is used here). 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

[2]Since we typically have more CPU cores than GPUs on a node, Pthreads are used to process the messages from multiple GPUs in parallel. The GPUs on the same node communicate without using MPI.

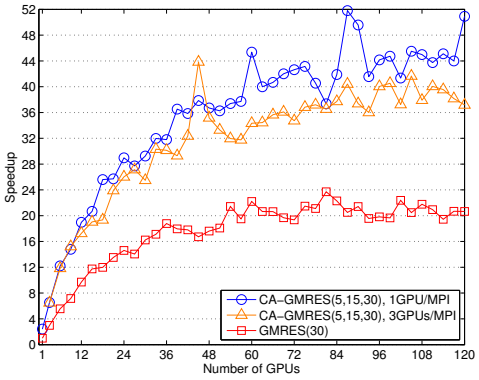


Fig. 6. Parallel Strong Scaling of CA-GMRES and GMRES on Distributed GPUs (over GMRES on one GPU), for the `G3_Circuit` matrix.

Number of GPUs (Matrix)	6	12	24	48
GMRES(30)				
Number of Restarts	39	35	130	79
Time (s)	3.51	3.13	10.60	10.54
CA-GMRES$(1, 30)$				
Number of Restarts	39	35	131	79
Times (s)	2.10	1.59	7.55	7.77
Speedup	1.60	1.97	1.40	1.36

Fig. 7. Parallel Weak Scaling Performance Studies for the `brick` matrices, starting with $n = 1$M on 6 GPUs.

around 60 GPUs because the local submatrix became too small for the GPU to obtain any strong scaling speedup. We expect that on a larger number of compute nodes, the MPI communication becomes more dominant, and the speedups obtained by avoiding the communication will increase [18]. The figure also shows 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 with our implementation and experimental setups, 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 MPI 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, staring from $1,035,351$ on 6 GPUs. To accomodate the large CPU memory usage during setup, we lauched one MPI per node, which manage three GPUs on the node, on 48 GPUs. Again, CA-GMRES outperformed GMRES for this weak scaling studies.

IV. CA Domain Decomposition Preconditioners

A. CA-Preconditioning Framework

As discussed in Section I, it is difficult to introduce preconditioning in CA Krylov methods. Instead of forming the basis vectors for the Krylov subspace $\mathcal{K}_k(A, \mathbf{q}_{:,1}) = \text{span}\{\mathbf{q}_{:,1}, A\mathbf{q}_{:,1}, \ldots, A^k\mathbf{q}_{:,1}\} = \text{span}\{\mathbf{q}_{:,1}, \mathbf{q}_{:,2}, \ldots, \mathbf{q}_{:,k+1}\}$, we must generate the basis vectors for the preconditioned

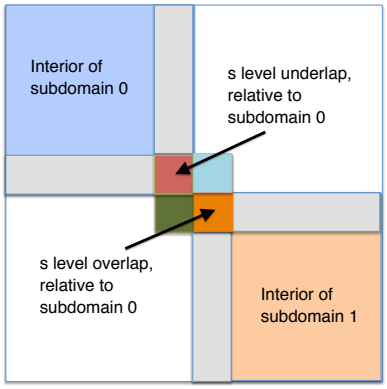


Fig. 8. Matrix Partitioning for the CA Preconditioner for two subdomains. The underlap and the overlap relative to subdomain 0 are shown.

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 k-th matrix power, each processor (or GPU) needs to know both the local vector elements at $\mathbf{i}^{(d,0)}$ and those at the ℓ-level overlap $\boldsymbol{\delta}^{(d,1:\ell)}$, where $\ell = s-k+1$ (for $k = 1, 2, \ldots, s$). Therefore, additional communication is needed if the action of M^{-1} to generate these vector elements of $\mathbf{z}_{:,j}$ at $\mathbf{i}^{(d,\ell)}$ requires any element of the input vector $\mathbf{q}_{:,j}$ aside from those at the row index set $\mathbf{i}^{(d,\ell)}$, where $\mathbf{i}^{(d,\ell)} = \mathbf{i}^{(d,0)} \bigcup \boldsymbol{\delta}^{(d,1:\ell)}$.

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 both the local submatrix $A^{(d)}$ and the s-level overlap $\boldsymbol{\delta}^{(d,1:s)}$. 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 since they are local in nature and are well suited for parallel computing.

We now propose a communication avoiding domain decomposition preconditioner. As explained in Section II, even block Jacobi preconditioner is difficult to integrate into a CA method since after each processor (or GPU) applies *Preco* by solving its local problem with its local submatrix $A^{(d)}$, each local *SpMV* requires the preconditioned input vector elements from its neighbors, introducing extra communication. To avoid this additional communication, we "shrink" the diagonal blocks to make them disjoint from the s-level overlaps. To this end, let us assume there are n_g non-overlapping subsets (subdomains or GPUs) of the row index set of A. Recall from our notation in Section III-C that in the view of the d-th subdomain the local diagonal block is $A^{(d,0)}$. 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 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 $\mathbf{i}^{(-s-1)}$ and $\boldsymbol{\delta}^{(\ell_1:\ell_2)}$ to represent $\mathbf{i}^{(d,-s-1)}$ and $\boldsymbol{\delta}^{(d,\ell_1:\ell_2)}$, where $\mathbf{i}^{(d,-s-1)} = \bigcup_{\ell \le -s-1} \boldsymbol{\delta}^{(d,\ell)}$ and $\boldsymbol{\delta}^{(d,\ell_1:\ell_2)} = \bigcup_{\ell_1 \le \ell \le \ell_2} \boldsymbol{\delta}^{(d,\ell)}$. Then, the square local matrix $A^{(d,s)}$ for the d-th subdomain 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. By definition $A^{(d,0)} = A(\mathbf{i}^{(0)}, \mathbf{i}^{(0)})$. Given $|A| = n$ and $\left|A^{(d,s)}\right| = n_d$, we define the standard rectangular n-by-n_d extension matrix $(R^{(d,0)})^T$ with zeros and ones, which extends by zero a vector of values associated with vertices $A^{(d,0)}$. The corresponding s-step 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 [3] with overlap s becomes:

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

By varying the amount of overlap, we obtain a *sequence* of s preconditioners applied at s iterations; for $k = 1, 2, \ldots, s$, we define

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

where $\ell = s - k + 1$. In this definition, the preconditioner changes at each iteration, shrinking both its underlap and overlap to match with the extended local submatrix 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. This is similar to the restricted additive Schwartz preconditioner. However, the s-step preconditioner shrinks the subdomain based on the iteration. 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). In our implementation, we do this by considering only the diagonal blocks of $A^{(d,s)}$ and using diagonal preconditioning for the two diagonal blocks $A(\boldsymbol{\delta}^{(-s:-1)}, \boldsymbol{\delta}^{(-s:-1)})$ and $A(\boldsymbol{\delta}^{(1:s)}, \boldsymbol{\delta}^{(1:s)})$ on the underlap and overlap, respectively. We also use a constant underlap so that the preconditioner is stationary. This *underlap* preconditioner, for s iterations with $k = 1, 2, \ldots, s$, is defined as

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

where $\ell = s - k + 1$ 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 to work effectively with the CA-GMRES without incurring any additional communication cost.

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.

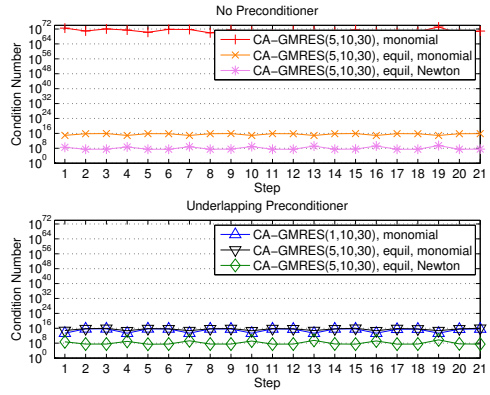


Fig. 10. Condition Number of Gram Matrix, `G3_Circuit` matrix, 6 GPUs.

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

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

where any traditional local subdomain preconditioner can be used for an inexact solution with $A(\mathbf{i}^{(-s-1)}, \mathbf{i}^{(-s-1)})$, including incomplete factorizations or a fixed number of iterations with a stationary method such as Jacobi or Gauss-Seidel. This formula defines the action of the preconditioner on each local subdomain; in addition, each processor must redundantly compute the action of the preconditioner on the overlap in a shrinking fashion based on the iteration number. Mathematically, the preconditioner is fixed and does not change from iteration to iteration.

B. Sample Implementation

In this section, we describe our implementation of the underlap preconditioner described in Section IV-A and present the setups for our experiments. As our first setup, to distribute the matrix A among the GPUs, we tested two matrix reordering algorithms: reverse Cuthill-McKee (RCM) [5] 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. Then, the computed solution is considered to have converged when the ℓ_2-norm of the initial residual is reduced by at least twelve orders of magnitude.

As the local solver, we investigated the stationary iterative methods (i.e., Jacobi or Gauss-Seidel iterations), 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 the local preconditioners on the CPU and copies them to the GPU. Then, at each iteration, the MPI process applies the preconditioners using either the sparse-matrix vector multiply or the sparse triangular solves of CuSparse in the Compressed Sparse Row (CSR) matrix storage format. The sparse matrix vector multiply with the coefficient matrix is performed using our own GPU kernel in the ELLPACKT format [16].

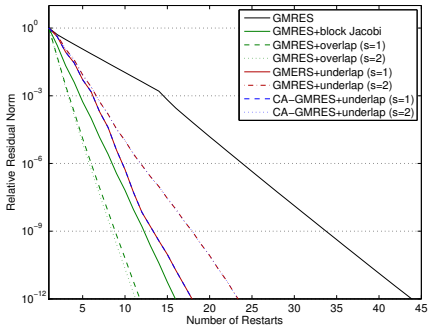
C. Convergence Studies with a Fixed GPU Count

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 example of the Trilinos software library. It arises from discretizing Poisson's equation with Dirichlet boundary conditions on a cube Ω, using a regular hexahedral mesh. The PDE is $\text{div}(M\nabla u) = f$ in Ω, $u = g$ on $\partial\Omega$, where M is a 3 by 3 material tensor, f a given source term, and g a given boundary term. Discretizing results in a linear system $Ax = b$. The material tensor M is always symmetric, and thus so is A. We can control the iteration count for solving $Ax = b$ by changing M. We set $M = \begin{pmatrix} 1 & 0 & \alpha \\ 0 & 1 & 0 \\ \alpha & 0 & 1 \end{pmatrix}$. When $\alpha = 0$, the problem takes few iterations. As α approaches 1, the problem takes more. When $\alpha > 1$, the matrix A is no longer positive definite. We present results for different values of α.

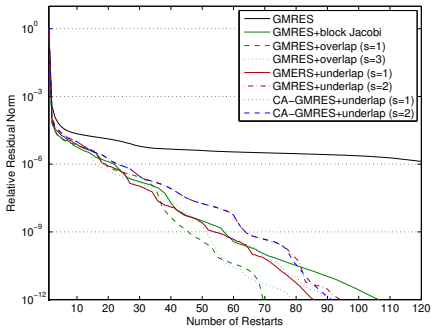
The Brick matrices come from the discretization of the same PDE on a 3-D brick-shaped mesh, in which the number of elements in two dimensions are fixed. The third dimension has four different types of material blocks, two of which have the element sizes graded.

To enhance CA-GMRES' numerical stability, before the iteration starts, we equilibrate A and b [8]. That is, we first scale its rows, and then its columns, by their ∞-norms. In addition, to improve stability when generating matrix powers, we 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 Hessenberg matrix H from the first restart in a Leja ordering [2]. Figure 10 shows the condition number of the Gram matrices generated during CholQR for the `G3_Circuit` matrix. These condition numbers are the square of the condition numbers of the basis vectors generated by *MPK* (see Section III-B). The figure shows that equilibration, using the Newton basis, and preconditioning all contribute to improve these condition numbers. While in our experiments, we used the same basis step size s and orthogonalization parameters for testing CA-GMRES with

[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

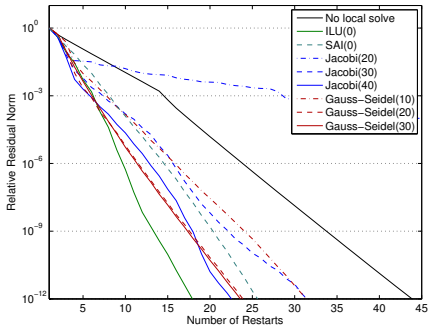


(a) `PDE_1M(0.0)` matrix, with restart = 20.

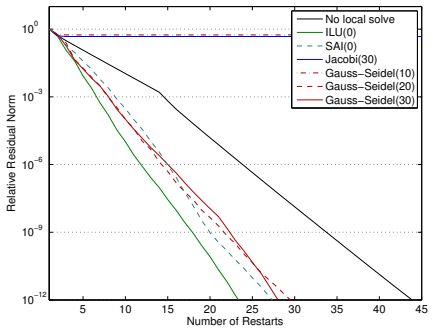


(b) `G3_Circuit` matrix, with restart = 30.

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.

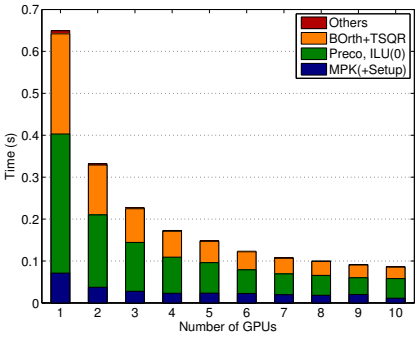
and without preconditioning, we could potentially use a larger s or omit reorthogonalization when preconditioning. This may significantly improve performance.

Figure 11 compares the convergence behavior of the relative residual norm, $\|\mathbf{b} - A\widehat{\mathbf{x}}\|_2/\|\mathbf{b}\|_2$, using different preconditioners, where the local solver is the 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 the iteration counts. However, the iteration count was significantly reduced using either an overlap or underlap preconditioner. In addition, the convergence of CA-GMRES 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.

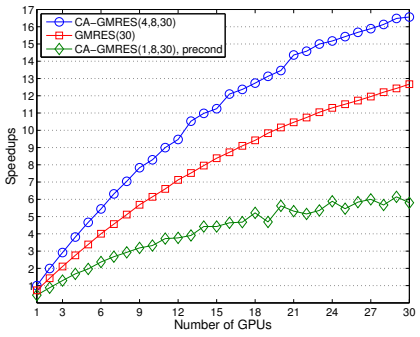
Figure 12 shows the solution convergence, when different local solvers were used in combination with our underlapping preconditioner. For instance, we investigated the Jacobi iteration as our local solver. This is because on some architectures including the GPU, the sparse triangular solution required for the ILU-based preconditioner often obtains much lower performance than *SpMV* required for *MPK* or for the Jacobi iteration (see Section IV-D). Unfortunately, even for this relatively well-conditioned system, in order to match the solution convergence obtained using ILU(0), a large number of Jacobi iterations was required when $s = 1$, and it did not converge when $s = 2$. Even though a fewer Gauss-Seidel iterations were required, the iteration count was still large, especially considering that a forward-substitution is needed at each Gauss-Seidel iteration. 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 (see Section IV-D).

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 the sparse triangular solver of cuSparse depends strongly on the sparsity pattern of the triangular factor [10]. 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 13(a) shows the breakdown of the average time spent in one restart loop. Even with the nested dissection ordering,

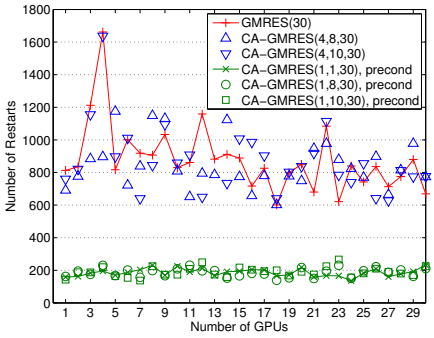


(a) Average Restart Time Breakdown, CA-GMRES(4, 8, 30).

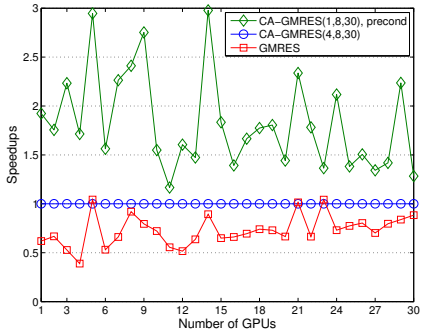


(b) Average Restart Time (over CA-GMRES on One GPU).

Fig. 13. Parallel Strong Scaling of Average Restart Time on Distributed GPUs for the G3_Circuit matrix.



(a) Number of Restarts.



(b) Total Solution Time (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.

though *Preco* and *MPK* perform the same number of floating-point operations, *Preco* required significantly longer time (e.g., by a factor of 4.67, where *MPK* includes the setup time), since the triangular solution is inherently serial. As a result, Figure 13(b) shows that the time per iteration of CA-GMRES was significantly longer when the preconditioner was used (e.g., by a factor of 3.41). However, as Figure 14(a) shows, the underlapping preconditioner significantly reduced the iteration count, and as Figure 14(b) shows, the total solution time was also greatly reduced using the preconditioner (e.g., by a factor of 2.95). Figure 15 shows the detailed performance results for this G3_Circuit matrix.

Compared to the G3_Circuit matrix, our PDE matrices are relatively well-conditioned, and SAI(0) is often effective as the local solver. Figure 17(a) shows the average breakdown of the restart loop time. 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 13(a)). As a result, compared with Figure 14(b) for the G3_Circuit matrix, Figure 17(b) shows greater speedups obtained using the underlap preconditioner for the PDE matrices. Figure 16 shows the detailed performance profiles with the PDE matrices, where for the PDE_10M matrices, we launched one MPI per node, which managed three GPUs on the node. The figure also shows that CA-GMRES with underlap preconditioner improves the performance of GMRES with the same preconditioner. We expect that this improvement will increase on a computer where the communication between the parallel processes or threads is more dominant.

We emphasize that the distribution of the matrix A among the GPUs not only affects the performance of CA-GMRES (e.g., storage, computation, and communication overheads associated with the s-step overlap), but it also determines the quality of the preconditioner (e.g., the size of the s-step underlap). For the results of the PDE matrices presented here, the matrix A was distributed among the GPUs based on the k-way graph partitioning algorithm. In contrast, when the matrix was distributed using the RCM ordering, the sizes of both the overlap and underlap increased quickly with the increase in the number of subdomains. As a result, even on a small number of GPUs (e.g., $n_g = 21$), some underlaps extended all the way inside the local subdomains, and the local preconditioners became the 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 the quick reduction in the speedups gained using the underlap preconditioner on a larger number of GPUs.

Number of GPUs		1	3	6	9	12	15	18	21	24	27	30
GMRES(30)		326.2 (813)	167.8 (1212)	73.0 (1001)	53.1 (1033)	47.5 (1159)	31.0 (889)	18.6 (601)	19.0 (680)	21.7 (840)	17.4 (713)	15.4 (669)
CA-GMRES(6, 30)		347.2 (1026)	116.4 (995)	142.0 (2257)	44.0 (1001)	25.6 (712)	29.2 (970)	23.1 (860)	24.5 (1027)	18.5 (823)	24.7 (1154)	19.3 (939)
CA-GMRES(8, 30)		201.8 (691)	88.5 (884)	38.7 (722)	42.2 (1131)	24.5 (795)	20.1 (774)	13.8 (602)	19.3 (947)	15.9 (824)	12.2 (665)	13.6 (773)
CA-GMRES(10, 30)		183.1 (760)	95.9 (1155)	45.1 (1011)	34.3 (1094)	16.7 (649)	22.0 (1007)	12.5 (640)	15.7 (918)	11.9 (739)	9.7 (627)	11.5 (776)
CA-GMRES(1, 1, 30)	ILU(0)	316.4 (158)	126.4 (183)	69.5 (187)	43.9 (169)	46.0 (212)	34.5 (194)	26.2 (167)	22.8 (159)	18.1 (136)	20.6 (160)	29.6 (233)
CA-GMRES(1, 8, 30)	ILU(0)	104.9 (162)	39.6 (175)	24.8 (201)	15.3 (167)	15.3 (197)	11.0 (166)	7.8 (139)	8.2 (150)	7.5 (151)	9.1 (187)	10.6 (211)
CA-GMERS(1, 10, 30)	ILU(0)	85.3 (143)	39.3 (187)	17.6 (154)	14.8 (173)	18.0 (249)	13.3 (217)	10.4 (200)	8.9 (174)	7.2 (156)	8.6 (190)	10.6 (225)

Fig. 15. 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.

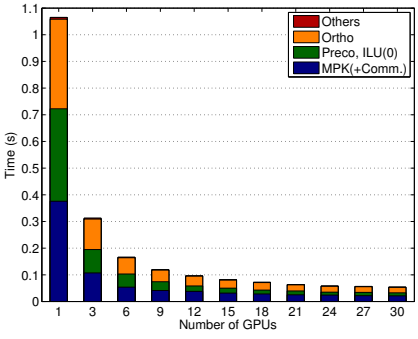
Number of GPUs		1	3	9	15	21	27	1	3	9	15	21	27
Underlap/Local Domain	min	0.00	0.04	0.07	0.06	0.08	0.09						
	max	0.00	0.05	0.10	0.15	0.21	0.21						
GMRES(20)		11.41 (41)	3.22 (41)	1.37 (41)	1.04 (41)	0.98 (41)	0.92 (41)	2911.3 (2558)	846.7 (2449)	364.4 (2726)	228.3 (2492)	195.2 (2676)	164.4 (2566)
GMRES(20)+SAI(0), $s = 1$		6.52 (17)	1.84 (16)	0.82 (18)	0.67 (20)	0.63 (23)	0.55 (22)	685.8 (463)	130.5 (292)	66.3 (400)	39.5 (353)	28.0 (318)	21.3 (278)
CA-GMRES(2, 10, 20)		8.11 (41)	2.29 (41)	1.00 (41)	0.81 (41)	0.61 (41)	0.59 (41)	1779.9 (2558)	496.1 (2449)	226.6 (2726)	149.2 (2492)	126.5 (2676)	108.8 (2566)
CA-GMRES(1, 10, 20)+SAI(0)		5.72 (17)	1.56 (16)	0.68 (18)	0.55 (20)	0.51 (23)	0.43 (22)	493.1 (463)	97.4 (312)	55.1 (462)	21.1 (259)	24.0 (377)	20.3 (359)

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

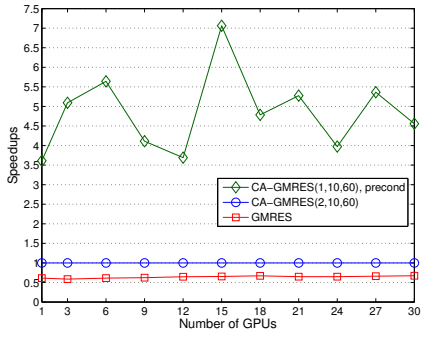
Number of GPUs	3	9	15	21	27	3	9	15	21	27
GMRES(40)	81.07 (44)	29.01 (44)	19.28 (44)	15.72 (44)	12.82 (44)	125.62 (68)	44.58 (68)	28.19 (68)	22.26 (68)	18.24 (68)
GMRES(40)+SAI(0), $s = 1$	49.27 (19)	17.07 (19)	11.11 (20)	8.69 (21)	7.28 (22)	82.07 (32)	29.25 (33)	15.37 (28)	14.40 (35)	11.45 (35)
CA-GMRES(2, 10, 40)	55.73 (44)	20.68 (44)	13.85 (44)	10.92 (44)	10.46 (44)	86.16 (68)	30.74 (68)	19.54 (68)	15.49 (68)	14.10 (68)
CA-GMRES(1, 10, 40)+SAI(0)	39.30 (19)	13.39 (19)	8.98 (20)	7.09 (21)	5.83 (22)	64.66 (32)	22.54 (33)	12.10 (28)	11.07 (35)	8.99 (35)

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

Fig. 16. 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.



(a) Breakdown of Average Restart Time.



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

Fig. 17. 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.

V. Conclusion

This paper presents the performance of CA-GMRES on a GPU cluster. CA-GMRES shows speedups of 2.5x over standard GMRES on up to 120 GPUs. We also proposed a framework based on domain decomposition to precondition CA Krylov methods. Though we use simple ILU(0) and SAI(0) preconditioners to solve on each GPU's domain, the number of GMRES iterations was greatly reduced. Our performance results on a GPU cluster 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 step size is used for *MPK* and *Preco*, there is a trade off when using a larger step size, between reducing communication and increasing the iteration count. 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.

Acknowledgments

This research was supported in part by NSF SDCI - National Science Foundation Award #OCI-1032815, "Collaborative Research: SDCI HPC Improvement: Improvement and Support of Community Based Dense Linear Algebra Software for Extreme Scale Computational Science," DOE grant #DE-SC0010042: "Extreme-scale Algorithms & Solver Resilience (EASIR)," NSF Keeneland - Georgia Institute of Technology Subcontract #RA241-G1 on NSF Prime Grant #OCI-0910735, and DOE MAGMA - Department of Energy Office of Science Grant #DE-SC0004983, "Matrix Algebra for GPU and Multicore Architectures (MAGMA) for Large Petascale Systems." This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Foundation 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 U.S. Department of Energy'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] 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.

[4] 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.

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

[6] 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.

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

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

[9] 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.

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

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

[12] 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.

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

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

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

[16] 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).

[17] 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).

[18] 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.