SAND2016-12307C

Improving performance of GMRES by reducing communication and pipelining global collectives

Ichitaro Yamazaki*, Mark Hoemmen†, Piotr Luszczek*, and Jack Dongarra*
*University of Tennessee, Knoxville, Tennessee, U.S.A.
†Sandia National Laboratories, Albuquerque, New Mexico, U.S.A.
iyamazak@icl.utk.edu, mhoemme@sandia.gov, luszczek@icl.utk.edu, and dongarra@icl.utk.edu

***Abstract*—**
We compare the performance of two iterative linear solvers, Pipelined GMRES and Communication-Avoiding (CA) GMRES, on a distributed-memory computer with a multicore node architecture. Pipelined GMRES depends on overlapping communication and computation. We implemented overlap in two different ways. The first uses nonblocking MPI (Message Passing Interface) collectives with OpenMP thread-parallel computational kernels. The second relies on a task scheduling system, QUARK, both for thread parallelism and to make MPI communication nonblocking. We also propose a variant of GMRES, Pipelined CA-GMRES, that combines the best features of both methods. It not only does fewer global all-reduces than standard GMRES, but also overlaps those all-reduces with other work. In our experiments, Pipelined CA-GMRES performed better than standard Pipelined GMRES by factors of up to $1.67\times$. In addition, even when using a relatively small number of compute nodes, our variant performed up to $1.22\times$ better than CA-GMRES.

I. Introduction

Krylov subspace projection methods iteratively solve large-scale linear systems of equations that are too big for other methods to solve. However, each Krylov iteration requires several communication. We use this term "communication" to include both "horizontal" data movement and synchronization between parallel processing units, as well as "vertical" data movement between memory hierarchy levels. Communication has become much more expensive on modern computers compared to computation, in terms of both throughput and energy consumption. Two approaches have been developed to reduce this communication cost. The first, "communication avoiding," redesigns the algorithm to communicate less. The second, "pipelining," hides the cost of communication by using nonblocking communication, then overlapping it with other work. These techniques may increase computational cost, but may nevertheless improve overall performance, due to the gap between the cost of communication and computation. Since we expect this gap to continue to widen [1], these techniques are likely to play an even more critical role in maintaining high performance of the linear solvers moving forward.

In this paper, we begin by comparing techniques to avoid or pipeline communication in a particular Krylov solver, the Generalized Minimum Residual (GMRES) method [2] for solving nonsymmetric linear systems of equations. We implement both Communication-Avoiding GMRES (CA-GMRES) [3] and Pipelined GMRES [4], then compare performance of the two approaches. Unlike previous Pipelined GMRES studies, we focus on the interaction between thread parallelism and overlap of computation and internode communication. To this end, we wrote two Pipelined GMRES implementations. The first uses threaded computational kernels for local computation, and nonblocking MPI calls for interprocess communication. The second one explores the potential of a shared-memory run-time system called QUARK [5] both to expose node-level task parallelism in computations, and to overlap those computations with communication tasks. Previous work studied the performance of either communication-avoiding (CA) or pipelined Krylov methods separately. However, to the best of our knowledge, we are the first to compare the performance of both methods in a single work.

In addition, we are first to combine the two approaches into a single new algorithm, *Pipelined CA-GMRES*. This builds on the observation behind pipelined methods, that global collectives will be the performance bottleneck at large scale, and our experiences that in our previous experiments using CA techniques, most of our performance improvement came from the block orthogonalization procedure to reduce the global communication. Hence, our algorithm focuses on reducing the number of all-reduces needed for orthogonalizing the Krylov basis vectors by combining CA-GMRES' block orthogonalization with Pipelined GMRES. While generating these Krylov basis vectors, the algorithm relies on the standard sparse-matrix vector multiply (*SpMV*) instead of its CA variant: the matrix powers kernel (*MPK*) [3]. *SpMV* requires only point-to-point communication among neighboring processes, and implementations can even overlap this with some local computation. More importantly, we avoid the computation, communication, and storage overheads of setting up the *MPK*, as well as its computational overhead and a potential increase in total communication volume of generating the basis vectors (*MPK* trades off these costs in favor of reducing the latency cost of point-to-point communication). In addition, not using *MPK* means that our implementation works with any preconditioner. This is essential for reducing the number of solver iterations in practice. Previous work, both by ourselves [6] and others [7], made attempts at preconditioning *MPK*, but effectivly preconditioning *MPK* remains to be a challenge. In this work, we overcome this challenge by trading off the communication for *SpMV* and preconditioning. In our performance comparison on a distributed-memory computer with up to 300 processes, we have seen that this combination

GMRES(A, M, $\mathbf{b}$, m):
$\mathbf{q}_1 = \mathbf{q}_1/\|\mathbf{q}_{:,1}\|_2$ (with $\widehat{\mathbf{x}} = \mathbf{0}$ and $\mathbf{q}_1 = \mathbf{b}$, initially)
for $j = 1, 2, \ldots, m$ **do**
  // *SpMV* with local submatrix of A, and optionally *Precond* with M:
1. `MPI_Isend` and `MPI_Irecv` for 1-level ghost of $\mathbf{q}_j$ among neighbors and then `MPI_Wait`
2. $\mathbf{v}_{j+1} := AM^{-1}\mathbf{q}_j$
  // *Orth* based on CGS:
3. $\mathbf{h}_{1:j,j} := Q_{1:j}^T \mathbf{v}_{j+1}$, with `MPI_Allreduce`
4. $\mathbf{q}_{j+1} := \mathbf{v}_{j+1} - Q_{1:j}\mathbf{h}_{1:j,j}$
5. $h_{j+1,j} := (\mathbf{q}_{j+1}^T \mathbf{q}_{j+1})^{1/2}$, with `MPI_Allreduce`
6. $\mathbf{q}_{j+1} := \mathbf{q}_{j+1}/h_{j+1,j}$
end for

Fig. 1. Pseudocode of GMRES to generate the orthonormal basis vectors $Q_{1:m+1}$ and the projected matrix H, where m is the restart length.

Pipelined GMRES(A, M, $\mathbf{b}$, ℓ, m):
$r_{1,1} := 1.0$
for $j = 1, 2, \ldots, m$ **do**
  // *SpMV* with local submatrix of A, and optionally *Precond* with M:
1. `MPI_Isend` and `MPI_Irecv` for 1-level ghost of $\mathbf{q}_j$ among neighbors and then `MPI_Wait`
2. $\mathbf{v}_{j+1} := AM^{-1}\mathbf{v}_j$
3. $k := j - \ell + 1$
4. **if** $j > \ell$ **then**
5. `MPI_Wait(tag = k)`
6. Update $\mathbf{r}_{1:k,k}$
7. Generate $\mathbf{h}_{1:k,k-1}$
  // *Orth* based on CGS:
8. $\mathbf{q}_k := (\mathbf{v}_k - Q_{1:k-1}\mathbf{r}_{1:k-1,k})/r_{k,k}$
9. $\mathbf{v}_{j+1} := (\mathbf{v}_{j+1} - V_{\ell:j}\mathbf{h}_{1:k-1,k-1})/h_{k,k-1}$
10. **end if**
11. $\mathbf{r}_{1:j+1,j+1} := [Q_{1:k-1}, V_{k:j+1}]^T \mathbf{v}_{j+1}$, with `MPI_Iallreduce(tag = j + 1)`
end for

Fig. 2. Pseudocode of pipelined GMRES to generate the orthonormal basis vectors $Q_{1:m+1}$ and the projected matrix H, where ℓ is the pipeline depth, and m is the restart length.

can obtain speedups of up to $1.67\times$ over our Pipelined GMRES implementation and $1.22\times$ over our CA-GMRES implementation.

II. Algorithms

In this section, we review the standard GMRES algorithm (Section II-A), and two variants thereof: Pipelined GMRES (Section II-B), and CA-GMRES (Section II-C).

A. GMRES

The Generalized Minimum Residual (GMRES) method [2] solves a nonsymmetric linear system of equations $Ax = b$. Its j-th iteration first generates a new basis vector $\mathbf{v}_{j+1} := A\mathbf{q}_j$ by applying *SpMV* to the previously orthonormalized basis vector $\mathbf{q}_j$. Then, GMRES computes the new basis vector $\mathbf{q}_{j+1}$ by orthonormalizing (*Orth*) $\mathbf{v}_{j+1}$ against the previously orthonormalized basis vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j$.

To reduce both the computational and storage costs of computing a large projection subspace, GMRES "restarts" 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 $\mathbf{c} := Q_{1:m+1}^T(\mathbf{b} - A\widehat{\mathbf{x}})$, $H := Q_{1:m+1}^T A Q_{1:m}$, and $\widehat{\mathbf{x}} := \widehat{\mathbf{x}} + Q_{1:m}\mathbf{g}$. Then, it restarts the iteration, using the residual vector $\mathbf{b} - A\widehat{\mathbf{x}}$ as its starting vector $\mathbf{q}_1$. 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. In comparison, for an n-by-n matrix A with $nnz(A)$ nonzeros, *SpMV* and *Orth* require a total of about $2m \cdot nnz(|A|)$ and $2m^3 n$ flops over the m iterations, respectively (i.e., $n, nnz(A) \gg m$). Figure 1 shows the pseudocode of restarted GMRES. The matrix and the vectors are distributed among the processes in a "one-dimensional" block row format. Hence, the one-level ghost entries of the vector that need to be communicated for *SpMV* are the nonlocal entries that are one edge away from the local entries in the adjacency graph of A.

Both *SpMV* and *Orth* require communication. This includes point-to-point messages or sparse neighborhood collectives for *SpMV*, and two global all-reduces in *Orth*. Beside these interprocess communications, there is intraprocess communication, that is, data movement between levels of the local memory hierarchy (for reading the sparse matrix and for reading and writing vectors, assuming that they do not fit in cache).

B. Pipelined GMRES

Pipelined Krylov methods [4] aim to hide the cost of the global all-reduces needed for *Orth*, by overlapping them with both local computation, and with the point-to-point communication in *SpMV*. Hiding the communication in a single *Orth* can lead to at most $2\times$ speedups, but pipelined methods may improve on this by overlapping multiple iterations. Pipelined methods have the additional advantage of mitigating the effects of random node performance variation ("jitter") and message delays [8]. In standard methods, these effects manifest as load imbalance that "piles up" at the global all-reduces. Finally, pipelined methods allow use of any preconditioner. This is essential to improving the convergence rate of the methods in practice [9], [10], [11].

Figure 2 shows the pseudocode of Pipelined GMRES. The algorithm generates an additional auxiliary upper-triangular matrix R such that $V_{1:j+1} = Q_{1:j+1}R_{1:j+1,1:j+1}$, where $V_{1:j+1}$ are the generated Krylov vectors and $Q_{1:j+1}$ are its orthonormal basis vectors. When computing $\mathbf{r}_{1:j+1,j+1}$, the last ℓ vectors, $V_{k:j+1}$ have not yet been orthogonalized (Line 11). Hence, after the corresponding synchronization, the column vector needs to be updated, assuming the orthogonality of the previous vectors (Line 6). Then, the orthonormal basis vectors Q are generated using R (Line 8).

In finite-precision arithmetic, the algorithm loses orthogonality among the basis vectors faster than in standard GMRES. To maintain numerical stability, Pipelined GMRES introduces the *change-of-basis* matrix, $B_{1:j+1,1:j}$, such that

$$AV_{1:j} = V_{1:j+1}B_{1:j+1,1:j}. \tag{1}$$

For instance, in [4], the matrix $B_{1:j+1,1:j}$ is defined to generate the Newton basis for the first ℓ steps [12], and then for the

CA-GMRES(A, M, $\mathbf{b}$, s, m):
$\mathbf{q}_1 := \mathbf{q}_1/\|\mathbf{q}_1\|_2$ (with $\widehat{\mathbf{x}} := \mathbf{0}$ and $\mathbf{q}_1 := \mathbf{b}$, initially)
for $j = 1, 1+s, \ldots, m$ **do**
  // Matrix Powers Kernel (*MPK*(s, A, M, $\mathbf{q}_1$)):
1. `MPI_Isend` and `MPI_Irecv` for s-level ghost of $\mathbf{q}_j$ among neighbors and then `MPI_Wait` to form $\mathbf{v}_j$
2. **for** $k = 1, 2, \ldots, s$ **do**
    // *SpMV* with local submatrix and $s-k$ ghost of A,
    // and optionally *Precond* with M:
3. $\mathbf{v}_{j+k} := AM^{-1}\mathbf{v}_{j+k-1}$
4. **end for**
  // Block Orthonormalization (*BOrth*):
  $R_{1:j,j+1:j+s} := Q_{1:j}^T V_{j+1:j+s}$ with `MPI_Allreduce`
  $Q_{j+1:j+s} := V_{j+1:j+s} - Q_{1:j} R_{1:j,j+1:j+s}$
5. // CholQR factorization:
  $G := Q_{j+1:j+s}^T Q_{j+1:j+s}$ with `MPI_Allreduce`
  compute $R_{j+1:j+s,j+1:j+s}$, Cholesky factor of G
  $Q_{j+1,j+s} := Q_{j+1:j+s} R_{j+1:j+s}^{-1}$
7. Extend projected Hessenberg matrix H with s columns
end for

Fig. 3. Pseudocode of CA-GMRES, where t and s are *MPK* and *BOrth* step sizes, respectively, and m is the restart length.

following j-th step, it uses all the information available from the orthogonalization at the $j-\ell$ step, i.e.,

$$B_{1:\ell+1,1:\ell} = \text{bidiag}\begin{pmatrix} \sigma_1 & \sigma_2 & \ldots & \sigma_\ell \\ 1 & 1 & \ldots & 1 \end{pmatrix},$$

$$B_{\ell+1:j+1,\ell+1:j} = \begin{pmatrix} h_{1,1} & \ldots & h_{1,j} \\ h_{2,1} & \ddots & \vdots \\ & \ddots & h_{j,j} \\ & & h_{j+1,j} \end{pmatrix}.$$

Hence, the projected matrix $H_{1:j+1,1:j}$ is computed by $H_{1:j+1,1:j} := R_{1:j+1,1:j+1} B_{1:j+1,1:j} R_{1:j,1:j}^{-1}$.

Each all-reduce is overlapped with the $\ell-1$ iterations, including the $\ell-1$ all-reduces and neighborhood collectives needed over the iterations. In addition, the pipelined algorithm only performs one all-reduce at each iteration, compared to the two all-reduces in standard GMRES. To avoid the second all-reduce, the algorithm assumes orthogonality of the basis vectors $Q_{1:j}$, and computes the vector norm $h_{j+1,j}$ implicitly.

Pipelining introduces two main trade-offs. First, since the auxiliary matrices H and G are indirectly computed, it could lose numerical stability due to the loss of orthogonality among the basis vectors. Second, it requires additional computation to apply the change-of-basis to the Krylov vectors V.

C. Communication-avoiding GMRES

Communication-avoiding (CA) Krylov methods are based on s-step methods [13], [14], [15], [16], [17]. These were originally proposed as a way to improve Krylov methods' theoretical upper bound on parallelism. s-step methods generate s times as many basis vectors per iteration as their conventional counterparts. In order to avoid communication, the CA variants of the computational kernels (e.g., the orthogonalization and matrix powers kernels), that can be implemented with the minimum communication costs, have been integrated into the s-step solvers [3], [18], [19], [20]. The effectiveness of such CA implementations to improve performance has already been demonstrated on current computers, including shared-memory multicore CPUs [3], distributed-memory CPUs [21], multiple GPUs on a single compute node [22], and a hybrid CPU/GPU cluster [6].

Figure 3 shows CA-GMRES' pseudocode. For the matrix powers kernel (*MPK*), to avoid the point-to-point communication needed for *SpMV*, the local submatrix of A on each process is extended with the $(s-1)$-level ghost boundary elements that are the nonlocal entries $s-1$ edge away from the local entries in the adjacency graph of A. To generate a new set of s Krylov vectors, each process first exchanges the s-level ghost entries of the starting vector $\mathbf{q}_j$. After this round of point-to-point communication, each process may independently apply *SpMV* s times without further communication. Though *MPK* reduces the communication latency to generate the s basis vectors by a factor of s, at each k-th step of *MPK*, it redundantly applies *SpMV* to the $(s-k)$-level ghost elements. This means extra computation.

Our block orthogonalization procedure uses block classical Gram Schmidt [23] to orthogonalize the new Krylov vectors generated by *MPK* against the previously orthonormalized basis vectors. We then orthonormalize each "block" of new basis vectors using the Cholesky QR (CholQR) factorization [24]. In our previous studies, these block orthogonalization procedures have performed well on both distributed-memory CPUs [21] and on a hybrid CPU/GPU cluster [6]. They are also easy to implement on new architectures, since their performance depends mainly on dense matrix-matrix multiply.

CA algorithms have three main drawbacks. First, at each step of *MPK*, each generated vector becomes increasingly linearly dependent against the previous basis vectors in that *MPK* round. This limits the basis length in practice. Mitigation techniques are the same as in Pipelined GMRES, namely choosing a different basis. Second, though *MPK* reduces the communication latency cost, it introduces redundant storage and computation among the neighboring processes, which are associated with the s-level ghost entries. Furthermore, the total communication volume could increase depending on the sparsity pattern of the matrix A. Third, though several preconditioners have been proposed that work with CA methods [7], [6], it is a challenge to integrate preconditioning into *MPK* without making it non-CA.

III. Algorithmic Variants

In this sections, we describe two algorithmic variants of the CA and pipelined methods that are designed to improve the performance of the solvers by combining the insights or the strengths of these two methods. We also consider an option to reduce the computational overhead associated with the pipelining method.

A. Avoiding all-reduces

In our previous studies with the CA methods [6], [21], [22], [25], most of our performance improvement came from using

the block algorithm to orthogonalize the basis vectors. This is mainly because obtaining the performance improvement from *MPK* remained to be a challenge due to the storage and computation overheads, and the potential increase in the total communication volume. In addition, though preconditioning is essential for reducing the solvers' iteration count in practice, it is a challenge to integrate preconditioning into *MPK*.

For our performance comparison in this paper, we focus on a slight variation of the CA method, that relies on the block orthogonalization to reduce the global communication cost, but performs the neighborhood communication before applying each *SpMV* to generate the Krylov vector. This variant goes along with the motivation for designing the pipelined methods, that at a large-scale, the parallel performance of the solver is limited by the all-reduce needed for the orthogonalization. In addition, this variant allows us to use any preconditioner, which is a great benefit in practice. This also leads to an interesting performance comparison of the two techniques, one to reduce and the other to pipeline the all-reduces. Both approaches generate a set of s Krylov vectors by simply calling *SpMV* s times but then one performs two all-reduces to orthogonalizes all of the s vectors at once, while the other pipelines the all-reduces.

Though this variant addresses the first trade-off of the CA method (i.e., the overheads associated with *MPK*), it could still suffer from the numerical instability due to the application of the matrix powers without orthogonalization. Hence, to maintain the numerical stability, we generate the Newton basis [12], whose shifts are the Ritz values computed at the first restart (our first restart cycle is based on the standard GMRES).

B. Pipelining block-orthogonalization

The CA method in Section III-A only performs the all-reduce every s steps. Though we generate the Newton basis, the step size s is limited due to the numerical instability (e.g., in our experiments, $s \leq 10$). Hence, the global communication can still become the performance bottleneck. To hide this communication cost, we consider another variant that pipelines the all-reduces needed for the block orthogonalization in our CA implementation. This variant can be also considered as an extension of the pipelined method, that integrates the block-orthogonalization procedure. Figure 4 shows the pseudocode of our implementation. Instead of launching the non-blocking all-reduce after generating each Krylov vector, this variant perform the all-reduce after a set of t basis vectors are generated. Once the required all-reduce is completed, we use the block procedure to orthogonalize the t vectors at once. When each all-reduce takes longer than a single *SpMV*, instead of increasing the pipeline depth, this variant allows us to hide the communication behind the block generation of t orthonormal basis vectors.

Compared to the previous pipelined method, this variant require the deeper pipelining of depth $t\ell$, which leads to the additional $t\ell$ iterations to drain the pipeline at the end of each restart cycle. However, we perform only the orthogonalization

Pipelined CA-GMRES(A, M, $\mathbf{b}$, ℓ, m):
repeat
for $j = 1, 1+s, \ldots, m$ **do**
  // Matrix Powers Kernel:
1. **for** $k = 1, 2, \ldots, t$ **do**
2. $i := j + k - \ell t + 1$
3. `MPI_Isend` and `MPI_Irecv` for 1-level ghost of $\mathbf{q}_j$ among neighbors and then `MPI_Wait`
4. $\mathbf{v}_{j+k} := AM^{-1}\mathbf{v}_{j+k-1}$
5. $\mathbf{v}_{j+k} := (\mathbf{v}_{j+k} - V_{i:j+k-1}\mathbf{h}_{1:i-1,i-1})/h_{i,i-1}$
6. **end do**
7. $i := j + t - \ell t + 1$
8. **if** $i > 0$ **then**
9. `MPI_Wait(tag = i)`
10. **for** $k = i, \ldots, i+t-1$ **do**
11. Update $\mathbf{r}_{1:k,k}$ using $R_{1:k-1}$
12. Generate $\mathbf{h}_{1:k,k-1}$
13. **end for**
  // Block orthonormalization:
14. $k := i + t - 1$
15. $Q_{i:k} := V_{i:k} - Q_{1:i-1}R_{1:i-1,i:k}$
16. $Q_{i:k} := Q_{i:k}R^{-1}_{i:k,i:k}$
  // generate next starting vector:
17. $\mathbf{v}_{j+t} := (\mathbf{v}_{j+t} - V_{i:j+t-1}\mathbf{h}_{1:i-1,i-1})/h_{i,i-1}$
18. **end if**
19. $R_{1:j+s+1,j:j+s+1} := [Q_{1:t}V_{t:j}]^T V_{j+1:j+t}$, with `MPI_Iallreduce(tag = j + 1)`
end for

Fig. 4. Pseudocode of pipelined GMRES with block-orthogonalization, where ℓ is the pipeline depth, t is the step size, and m is the restart length.

for these additional iterations, while the orthogonalization is not performed for the first $t\ell$ iterations. Hence, just like the previous pipelining method, the main computational overhead comes from the application of the change-of-the-basis to the Krylov vectors V (Line 5 in Figure 4). On the other hand, in practice, the maximum step size t, that this variant can take, may be smaller than the step size, s, used in the previous CA implementation since the pipelined method often suffers from the numerical instability when used with a large pipeline depth. (e.g., $s = 10$ and $t = 5$ with $\ell = 2$ in our experiments). Hence, this variant trades off the benefit of the block orthogonalization (e.g., intra-process communication) to pipeline the global communication. We expect that at the large-scale, the global communication becomes the performance bottleneck, and this variant obtains a higher performance.

C. Forming partial change-of-basis

The pipelined GMRES with the block orthogonalization in Section III-B orthogonalizes the basis vectors Q using BLAS-3, and compared to the previous pipelining method in Section II-B, it could reduce the communication latency cost of the orthogonalization by a factor of t. However, though it does not require the inter-process communication, the change-of-the-basis is applied to the vectors V one vector at a time using BLAS-2 kernels.

In the previous approach [4], to maintain the numerical stability, at the j-th step, all the information from the orthogonalization procedure at the $(j-\ell)$-th step was used. However, the equation (1) holds for any matrix B of full column rank. In our experiments, in order to reduce the computational

overhead, we explore the change-of-basis B that only applies *partial* orthogonalization of the basis vectors Q to the Krylov vectors V. Since the main motivation of the change-of-the-basis is to avoid the resulting vector $\mathbf{v}_{j+k}$ to become numerically linearly dependent to the previous vectors, we will remove only the components of $\mathbf{v}_{j+k}$, which have large magnitude. In other words, we will set the elements of B, whose magnitude is less than a specified tolerance τ, to be zero (e.g., in our experiments, $\tau = h_{i,i-1}\|A\|_2 10^{-3}$, where $\|A\|_2$ is approximated by the largest Ritz value compute at the first restart). As the small elements are discarded, the value of $h_{i,i-1}$ used to normalize the Krylov vector is updated. More sophisticated schemes have been previously used, for example, to reorthogonalize the basis vectors for computing eigenvalues [26], [27].

IV. Implementations

To compare the performance of different variants of GMRES on a distributed-memory computer with multicore CPUs, we designed two implementations of each algorithm; one that relies on the threaded computational kernels and non-blocking MPI collectives, and the other one based on a shared-memory runtime system to locally schedule both the computational and communication tasks of the process. We chose these two programming models considering the programability and performance of our implementations.

A. Implementation with dynamic scheduler

The effectiveness of a sequential task-based programming model to exploit the compute power of the modern manycore node architectures has been demonstrated through superscalar schedulers [28], [29], [30], [31] and runtime systems [32], [33], [34]. To utilize this programming model, the programmer inserts the tasks along with their data access types (e.g., input or output). Then, the task dependencies are automatically derived to execute the tasks consistently with their sequential execution. Hence, the programability of the parallel code is often at about the same level as that of a sequential code. This lead to the recent adaptation of the programming model by the OpenMP standard. In this work, we used this programming model to execute our solver on a distributed-memory system, relying on a shared-memory runtime system to locally schedule both the computational and communication tasks of the process on the multicore node architecture. For the runtime system, we focused on QUARK [5] which was developed for executing the linear algebra algorithms on a shared-memory multicore architectures [35]. Our choice is mainly due to our familiarity with the runtime system, but as we detail below, QUARK provides many features that are useful for our studies but not available, for example, in OpenMP, yet.[1] Figures 5 and 6 show our QUARK implementation of GMRES and wrapper for *SpMV* tasks, respectively. They preserve the structure of the sequential algorithm in Figure 1, and enable a high productivity.

```
iter = 0;
while iter < maxiters do
  int stop_iter = min(maxiters, iter + restart);
  int restart_i = stop_iter - iter;
  for (j = 0; iter<stop_iter; j++, iter++) {
    // neighborhood comm for SpMV
    QUARK_CORE_zspmv_gather(iter, A, G(0, 0));
    for (i = 0; i < mt; i++)
      QUARK_CORE_zspmv_gemv(
        // SpMV: Q(:, j+1) := A*Q(:, j)
        i, A.mbi[i], A, G(0, 0), Q(i, j+1),
        // GEMV: H(:, j) := Q(:, 0:j)'*Q(:, j+1)
        j+1, Q(i, 0), ldq, T(i), ione);
    // local accumulation and global reduce, H(1:j, j) := sum_{k=0}^{mt-1} T(k)
    QUARK_CORE_zgeadd_dist(
      NoTrans, j+1, ione, 0, mt-1,
      zone, T(0), ldh, zone, &H(0, j), ldh);
    for (i = 0; i < mt; i++)
      QUARK_CORE_zgemv_dot(
        NoTrans, A.mbi[i], j+1,
        // GEMV: Q(:, j+1) -= Q(:, 1:j)*H(1:j, j)
        zmone, Q(i, 0), ldq, &H(0, j), ione, zone, Q(i, j+1), ione
        // DOT: T(i) := Q(i, j+1)'*Q(i, j+1)
        T(i));
    // local accumulation and global reduce, H(j+1, j) := sum_{k=0}^{mt-1} T(i)
    QUARK_CORE_zgeadd_dist(
      NoTrans, 1, 1, 0, mt-1,
      zone, T(0), ldt, zone, &H(j+1, j), ldh
    for (i = 0; i < mt; i++)
      QUARK_CORE_zlascal_copy(
        UpperLower, A.mbi[i], ione,
        // LASCL: Q(:, j+1) /= H(j+1, j)
        &H(j+1, j), Q(i, j+1), ldq,
        // LACPY: G_local := Q(:, j+1) for next SpMV
        G_local(i, 0), A.global_m);
  end for
  // prepare to restart
end while
```

Fig. 5. GMRES implementation with QUARK, where mt is the number of local blocks and mbi[i] is the i-th block size.

Our main motivation for using the runtime system is to ensure our all-reduces to progress and be overlapped with other tasks. In order to do this, we wrap both the neighborhood communication needed for *SpMV* (using MPI_Isend and MPI_Irecv, and then MPI_Wait) and the global collective needed for *Orth* (using MPI_Allreduce) in tasks. We rely on MPI_THREAD_MULTIPLE support for the multiple independent communication tasks to be executed from different threads at the same time. With this programming model, a few physical cores could be idle while the communication task assigned to the core is waiting for the corresponding communication tasks to be executed by other processes. However, in many cases, there are only limited number of communication tasks being executed at a time (i.e., small pipelining depth, ℓ), and on many-core system, the idle time may not be significant. To reduce the idle time of the cores, QUARK allows us to set the priorities of the tasks. Using the priority tags, we encourage QUARK to schedule the tasks in a specific order such that the corresponding communication tasks are scheduled by all the processes around the same time, reducing the idel time but still allowing a flexible order of execution.

To overlap the neighborhood communication with the local computation for *SpMV*, we split the local submatrix into two

[1]Node-local matrix is stored in the column-major format. Hence, the tasks work on the data which is not contiguous in the memory. This violates the requirement that the tasks in OpenMP work on the contiguous data blocks.

```
void QUARK_CORE_zspmv_gemv(
      // arguments for SpMV
      int i, int mb, Sparse_desc A, Complex64_t *x, Complex64_t *y,
      // arguments for GEMV
      int n, Complex64_t *Z, int ldz, Complex64_t *w, int incw)
{
  // subroutine to be executed
  Task *task = QUARK_Task_Init(quark, CORE_zspmv_gemv_quark, task_flags );
  // arguments for SpMV, y = A*x
  Pack_Arg(task, sizeof(int),                  &A.blk[i], VALUE);
  Pack_Arg(task, sizeof(int),                  &mb,       VALUE);
  Pack_Arg(task, sizeof(Sparse_desc),          &A,        VALUE);
  Pack_Arg(task, sizeof(Complex64_t)*A.n,      x,         NODEP);
  Pack_Arg(task, sizeof(Complex64_t)*mb,       y,         INOUT | LOCALITY);
  // arguments for GEMV, w = Z'*y
  Pack_Arg(task, sizeof(int),                  &n,        VALUE);
  Pack_Arg(task, sizeof(Complex64_t)*mb*n,     Z,         INPUT);
  Pack_Arg(task, sizeof(int),                  &ldz,      VALUE);
  Pack_Arg(task, sizeof(Complex64_t)*mb,       w,         INOUT);

  // i-th SpMV depends on neighboring tiles
  for (int k=0; k<A.neighbors[i][0]; k++) {
    int offset = A.neighbors[i][k+1];
    Pack_Arg(task, sizeof(Complex64_t)*mbk, &x[offset], INPUT);
  }
}
```

Fig. 6. QUARK wrapper for *SpMV* followed by *GEMV*. When all the dependencies are satisfied, CORE_zspmv_gemv_quark is executed.

parts, interior points, that are only connected to the local points, and the local interface that are connected to a non-local point. With this partition, *SpMV* with the interior points can be performed while waiting for the neighborhood communication. After the communication is completed, *SpMV* on the interface is performed. To schedule on n_t physical cores, we split the interior submatrix into $n_t - 1$ parts, leaving one core for the communication. Unfortunately, when the same partitioning is used for the orthogonalization, this leads to load imbalance among the orthogonalization tasks since the interface is often much smaller than the interior. To reduce this load imbalance, we tried repartitioning the vectors into n_t parts of an equal size for the orthogonalization, but this lead to the data movement between the physical cores, slowing down the iterative process. Alternatively, we could simply append the interface to the last block of the interior. However, to leave one core for the point-to-point communication, the submatrix is partitioned into $t-1$ blocks for *SpMV*, and appending the interface to the last block will leave one core idle for the orthogonalization process. For our experiments, we used the same partitioning for *SpMV* and *Orth*, which obtained the best performance in most cases.

Another useful feature of QUARK is the "locality" tag. In order to obtain a high-performance of the Krylov solver, the computational tasks need to be scheduled on the cores that are close to the required data. Using the locality tags, we encourage QUARK to schedule the tasks on the cores that are in the vicinity of the data. Another plausible approach merges the computational kernels that work on the same data into a single task. This also reduces the number of tasks and the scheduling overhead of the small tasks appearing in the sparse iterative solvers. As shown in Figures 5 and 6, we use both approaches to improve the solver's performance.

B. Implementation with non-blocking communication

Our second implementation relies on the non-blocking point-to-point and all-reduce communication supported by MPI, and the threaded computational kernels (e.g., threaded MKL for sparse and dense matrix operations). Our MPI implementation is almost identical to our QUARK implementation, except that we directly call the core functions without the QUARK wrappers (e.g., CORE_zspmv_gemv instead of QUARK_CORE_zspmv_gemv in Figure 6). To provide enough computation for the thread kernels to exploit the parallelism, we do not partition the local submatrices. The only exception is for *SpMV*, where the submatrix is partitioned into the interior and interface in order to hide the neighborhood communication behind *SpMV* with the interior points.

Compared to the task-based model in Section IV-A, the pure MPI-based implementation has trade-offs in term of programability. Clearly, the task-based model could introduce difficulty to keep track of the task dependencies, especially with the pipelined methods which have several independent tasks from different phase of iteration. However, with the task-based model, we do not have to worry about draining the pipeline for orthogonalizing the last ℓ basis vectors since the orthogonalization will be scheduled at any time after the corresponding all-reduce is completed (with a low priority).[2] Moreover, unlike our QUARK implementation, where several cores may be idle waiting for the communication tasks to be scheduled by other processes, our MPI implementation can potentially utilize all the cores. However, in our experiments using the progress thread of MPICH, the performance of our solver was significantly degraded if we do not leave one core open for the progress thread.

V. Performance Results

We now study the performance of our particular implementations of the GMRES solvers discussed in this paper. First in Sections V-A and V-B, we discuss our experiment setups and benchmark the performance of MPI used in our experiments. Then, in Section V-C, we compare the performance of different solvers, using one thread per process. Finally, in Section V-D, we compare the performance of our two implementations, using multiple threads per process.

A. Experiment setup

We conducted our experiments on the Tsubame2 Computer at the Tokyo Institute of Technology. Each of its nodes features two six-core Intel Xeon CPUs, which are connected by 80Gbps QDR InfiniBand. We compiled our code using `mpicc` from MPICH version 3.2 (configured with `--enable-threads=multiple`), and linked it to MKL version XE2013.1.046. We initialized the MPI library with `MPI_THREAD_MULTIPLE`, and used `MPI_Iallreduce` implemented using TCP and IP-over-Infiniband. Unless otherwise specified, our test matrix is a five-point 2D Laplace with a

[2]We plan to study the effects of the block sizes, as a tuning parameter, to schedule these independent tasks on manycore systems.

#bytes	$t_{\text{ovrl}}[\mu\text{sec}]$	$t_{\text{pure}}[\mu\text{sec}]$	$t_{\text{CPU}}[\mu\text{sec}]$	overlap[%]
0	255.83	230.29	242.35	89.46
8	312.37	242.53	272.48	74.37
16	268.53	225.00	254.62	82.91
32	264.67	222.07	251.30	83.05
64	281.10	237.46	249.84	82.53
128	267.30	227.92	253.52	84.47
256	278.94	227.63	265.70	80.69

Fig. 7. Communication and computation overlap, $n_p = 240$, progress threads.

square grid, $n_x \times n_x$. Previously, the numerics and performance of the pipelined and CA GMRES have been separately studied using various matrices. In this paper, we focus on comparing the performance of the two methods. Due to the lack of space, instead of showing high-level performance using different types of matrices, we provide detailed performance results focusing on this particular class of matrices, which was used in many studies (including [4]) and provides representative performance. For the solver performance, we report the best performance among five runs.

As discussed in Section IV-A, when we use the MPI's progress thread, the solver often performs better, leaving one spare core for the progress thread. It was also critical to bind each process to a set of specific cores. For instance, when using one solver thread per process, we bound each process to two unique cores. In other cases, we bound each process to a socket with six cores and launched five threads inside the process leaving one core available for the progress thread. We also found that in some cases, for example with a large pipeline depth, using a progress thread could lower the solver performance. This could be due to the fact that between each `MPI_Iallreduce` and corresponding `MPI_Wait`, there are other MPI calls, like `MPI_Wait` matching `MPI_Isend` and `MPI_Irecv` for *SpMV* as well as `MPI_Iallreduce` still in-progress from the previous iteration (which may allow `MPI_Iallreduce` to progress without the progress thread). In all, `MPI_Iallreduce` is only one of the in-progress communication that MPI's internal progress thread must advance. In this paper, we focus on demonstrating the behavior of different solvers, showing the results with a progress thread for the consistency. We plan to extend our benchmark results with additional configurations in a future report.

B. Non-blocking all-reduce

We have tested different implementations and versions of MPI and different installation and runtime configurations. Due to the limited space, we only present results of MPICH used in our experiments. Figure 7 tests how well `MPI_Iallreduce` overlaps with computation, using Intel MPI Benchmark (IMB). The time for a call to a non-blocking collective immediately followed by a call to `MPI_Wait` is measured by t_{pure} – a *purely* communication-bound execution. The computation time t_{CPU} measures the time taken by a a repeated computation of a small in-cache dense matrix-vector multiply that is supposed to take as long as the pure call but with the actual non-blocking communication happening in the background. It is clear that $t_{\text{pure}} \leq t_{\text{CPU}}$ and the equality holds only if there is no interference between the communication and computation.

#bytes	80	160	240	320	400	480	560	640
ℓ calls `MPI_Iallreduce` followed by `MPI_Waitall`, progress threads								
$n_p = 60$	4.62	4.86	5.55	6.02	6.10	6.83	6.62	6.45
120	4.22	4.81	6.32	5.98	6.43	6.76	7.11	6.48
ℓ calls to `MPI_Allreduce` from n_t threads per process, $n_p = 20$.								
$n_t = 2$	8.84	9.27	8.78	8.38	8.41	9.05	8.91	9.09
6	7.75	8.51	8.19	8.77	9.18	9.60	10.21	10.57

Fig. 8. Pipeline results with pipeline depth $\ell = 10$ and process count n_p.

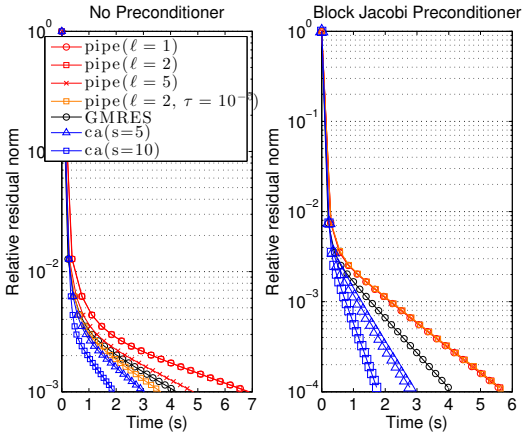


Fig. 9. Convergence of GMRES(60) with respect to the iteration time for 2D Laplace, $n_x = 512$. Time was measured at each restart, identified by marker. All the solvers converged equivalently with respect to the iteration count.

Total time to finish the simultaneous communication and computation is marked as t_{ovrl} and, accordingly, the percentage of overlap is reported as $(t_{\text{pure}} + t_{\text{CPU}} - t_{\text{ovrl}})/min(t_{\text{pure}}, t_{\text{CPU}})$. The benchmark shows good overlaps of between $74 \sim 89\%$.

We next study if the pipelined all-reduces can be overlapped on each other either through multiple calls to `MPI_Iallreduce` followed by `MPI_Waitall` or multiple calls to `MPI_Allreduce` from different threads at a time. Figure 10 shows the ratio of the time needed to pipeline ten all-reduces over the time needed to perform one `MPI_Iallreduce` followed by `MPI_Wait`. These results indicate that with our current implementation, the pipelining of all-reduce is insignificant, and a deeper pipeline is mainly for providing more computation to hide the all-reduce behind.

C. Comparison of different solvers – pure MPI

In this section, we study the performance of different variants of the solver using one thread per process (without using QUARK). Figure 9 shows the convergence results of different solvers. We found that the solver often loses its numerical stability faster using a larger pipeline depth ℓ than using a larger step size s (e.g., due to the loss of the orthogonality among the basis vectors). For this experiments, we only use 12 processes, and the pipelined GMRES did not improve the performance. However, as we will show later, with a large enough number of processes, we could stably use a sufficiently large pipeline depth or step size to obtain a good speedup (e.g., we need $s = 5$ or $\ell = 2$ to obtain speedups, but the solver was stable with $s = 10$ or $\ell = 10$). Though we do

ℓ	$n_p = 120$		$n_p = 180$		$n_p = 240$		$n_p = 300$	
	block	non	block	non	block	non	block	non
0	0.40		0.37		0.34		0.35	
2	0.42	0.35	0.31	0.28	0.27	0.21	0.26	0.21
5	0.41	0.34	0.31	0.25	0.27	0.21	0.26	0.20
10	0.40	0.33	0.31	0.25	0.27	0.21	0.26	0.20

Fig. 10. Time in seconds of 20 restart-cycles of pipelined GMRES(30) for 2D Laplace ($n_x = 1024$) and number of processes, n_p. "block" and "non" use blocking and non-blocking all-reduces. When $\ell = 0$, standard GMRES is used.

not show the convergence for the rest of the experiments, all the solvers obtained the equivalent convergence. This is true even with the preconditioner since our CA implementation communicates for each *SpMV* and can use any preconditioner. The right plot of Figure 9 used the block Jacobi preconditioner, where each process applies the sparse approximate inverse of its local submatrix.

Figure 10 shows the performance of Pipelined GMRES with different configurations. In the table, under "block," we replaced MPI_Iallreduce with MPI_Allreduce to study the effects of overlap, while when $\ell = 0$, we used the standard GMRES. Even with the blocking all-reduce, Pipelined GMRES improved the GMRES' performance because it only performs one all-reduce per iteration, compared to two all-reduces performed by the standard GMRES. Then, using non-blocking all-reduce, the performance was further improved by a factor of about $1.3\times$.

Figure 11 compares the performance of different solvers. Even though Pipelined GMRES performs extra computation, it improved the performance of standard GMRES when the communication latency of all-reduce became significant with a large enough number of processes. On the other hand, CA-GMRES obtained the speedups even on a small number of processes, reducing both the inter-process and intra-process communication. Furthermore, Pipelined CA-GMRES improved the performance of Pipelined GMRES using block-orthogonalization, and obtained the speedups of up to $1.67\times$ over the best pipelined performance. Pipelined CA-GMRES also obtained the speedups of up to $1.22\times$ or $1.09\times$ over CA-GMRES with $s = 5$ or 10 by pipelining the all-reduce. We observed that as the process count increased, the latency cost of all-reduce, and hence the speedup from pipelining the all-reduces, increased.

Figure 12 shows the similar performance results for solving 27-points 3D problems. Since the coefficient matrix has more nonzeros per row and a smaller restart cycle was used, the block orthogonalization lead to a less performance improvement, compared to those obtained for the 2D problems in Figure 11. As shown in Figure 13, we have also observed similar results for the matrices from the University of Florida Sparse Matrix Collection (the matrices are equilibrated using the largest elements in each row and column, and distributed using METIS). At this small-scale, Pipelined CA-GMRES improved the performance of Pipelined GMRES, but pipelining the all-reduce did not improve the performance of CA-GMRES (e.g., the latency cost has been reduced by block orthogonalization and may not be significant enough). With a larger number of processes, we expect Pipelined CA-GMRES to improve performance of CA-GMRES.

D. Comparison of two implementations – MPI+Threads

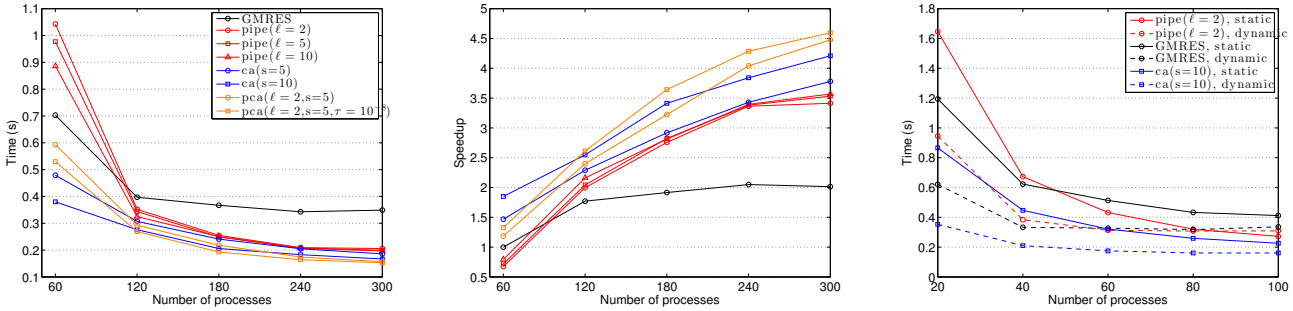
We now compare the performance of our two implementations, one using threaded MKL and nonblocking MPI collectives, and the other using QUARK. Figure 14 shows the solvers' performance on one node with different process / thread configurations. For these experiments, we did not use Pipelined GMRES and we disabled progress threads. For the very tall and skinny dense matrices appearing in the orthogonalization procedure, the threaded MKL may not be optimized (e.g., DSYRK), and our QUARK implementation may perform better, leading to the higher performance of the solver. At the end, in most cases, the optimal performance was obtained using all the cores with one process either per socket or per core. We also see that even on one node, CA-GMRES obtained good speedups over standard GMRES.

The execution traces in Figure 15 clearly show the different behavior of the solvers. For instance, GMRES has the multiple synchronization points at each iteration, while CA-GMRES performs s *SpMV* before the block orthogonalization. Pipelined GMRES breaks these synchronization points, but has more computation in the matrix-vector multiplies. In addition, in our QUARK implementation, compared to standard GMRES, CA-GMRES often spends more time in point-to-point communication for *SpMV*. This is because CA solver schedules s *SpMV*'s without a global synchronization, while the standard solver globally synchronizes between each *SpMV* due to the all-reduce needed for the orthogonalization. Hence, it is more likely that in the CA solver, the corresponding communication tasks are scheduled by different processes at different time, increasing the communication time.

Figure 11(c) compares the performance of our two implementations. With a relatively small number of processes, our QUARK implementation could utilize the physical cores more effectively, obtaining higher performance than our MPI implementation (both in Figures 11 and 16). However, with a larger number of processes, our MPI implementation seems to gain more advantage. This may be because QUARK is not scheduling the communication tasks at the earliest time, and we are looking to improve its performance. Finally, Figure 16 compares the solver performance using our MPI implementation with multiple threads per process. As before, Pipelined CA-GMRES obtained the speedups of up to $1.34\times$ over Pipelined GMRES through block orthogonalization. By pipelining the all-reduces, Pipelined CA-GMRES also obtained the speedups of up to $1.12\times$ or $1.02\times$ over CA-GMRES with $s = 5$ or 10, respectively.

VI. Conclusion

We began this work by comparing the performance of Pipelined GMRES and Communication-Avoiding (CA) GMRES on a distributed-memory computer. We implemented the solvers in two different ways. The first way builds on a



(a) Time in seconds, six processes per node, one thread per process.

(b) Speedup over GMRES on 10 nodes, six processes per node, one thread per process.

(c) Time for two different implementations, one process per socket, five threads per process.

Fig. 11. 20 restart cycles of GMRES(30) with 2D Laplace ($n_x = 1024$).

ℓ	s/t	τ	number of processes 60	120	180	240
GMRES						
–	–	–	2.10 (1.00)	1.25 (1.00)	0.88 (1.00)	0.64 (1.00)
Pipelined						
2	–	–	2.36 (0.89)	1.36 (0.92)	0.88 (1.00)	0.68 (1.00)
5	–	–	2.32 (0.91)	1.27 (0.98)	0.84 (1.05)	0.65 (1.05)
10	–	–	2.20 (0.95)	1.19 (1.05)	0.83 (1.06)	0.61 (1.11)
CA						
–	5	–	1.85 (1.14)	1.06 (1.18)	0.74 (1.19)	0.49 (1.38)
–	10	–	1.75 (1.20)	1.04 (1.20)	0.70 (1.26)	0.47 (1.45)
Pipelined CA						
2	5	0.0	2.03 (1.03)	1.13 (1.11)	0.78 (1.13)	0.51 (1.33)
2	5	0.001	1.96 (1.07)	1.07 (1.17)	0.72 (1.22)	0.49 (1.39)

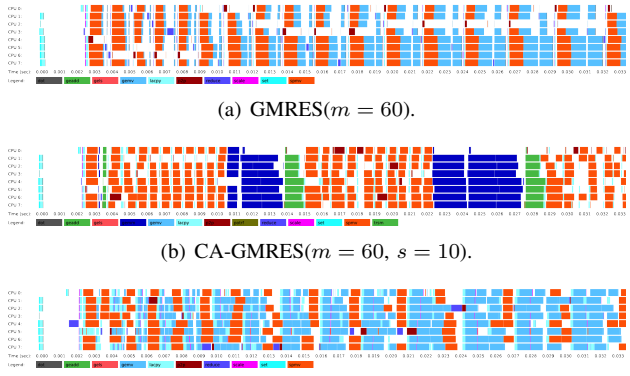
Fig. 12. Time in seconds for 20 restart-cycles and $m = 20$, with 27-points 3D Laplace ($n_x = 128$). The numbers in parenthesis are the speedups over GMRES with the same processor count.

	n (M)	$\frac{nnz}{n}$	GMRES (s)	Pipelined	CA	Pipelined CA
G3_Circuit	1.6	4.8	0.43	1.30	1.59	1.54
thermal2	1.2	7.0	0.43	1.54	1.87	1.72
atmosmodd	1.3	6.9	0.74	1.64	1.95	1.72

Fig. 13. Speedups over GMRES with $(m, \ell, s, t) = (30, 2, 10, 5)$ on 240 processes for matrices from UF Sparse Matrix Collection.

	$n_p \cdot n_t$ 1 · 1	1 · 3	1 · 6	1 · 12	2 · 1	2 · 3	2 · 6	12 · 1
GMRES($m = 30$)								
mpi	15.6	9.6	9.2	6.7	7.6	5.5	4.8	4.6
dyn	16.3	10.7	7.6	6.3	7.8	5.5	4.8	4.6
CA-GMRES($m = 30$, $s = 10$)								
mpi	11.5	7.3	7.4	7.1	5.9	3.8	3.8	2.4
dyn	10.3	6.9	4.1	3.6	5.2	3.5	2.4	2.1
GMRES($m = 60$)								
mpi	52.4	32.3	24.3	29.3	25.5	18.6	16.1	13.3
dyn	53.9	34.9	25.3	20.0	25.9	20.5	16.3	15.3
CA-GMRES($m = 60$, $s = 10$)								
mpi	31.7	19.3	21.4	18.6	16.3	10.3	10.6	6.4
dyn	27.7	17.2	9.9	7.8	13.7	8.9	5.9	5.0

Fig. 14. Time in seconds for 10 restart-cycles with 2D Laplace ($n_x = 1024$) using different $n_p \cdot n_t$, where n_p and n_t are the number of processes and the number of threads per process, respectively. "mpi" and "dyn" denote our MPI and QUARK implementations.



(a) GMRES($m = 60$).

(b) CA-GMRES($m = 60$, $s = 10$).

(c) pipelined GMRES($m = 60$, $\ell = 2$).

Fig. 15. Execution traces of one of two processes, used for GMRES solvers.

threaded BLAS and LAPACK library and nonblocking MPI collectives. The second uses the QUARK shared-memory run-time system to schedule computational and communications tasks on each MPI process. We also developed a new algorithm, Pipelined CA-GMRES, that combines the strengths of the above two methods. It uses fewer global all-reduces than standard GMRES, by applying the same block orthogonalization approach as CA-GMRES. Like Pipelined GMRES, it overlaps those all-reduces with useful work, thus making CA-GMRES less synchronous. In our experiments, Pipelined CA-GMRES performed up to 1.67 times better than Pipelined GMRES, thanks to the use of block orthogonalization. Thanks to overlapping the all-reduces with useful work, Pipelined CA-GMRES performed up to 1.22 times better than CA-GMRES when the same step size is used (i.e., $t = s$), and 1.09 times better when the total pipeline depth is equal to the step size (i.e., $\ell t = s$).

The performance of these solvers depends on many factors, including the hardware, the underlying software libraries, and the configurations used to run the solver. In future work, we plan more extensive experiments in order to understand these factors better. These experiments will include running on a hybrid CPU/GPU cluster, where we have access to the source code of the optimized GPU kernels. We also plan to explore other task-parallel run-time systems, as well as the

ℓ	s/t	τ	number of processes 20	40	60	80	100
GMRES							
–	–	–	1.19	0.62	0.50	0.43	0.41
Pipelined							
2	–	–	1.66	0.66	0.43	0.34	0.28
5	–	–	1.59	0.63	0.43	0.32	0.27
10	–	–	1.48	0.59	0.40	0.31	0.28
CA							
–	5	–	1.04	0.50	0.36	0.29	0.26
–	10	–	0.86	0.45	0.33	0.25	0.23
Pipelined CA							
2	5	0.0	1.22	0.52	0.35	0.27	0.23
2	5	0.001	1.10	0.49	0.33	0.26	0.23

Fig. 16. Time in seconds for twenty restart cycles and $m = 20$, 2D Laplace ($n_x = 1024$), one process per socket, five thread per process.

development version of OpenMPI, where we have a close collaboration with the developers. We are also looking for another opportunity to run our solvers at a larger scale (e.g., through XSEDE). Though in our experiments, the performance of Pipelined GMRES was lower than that of CA-GMRES, we expect the pipelined variant to perform better at larger scales. We have observed that the pipelined method can lose its numerical stability when used with a large pipeline depth and restart cycles. We are investigating techniques to improve the numerical stability of our solver. Our solver is currently maintained in a private Bitbucket repository. We plan to release implementations of some of these solvers through the Trilinos (trilinos.org) project.

REFERENCES

[1] D. A. Patterson, Latency lags bandwith, Communications of the ACM 47 (10).

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

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

[4] P. Ghysels, T. Ashby, K. Meerbergen, W. Vanroose, Hiding global communication latency in the GMRES algorithm on massively parallel machines, SIAM J. Sci. Comput. 35 (2013) C48–C71.

[5] A. YarKhan, J. Kurzak, J. Dongarra, QUARK users' guide: QUeueing And Runtime for Kernels, Tech. Rep. ICL-UT-11-02, University of Tennessee, Innovative Computing Laboratory (2011).

[6] I. Yamazaki, S. Rajamanickam, E. Boman, M. Heroux, M. Hoemmen, S. Tomov, Domain decomposition preconditioners for communication-avoiding Krylov methods on hybrid CPU/GPU cluster, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC), 2014, pp. 933–944.

[7] L. Grigori, S. Moufawad, Communication avoiding ILU0 preconditioner, SIAM J. Sci. Comput. 37 (2015) C217–C246, also available as an INRIA technical report, RR-8266.

[8] H. Morgan, M. G. Knepley, P. Sanan, L. R. Scott, A stochastic performance model for pipelined Krylov methods, CoRR abs/1602.04873.

[9] P. Ghysels, W. Vanroose, Hiding global synchronization latency in the preconditioned conjugate gradient algorithm, Parallel Computing 40 (7) (2014) 224 – 238.

[10] P. Sanan, S. M. Schnepp, D. A. May, Pipelined, flexible Krylov subspace methods, SIAM J. Sci. Comput. 38 (5) (2016) C441–C470.

[11] P. Eller, W. Gropp, Non-blocking preconditioned conjugate gradient methods for extreme-scale computing, SC15 poster session: International Conference for High Performance Computing, Networking, Storage and Analysis (2015).

[12] Z. Bai, D. Hu, L. Reichel, A Newton basis GMRES implementation, IMA Journal of Numerical Analysis 14 (1994) 563–581.

[13] J. van Rosendale, Minimizing inner product data dependence in conjugate gradient iteration, in: IEEE Internatational Conference for Parallel Processing, 1983, pp. 44–46.

[14] A. Chronopoulos, C. Gear, s-step iterative methods for symmetric linear systems, J. Comput. Appl. Math. 25 (1989) 153–168.

[15] R. Leland, The effectiveness of parallel iterative algorithms for solution of large sparse linear systems, Ph.D. thesis, University of Oxford (1989).

[16] H. Walker, Implementation of the GMRES method using Householder transformations, SIAM J. Sci. Stat. Comput. 9 (1988) 152–163.

[17] E. de Sturler, H. van der Vorst, Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers, Applied Numerical Mathematics 18 (2005) 441–459.

[18] M. Hoemmen, Communication-avoiding Krylov subspace methods, Ph.D. thesis, EECS Department, University of California, Berkeley (2010).

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

[20] J. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-optimal parallel and sequential QR and LU factorizations, SIAM J. Sci. Comput. 34 (2012) A206–A239.

[21] I. Yamazaki, 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 Software (HPCC), 2011, pp. 345–354.

[22] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, J. Dongarra, Improving the performance of CA-GMRES on multicores with multiple GPUs, in: IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2014, pp. 382–391.

[23] G. Stewart, Block Gram-Schmidt orthogonalization, SIAM J. Sci. Comput. 31 (2007) 761–775.

[24] A. Stathopoulos, K. Wu, A block orthogonalization procedure with constant synchronization requirements, SIAM J. Sci. Comput. 23 (2002) 2165–2182.

[25] I. Yamazaki, S. Tomov, T. Dong, J. Dongarra, Mixed-precision orthogonalization scheme and adaptive step size for improving the stability and performance of CA-GMRES on GPUs, in: Proceedings of International Meeting on High Performance Computing for Computational Science (VECPAR), 2014, pp. 17–30.

[26] B. Parlett, D. Scott, The Lanczos algorithm with selective orthogonalization, Mathematics of Computation 33 (145) (1979) 217–238.

[27] H. Simon, The Lanczos algorithm with partial reorthogonalization, Mathematics of Computation 42 (165) (1984) 115–142.

[28] J. M. Pérez, P. Bellens, R. M. Badia, J. Labarta, Cellss: Making it easier to program the cell broadband engine processor, IBM Journal of Research and Development 51 (5) (2007) 593–604.

[29] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, G. Quintana-Ortí, Parallelizing dense and banded linear algebra libraries using SMPSs, Concurr. Comput. 21 (18) (2009) 2438–2456.

[30] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, OMPSS: A proposal for programming heterogeneous multi-core architectures, Parallel Processing Letters 21 (2011) 173–193.

[31] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, S. Thibault, Achieving high performance on supercomputers with a sequential task-based programming model, Tech. Rep. RR-8927, Universiśe de Bordeaux (2016).

[32] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurr. Comput. 23 (2011) 187–198.

[33] M. Tillenius, Superglue: A shared memory framework using data versioning for dependency-aware task-based parallelization, SIAM Journal on Scientific Computing 37 (6) (2015) C617–C642.

[34] A. YarKhan, Dynamic task execution on shared and distributed memory architectures, Ph.D. thesis, University of Tennessee (2012).

[35] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical linear algebra on emerging architectures: The plasma and magma projects, in: Journal of Physics: Conference Series, Vol. 180, IOP Publishing, 2009, p. 012037.