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 pipelined and s-step GMRES, respectively referred to as ℓ-GMRES and s-GMRES, on distributed-multicore CPUs. Compared to standard GMRES, s-GMRES requires fewer all-reduces, while ℓ-GMRES overlaps the all-reduces with computation. To combine the best features of two algorithms, we propose another variant, (ℓ, t)-GMRES, that not only does fewer global all-reduces than standard GMRES, but also overlaps those all-reduces with other work. We implemented the thread-parallelism and communication-overlap in two different ways. The first uses nonblocking MPI collectives with thread-parallel computational kernels. The second relies on a shared-memory task scheduler. In our experiments, (ℓ, t)-GMRES performed better than ℓ-GMRES by factors of up to $1.67\times$. In addition, though we only used 50 nodes, when the latency cost became significant, our variant performed up to $1.22\times$ better than s-GMRES by hiding all-reduces.

I. Introduction

Krylov subspace projection methods iteratively solve large systems of linear equations. Krylov methods can solve problems too large for other kinds of algorithms like factorizations, as well as problems where the coefficient matrix A is only available as a function that takes an input vector $\mathbf{x}$ and returns the resulting vector $\mathbf{y}$ of the product $\mathbf{y} := A\mathbf{x}$. However, on the current computers, the performance of the Krylov methods is often dominated by communication [1], as the communication has become much more expensive compared to computation, in terms of both throughput and energy consumption. We use the term "communication" to include both "horizontal" data movement between parallel processing units, as well as "vertical" data movement between memory hierarchy levels. Two approaches have been developed to reduce this communication cost. The first, "communication avoiding" (CA), is based on an s-step method which redesigns the algorithm to communicate less by generating a set of s basis vectors at a time [1]. The second, "pipelining," redesigns algorithms to hide the cost of communication by exploiting nonblocking communication and pipelining the iterations [2]. Both s-step and pipelining techniques may increase computational cost, but may nevertheless improve overall performance, due to the gap between the cost of communication and computation.

In this paper, we begin by comparing techniques to avoid or pipeline communication in a particular Krylov solver, the Generalized Minimum Residual (GMRES) method [3] for solving nonsymmetric linear systems. To the best of our knowledge, we are the first to compare the performance of the two methods in a single work. Furthermore, unlike previous studies, we focus on the interaction between thread parallelism and overlap of computation and internode communication. To this end, we wrote two implementations. The first uses threaded computational kernels and nonblocking MPI collectives. The second one relies on a shared-memory run-time system called QUARK [4], both to expose thread-level task parallelism in computations and to overlap those computations with communication tasks. We refer to our s-step and pipelined implementations as s-GMERS and ℓ-GMRES, respectively.

Our implementations focus on reducing the cost of global all-reduces needed for orthogonalizing the basis vectors. This builds both on the observation behind pipelined methods, that global collectives will be the performance bottleneck at large scale, and on our experiences using s-step GMRES, where most of our performance improvement came from the block orthogonalization to reduce global communication. Hence, to generate the Krylov vectors, s-GMRES relies on standard sparse-matrix vector multiply (*SpMV*) instead of its CA variant, the matrix powers kernel (*MPK*) [5]. Hence, s-GMRES communicates to generate each Krylov vector, but it is only the point-to-point communication among neighboring processes, which we aim to overlap with local computation of *SpMV*. By not using *MPK*, we avoid the computation, communication, and storage overheads of setting up *MPK*, as well as its computation overhead and a potential increase in total communication volume of generating the basis vectors (*MPK* trades off these overheads 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 [6], [7] made attempts at preconditioning *MPK*, but effectively preconditioning *MPK* remains a challenge. We overcome this challenge by trading some communication (using *SpMV* instead of *MPK*) in favor of having the freedom to pick any preconditioner.

We then combine these two approaches, pipelining and s-step, into a single new algorithm, *Pipelined s-step GMRES*, or (t, ℓ)-GMERS in short. This is achieved by combining s-GMRES' block orthogonalization with ℓ-GMRES. In our performance comparison on a distributed-memory computer with up to 300 processes, we have seen that this combination can improve the performance of ℓ-GMRES by a factor of

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* M:
1. `MPI_Isend` and `MPI_Irecv` for $\mathbf{q}_j$'s 1-level ghost with neighbors and then `MPI_Wait`
2. $\mathbf{v}_{j+1} := AM^{-1}\mathbf{q}_j$
  // *Orth* based on Clasical Gram-Schmidt (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. GMRES to generate the orthonormal basis vectors $Q_{1:m+1}$ and the projected matrix H, where m is the restart length.

$1.67\times$ and that of s-GMRES by $1.22\times$.

II. Algorithms

In this section, we review the standard GMRES algorithm, and two variants thereof: pipelined and s-step GMRES.

A. GMRES

The Generalized Minimum Residual (GMRES) method [3] solves a nonsymmetric linear system $A\mathbf{x} = \mathbf{b}$. Its j-th iteration first applies *SpMV* to the previously orthonormalized basis vector $\mathbf{q}_j$ and generates a new Krylov vector $\mathbf{v}_{j+1} := A\mathbf{q}_j$. Then, GMRES computes the new basis vector $\mathbf{q}_{j+1}$ by orthonormalizing (*Orth*) $\mathbf{v}_{j+1}$ against $\mathbf{q}_1, \ldots, \mathbf{q}_j$.

To reduce both the computational and storage costs of computing a large projection subspace, GMRES "restarts" the iteration 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, the iteration is restarted, using the residual vector $\mathbf{b} - A\widehat{\mathbf{x}}$ as its starting vector $\mathbf{q}_1$. The matrix H, a by-product of orthogonalization, has upper Hessenberg form, and the cost of solving the least-squares problem is small relative to the cost of generating the basis vectors for typical restart lengths and sizes of the matrix A. Figure 1 shows pseudocode for restarted GMRES. In this paper, we focus on the right preconditioning, but our discussion can be extended to the left-preconditioning.

We distribute the matrix and vectors over MPI processes in a "one-dimensional" block row format. Both *SpMV* and *Orth* require communication. This includes two global all-reduces in *Orth*, and point-to-point neighborhood messages for *SpMV* to exchange the one-level ghost entries of the vector, which are the nonlocal entries one edge away from the local entries in the adjacency graph of A. Besides these instances of interprocess communication, there are also those in the intraprocess context: the data movement between levels of the local memory hierarchy.

B. Pipelined GMRES

Pipelined GMRES [2] aims to hide the cost of the global all-reduce in *Orth*, by overlapping them with both local computation and with *SpMV*'s point-to-point communication. Hiding the communication in a single *Orth* can lead to at most $2\times$ speedups, but a greater speedup may be possible by pipelining 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 multiple global all-reduce calls. Finally, pipelined methods allow the domain scientist to use any preconditioner. This is essential in practice for good convergence rates, or even to converge at all on hard problems.

ℓ-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* M:
1. `MPI_Isend` and `MPI_Irecv` for $\mathbf{v}_j$'s 1-level ghost with neighbors and then `MPI_Wait`
2. $\mathbf{v}_{j+1} := AM^{-1}\mathbf{v}_j$
3. $k := \max(1, j - \ell + 1)$
4. **if** $j > \ell$ **then**
5. `MPI_Wait(tag = k)`
6. Update $\mathbf{r}_{1:k,k}$, and generate $\mathbf{h}_{1:k,k-1}$
  // *Orth* based on CGS:
7. $\mathbf{q}_k := (\mathbf{v}_k - Q_{1:k-1}\mathbf{r}_{1:k-1,k})/r_{k,k}$
  // change-of-basis to generate next vector:
8. $\mathbf{v}_{j+1} := (\mathbf{v}_{j+1} - V_{\ell:j}\mathbf{h}_{1:k-1,k-1})/h_{k,k-1}$
9. **end if**
10. $\mathbf{r}_{1:j+1,j+1} := [Q_{1:k-1}, V_{k:j+1}]^T \mathbf{v}_{j+1}$, with `MPI_Iallreduce(tag = j + 1)`
end for

Fig. 2. Pipelined ℓ-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.

Figure 2 shows the pipelined ℓ-GMRES. Compared to the standard algorithm, ℓ-GMRES generates an extra small 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 their orthonormalized versions. When computing the $(j+1)$-th column vector $\mathbf{r}_{1:j+1,j+1}$, the last ℓ vectors $V_{k:j+1}$ have not yet been orthogonalized (Line 10). Hence, after the corresponding synchronization, the column vector must be updated, assuming orthogonality of the previous vectors holds (Line 6). Then, the orthonormal basis vectors Q are generated using R (Line 7).

In finite-precision arithmetic, the algorithm loses orthogonality among the basis vectors faster than in standard GMRES. To maintain numerical stability, ℓ-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 [2], the matrix $B_{1:j+1,1:j}$ is defined to generate the Newton basis with shifts σ_k for the first ℓ steps, i.e.,

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

In our experiments, the shifts are the Ritz values, in a Leja order, computed at the first restart (our first restart cycle is based on standard GMRES). Then, for the following j-th step, it uses all the information available from the orthogonalization

(h, s)-GMRES(A, M, $\mathbf{b}$, h, 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 $j = 1, 2, \ldots, m$ **do**
1. **for** $i = 1, i+h, \ldots, s$ **do**
   // Matrix Powers Kernel (*MPK*(h, A, M, $\mathbf{q}_1$)):
2. `MPI_Isend` and `MPI_Irecv` for $\mathbf{q}_j$'s h-level ghost with neighbors and then `MPI_Wait` to form $\mathbf{v}_j$
3. **for** $k = i, i+1, \ldots, i+h-1$ **do**
   // *SpMV* with local submatrix and $s-k$ ghost of A,
   // and optionally *Precond* M:
3. $\mathbf{v}_{j+k} := AM^{-1}\mathbf{v}_{j+k-1}$
4. **end for**
5. **end for**
   // Block Orthonormalization (*BOrth*):
   $R_{1:j,j+1:j+s} := Q_{1:j}^T V_{j+1:j+s}$ with `MPI_Allreduce`
   $Q_{j+1:j+s} := V_{j+1:j+s} - Q_{1:j} R_{1:j,j+1:j+s}$
6. // 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. s-step GMRES with two step sizes h and s, and restart length m.

at the $(j-\ell)$-th step, i.e.,

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

where $k := j - \ell + 1$, and the projected matrix is computed by $H_{1:j+1,1:j} := R_{1:j+1,1:j+1} B_{1:j+1,1:j} R_{1:j,1:j}^{-1}$.

It takes extra computation to change basis, but ℓ-GMRES overlaps each all-reduce with ℓ iterations, including the $\ell - 1$ all-reduces and ℓ neighborhood collectives needed over the iterations. In addition, the algorithm only performs one all-reduce per iteration, compared with 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 [2]. However, even with the change-of-basis, the algorithm could lose numerical stability due to loss of orthogonality among the basis vectors, and the pipeline depth ℓ must be kept small.

C. s-step GMRES

s-step methods [9] were originally proposed as a way to improve Krylov methods' theoretical upper bound on parallelism. To avoid communication, the CA variants of the computational kernels (e.g., the orthogonalization and *MPK*), which can be implemented with the minimum communication costs, have been integrated into the s-step solvers [5], [1], [10], [11]. Such CA methods generate s basis vectors with the same communication latency of their conventional counterparts to generate a single vector. The effectiveness of such CA kernels to improve performance has been demonstrated on various architectures [5], [12], [13], [6].

To avoid the point-to-point communication needed for *SpMV*, *MPK* extends the local submatrix of A on each process with the $(s-1)$-level ghost boundary elements that are the nonlocal entries $s-1$ edges away from the local entries in the adjacency graph of A. To generate a new set of s Krylov vectors, each process first gatheres the s-level ghost entries of the starting vector $\mathbf{q}_j$ from the neighboring processes. 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, each k-th step of *MPK* applies *SpMV* to the $(s-k)$-level ghost elements. This requires extra computation.

To reduce the cost of all-reduces needed for the orthogonalization, a block procedure (*BOrth*) orthogonalizes a set of s basis vectors at a time. In our implementation, we used block classical Gram-Schmidt [14] to orthogonalize the new Krylov vectors generated by *MPK* against the previously orthonormalized basis vectors. We then orthonormalize the block of new basis vectors using the Cholesky QR (CholQR) factorization [15]. Compared to *MPK*, both *BOrth* and CholQR are easier to implement efficiently since they depend mainly on dense matrix-matrix multiply. In our previous studies, these block procedures performed well on distributed-memory computers [12], [6].

Figure 3 shows the resulting s-step GMRES. To reduce the overheads associated with *MPK*. we rely on two different step sizes h and s for *MPK* and *BOrth*, respectively. By using a smaller step size for *MPK*, we can control the overhead associated with *MPK* and maintain the quality of the DD preconditioner. The algorithm have three drawbacks. First, at each step of *MPK*, each generated vector increasingly becomes linearly dependent wrt. the previous basis vectors in that *MPK* round. Mitigation techniques are the same as in ℓ-GMRES, namely choosing a different basis, but the step size is limited in practice. Second, *MPK* reduces the communication latency cost, but it introduces redundant storage and computation among the neighboring processes. Furthermore, the total communication volume could increase depending on the sparsity pattern of the matrix A. Third, though several preconditioners have been proposed for *MPK* [6], [7], it is still a challenge to integrate preconditioning into *MPK*.

III. Algorithmic Variants

In this section, we describe two algorithmic variants of the s-step and pipelined methods that are designed to improve performance by combining the strengths of these two methods. We also consider an option to reduce the computational overhead associated with pipelining.

A. Avoiding all-reduces

In our previous studies with s-step methods [5], [6], [12], [13], most performance improvements came from using the block orthogonalization procedure. This is mainly because at large scales, the solver's parallel performance is often limited by the all-reduces needed for orthogonalization, and not by the neighborhood collective for *SpMV*.

In this paper, we focus on a variant of the s-step method that relies on block orthogonalization to reduce the global

(ℓ,t)-GMRES(A, M, $\mathbf{b}$, t, ℓ, m):
repeat
for $j = 1, 1+t, \ldots, m$ **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 $\mathbf{q}_j$'s 1-level ghost with 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. $Q_{i:k} := Q_{i:k}R^{-1}_{i:k,i:k}$
  // change-of-basis to 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. **else**
19. $k := 0$
20. **end if**
21. $R_{1:j+t+1,j+1:j+t} := [Q_{1:k}V_{k+1:j+t}]^T V_{j+1:j+t}$, with `MPI_Iallreduce(tag = j + 1)`
end for

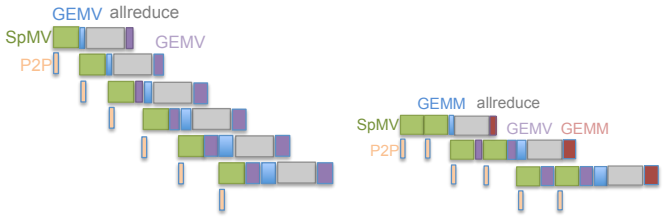
Fig. 4. Pipelined t-step GMRES, where ℓ is the pipeline depth, t is the step size, and m is the restart length.

communication cost but uses standard *SpMV* to generate the Krylov vectors (i.e., $h = 1$). For the rest of the paper, we refer to the resulting implementation as s-GMRES for simplicity. This variant also allows us to use any preconditioner, which is a great benefit in practice. This leads to an interesting performance comparison of the two techniques, one to reduce and the other to pipeline the all-reduces. Both approaches generate each Krylov vector by calling *SpMV* (with sparse neighborhood collectives) but then s-GMRES performs two all-reduces to block-orthogonalize the s vectors at once, while ℓ-GMRES hides and pipelines the all-reduces.

s-GMRES addresses two of *MPK*'s trade-offs: the overheads associated with *MPK*, and the challenge to precondition *MPK*. However, it could still suffer from numerical instability due to the application of matrix powers without orthogonalization. Hence, to maintain numerical stability, just as in ℓ-GMRES, we use the Newton basis [16], whose shifts are the Ritz values computed at the first restart.

B. Pipelining block orthogonalization

s-GMRES in Section III-A only performs an all-reduce every s steps. Despite use of a Newton basis, numerical stability still limits the step size s (e.g., in our experiments, $s \leq 10$). Furthermore, the algorithm is still block synchronous. To hide these synchronization points, we consider a combination of s-GMRES and ℓ-GMRES. The extension not only uses s-GMRES' block orthogonalization, but also pipelines the orthogonalization's all-reduces. Figure 4 shows the resulting *pipelined s-step GMRES*, where to distinguish from the step



(a) Pipelined($\ell = 3$). (b) Pipelined t-step($\ell = 2$, $t = 2$).

Fig. 5. Illustration of pipelined GMRES. Blocks colored in green, blue, purple, and gray represent *SpMV*, computation of R, *Orth*, and all-reduce, respectively. Blue and purple blocks of Pipelined t-step block-orthogonalize t vectors, while those of Pipelined orthogonalize one vector at a time.

size used in s-GMRES, we use t to denote the step size in this new (ℓ, t)-GMRES. Instead of launching a nonblocking all-reduce after generating each Krylov vector, this variant does so after generating t basis vectors. After the all-reduce, the algorithm block-orthogonalizes the t vectors all at once. If each all-reduce takes longer than a single *SpMV*, this variant allows us to hide the communication behind the block generation of t orthonormal basis vectors, without needing to increase the pipeline depth ℓ. Figure 5 illustrates two benefits of (ℓ, t)-GMRES over ℓ-GMRES. First, it improves the intraprocess performance of orthogonalization by using BLAS-3 instead of BLAS-2 kernels. Second, it does a factor of t fewer global all-reduces.

Like ℓ-GMRES, (ℓ, t)-GMRES requires an extra $t\ell$ iterations at the end of each restart cycle to drain the pipeline. However, we perform only the orthogonalization for these additional iterations, while the orthogonalization is not performed for the first $t\ell$ iterations. Hence, just like in ℓ-GMRES, (ℓ, t)-GMRES' main computational overhead comes from the change-of-basis (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 s-GMRES. This is because the pipelined method often suffers from numerical instability when used with a large pipeline depth (e.g., $s = 10$ and $t = 5$ with $\ell = 2$ in our experiments). Specifically, since (ℓ, t)-GMRES generates the Krylov vector without orthogonalizing it against the $t\ell$ previous vectors, for (ℓ, t)-GMRES to be stable, ℓ-GMRES must be stable with the pipeline depth of $t\ell$. Hence, (ℓ, t)-GMRES trades off the s-GMRES's benefits of block orthogonalization in order to pipeline the global all-reduces. We expect that at large scales, the latter will be the performance bottleneck, thus making (ℓ, t)-GMRES attractive.

C. Forming partial change-of-basis

(ℓ, t)-GMRES in Section III-B orthogonalizes the basis vectors using BLAS-3, and compared with ℓ-GMRES in Section II-B, it reduces the communication latency cost of the orthogonalization by a factor of t. However, though it does not require interprocess communication, the change-of-basis is applied to the vectors V one vector at a time using BLAS-2 kernels (Line 5 in Figure 4).

Equation (1) holds for any matrix B of full column rank. 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-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.

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 nonblocking MPI collectives, and one that is 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. For example, as a solver developer, we prefer that the progress of the non-blocking collective is handled at the lower level of the software stack (i.e., by MPI or runtime).

A. Implementation with QUARK 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 [17], [18], [19], [20], [21], [22]. To utilize this programming model, the programmer sequentially 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 in parallel, consistently with their sequential execution. Hence, the programability of the parallel code can be at about the same level as that of a sequential code. At the same time, the task execution breaks the synchronization points, allowing independent tasks from different kernels to be executed at the same time. This may lead to more efficient utilization of many cores and higher performance than using the traditional threaded computational kernels. This led to the recent adaptation of the programming model by the OpenMP standard.

In this work, we used this task-based programming model to execute our solver on a distributed-memory computer, relying on a shared-memory runtime system to locally schedule both the computational and communication tasks of the process on shared-memory multicores. Since each runtime system only handles the local tasks of the process, it does not construct the global directed acyclic graph (DAG) of all the processes, which often limits the parallel scalability of a superscalar scheduler. For the runtime system, we focused on QUARK [22], which was developed for executing the linear algebra algorithms on a shared-memory multicore architectures. Our choice is mainly due to our familiarity with the runtime

```
void QUARK_CORE_zspmv_gather(sparse_desc A, Complex64_t *g) {
  Task *task = Task_Init(quark, CORE_zspmv_gather_quark, task_flags );
  Pack_Arg(task, sizeof(sparse_desc),                    &A,    VALUE);
  Pack_Arg(task, sizeof(Complex64_t)*A.global_m,          g,    NODEP);

  // INPUT only for local "underlap" tiles to be sent
  for (int k=0; k<A.send_blocks[0]; k++)
    Pack_Arg(task, sizeof(Complex64_t)*A.mb,
                   &g[A.send_blocks[k+1]], INPUT);

  // OUTPUT only for non-local "ghost" tiles to be received
  for (int k=0; k<A.recv_blocks[0]; k++)
    Pack_Arg(task, sizeof(Complex64_t)*A.mb,
                   &g[A.recv_blocks[k+1]], OUTPUT);
```

(a) QUARK wraper.

```
void CORE_zspmv_gather(int iter, sparse_desc A, Complex64_t *g) {
  for (p=0; p<A.num_mpi; p++) { // TODO: access only neigbor processes
    if (p != A.mpi_id) then
      int count = A.num_send_vecs[p];
      if (count > 0) then
        // prepare buffer for MPI_Isend
        for (i=0; i<count; i++) // vector elements to be sent to p
          send_buffer[send+i] = g[A.send_vecs[p][i]];
        // start MPI_Isend
        MPI_Isend(&send_buffer[send], count, MPI_DOUBLE, p,
                  iter, MPI_COMM_WORLD, &(A.send[p][request_id]));
        send += count;
      end if
      // set up MPI_Irecv
      ...

  for (p=0; p<A.num_mpi; p++) do
    if (p != A.mpi_id) then
      // wait for MPI_Isend
      if (A.num_send_vecs[p] > 0)
        MPI_Wait(&(A.send[p][request_id]), &status);
      // wait for MPI_Irecv
      ...
```

(b) core subroutine.

Fig. 6. Communication task for *SpMV*'s point-to-point communication.

system, but QUARK provides some features that are useful for our studies but not yet available, for example, in OpenMP. For instance, for each process to exploit the thread-parallelism, its matrix or vector operation is split into multiple tasks, where each task works on block rows of the process-local dense or sparse submatrices which are stored either in the column-major format or in the compressed sparse row (CSR) format, respectively. Hence, the task works on the data that may not be contiguous in the memory. This violates the requirement that the tasks in OpenMP work on the contiguous data blocks. 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 is to merge 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.

Our main motivation for using the runtime system is to ensure our all-reduces overlap with other tasks. To accomplish 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. At each iteration, each process inserts a single communication task before inserting the independent computation tasks for *SpMV* or *Orth* to be executed on multiple cores. We rely on MPI_THREAD_MULTIPLE support for the independent communication tasks to be executed from different threads at the same time (e.g., the neighborhood communication and the all-reduce, or the all-reduces from different iterations, both in pipelined GMRES). Since these tasks block and wait for the communication to complete, some 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, only a limited number of communication tasks are being executed at a time (i.e., small pipelining depth, ℓ), and on a 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 idle time but still allowing an out-of-order execution of the tasks.

Figure 6 shows our QUARK wrapper, which defines the data dependencies, and the core subroutine, which is executed when all the data dependencies are satisfied, for the neighborhood communication of *SpMV*. QUARK allows us to specify the data dependencies through for-loops. For each process, this communication task has the input dependencies to all the local blocks of the input vector of *SpMV*, that are owned by this process and contain the vector elements which need to be sent to its neighboring processes. Then, the communication task has the output dependencies to the non-local blocks of the input vector, which contains the ghost elements for *SpMV*. For each neighbor process, the core subroutine packs the local vector elements, which need to be sent to the process, into a communication buffer and launch a non-blocking send. Similarly, the core subroutine launchs the non-blocking receive to gather all the non-local vectors elements from the neighbors. Then, it waits for the completion of the exchange, and expands the received elements into the global input vector.

To overlap the neighborhood communication with the local computation for *SpMV*, we split the local submatrix into two parts: the *interior* that are only connected to the local elements in the adjacency graph of A, and the *local interface* that are connected to a non-local element. With this partition, *SpMV* with the interior can be performed along with 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 data movement between the physical cores, slowing down the iterative process. Alternatively, we could append the interface to the last block of the interior. However, this will leave one core idle during the orthogonalization because the submatrix is partitioned into $n_t - 1$ blocks for *SpMV* (to leave one core for communication). For our experiments, we did not repartition the vectors for *Orth*, hence, using the same partitioning for *SpMV* and *Orth*, which obtained the best performance in most cases.

```
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++) do
    // 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,
        // scale: Q(:, j+1) /= H(j+1, j)
        &H(j+1, j), Q(i, j+1), ldq,
        // copy: G_local := Q(:, j+1) for next SpMV
        G_local(i, 0), A.global_m);
  end for
  // prepare to restart
end while
```

Fig. 7. GMRES implementation with QUARK, where mt is the number of local blocks and mbi[i] is the i-th block size and the "parallel" for-loops allow the execution of independent tasks. The communication tasks are identified by the italic letters in the comments, *neighborhood comm and global reduce.*

Putting all these together, Figure 7 shows our QUARK implementation of GMRES. They preserve the structure of the sequential algorithm in Figure 1 and enable high productivity. At each restart, there is an implicit synchronization to solve the least-square problem, where we insert the explicit synchronization to check for the convergence. This synchronization also reduces the number of the tasks that QUARK manage, reducing the scheduling overhead.

B. Implementation with MPI nonblocking communication

Our second implementation relies on the MPI's nonblocking point-to-point and all-reduce communication support, and the threaded computational kernels (e.g., threaded MKL for sparse and dense matrix operations). This MPI implementation is almost identical to our QUARK implementation, except that we directly call the core subroutines without the QUARK wrappers (e.g., calling CORE_zspmv_gather instead of

QUARK_CORE_zspmv_gather of Figure 6). To provide enough computation for the threaded 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 so that the neighborhood communication is hidden behind *SpMV* with the interior points.

Compared to the task-based model in Section IV-A, this MPI-based implementation has trade-offs in terms of programability. Clearly, the task-based model could introduce a challenge to keep track of the task dependencies, especially with the pipelined methods that have several independent tasks from different phases of iteration.[1] 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).

V. Performance Results

We now study the solver performance on a distributed-memory multicore CPUs. 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, and in Section V-D, we compare the performance of our two different implementations of the solvers, 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, and these nodes are connected by 80Gbps QDR InfiniBand. We compiled our code using mpicc from MPICH version 3.2 that implements MPI_Iallreduce using TCP and IP-over-Infiniband. We configured MPICH with --enable-threads=multiple and initialized the MPI library using MPI_THREAD_MULTIPLE. We chose to use MPICH mainly because of its MPI_Iallreduce's capability to overlap the communication with the computation (see Section V-B). For our computational tasks, we linked our code with threaded MKL version XE2013.1.046 (setting MKL_NUM_THREADS to be one for our QUARK implementation). Previous work used different matrices to study the numerics and performance of the pipelined and s-step GMRES separately. In this paper, we compare the performance of the pipelined and s-step methods. We focus on five-point 2D Laplace with square grids, $n_x \times n_x$, which were used in many studies including [2] and provide analyzable performance, but we also provide a few results using 3D and other types of matrices. We report the best performance among five runs.

Our MPI implementation can potentially utilize all the cores unlike our QUARK implementation, where several cores may be idle waiting for the communication tasks to be scheduled by other processes. However, when we use our MPI implementation with the progress thread (by setting MPICH_ASYNC_PROGRESS to be one), the performance of our solver was significantly degraded if we did not leave one spare core open for the progress thread.

[1] 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. 8. Communication and computation overlap, $n_p = 240$, progress threads.

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 (e.g., 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 or MPI_Irecv for *SpMV* as well as MPI_Iallreduce still in-progress from the previous iteration (these waits may allow MPI_Iallreduce to advance without the progress thread). It was also critical to bind the process to a set of specific cores for the QUARK implementation. We disabled hwloc for QUARK and let OS schedule the tasks on the cores that the process is bound to.

Since our focus in this paper is not on the MPI's performance, we focus on studying the solver performance with a particular setup (when using MPI_Iallreduce, the progress thread was always enabled for consistency). We plan to extend our benchmark results in a future report.

B. Benchmarking non-blocking all-reduce

We have tested both MPI's and solver's performance using various implementations, versions, and configurations of MPIs. Due to the limited space, we only present results of MPICH used in our experiments, that provided good pipelining performance of MPI_Iallreduce on our testbed. Figure 8 tests how well nonblocking all-reduce overlaps with computation, using Intel MPI Benchmark (IMB). The time between a call to MPI_Iallreduce 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 repeated computation of a small in-cache dense matrix-vector multiply that is supposed to take as long as t_{pure} but with the actual nonblocking 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. Total time to finish the simultaneous communication and computation is denoted by t_{ovrl}, and 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 $74 \sim 89\%$.

#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/process, $n_p = 20$.								
$n_t = 2$	9.74	9.66	9.77	9.42	9.75	9.32	9.61	9.25
5	8.79	8.97	8.72	9.26	8.50	10.58	10.87	10.50
ℓ calls to `MPI_Iallreduce`/`MPI_Wait` from n_t threads/process, $n_p = 20$								
$n_t = 2$	9.86	10.21	9.81	9.96	9.77	10.08	9.97	9.90
5	8.66	9.23	8.11	9.31	9.79	11.14	12.59	11.93

Fig. 9. Pipeline results with pipeline depth $\ell = 10$ and process count n_p.

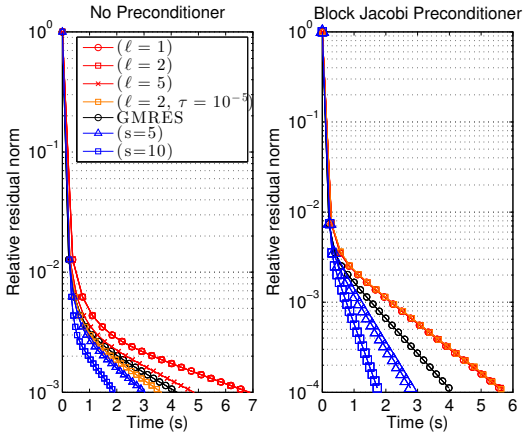


Fig. 10. Convergence with respect to the iteration time for 2D Laplace, $n_x = 512$. Time was measured at each restart, identified by marker ($m = 30$). All the solvers converged equivalently with respect to the restart count.

We next study if the pipelined all-reduces can overlap on each other either through multiple calls to `MPI_Iallreduce` followed by `MPI_Waitall` or simultaneous calls to `MPI_Allreduce` from different OpenMP threads. Each thread uses a unique communicator. Figure 9 shows the ratio of the time needed to execute 10 all-reduces using the above two approaches over the time needed to perform one all-reduce. Hence, the ratio of one indicates the perfect overlap, while the ratio of 10 or greater means no overlap. These results indicate that the recursive calls to `MPI_Iallreduce` followed by `MPI_Waitall` achieve a greater overlap, giving advantage to our MPI implementation over our QUARK implementation, in terms of overlapping the pipelined all-reduces with each other. We also tried launching `MPI_Iallreduce` followed by `MPI_Wait` from different threads, but we did not observe any improvement over launching `MPI_Allreduce`.

C. Comparing solver performance – pure MPI

We now study the performance of different solvers using one thread per process (without using QUARK). Figure 10 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 orthogonality among the basis vectors). For these experiments, we only used 12 processes, and ℓ-GMRES did not improve the performance of GMRES. However, as we will show later, with a larger process count, we could stably use a sufficiently large pipeline depth or step size to obtain a good speedup (e.g., we needed $s = 5$ or $\ell = 2$ to obtain speedups, but the solver was stable using $s = 10$ or $\ell = 10$). Though we do not show the convergence for the remaining experiments, all the solvers obtained an equivalent convergence. This is true even with preconditioning since our s-GMRES communicates for each *SpMV* and can use any preconditioner. The right plot of Figure 10 used the block Jacobi preconditioner, where each process applied the sparse approximate inverse of its local submatrix.

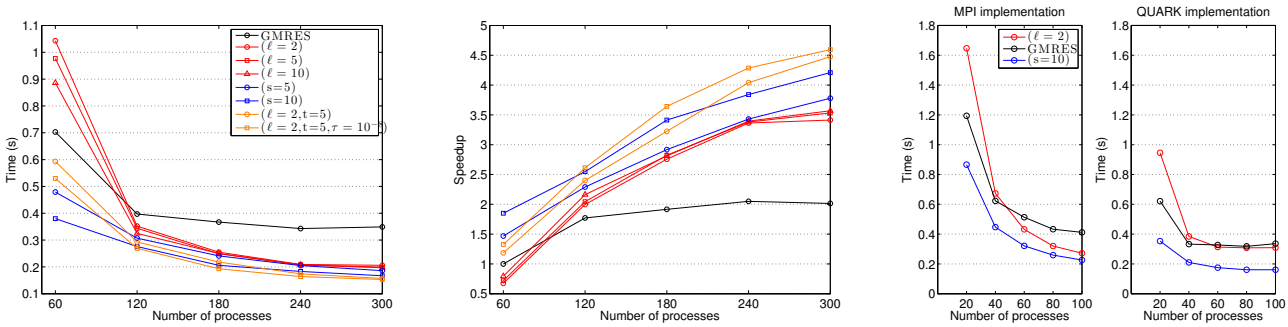
	$n_p = 120$		$n_p = 180$		$n_p = 240$		$n_p = 300$	
ℓ	block	non	block	non	block	non	block	non
0	0.40		0.37		0.34		0.35	
2	0.42	0.35	0.31	0.28	0.27	0.21	0.26	0.21
5	0.41	0.34	0.31	0.25	0.27	0.21	0.26	0.20
10	0.40	0.33	0.31	0.25	0.27	0.21	0.26	0.20

Fig. 11. Time in seconds of 20 restart-cycles of ℓ-GMRES for 2D Laplace ($n_x = 1024$,$m = 30$) and process count n_p. "block" and "non" use blocking and nonblocking all-reduces. With $\ell = 0$, standard GMRES is used.

Figure 11 shows the ℓ-GMRES' performance with different configurations. For comparison, we also show the performance of the standard GMRES (i.e., $\ell = 0$). In the table, under "block," we replaced `MPI_Iallreduce` with `MPI_Allreduce` to study the effects of overlap. Even with the blocking all-reduce, ℓ-GMRES improved the GMRES' performance because it only performs one all-reduce per iteration, compared to two all-reduces performed by GMRES. Then, using nonblocking all-reduce, the performance was further improved by a factor of about $1.3\times$.

Figures 12(a) and 12(b) compare the performance of different solvers. Even though ℓ-GMRES performs extra computation, it performed better than GMRES when the all-reduces' communication latency became significant with a large enough number of processes. s-GMRES obtained speedups even on a small number of processes since its block orthogonalization improves the cache efficiency. Furthermore, (ℓ, t)-GMRES got speedups of up to $1.67\times$ over ℓ-GMRES due to use of block orthogonalization. Using the same step size (i.e., $s = t = 5$), (ℓ, t)-GMRES also obtained speedups of up to $1.22\times$ over s-GMRES due to pipelining the all-reduces. It also got speedups of up to $1.09\times$ over s-GMRES with $s = 10$, where both solvers synchronize after 10 iterations. We observed that as the process count increased, the latency cost of an all-reduce, and hence the speedup from pipelining the all-reduces, increased.

Figure 13 shows the similar performance results for solving 27-point 3D problems. Since the coefficient matrix has more nonzeros per row and we used a smaller restart cycle, block orthogonalization led to a smaller performance improvement, compared to that for the 2D problems in Figure 12. At this small scale, (ℓ, t)-GMRES performed better than ℓ-GMRES, but pipelining the all-reduce did not improve the performance of s-GMRES (the latency savings due to block orthogonalization likely dominated). With a larger number of processes, we expect (ℓ, t)-GMRES to improve performance of s-GMRES. As shown in Figure 14, we have also observed similar results



(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) Performance of two different implementations, one process per socket, five threads per process.

Fig. 12. Execution time and speedup of 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)
ℓ-GMRES						
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)
s-GMRES						
–	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)
(ℓ,t)-GMRES						
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. 13. Time in seconds for 20 restart-cycles and $m = 20$, with 27-point 3D Laplace ($n_x = 128$). The numbers in parenthesis are the speedups over GMRES with the same processor count.

	n (M)	$\frac{nnz}{n}$	time	(ℓ)	(s)	(ℓ,t)
G3_Circuit	1.6	4.8	0.43	1.31	1.48	1.55
thermal2	1.2	7.0	0.43	1.54	1.60	1.65
atmosmodd	1.3	6.9	0.74	1.78	1.95	1.99

Fig. 14. Speedups over GMRES with $(m, \ell, s, t) = (30, 2, 10, 5)$ on 240 processes from UF Sparse Matrix Collection, where "$\frac{nnz}{n}$" is the average number of nonzero entries per row, and "time" shows the time in seconds for 20 GMRES restart-cycles. The matrices were equilibrated using the largest elements in each row and column, and distributed using METIS.

for the matrices from the University of Florida Sparse Matrix Collection, showing (ℓ, t)-GMRES improved the performance of both ℓ-GMRES and s-GMRES.

D. Comparison of two implementations – MPI+Threads

We now compare the performance of our two implementations: one using nonblocking MPI collectives, and the other using QUARK. Figure 15 shows the solvers' performance on one node with different process / thread configurations. For these experiments in the figure, we did not use ℓ-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 obtain higher kernel performance, leading to the higher performance of the solver. At the end, in most cases, the optimal performance was

	$n_p \cdot n_t$ 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
task	16.3	10.7	7.6	6.3	7.8	5.5	4.8	4.6
s-GMRES($m = 30$, $s = 10$)								
mpi	11.5	7.3	7.4	7.1	5.9	3.8	3.8	2.4
task	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
task	53.9	34.9	25.3	20.0	25.9	20.5	16.3	15.3
s-GMRES($m = 60$, $s = 10$)								
mpi	31.7	19.3	21.4	18.6	16.3	10.3	10.6	6.4
task	27.7	17.2	9.9	7.8	13.7	8.9	5.9	5.0

Fig. 15. Time in seconds for 10 restart-cycles with 2D Laplace ($n_x = 1024$) using different $n_p \times n_t$, where n_p and n_t are the number of processes and the number of threads per process, respectively. "mpi" and "task" denote our MPI and QUARK implementations.

	ℓ	s/t	τ	number of processes 20	40	60	80	100
GMRES	–	–	–	1.19	0.62	0.50	0.43	0.41
ℓ-GMRES	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
s-GMRES	–	5	–	1.04	0.50	0.36	0.29	0.26
	–	10	–	0.86	0.45	0.33	0.25	0.23
(ℓ,t)-GMRES	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.

obtained using all the cores of the node with one process either per socket or per core. We also see that even on one node, s-GMRES obtained good speedups over standard GMRES.

Figure 12(c) compares the parallel scaling of our two implementations. With a relatively small number of processes, our QUARK implementation could utilize the cores more effectively, obtaining higher performance than our MPI implementation (both in Figures 12 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, or cannot effectively pipeline the all-reduces (see Table 9). Finally, Figure 16 compares the solver performance using our

MPI implementation with multiple threads per process. As before, (ℓ, t)-GMRES obtained the speedups of up to $1.34\times$ over ℓ-GMRES through block orthogonalization. By pipelining the all-reduces, (ℓ, t)-GMRES also obtained the speedups of up to $1.12\times$ or $1.02\times$ over s-GMRES with $s = 5$ or 10, respectively.

VI. Conclusion

We began this work by comparing the performance of pipelined ℓ-GMRES and s-step s-GMRES on a distributed-memory computer. We implemented the solvers in two different ways. The first way builds on a threaded BLAS and LAPACK libraries and nonblocking MPI collectives. The second uses the QUARK shared-memory run-time system to schedule computational and communication tasks of each MPI process. We also developed a new algorithm, pipelined s-step (ℓ, t)-GMRES, that combines the strengths of the above two methods. It uses fewer global all-reduces than standard or ℓ-GMRES, by applying the same block orthogonalization approach as s-GMRES. In addition, like ℓ-GMRES, it overlaps those all-reduces with useful work, thus making it less synchronous than s-GMRES. In our experiments, (ℓ, t)-GMRES performed up to $1.67\times$ better than ℓ-GMRES, thanks to the use of block orthogonalization. Thanks to overlapping the all-reduces with useful work, (ℓ, t)-GMRES performed up to $1.22\times$ better than s-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 working with 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 or ECP). Though in our experiments, the performance of ℓ-GMRES was lower than that of s-GMRES, we expect the pipelined variant to perform better at larger scales. We have observed that ℓ-GMRES 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 the solver (e.g., reorthogonalization). Also, though the performance may not be limited by the point-to-point communication, we plan to integrate an option for (ℓ, t)-GMRES to use *MPK*. Our code is currently maintained in a private Bitbucket repository. We plan to release implementations of some of these solvers through the Trilinos (trilinos.org) project.

Acknowledgments

We thank Xi Luo at the University of Tennessee for helpful discussions on the non-blocking all-reduce communication. This research was supported in part by the U.S. Department of Energy Office of Science under Award Numbers DE-FG0213ER26137 and DE-SC0010042, and the U.S. National Science Foundation under Award Number 1339822. Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] M. Hoemmen, Communication-avoiding Krylov subspace methods, Ph.D. thesis, EECS Dep't, Univ. of Calif., Berkeley (2010).

[2] 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.

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

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

[5] M. Mohiyuddin, M. Hoemmen, J. Demmel, K. Yelick, Minimizing communication in sparse matrix solvers, in: Proc. of the Conf. on High Perf. Comput. Networking, Storage and Analysis (SC), 2009, pp. 36:1–36:12.

[6] I. Yamazaki, et al., Domain decomposition preconditioners for communication-avoiding Krylov methods on hybrid CPU/GPU cluster, in: Proc. of the Conf. on High Perf. Comput. 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.

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

[9] J. van Rosendale, Minimizing inner product data dependence in conjugate gradient iteration, in: IEEE Int'l Conf. for Par. Proc., 1983, pp. 44–46.

[10] G. Ballard, et al., Communication lower bounds and optimal algorithms for numerical linear algebra, Acta Numerica 23 (2014) 1–155.

[11] 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.

[12] 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-Gen. High-Perf. Scientific Software (HPCC), 2011, pp. 345–354.

[13] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, J. Dongarra, Improving the performance of CA-GMRES on multicores with multiple GPUs, in: IEEE Int'l Par. and Dist. Proc. Symposium (IPDPS), 2014, pp. 382–391.

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

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

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

[17] 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.

[18] R. M. Badia, et al., Parallelizing dense and banded linear algebra libraries using SMPSs, Concurr. Comput. 21 (18) (2009) 2438–2456.

[19] A. Duran, et al., OMPSS: A proposal for programming heterogeneous multi-core architectures, Parallel Processing Letters 21 (2011) 173–193.

[20] 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.

[21] M. Tillenius, SuperGlue: A shared memory framework using data versioning for dependency-aware task-based parallelization, SIAM J. Sci. Comput. 37 (6) (2015) C617–C642.

[22] A. YarKhan, Dynamic task execution on shared and distributed memory architectures, Ph.D. thesis, Univ. of Tenn. (2012).